



Community Experience Distilled

Implementing Splunk: Big Data Reporting and Development for Operational Intelligence

Learn to transform your machine data into valuable IT and business insights with this comprehensive and practical tutorial

Vincent Bumgarner

[PACKT]
PUBLISHING

Implementing Splunk: Big Data Reporting and Development for Operational Intelligence

Learn to transform your machine data into valuable IT and business insights with this comprehensive and practical tutorial

Vincent Bumgarner



BIRMINGHAM - MUMBAI

Implementing Splunk: Big Data Reporting and Development for Operational Intelligence

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: January 2013

Production Reference: 1140113

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-328-8

www.packtpub.com

Cover Image by Vincent Bumgarner (vincent.bumgarner@gmail.com)

Credits

Author

Vincent Bumgarner

Project Coordinator

Anish Ramchandani

Reviewers

Mathieu Dessus

Cindy McCririe

Nick Mealy

Proofreader

Martin Diver

Indexer

Tejal Soni

Acquisition Editor

Kartikey Pandey

Graphics

Aditi Gajjar

Lead Technical Editor

Azharuddin Sheikh

Production Coordinator

Nitesh Thakur

Technical Editors

Charmaine Pereira

Varun Pius Rodrigues

Cover Work

Nitesh Thakur

Copy Editors

Brandt D'Mello

Aditya Nair

Alfida Paiva

Laxmi Subramanian

Ruta Waghmare

About the Author

Vincent Bumgarner has been designing software for nearly 20 years, working in many languages on nearly as many platforms. He started using Splunk in 2007 and has enjoyed watching the product evolve over the years.

While working for Splunk, he helped many companies, training dozens of users to drive, extend, and administer this extremely flexible product. At least one person at every company he worked with asked for a book on Splunk, and he hopes his effort helps fill their shelves.

I would like to thank my wife and kids as this book could not have happened without their support. A big thank you to all of the reviewers for contributing their time and expertise, and special thanks to SplunkNinja for the recommendation.

About the Reviewers

Mathieu Dessus is a security consultant for Verizon in France and acts as the SIEM leader for EMEA. With more than 12 years of experience in the security area, he has acquired a deep technical background in the management, design, assessment, and systems integration of information security technologies. He specializes in web security, Unix, SIEM, and security architecture design.

Cindy McCririe is a client architect at Splunk. In this role, she has worked with several of Splunk's enterprise customers, ensuring successful deployment of the technology. Many of these customers are using Splunk in unique ways. Sample use cases include PCI compliance, security, operations management, business intelligence, Dev/Ops, and transaction profiling.

Nick Mealy was an early employee at Splunk and worked as the Mad Scientist / Principal User Interface Developer at Splunk from March 2005 to September 2010. He led the technical design and development of the systems that power Splunk's search and reporting interfaces as well as on the general systems that power Splunk's configurable views and dashboards. In 2010, he left Splunk to found his current company, Sideview, which is creating new Splunk apps and new products on top of the Splunk platform. The most widely known of these products is the Sideview Utils app, which has become very widely deployed (and will be discussed in *Chapter 8, Building Advanced Dashboards*). Sideview Utils provides new UI modules and new techniques that make it easier for Splunk app developers and dashboard creators to create and maintain their custom views and dashboards.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
<hr/>	
Chapter 1: The Splunk Interface	7
<hr/>	
Logging in to Splunk	7
The Home app	8
The top bar	11
Search app	13
Data generator	13
The Summary view	14
Search	16
Actions	17
Timeline	18
The field picker	19
Fields	19
Search results	21
Options	22
Events viewer	23
Using the time picker	25
Using the field picker	26
Using Manager	27
Summary	29
Chapter 2: Understanding Search	31
<hr/>	
Using search terms effectively	31
Boolean and grouping operators	32
Clicking to modify your search	34
Event segmentation	34
Field widgets	34
Time	35

Using fields to search	35
Using the field picker	35
Using wildcards efficiently	36
Only trailing wildcards are efficient	36
Wildcards are tested last	36
Supplementing wildcards in fields	37
All about time	37
How Splunk parses time	37
How Splunk stores time	37
How Splunk displays time	38
How time zones are determined and why it matters	38
Different ways to search against time	39
Specifying time in-line in your search	41
_indextime versus _time	42
Making searches faster	42
Sharing results with others	43
Saving searches for reuse	46
Creating alerts from searches	48
Schedule	49
Actions	51
Summary	52
Chapter 3: Tables, Charts, and Fields	53
About the pipe symbol	53
Using top to show common field values	54
Controlling the output of top	56
Using stats to aggregate values	57
Using chart to turn data	61
Using timechart to show values over time	63
timechart options	65
Working with fields	66
A regular expression primer	66
Commands that create fields	68
eval	68
rex	69
Extracting loglevel	70
Using the Extract Fields interface	70
Using rex to prototype a field	73
Using the admin interface to build a field	75
Indexed fields versus extracted fields	77
Summary	80

Chapter 4: Simple XML Dashboards	81
The purpose of dashboards	81
Using wizards to build dashboards	82
Scheduling the generation of dashboards	91
Editing the XML directly	91
UI Examples app	92
Building forms	92
Creating a form from a dashboard	92
Driving multiple panels from one form	97
Post-processing search results	104
Post-processing limitations	106
Panel 1	106
Panel 2	107
Panel 3	108
Final XML	108
Summary	110
Chapter 5: Advanced Search Examples	111
Using subsearches to find loosely related events	111
Subsearch	111
Subsearch caveats	112
Nested subsearches	113
Using transaction	114
Using transaction to determine the session length	115
Calculating the aggregate of transaction statistics	117
Combining subsearches with transaction	118
Determining concurrency	122
Using transaction with concurrency	122
Using concurrency to estimate server load	123
Calculating concurrency with a by clause	124
Calculating events per slice of time	129
Using timechart	129
Calculating average requests per minute	131
Calculating average events per minute, per hour	132
Rebuilding top	134
Summary	141
Chapter 6: Extending Search	143
Using tags to simplify search	143
Using event types to categorize results	146
Using lookups to enrich data	150
Defining a lookup table file	150

Defining a lookup definition	152
Defining an automatic lookup	154
Troubleshooting lookups	157
Using macros to reuse logic	157
Creating a simple macro	158
Creating a macro with arguments	159
Using eval to build a macro	160
Creating workflow actions	160
Running a new search using values from an event	161
Linking to an external site	163
Building a workflow action to show field context	165
Building the context workflow action	165
Building the context macro	167
Using external commands	170
Extracting values from XML	170
xmlkv	170
XPath	171
Using Google to generate results	172
Summary	172
Chapter 7: Working with Apps	173
<hr/>	
Defining an app	173
Included apps	174
Installing apps	175
Installing apps from Splunkbase	175
Using Geo Location Lookup Script	176
Using Google Maps	178
Installing apps from a file	178
Building your first app	179
Editing navigation	182
Customizing the appearance of your app	184
Customizing the launcher icon	185
Using custom CSS	185
Using custom HTML	187
Custom HTML in a simple dashboard	187
Using ServerSideInclude in a complex dashboard	188
Object permissions	191
How permissions affect navigation	192
How permissions affect other objects	192
Correcting permission problems	193
App directory structure	194

Adding your app to Splunkbase	196
Preparing your app	196
Confirming sharing settings	196
Cleaning up our directories	197
Packaging your app	198
Uploading your app	199
Summary	200
Chapter 8: Building Advanced Dashboards	201
<hr/>	
Reasons for working with advanced XML	201
Reasons for not working with advanced XML	202
Development process	202
Advanced XML structure	203
Converting simple XML to advanced XML	205
Module logic flow	210
Understanding layoutPanel	213
Panel placement	214
Reusing a query	215
Using intentions	217
stringreplace	217
addterm	218
Creating a custom drilldown	219
Building a drilldown to a custom query	219
Building a drilldown to another panel	222
Building a drilldown to multiple panels using HiddenPostProcess	224
Third-party add-ons	228
Google Maps	228
Sideview Utils	230
The Sideview Search module	231
Linking views with Sideview	232
Sideview URLLoader	232
Sideview forms	235
Summary	241
Chapter 9: Summary Indexes and CSV Files	243
<hr/>	
Understanding summary indexes	243
Creating a summary index	244
When to use a summary index	245
When to not use a summary index	246
Populating summary indexes with saved searches	247
Using summary index events in a query	249
Using sistats, sitop, and sitimechart	251

How latency affects summary queries	254
How and when to backfill summary data	256
Using fill_summary_index.py to backfill	256
Using collect to produce custom summary indexes	258
Reducing summary index size	261
Using eval and rex to define grouping fields	262
Using a lookup with wildcards	264
Using event types to group results	267
Calculating top for a large time frame	269
Storing raw events in a summary index	273
Using CSV files to store transient data	275
Pre-populating a dropdown	276
Creating a running calculation for a day	276
Summary	278
Chapter 10: Configuring Splunk	279
Locating Splunk configuration files	279
The structure of a Splunk configuration file	280
Configuration merging logic	281
Merging order	281
Merging order outside of search	281
Merging order when searching	282
Configuration merging logic	283
Configuration merging example 1	284
Configuration merging example 2	284
Configuration merging example 3	285
Configuration merging example 4 (search)	288
Using btool	290
An overview of Splunk .conf files	292
props.conf	292
Common attributes	292
Stanza types	296
Priorities inside a type	298
Attributes with class	299
inputs.conf	300
Common input attributes	300
Files as inputs	301
Network inputs	306
Native Windows inputs	308
Scripts as inputs	309
transforms.conf	310
Creating indexed fields	310
Modifying metadata fields	312
Lookup definitions	315
Using REPORT	318

Chaining transforms	320
Dropping events	321
fields.conf	322
outputs.conf	323
indexes.conf	323
authorize.conf	325
savedsearches.conf	326
times.conf	326
commands.conf	326
web.conf	326
User interface resources	326
Views and navigation	326
Appserver resources	327
Metadata	328
Summary	331
Chapter 11: Advanced Deployments	333
<hr/>	
Planning your installation	333
Splunk instance types	334
Splunk forwarders	334
Splunk indexer	336
Splunk search	337
Common data sources	337
Monitoring logs on servers	337
Monitoring logs on a shared drive	338
Consuming logs in batch	339
Receiving syslog events	340
Receiving events directly on the Splunk indexer	340
Using a native syslog receiver	341
Receiving syslog with a Splunk forwarder	343
Consuming logs from a database	343
Using scripts to gather data	345
Sizing indexers	345
Planning redundancy	348
Indexer load balancing	348
Understanding typical outages	349
Working with multiple indexes	350
Directory structure of an index	350
When to create more indexes	351
Testing data	351
Differing longevity	351
Differing permissions	352
Using more indexes to increase performance	353

The lifecycle of a bucket	354
Sizing an index	355
Using volumes to manage multiple indexes	356
Deploying the Splunk binary	358
Deploying from a tar file	359
Deploying using msiexec	359
Adding a base configuration	360
Configuring Splunk to launch at boot	360
Using apps to organize configuration	361
Separate configurations by purpose	361
Configuration distribution	366
Using your own deployment system	366
Using Splunk deployment server	367
Step 1 – Deciding where your deployment server will run	367
Step 2 – Defining your deploymentclient.conf configuration	368
Step 3 – Defining our machine types and locations	368
Step 4 – Normalizing our configurations into apps appropriately	369
Step 5 – Mapping these apps to deployment clients in serverclass.conf	369
Step 6 – Restarting the deployment server	373
Step 7 – Installing deploymentclient.conf	373
Using LDAP for authentication	374
Using Single Sign On	375
Load balancers and Splunk	376
web	376
splunktcp	376
deployment server	377
Multiple search heads	377
Summary	378
Chapter 12: Extending Splunk	379
Writing a scripted input to gather data	379
Capturing script output with no date	380
Capturing script output as a single event	382
Making a long-running scripted input	384
Using Splunk from the command line	385
Querying Splunk via REST	387
Writing commands	390
When not to write a command	390
When to write a command	392
Configuring commands	392
Adding fields	393
Manipulating data	394

Transforming data	396
Generating data	401
Writing a scripted lookup to enrich data	403
Writing an event renderer	406
Using specific fields	406
Table of fields based on field value	408
Pretty print XML	411
Writing a scripted alert action to process results	413
Summary	416
Index	417

Preface

Splunk is a powerful tool for collecting, storing, alerting, reporting, and studying machine data. This machine data usually comes from server logs, but it could also be collected from other sources. Splunk is by far the most flexible and scalable solution available to tackle the huge problem of making machine data useful.

The goal of this book is to serve as an organized and curated guide to Splunk 4.3. As the documentation and community resources available for Splunk are vast, finding the important pieces of knowledge can be daunting at times. My goal is to present what is needed for an effective implementation of Splunk in as concise and useful a manner as possible.

What this book covers

Chapter 1, The Splunk Interface, walks the reader through the user interface elements.

Chapter 2, Understanding Search, covers the basics of the search language, paying particular attention to writing efficient queries.

Chapter 3, Tables, Charts, and Fields, shows how to use fields for reporting, then covers the process of building our own fields.

Chapter 4, Simple XML Dashboards, first uses the Splunk web interface to build our first dashboards. It then examines how to build forms and more efficient dashboards.

Chapter 5, Advanced Search Examples, walks the reader through examples of using Splunk's powerful search language in interesting ways.

Chapter 6, Extending Search, exposes a number of features in Splunk to help you categorize events and act upon search results in powerful ways.

Chapter 7, Working with Apps, covers the concepts of an app, helps you install a couple of popular apps, and then helps you build your own app.

Chapter 8, Building Advanced Dashboards, explains the concepts of advanced XML dashboards, and covers practical ways to transition from simple XML to advanced XML dashboards.

Chapter 9, Summary Indexes and CSV Files, introduces the concept of summary indexes, and how they can be used to increase performance. It also discusses how CSV files can be used in interesting ways.

Chapter 10, Configuring Splunk, explains the structure and meaning of common configurations in Splunk. It also explains the process of merging configurations in great detail.

Chapter 11, Advanced Deployments, covers common questions about multimachine Splunk deployments, including data inputs, syslog, configuration management, and scaling up.

Chapter 12, Extending Splunk, demonstrates ways in which code can be used to extend Splunk for data input, external querying, rendering, custom commands, and custom actions.

What you need for this book

To work through the examples in this book, you will need an installation of Splunk, preferably a non-production instance. If you are already working with Splunk, then the concepts introduced by the examples should be applicable to your own data.

Splunk can be downloaded for free from <http://www.splunk.com/download>, for most popular platforms.

The sample code was developed on a Unix system, so you will probably have better luck using an installation of Splunk that is running on a Unix operating system. Knowledge of Python is necessary to follow some of the examples in the later chapters.

Who this book is for

This book should be useful for new users, seasoned users, dashboard designers, and system administrators alike. This book does not try to act as a replacement for the official Splunk documentation, but should serve as a shortcut for many concepts.

For some sections, a good understanding of regular expressions would be helpful. For some sections, the ability to read Python would be helpful.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "If a field value looks like `key=value` in the text of an event, you will want to use one of the field widgets."

A block of code is set as follows:

```
index=myapplicationindex
(
  sourcetype=security
  AND
  (
    (bob NOT error)
    OR
    (mary AND warn)
  )
)
```



When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:



```
<searchPostProcess>
  timechart span=1h sum(count) as "Error count" by network
</searchPostProcess>
<title>Dashboard - Errors - errors by network timechart</title>
```

Any command-line input or output is written as follows:

```
ERROR LogoutClass error, ERROR, Error! [user=mary, ip=3.2.4.5]
WARN AuthClass error, ERROR, Error! [user=mary, ip=1.2.3.3]
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Quickly create a simple dashboard using the wizard interface that we used before, by selecting **Create | Dashboard Panel**."

 Warnings or important notes appear in a box like this. 

 Tips and tricks appear like this. 

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

The Splunk Interface

This chapter will walk you through the most common elements in the Splunk interface, and will touch upon concepts that are covered in greater detail in later chapters. You may want to dive right into search, but an overview of the user interface elements might save you some frustration later. We will walk through:

- Logging in and app selection
- A detailed explanation of the search interface widgets
- A quick overview of the admin interface

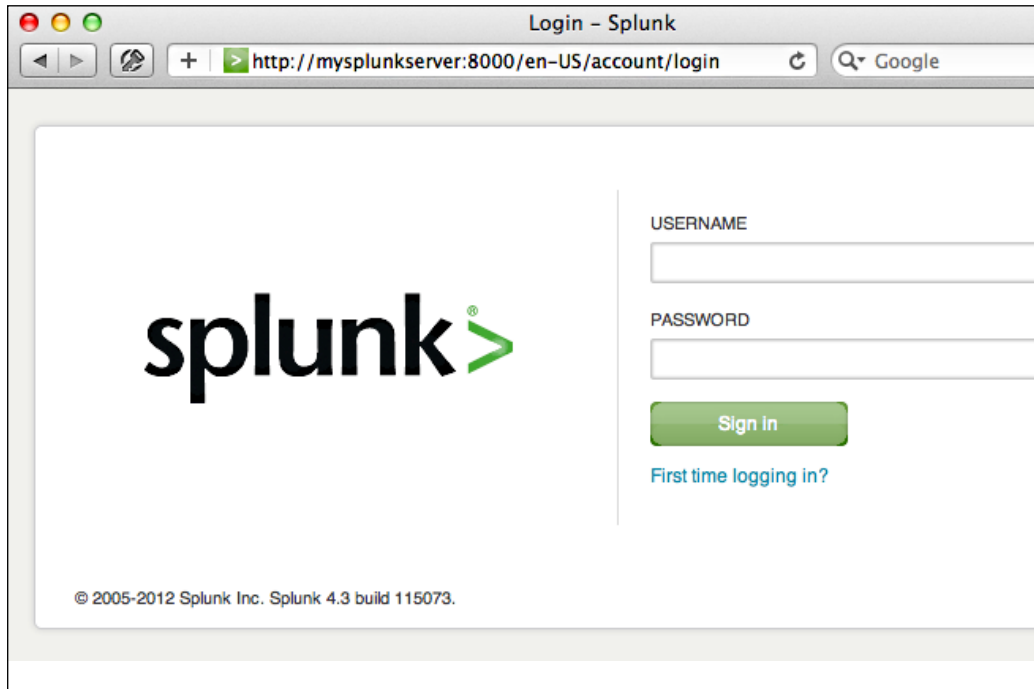
Logging in to Splunk

The Splunk interface is web-based, which means that no client needs to be installed. Newer browsers with fast Javascript engines, such as Chrome, Firefox, and Safari, work better with the interface.

As of Splunk Version 4.3, no browser extensions are required. Splunk Versions 4.2 and earlier require Flash to render graphs. Flash can still be used by older browsers, or for older apps that reference Flash explicitly.

The default port for a Splunk installation is 8000. The address will look like `http://mysplunkserver:8000` or `http://mysplunkserver.mycompany.com:8000`. If you have installed Splunk on your local machine, the address can be some variant of `http://localhost:8000`, `http://127.0.0.1:8000`, `http://machinename:8000`, or `http://machinename.local:8000`.

Once you determine the address, the first page you will see is the login screen.

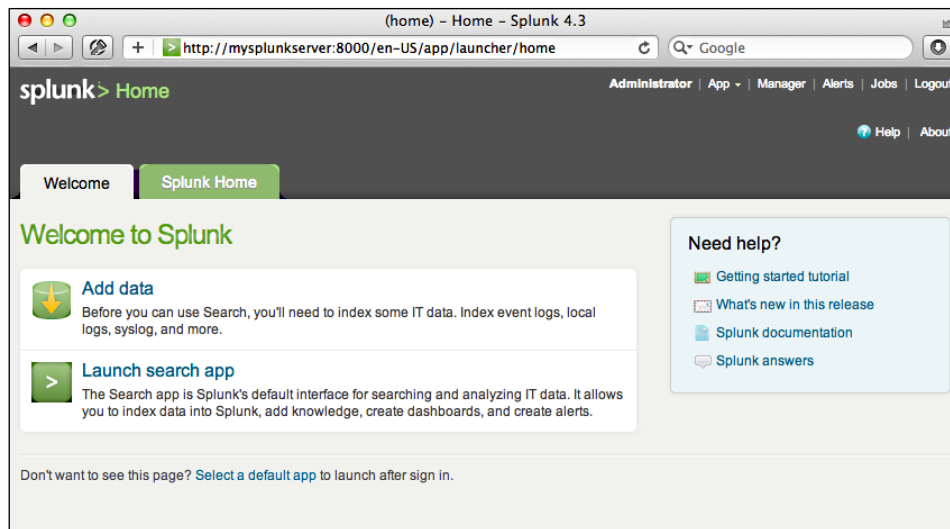


The default username is *admin* with the password *changeme*. The first time you log in, you will be prompted to change the password for the admin user. It is a good idea to change this password to prevent unwanted changes to your deployment.

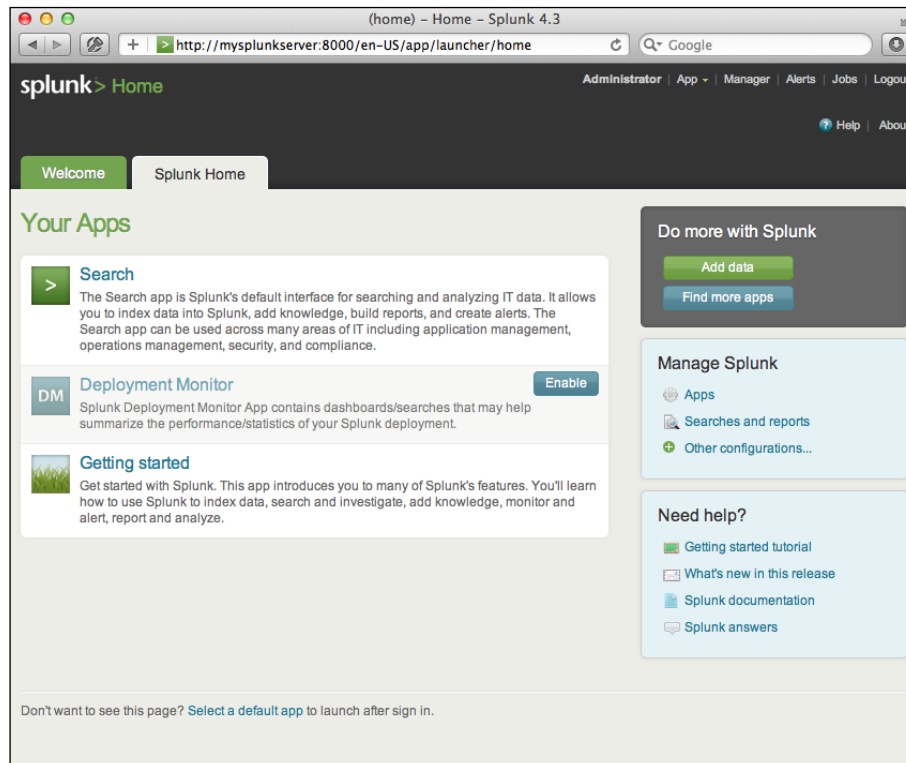
By default, accounts are configured and stored within Splunk. Authentication can be configured to use another system, for instance LDAP.

The Home app

After logging in, the default app is **Home**. This app is a launching pad for apps and tutorials.



The **Welcome** tab provides two important shortcuts, **Add data** and **Launch search app**. These links appear again on the second tab, **Splunk Home**.



The **Your Apps** section shows the apps that have GUI elements on your instance of Splunk.

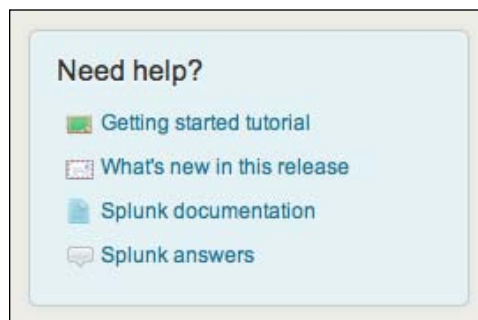


App is an overloaded term in Splunk. An app doesn't necessarily have a GUI at all; it is simply a collection of configurations wrapped into a directory structure that means something to Splunk. We will discuss apps in a more detailed manner in *Chapter 7, Working with Apps*.

Under **Do more with Splunk**, we find:

- **Add data:** This links to the **Add Data to Splunk** page. This interface is a great start for getting local data flowing into Splunk. The new **Preview data** interface takes an enormous amount of complexity out of configuring dates and line breaking. We won't go through those interfaces here, but we will go through the configuration files that these wizards produce in *Chapter 10, Configuring Splunk*.
- **Find more apps:** This allows you to find and install more apps from **Splunkbase**. Splunkbase (<http://splunk-base.splunk.com/>) is a very useful community-driven resource where Splunk users and Splunk employees post questions, answers, code snippets, and apps.

Manage Splunk takes the user to the **Manager** section of Splunk. The **Manager** section is used to configure most aspects of Splunk. The options provided change depending on the capabilities of the user. We will use the **Manager** section throughout the book as we learn about different objects.



Getting started tutorial provides a quick but thorough overview of the major functionality of Splunk.

Splunk documentation takes you to the official Splunk documentation. The documentation, hosted at `splunk.com`, is truly vast.



Two quick notes about the Splunk documentation:

To get to documentation for search and reporting commands, quick help is provided while searching, and a link to the documentation for that command is provided through the interface.

When working directly with configuration files, the fastest route to the documentation for that file is to search for `splunk name.conf` using your favorite search engine. The documentation is almost always the first link.

Splunk answers goes to the Splunkbase site we just mentioned. Splunkbase and Splunk Answers used to be different sites but were merged into one site.

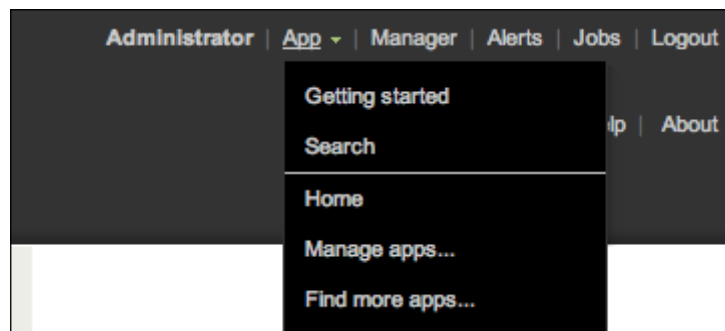
The top bar

The bar across the top of the window contains information about where you are as well as quick links to preferences, other apps, and administration.

The current app is specified in the upper-left corner.

splunk > Home

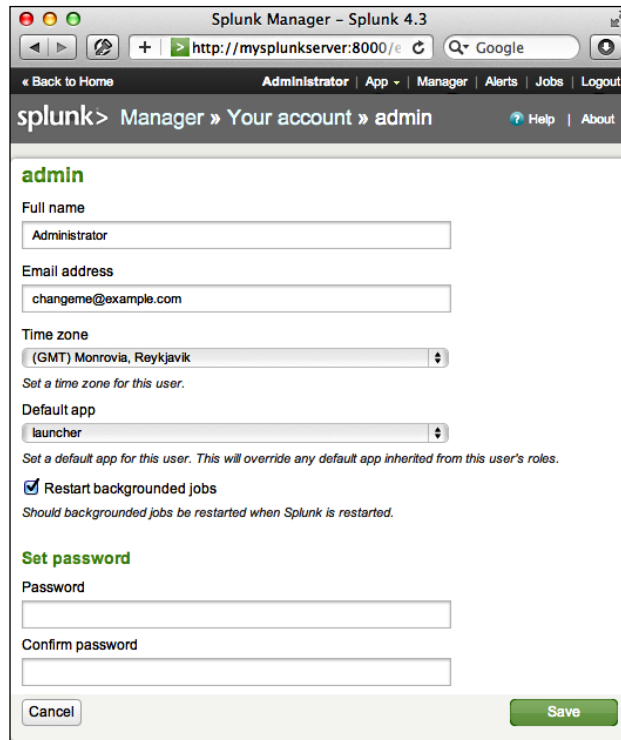
Clicking on the Splunk logo or the text takes you to the default page for that app. In most apps, the text next to the logo is simply changed, but the whole block can be customized with logos and alternate text by modifying the app's CSS. We will cover this in *Chapter 7, Working with Apps*.



The upper-right corner of the window contains action links that are almost always available:


- The name of the user that is currently logged in appears first. In this case, the user is **Administrator**. Clicking on the username takes you to the **Your account** page.
- The **App** menu provides quick links to installed apps and to app administration. Only apps with GUI components that the current user has permissions to see will be listed in this menu.
- The **Manager** link is always available at the top of the window. The availability of options on the **Manager** page is controlled by the role of the user.
- The **Jobs** link pops up the **Jobs** window. The **Jobs** window provides a listing of current and past search jobs that have been run on this Splunk instance. It is useful for retrieving past results as well as determining what searches are using resources. We will discuss this interface in detail in *Chapter 2, Understanding Search*.
- **Logout** ends the session and forces the user to log in again.

The following screenshot shows what the **Your account** page looks like:



This form presents the global preferences that a user is allowed to change. Other settings that affect users are configured through permissions on objects and settings on roles.

- **Full name** and **Email address** are stored for the administrator's convenience.
- **Time zone** can be changed for each user. This is a new feature in Splunk 4.3.

 Setting the time zone only affects the time zone used to display the data. It is very important that the date is parsed properly when events are indexed. We will discuss this in detail in *Chapter 2, Understanding Search*.

- **Default app** controls where you first land after login. Most users will want to change this to **search**.
- **Restart backgrounded jobs** controls whether unfinished queries should run again if Splunk is restarted.
- **Set password** allows you to change your password. This is only relevant if Splunk is configured to use internal authentication. For instance, if the system is configured to use Windows Active Directory via LDAP (a very common configuration), users must change their password in Windows.

Search app

The search app is where most actions in Splunk start.

Data generator

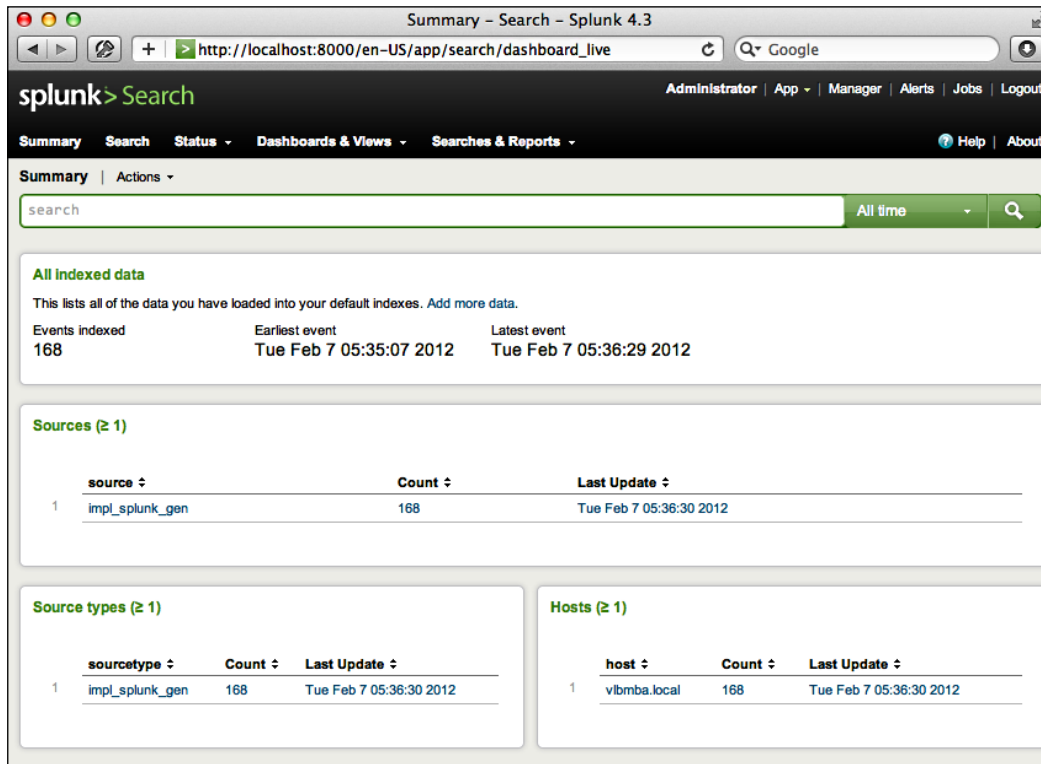
If you want to follow the examples that appear in the next few chapters, install the *ImplementingSplunkDataGenerator* demo app by following these steps:

1. Download *ImplementingSplunkDataGenerator.tar.gz* from the code bundle available on the site <http://www.packtpub.com/support>.
2. Choose **Manage apps...** from the **Apps** menu.
3. Click on the button labeled **Install app from file**.
4. Click on **Choose File**, select the file, and then click on **Upload**.

This data generator app will produce about 16 megabytes of output per day. The app can be disabled so that it stops producing data by using **Manage apps...**, under the **App** menu.

The Summary view

The user is initially presented with the **Summary** view, which contains information about what data that user searches by default. This is an important distinction—in a mature Splunk installation, not all users will always search all data by default.



Let's start below the app name and discuss all the new widgets. The first widget is the navigation bar.



On most pages we encounter from now on, you will see this navigation bar. Items with downward triangles are menus. Items without a downward triangle are links. We will cover customizing the navigation bar in *Chapter 7, Working with Apps*.

Next we find the search bar. This is where the magic starts. We'll go into great detail shortly.



The **All indexed data** panel shows statistics for all indexed data. Remember that this only reflects indexes that this particular user searches by default. There are other events that are indexed by Splunk, including events Splunk indexes about itself. We will discuss indexes in *Chapter 9, Building Advanced Dashboards*.

All indexed data		
This lists all of the data you have loaded into your default indexes. Add more data.		
Events indexed	Earliest event	Latest event
5,210	Sun Feb 5 06:30:25 2012	Tue Feb 7 06:05:08 2012

The next three panels give a breakdown of your data using three important pieces of metadata – **source**, **sourcetype**, and **host**.

Sources (≥ 2)		
source ↓	Count ↓	Last Update ↓
1 impl_splunk_gen	3,403	Tue Feb 7 06:05:09 2012
2 /var/log/system.log	1,807	Tue Feb 7 06:05:03 2012

Source types (≥ 2)		
sourcetype ↓	Count ↓	Last Update ↓
1 impl_splunk_gen	3,403	Tue Feb 7 06:05:09 2012
2 mac_system	1,807	Tue Feb 7 06:05:03 2012

Hosts (≥ 1)		
host ↓	Count ↓	Last Update ↓
1 vlmba.local	5,210	Tue Feb 7 06:05:09 2012

A **source** in Splunk is a unique path or name. In a large installation, there may be thousands of machines submitting data, but all data at the same path across these machines counts as one source. When the data source is not a file, the value of the source can be arbitrary, for instance the name of a script or network port.

A **source type** is an arbitrary categorization of events. There may be many sources across many hosts in the same source type. For instance, given the sources `/var/log/access.2012-03-01.log` and `/var/log/access.2012-03-02.log` on the hosts `fred` and `wilma`, you could reference all of these logs with source type `access` or any other name you like.

A **host** is a captured hostname for an event. In majority of the cases, the **host** field is set to the name of the machine where the data originated. There are cases where this is not known, so the host can also be configured arbitrarily.

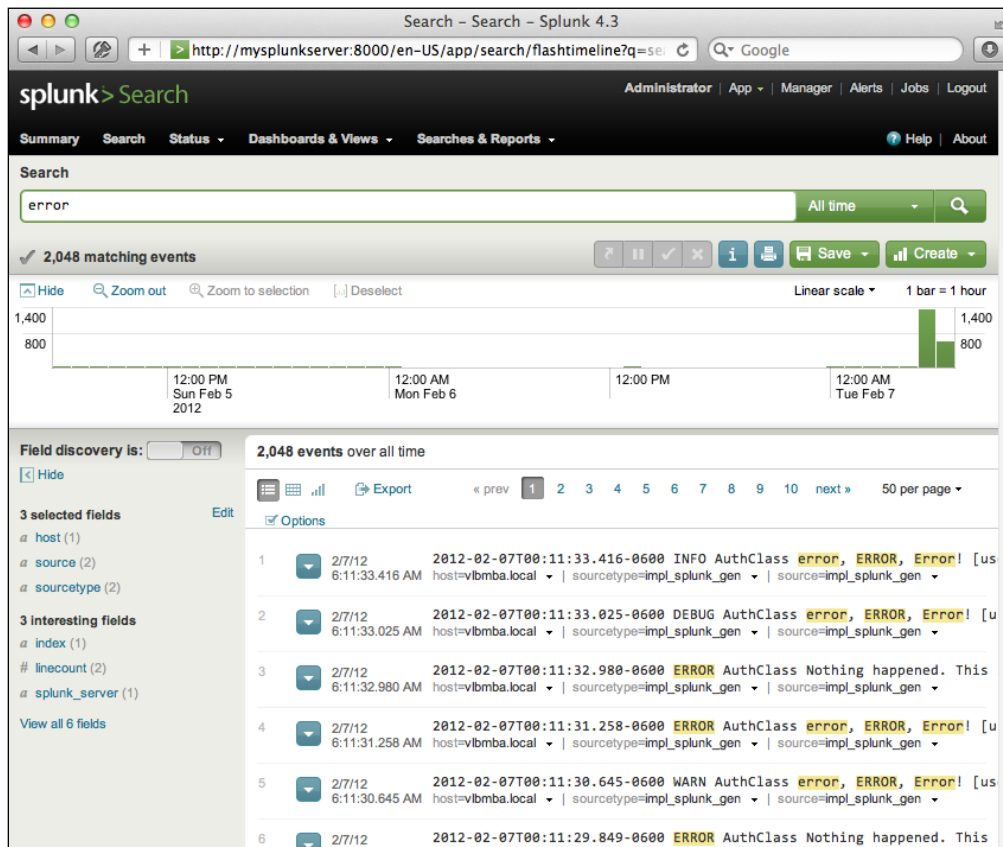
Search

We've finally made it to search. This is where the real power of Splunk lies.

For our first search, we will search for the word `error`. Click in the search bar, type the word `error`, and then either press *Enter* or click on the magnifying glass on the right of the bar.



Upon initiating the search, we are taken to the search results page.



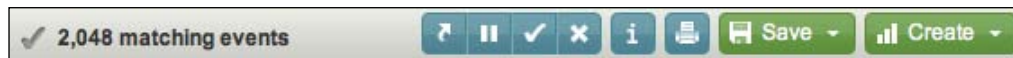


Note that the URL in the browser has changed to **flashtimeline**. You may see references to flashtimeline from time to time. It is simply another name for the search interface.

See the *Using the time picker* section for details on changing the time frame of your search.

Actions

Let's inspect the elements on this page. Below the search bar itself, we have the event count, actions icons, and menus.



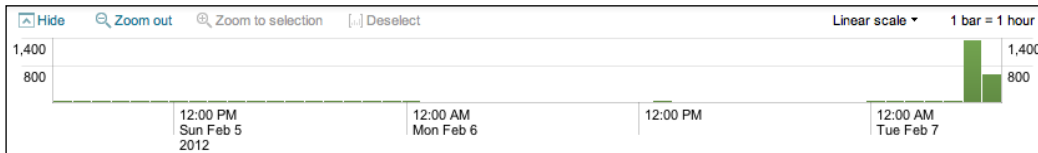
Starting at the left, we have:

- The number of events matched by the base search. Technically, this may not be the number of results pulled from disk, depending on your search. Also, if your query uses commands, this number may not match what is shown in the event listing.
- **Send to background** (↻), which sends the currently running search to the background, where it will continue to run. Jobs sent to the background and past jobs can be restored from the **Jobs** window.
- **Pause** (⏸), which causes the current search to stop locating events but keeps the job open. This is useful if you want to inspect the current results to determine whether you want to continue a long running search.
- **Finalize** (✅), which stops the execution of the current search but keeps the results generated so far. This is useful when you have found enough and want to inspect or share the results found so far.
- **Cancel** (✖), which stops the execution of the current search and immediately deletes the results.
- **Job Inspector** (i), which opens the **Search job inspector** window, which provides very detailed information about the query that was run.
- **Print** (🖨), which formats the page for printing and instructs the browser to print.

- **Save**, which provides different options for saving the search or the results. We will discuss this later in this chapter.
- **Create**, which provides wizard-like interfaces for building different objects from this search. We will discuss these options in *Chapter 4, Simple XML Dashboards*.

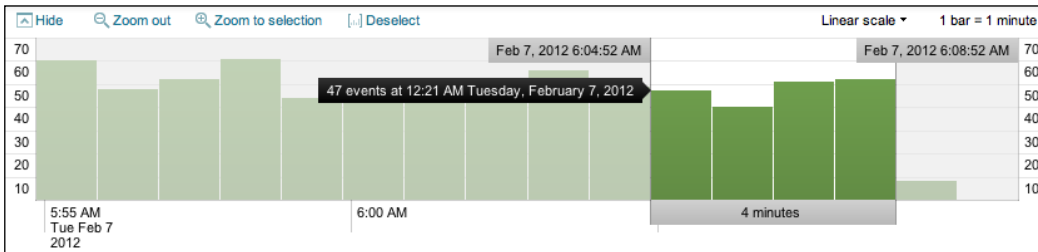
Timeline

Below the actions icons, we have the timeline.



Along with providing a quick overview of the event distribution over a period of time, the timeline is also a very useful tool for selecting sections of time. Placing the pointer over the timeline displays a pop up for the number of events in that slice of time. Clicking on the timeline selects the events for a particular slice of time.

Clicking and dragging selects a range of time.



Once you have selected a period of time, clicking on **Zoom to selection** changes the time frame and re-runs the search for that specific slice of time. Repeating this process is an effective way to drill down to specific events.

Deselect shows all events for the time range selected in the time picker.

Zoom out changes the timeframe to a larger timeframe around the events in the current timeframe.

The field picker

To the left of the search results, we find the field picker. This is a great tool for discovering patterns and filtering search results.

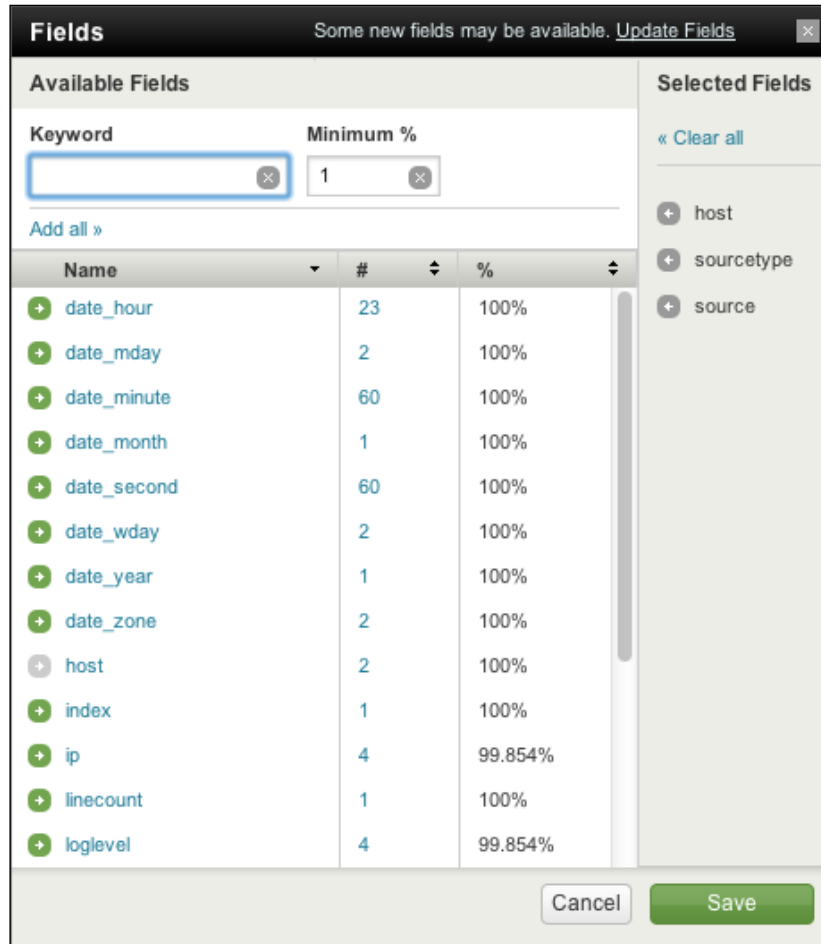


Fields

The fields list contains two lists:

- Selected fields, which have their values displayed under the search event in the search results
- Interesting fields, which are other fields that Splunk has picked out for you

The **Edit** link next to selected fields and the **View all 30 fields** link at the bottom of the field picker both take you to the **Fields** window.





Search results


We are almost through all of the widgets on the page. We still have a number of items to cover in the search results section though, just to be thorough.

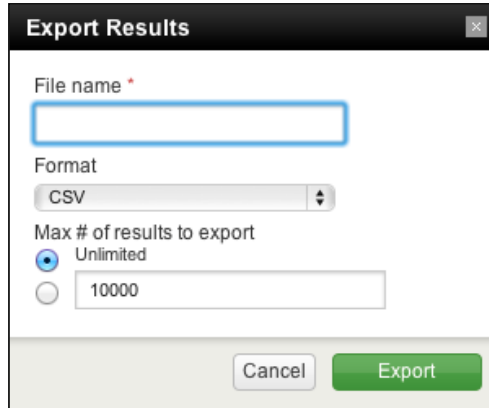
16,185 events in the last 24 hours (from 5:00:00 PM February 3 to 5:30:16 PM February 4, 2012)			
1	2/4/12 5:29:41.550 PM	2012-02-04T17:29:41.550-0600	ERROR AuthClass error, ERROR, Error! [user=jack, ip=4.3.2.1] host=vlbmba.local sourcetype=impl_splunk_gen source=impl_splunk_gen
2	2/4/12 5:29:40.561 PM	2012-02-04T17:29:40.561-0600	ERROR AuthClass error, ERROR, Error! [user=jack, ip=4.3.2.1] host=vlbmba.local sourcetype=impl_splunk_gen source=impl_splunk_gen
3	2/4/12 5:29:40.137 PM	2012-02-04T17:29:40.137-0600	ERROR LogoutClass error, ERROR, Error! [user=bob, ip=1.2.3.3] host=vlbmba.local sourcetype=impl_splunk_gen source=impl_splunk_gen
4	2/4/12 5:29:39.992 PM	2012-02-04T17:29:39.992-0600	ERROR FooClass Hello world. [user=mary, ip=4.3.2.1] host=vlbmba.local sourcetype=impl_splunk_gen source=impl_splunk_gen
5	2/4/12	2012-02-04T17:29:38.215-0600	ERROR FooClass error, ERROR, Error!

Starting at the top of this section, we have the number of events displayed. When viewing all results in their raw form, this number will match the number above the timeline. This value can be changed either by making a selection on the timeline or by using other search commands.

Next, we have actions that affect these particular results. Starting at the left we have:

- **Events List** () , which will show the raw events. This is the default view when running a simple search, as we have done so far.
- **Table** () , which shows a table view of the results. This is the default view when any reporting commands are used. When looking at raw events, this view will show a table with the time of the event, any selected fields, and finally the raw event.

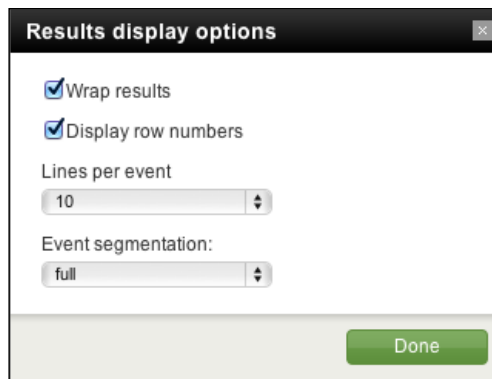
- **Results Chart** (), which shows a chart, if the data allows. For simple searches, charts don't make sense, but they are very useful for reporting.
- **Export**, which allows you to export these particular results to CSV, Raw events, XML, or JSON. New to Splunk 4.3 is the ability to export an unlimited number of results from the web interface.



- **Options** presents display options for the event viewer. See the following section for a discussion about these options.
- To the right, you can choose a page of results and change the number of events per page.

Options

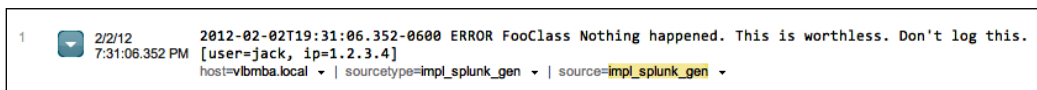
The items presented in the options pop up deserve a short discussion.



- **Wrap results** controls whether events are wrapped at the right edge of the browser window.
- **Display row numbers** toggles the display of the row number to the left of each event.
- **Lines per event** changes the maximum number of lines of an event displayed in the browser per event. There are a few things to note here:
 - All lines of the event are indexed and searchable
 - If the value for this setting is too large, and if a search returns many large messages, your browser may have trouble rendering what it is told to display
 - Events with many lines will have a link at the bottom to see more lines in the event
- The most interesting option here is **Event segmentation**. This setting changes what text is highlighted as you mouse over events. We will discuss this further in *Chapter 2, Understanding Search*.

Events viewer

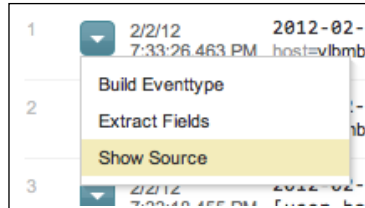
Finally, we make it to the actual events. Let's examine a single event.



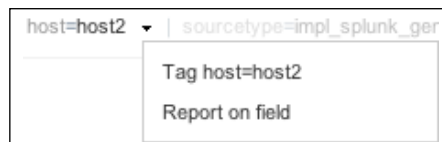
Starting at the left, we have:

- The event number: Raw search results are always returned in the order "most recent first".
- The event options menu (☑): This menu contains workflow actions, a few of which are always available.
 - **Build Eventtype**: Event types are a way to name events that match a certain query. We will dive into event types in *Chapter 6, Extending Search*.
 - **Extract Fields**: This launches an interface for creating custom field extractions. We will cover field extraction in *Chapter 3, Tables, Charts, and Fields*.
 - **Show Source**: This pops up a window with a simulated view of the original source.

- Next appear any workflow actions that have been configured. Workflow actions let you create new searches or links to other sites using data from an event. We will discuss workflow actions in *Chapter 6, Extending Search*.

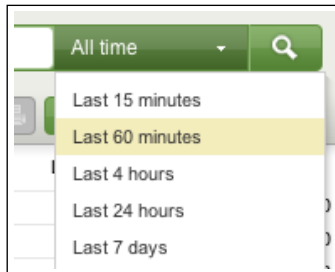


- Next comes the parsed date from this event, displayed in the time zone selected by the user. This is an important and often confusing distinction. In most installations, everything is in one time zone – the servers, the user, and the events. When one of these three things is not in the same time zone as the others, things can get confusing. We will discuss time in great detail in *Chapter 2, Understanding Search*.
- Next, we see the raw event itself. This is what Splunk saw as an event. With no help, Splunk can do a good job finding the date and breaking lines appropriately, but as we will see later, with a little help, event parsing can be more reliable and more efficient.
- Below the event are the fields that were selected in the field picker. Clicking on the value adds the field value to the search. Each field value also has a menu:
 - Tag `fieldname=value` allows you to create a tag that can be used for classification of events. We will discuss tags in *Chapter 6, Extending Search*.
 - **Report on field** launches a wizard showing the values of this field over time.
 - Workflow actions can also appear in these field menus, allowing you to create actions that link to new searches or external sites by using a particular field value.



Using the time picker

Now that we've looked through all of the widgets, let's use them to modify our search. First we will change our time. The default setting of **All time** is fine when there are few events, but when Splunk has been gathering events for weeks or months, this is less than optimal. Let's change our search time to one hour.



The search will run again, and now we only see results for the last hour. Let's try a custom time. **Date** is the first option.

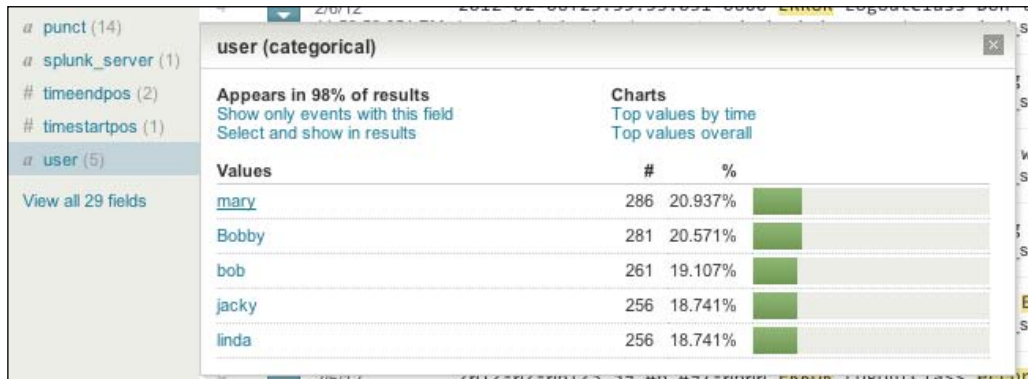
If you know specifically when an event happened, you can drill down to whatever time range you want here. We will examine the other options in *Chapter 2, Understanding Search*.



The time zone used in **Custom Time Range** is the time zone selected in the user's preferences, which is by default the time zone of the Splunk server.

Using the field picker

The field picker is very useful for investigating and navigating data. Clicking on any field in the field picker pops open a panel with a wealth of information about that field in the results of your search.



Looking through the information, we observe:

- *Appears in X% of results* tells you how many events contain a value for this field.
- **Show only events with this field** will modify the query to only show events that have this field defined.
- **Select and show in results** is a shortcut for adding a field to your selected fields.
- **Top values by time** and **Top values overall** present graphs about the data in this search. This is a great way to dive into reporting and graphing. We will use this as a launching point later.
- The chart below the links is actually a quick representation of the top values overall. Clicking on a value adds that value to the query. Let's click on **mary**.



This will rerun the search, now looking for errors that affect only the user mary. Going back to the field picker and clicking on other fields will filter the results even more. You can also click on words in the results, or values of fields displayed underneath events.

Using Manager

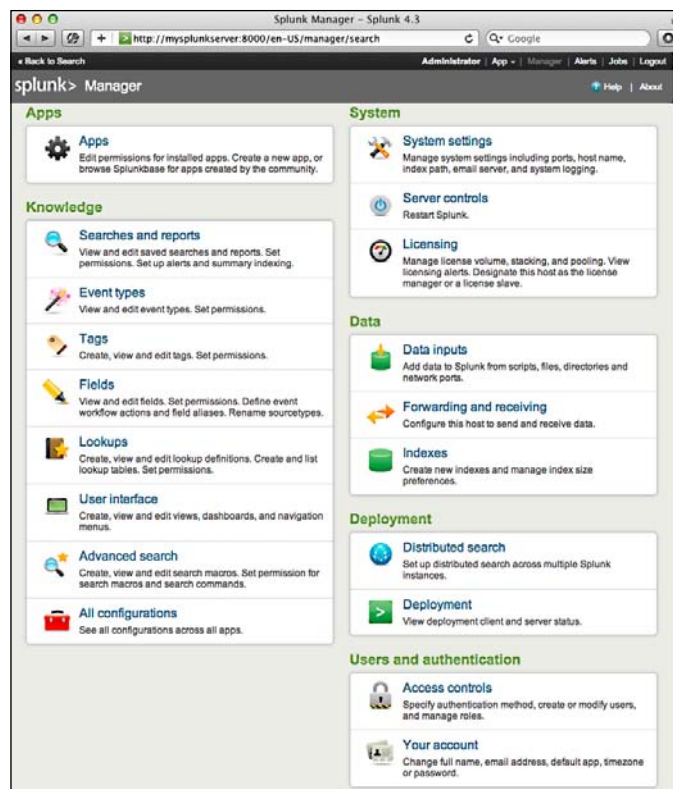
The **Manager** section, in a nutshell, is an interface for managing configuration files. The number of files and options in these configuration files is truly daunting, so the web interface concentrates on the most commonly used options across the different configuration types.



Splunk is controlled exclusively by plain text configuration files. Feel free to take a look at the configuration files that are being modified as you make changes in the admin interface. You will find them in `$(SPLUNK_HOME)/etc/system/local/` and `$(SPLUNK_HOME)/etc/apps/`.

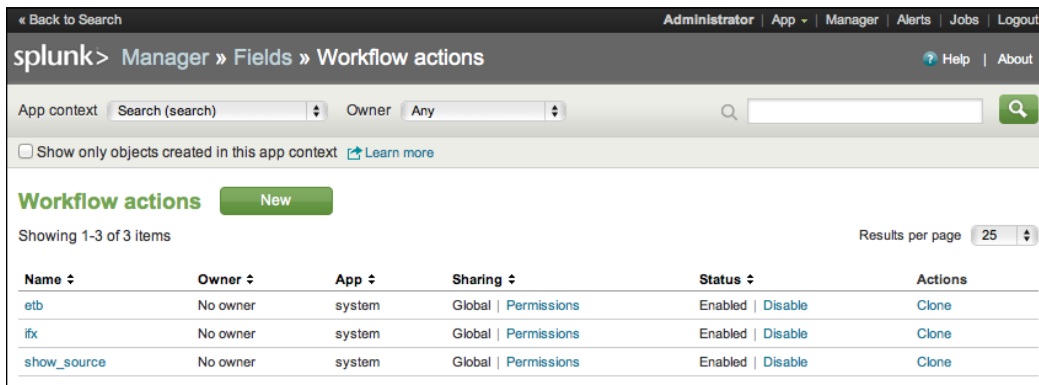
You may notice configuration files with the same name in different locations. We will cover, in detail, the different configuration files, their purposes, and how these configurations merge together, in *Chapter 10, Configuring Splunk*. Don't start modifying the configurations directly until you understand what they do and how they merge.

Clicking on **Manager**, on the top bar, takes you to the **Manager** page.



The options are organized into logical groupings, as follows:

- **Apps:** This interface allows you to easily add new apps and manage apps that are currently installed. If you installed the *ImplementingSplunkDataGenerator* app, you have already seen this interface.
- **Knowledge:** Each of the links under **Knowledge** allows you to control one of the many object types that are used at search time. The following screenshot shows an example of one object type, workflow actions.



Let's cover the administration of each object type that we will cover in later chapters:

- **System:** The options under this section control system-wide settings.
 - **System settings** covers network settings, the default location to store indexes, outbound e-mail server settings, and how much data Splunk logs about itself
 - **Server controls** contains a single page that lets you restart Splunk from the web interface
 - **Licensing** lets you add license files or configure Splunk as a slave to a Splunk license server
- **Data:** This section is where you manage data flow.
 - **Data Inputs:** Splunk can receive data by reading files (either in batch mode or in real time), listening to network ports, or running scripts
- **Forwarding and receiving:** Splunk instances don't typically stand alone. Most installations consist of at least one Splunk indexer and many Splunk forwarders. Using this interface, you can configure each side of this relationship and more complicated setups (we will discuss this in a more detail in *Chapter 11, Advanced Deployments*):

- **Indexes:** An Index is essentially a datastore. Under the covers, it is simply a set of directories, created and managed by Splunk. For small installations, a single index is usually acceptable. For larger installations, using multiple indexes allows flexibility in security, retention, and performance tuning, and better use of hardware. We will discuss this further in *Chapter 10, Configuring Splunk*.
- **Deployment:** The two options here relate to distributed deployments. (we will cover these options in detail in *Chapter 11, Advanced Deployments*):
 - **Distributed Search:** Any Splunk instance running searches can utilize itself and other Splunk instances to retrieve results. This interface allows you to configure access to other Splunk instances.
 - **Deployment:** Splunk includes a deployment server component to aid in distributing configurations to the many instances that can be involved in a distributed installation. There is no need to use the deployment server, particularly if you already have something to manage configurations.
- **Users and authentication:** This section provides authentication controls and an account link.
 - **Access controls:** This section is for controlling how Splunk authenticates users and what users are allowed to see and do. We will discuss this further in *Chapter 10, Configuring Splunk*.
 - **Your account:** We saw this earlier when we clicked on the name of the user currently logged in on the top bar.

Summary

As you have seen in this chapter, the Splunk GUI provides a rich interface for working with search results. We have really only scratched the surface and will cover more elements as we use them in later chapters.

In the next chapter, we will dive into the nuts and bolts of how search works, so that you can make efficient searches to populate the cool reports we will make in *Chapter 3, Tables, Charts, and Fields*, and beyond.

2

Understanding Search

To successfully use Splunk, it is vital that you write effective searches. Using the index efficiently will make your initial discoveries faster, and the reports you create will run faster for you and others. In this chapter, we will cover:

- How to write effective searches
- How to search using fields
- Understanding time
- Saving and sharing searches

Using search terms effectively

The key to creating an effective search is to take advantage of the index. Splunk's index is effectively a huge word index, sliced by time. The single most important factor for the performance of your searches is how many events are pulled from disk. The following few key points should be committed to memory:

- **Search terms are case insensitive:** Searches for `error`, `Error`, `ERROR`, and `ErRoR` are all the same thing.
- **Search terms are additive:** Given the search item `mary error`, only events that contain *both* words will be found. There are Boolean and grouping operators to change this behavior; we will discuss these later.
- **Only the time frame specified is queried:** This may seem obvious, but it's a big difference from a database, which would always have a single index across all events in a table. Since each index is sliced into new buckets over time, only the buckets that contain events for the time frame in question need to be queried.
- **Search terms are words, not parts of words:** A search for `foo` will not match `foobar`.

With just these concepts, you can write fairly effective searches. Let's dig a little deeper, though:

- **A word is anything surrounded by whitespace or punctuation:** For instance, given the log line `2012-02-07T01:03:31.104-0600 INFO AuthClass Hello world. [user=Bobby, ip=1.2.3.3]`, the "words" indexed are `2012, 02, 07T01, 03, 31, 104, 0600, INFO, AuthClass, Hello, world, user, Bobby, ip, 1, 2, 3, and 3`. This may seem strange, and possibly a bit wasteful, but this is what Splunk's index is really *really* good at – dealing with huge numbers of words across huge numbers of events.
- **Splunk is not grep with an interface:** One of the most common questions is whether Splunk uses regular expressions for search. Technically, the answer is no, but most of what you would do with regular expressions is available in other ways. Using the index as it is designed is the best way to build fast searches. Regular expressions can then be used to further filter results or extract fields.
- **Numbers are not numbers until after they have been parsed at search time:** This means that searching for `foo>5` will not use the index as the value of `foo` is not known until it has been parsed out of the event at search time. There are different ways to deal with this behavior, depending on the question you're trying to answer.
- **Field names are case sensitive:** When searching for `host=myhost`, `host` must be lowercase. Likewise, any extracted or configured fields have case sensitive field names, but the values are case insensitive.
 - `Host=myhost` will not work
 - `host=myhost` will work
 - `host=MyHost` will work
- **Fields do not have to be defined before indexing data:** An **indexed field** is a field that is added to the metadata of an event at index time. There are legitimate reasons to define indexed fields, but in the vast majority of cases it is unnecessary and is actually wasteful. We will discuss this in *Chapter 3, Tables, Charts, and Fields*.

Boolean and grouping operators

There are a few operators that you can use to refine your searches (note that these operators *must* be in uppercase to not be considered search terms):

- **AND** is implied between terms. `error mary` is the same as `error AND mary`.

- **OR** allows you to specify multiple values. `error OR mary` means "find any event that contains either word".
- **NOT** applies to the next term or group. `error NOT mary` would find events that contain `error` but do not contain `mary`.
- `""` identifies a phrase. `"Out of this world"` will find this exact sequence of words. `Out of this world` would find any event that contains *all* of these words, but not necessarily in that order.
- `()` is used for grouping terms. Parentheses can help avoid confusion in logic. For instance, these two statements are equivalent:
 - `bob error OR warn NOT debug`
 - `(bob AND (error OR warn)) AND NOT debug`
- `=` is reserved for specifying fields. Searching for an equal sign can be accomplished by wrapping it in quotes.
- `[]` is used to perform a subsearch. We will discuss this in *Chapter 5, Advanced Search Examples*.

You can use these operators in fairly complicated ways, if you want to be very specific, or even to find multiple sets of events in a single query. The following are a few examples:

- `error mary NOT jacky`
- `error NOT (mary warn) NOT (jacky error)`
- `index=myapplicationindex (sourcetype=sourcetype1 AND ((bob NOT error) OR (mary AND warn))) OR (sourcetype=sourcetype2 (jacky info))`

This can also be written with some whitespace for clarity:

```
index=myapplicationindex
(
  sourcetype=security
  AND
  (
    (bob NOT error)
    OR
    (mary AND warn)
  )
)
OR
(
  sourcetype=application
  (jacky info)
)
```



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

Clicking to modify your search

Though you can probably figure it out by just clicking around, it is worth discussing the behavior of the GUI when moving your mouse around and clicking.

- Clicking on any word or field value will add that term to the search.
- Clicking on a word or field value that is already in the query will remove it from the query.
- Clicking on any word or field value while holding down *Alt* (option on the Mac) will append that search term to the query, preceded by `NOT`. This is a very handy way to remove irrelevant results from query results.

Event segmentation

In *Chapter 1, The Splunk Interface*, we touched upon this setting in the **Options** dialog. The different options change what is highlighted as you mouse over the text in the search results, and therefore what is added to your query when clicked on. Let's see what happens to the phrase `ip=10.20.30.40` with each setting:

- **inner** highlights individual words between punctuation. Highlighted items would be `ip`, `10`, `20`, `30`, and `40`.
- **outer** highlights everything between whitespace. The entire phrase `ip=10.20.30.40` would be highlighted.
- **full** will highlight everything from the beginning of the block of text as you move your mouse. Rolling from left to right would highlight `ip`, then `ip=10`, then `ip=10.20`, then `ip=10.20.30`, and finally `ip=10.20.30.40`. This is the default setting and works well for most data.
- **raw** disables highlighting completely, allowing the user to simply select the text at will. Some users will prefer this setting as it takes away any unexpected behavior. It is also slightly faster as the browser is doing less work.

Field widgets

Clicking on values in the field picker or in the field value widgets underneath an event will append the field value to a query. For instance, if `ip=10.20.30.40` appears under your event, clicking on the value will append `ip=10.20.30.40` to your query.



If a field value looks like `key=value` in the text of an event, you will want to use one of the field widgets instead of clicking on the raw text of the event. Depending on your event segmentation setting, clicking on the word will either add `value` or `"key=value"`. The former will not take advantage of the field definition; instead, it will simply search for the word. The latter will work for events that contain the exact quoted text but not for other events that actually contain the same field value extracted in a different way.

Time

Clicking on the time next to an event will change the search to only find events that happened in that second.



To zoom in to a short time frame, one convenient approach is to click on the time of an event to search only that second, then click on **Zoom out** above the timeline until the appropriate time frame is reached.

Using fields to search

When we explored the GUI in *Chapter 1, The Splunk Interface*, you probably noticed fields everywhere. Fields appear in the field picker on the left and under every event. Where fields actually come from is transparent to the user, who simply searches for `key=value`. We will discuss adding new fields in *Chapter 3, Tables, Charts, and Fields*, and in *Chapter 10, Configuring Splunk*.


Using the field picker

The field picker gives us easy access to the fields currently defined for the results of our query. Clicking on any field presents us with details about that field in our current search results.

Values	#	%
red	12,741	4.474%
green	12,621	4.432%

As we go through the following items in this widget, we see a wealth of information right away:

- *Appears in X% of results* is a good indication of whether we are getting the results we think we're getting. If every event in your results should contain this field, and this is not 100 percent, either your search can be made more specific or a field definition needs to be modified.
- **Show only events with this field** adds `fieldname="*"` to your existing search to make sure you only get events that have this field.

 If the events you are searching for always contain the name of the field, in this case `network`, your query will be more efficient if you also add the field name to the query. In this case, the query would look like this: `sourcetype="impl_splunk_gen" network="*" network`.

- **Select and show in results** adds the field to the **selected fields** list at the top of the field picker and displays the field value under each event.
- **Charts** contains the following links, which we will use as starting points for examples in *Chapter 3, Tables, Charts, and Fields*:
 1. **Top values by time** shows a graph of the most common values occurring in the time frame searched.
 2. **Top values overall** shows a table of the most common values for this field for the time frame searched.
- **Values** shows a very useful snapshot of the top ten most common values.

Using wildcards efficiently

Though the index is based on words, it is possible to use wildcards when needed, although some care must be taken.

Only trailing wildcards are efficient

Stated simply, `bob*` will find events containing Bobby efficiently, but `*by` or `*ob*` will not. The latter cases will scan all events in the time frame specified.

Wildcards are tested last

Wildcards are tested *after* all other terms. Given the search: `authclass *ob* hello world`, all other terms besides `*ob*` will be searched *first*. The more you can limit the results using full words and fields, the better your search will perform.

Supplementing wildcards in fields

Given the following events, a search for `world` would return both events:

```
2012-02-07T01:04:31.102-0600 INFO AuthClass Hello world. [user=Bobby, ip=1.2.3.3]
2012-02-07T01:23:34.204-0600 INFO BarClass Goodbye. [user=Bobby, ip=1.2.3.3, message="Out of this world"]
```

What if you only wanted the second event, but all you know is that the event contains `world` somewhere in the field `message`? The query `message="*world"` would work but is very inefficient because Splunk must scan every event looking for `*world*` and then determine whether `world` is in the field `message`.

You can take advantage of the behavior mentioned before—wildcards are tested last. Rewriting the query as `world message="*world"` gives Splunk a chance to find all records with `world`, then inspect those events for the more specific wildcard condition.

All about time

Time is an important and confusing topic in Splunk. If you want to skip this section, absorb one concept—time *must* be parsed properly on the way into the index as it cannot be changed later without indexing the raw data again.

How Splunk parses time

Given the date `11-03-04`, how would you interpret this date? Your answer probably depends on where you live. In the United States, you would probably read this as November 3, 2004. In Europe, you would probably read this as March 11, 2004. It would also be reasonable to read this as March 4, 2011.

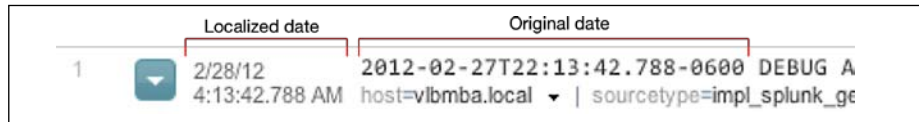
Luckily, most dates are not this ambiguous, and Splunk makes a good effort. It is absolutely worth the trouble to give Splunk a little help by configuring the time format. We'll discuss the relevant configurations in *Chapter 10, Configuring Splunk*.

How Splunk stores time

Once the date is parsed, the date stored in Splunk is always stored as GMT epoch. **Epoch time** is the number of seconds since January 1, 1970, the birthday of Unix. By storing all events using a single time zone, there is never a problem lining up events that happen in different time zones. This, of course, only works properly if the time zone of the event can be determined when it is indexed. This numeric value is stored in the field `_time`.

How Splunk displays time

The text of the original event, and the date it contains, is never modified. It is always displayed as it was received. The date displayed to the left of the event is determined by the time zone of the Splunk instance or the user's preference as specified in **Your account**.



How time zones are determined and why it matters

Since all events are stored according to their GMT time, the time zone of an event only matters at parse time, but it is vital to get it right. Once the event is written into the index, it cannot be changed without re-indexing the raw data.

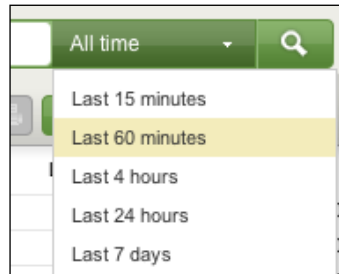
The time zone can come from a number of places, in this order of precedence:

- The time zone specified in the log. For instance, the date `2012-02-07T01:03:23.575-0600`, `-0600` indicates that the zone is 6 hours behind GMT. Likewise, `Tue 02 Feb, 01:03:23 CST 2012` represents the same date.
- The configuration associated with a source, host, or source type, in that order. This is specified in `props.conf`. This can actually be used to override the time zone listed in the log itself, if needed. We will discuss this in *Chapter 10, Configuring Splunk*.
- The time zone of the Splunk instance forwarding the events. The time zone is relayed along with the events, just in case it is not specified elsewhere. This is usually an acceptable default. The exception is when different logs are written with different time zones on the same host, without the time zone in the logs. In that case, it needs to be specified in `props.conf`.
- The time zone of the Splunk instance parsing the events. This is sometimes acceptable and can be used in interesting ways in distributed environments.

The important takeaway, again, is that the time zone needs to be known at the time of parsing and indexing the event.

Different ways to search against time

Now that we have our time indexed properly, how do we search against time? The time picker provides a neat set of defaults for relative time.



These options search back from the present to a relative point in time, but sometimes, you need to search over a specific period of time.

The last option, **Custom time...**, provides an interface that helps specify specific times.

- **Date** is the first option.

If you know specifically when an event happened, you can drill down to whatever time range you want here. The time zone here is what you have chosen in **Your account**, or the system default if you didn't change it. This may not be the time zone of the events you are looking for.

- **Relative** lets you choose a time in the past.

The screenshot shows a dialog box titled "Custom Time Range". Under "Range Type", the "Relative" radio button is selected. The "Earliest time" is set to "2" hours ago, with a "Snap to" dropdown set to "Minute". The "Search language equivalent" is "-2h@m" and the "Effective range" is "Feb 8, 2012 11:00:00.000 PM - (now)". "Cancel" and "Apply" buttons are at the bottom right.

The end of the search will always be the current time. The **Snap to** option lets you choose a unit to round down to. For instance, if the current time is 4:32 and you choose 2 for the **Hour(s) ago** option, and **Hour** for the **Snap to** option, the earliest time for the search will be 2:00. **Effective range** will tell you what time range is being searched.

Note the text under **Search language equivalent**. This is the way you express relative times in Splunk. We will see this often as we move forward.

- Like **Relative** time, **Real-time** lets you choose a time in the past and shows you the search language equivalent. A real-time search is different in that it continues to run, continuously updating your query results, but only keeps the events with a parsed date that is newer than the time frame specified.

The screenshot shows a dialog box titled "Custom Time Range". Under "Range Type", the "Real-time" radio button is selected. The "Time window" is set to "15" minutes, with a "Minute(s)" dropdown. The "Search language equivalent" is "rt-15m". "Cancel" and "Apply" buttons are at the bottom right.

- Lastly, we have **Advanced search language**.

If you noticed, we have selected the **2** for the **Hour(s) ago** option, and **Minute** for the **Snap to** option in the **Relative** tab. The search language equivalent for this selection is **-2h@m**, which means "go back 2 hours (7,200 seconds) from this moment, and then snap to the beginning of the minute that second falls in". So, given the time 15:11:23, the relative time would be 13:11:00. The language is very powerful and can be used whenever a search is specified.

Specifying time in-line in your search

You can also directly use relative and exact times in your searches. For instance, given the search item `bob error`, you can specify directly in the search the time frame you want to use, using the fields **earliest** and **latest**.

- To search for errors affecting `bob` in the last 60 minutes, use `earliest=-60m bob error`
- To search for errors affecting `bob` in the last 3 hours, snap to the beginning of the hour using `earliest=-3h@h bob error`
- To search for errors affecting `bob` yesterday, use `earliest=-1d@d latest=0d@d bob error`
- To search for errors affecting `bob` since Monday at midnight, use `earliest=-0@w1 bob error`



You cannot use different time ranges in the same query; for instance, in a Boolean search, `(earliest=-1d@d latest=-0d@d bob error)` OR `(earliest=-2d@d latest=-1d@d mary error)` will not work. The `append` command provides a way of accomplishing this.

`_indextime` versus `_time`

It is important to note that events are generally not received at the same time as stated in the event. In most installations, the discrepancy is usually of a few seconds, but if logs arrive in batches, the latency can be much larger. The time at which an event is actually written in the Splunk index is kept in the internal field `_indextime`. The time that is parsed out of the event is stored in `_time`.

You will probably never search against `_indextime`, but you should understand that the time you are searching against is the time parsed from the event, not the time at which the event was indexed.

Making searches faster

We have talked about using the index to make searches faster. When starting a new investigation, following a few steps will help you get results faster:

1. Set the time to the minimum time that you believe will be required to locate relevant events. For a chatty log, this may be as little as a minute. If you don't know when the events occurred, you might search a larger time frame and then zoom in by clicking on the timeline while the search is running.
2. Specify the index if you have multiple indexes. It's good to get into the habit of starting your queries with the index name, for example, `index=myapplicationindex error bob`.
3. Specify other fields that are relevant. The most common fields to specify are `sourcetype` and `host`, for example, `index=myapplicationindex sourcetype="impl_splunk_gen" error bob`.



If you find yourself specifying the field `source` on a regular basis, you could probably benefit from defining more source types. Avoid using the `sourcetype` field to capture other information, for instance `datacenter` or `environment`. You would be better off using a lookup against `host` or creating another indexed field for those cases.

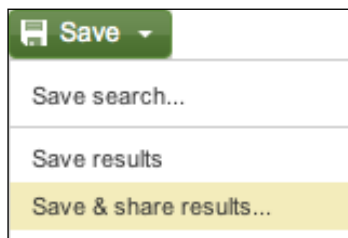
4. Add more words from the relevant messages as and when you find them. This can be done simply by clicking on words or field values in events or field values in the field picker, for example, `index=myapplicationindex sourcetype="impl_splunk_gen" error bob authclass OR fooclass`.
5. Expand your time range once you have found the events that you need, and then refine the search further.
6. Disable **Field discovery** (at the top of the field picker). This can greatly improve speed, particularly if your query retrieves a lot of events. Extracting all of the fields from events simply takes a lot of computing time, and disabling this option prevents Splunk from doing all of that work when not needed.



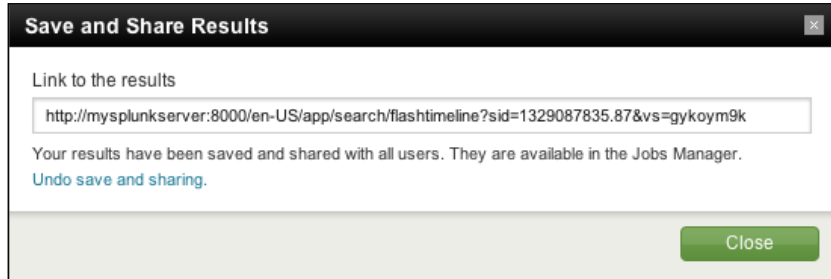
If the query you are running is taking a long time to run, and you will be running this query on a regular basis – perhaps for an alert or dashboard – using a summary index may be appropriate. We will discuss this in *Chapter 9, Summary Indexes and CSV Files*.

Sharing results with others

It is often convenient to share a specific set of results with another user. You could always export the results to a CSV file and share it, but this is cumbersome. Instead, to use a URL for sharing, start by choosing **Save & share results...** from the **Save** menu.

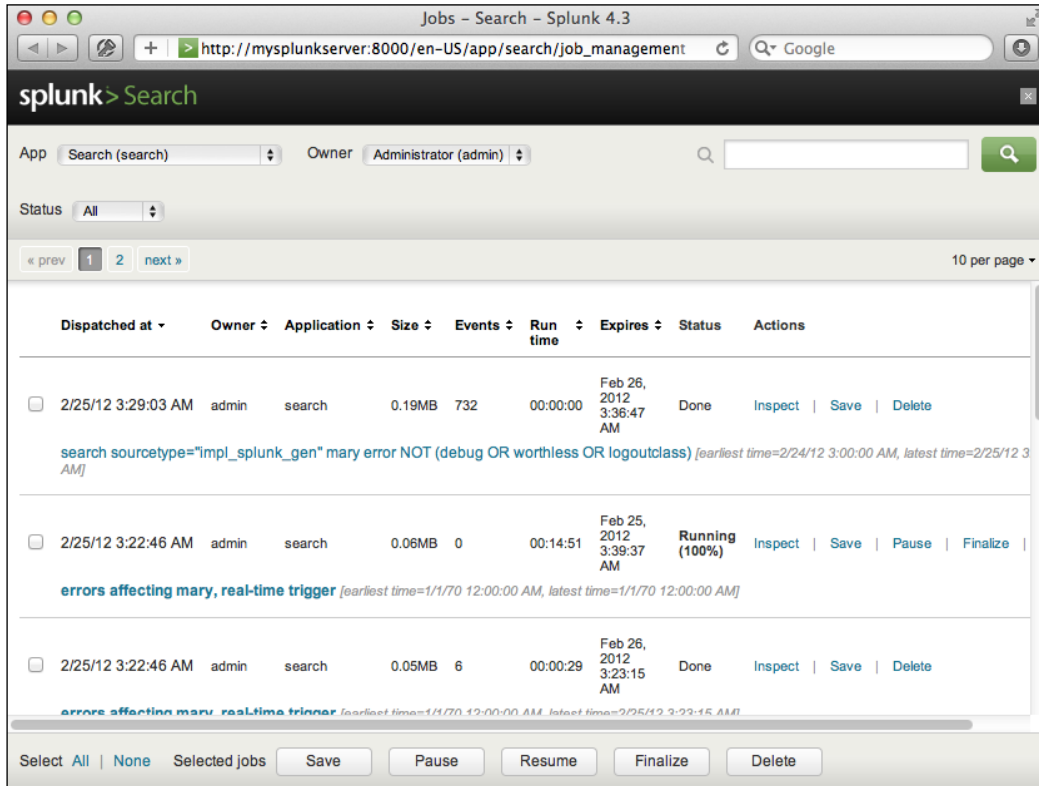


This opens the **Save and Share Results** panel.



The URL under **Link to the results** can be copied and sent to other users. A user visiting this URL will see exactly the same results you did, assuming the job has not expired.

The results are also available in the **Jobs** window. Clicking on the **Jobs** link in the top bar opens the **Jobs** window.



The **App** menu, **Owner** menu, **Status** menu, and search bar let you filter what jobs are displayed.

The table has the following columns:

- **Dispatched at** is the time at which the search started.
- **Owner** is the user that started the job. Sometimes jobs will appear with **system** as the user if the saved search is configured in an application but not owned by a particular user.
- **Application** specifies the application in which the search was started. This is useful for locating your searches as well as unfamiliar searches being fired off by other apps.
- **Size** is the amount of disk space being used to store the results of this query.
- **Events** shows the number of events that were matched by the search. In a complicated search or report, the results returned may be different from this number.
- **Run time** is how long a search took to run or the elapsed time if the search is still running.
- **Expires** is the time at which the results will be removed from disk.
- **Status** lets you see and sort searches based on whether they are still running.



One simple way to find running jobs is to change the **Status** menu to **Running** and click the magnifying glass.

- **Actions** provides the following links to affect a search or its results:
 - **Inspect** shows detailed information about the query. We will cover the search job inspector in *Chapter 5, Advanced Search Examples*.
 - **Save** keeps the search results indefinitely.
 - **Pause** pauses the execution of a job.
 - **Finalize** stops the execution but keeps the results located up to this point.
 - **Delete** removes the results from the disk immediately. It is generally not necessary to delete search results as they will expire on their own.
- The search bar at the top of this window is useful for finding present and past jobs.

Saving searches for reuse

Let's build a query, save it, and make an alert out of it.

First, let's find errors that affect `mary`, one of our most important users. This can simply be the query `mary error`. Looking at some sample log messages that match this query, we see that some of these events probably don't matter (the dates have been removed to shorten the lines).

```
ERROR LogoutClass error, ERROR, Error! [user=mary, ip=3.2.4.5]
WARN AuthClass error, ERROR, Error! [user=mary, ip=1.2.3.3]
ERROR BarClass Hello world. [user=mary, ip=4.3.2.1]
WARN LogoutClass error, ERROR, Error! [user=mary, ip=1.2.3.4]
DEBUG FooClass error, ERROR, Error! [user=mary, ip=3.2.4.5]
ERROR AuthClass Nothing happened. This is worthless. Don't log this.
[user=mary, ip=1.2.3.3]
```

We can probably skip the `DEBUG` messages; the `LogoutClass` messages look harmless; and the last message actually *says* that it's worthless.

`mary error NOT debug NOT worthless NOT logoutclass` limits the results to:

```
WARN AuthClass error, ERROR, Error! [user=mary, ip=1.2.3.3]
ERROR BarClass Hello world. [user=mary, ip=4.3.2.1]
```

For good measure, let's add the `sourcetype` field and some parentheses.

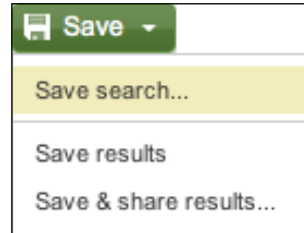
```
sourcetype="impl_splunk_gen" (mary AND error) NOT debug NOT worthless
NOT logoutclass
```

Another way of writing the same thing is as follows:

```
sourcetype="impl_splunk_gen" mary error NOT (debug OR worthless OR
logoutclass)
```

So that we don't have to type our query every time, we can save this search for quick retrieval.

First, choose **Save search...** from the **Save** menu.



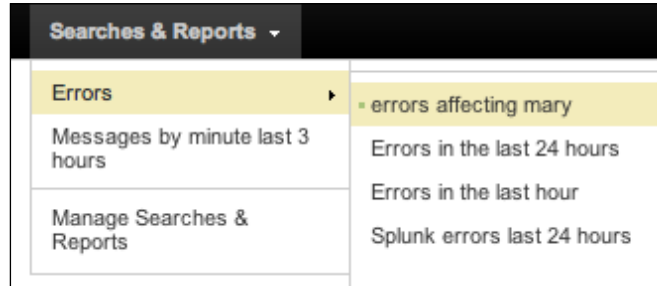
The **Save Search** window appears.

A screenshot of the 'Save Search' dialog box. The dialog has a title bar with 'Save Search' and a close button. It contains the following fields and options:

- * Search name**: A text input field containing 'errors affecting mary'.
- * Search string**: A text input field containing 'error user="mary"'. There is a small icon in the bottom right corner of the field.
- Time range**: Two input fields. The first contains '-24h@h' and the second contains 'now'. Below these fields is explanatory text: '-1d (one day ago), now (triggering time)', 'rt-1d (one day ago in real-time), rt(triggering time)', and 'Time specifiers: y, mon, d, h, m, s' followed by a blue link 'Learn more'.
- Share**: Two radio button options. The first is 'Keep search private' (selected) and the second is 'Share as read-only to all users of current app'.
- Below the radio buttons is the text: 'Additional permission settings available in Manager » Searches and reports'.
- At the bottom left is a 'Cancel' button. At the bottom right is a green 'Finish »' button.

Enter a value for **Search name**, in our case, `errors affecting mary`. The time range is filled in based on what was selected in the time picker. **Share** lets you specify whether other users should be able to see this search in their menus. Standard users will not have the ability to share their searches with others.

The search is then available in the **Searches & Reports** menu under **Errors**.



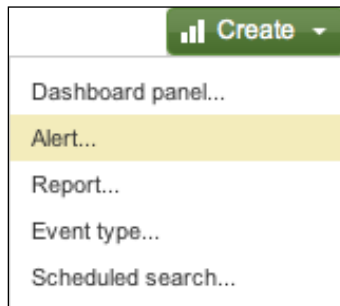
Selecting the search from the menu runs the search using the latest data available.



Note the small square next to **errors affecting mary**. This indicates that this search is not shared and is only viewable by its owner.

Creating alerts from searches

Any saved search can also be run on a schedule. One use for scheduled searches is firing alerts. To get started, choose **Alert...** from the **Create** menu.

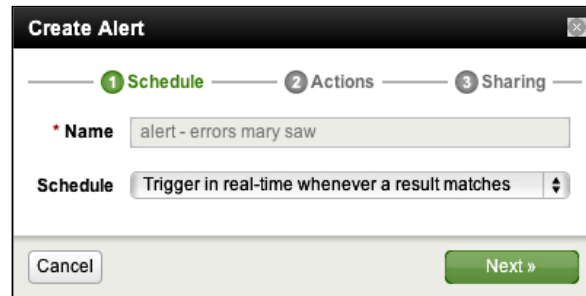


A wizard interface is presented, covering three steps.

Schedule

The **Schedule** step provides the following options:

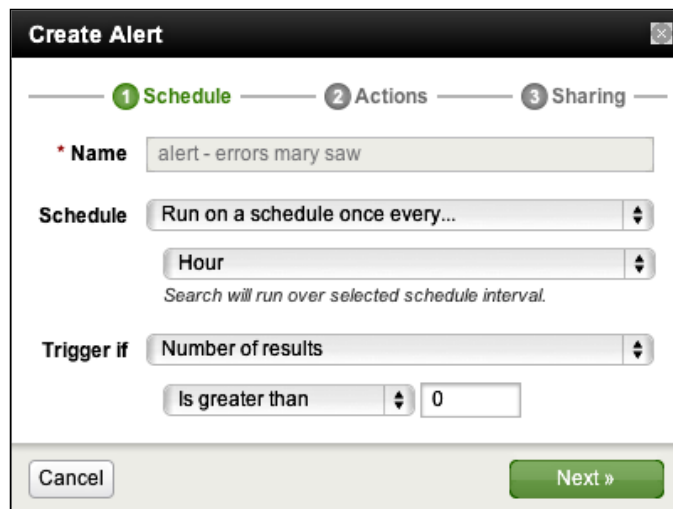
- **Trigger in real-time whenever a result matches:** This option will leave a real-time search running *all the time* and will *immediately* fire an alert whenever an event is seen.



The screenshot shows the 'Create Alert' dialog box with the 'Schedule' step selected. The 'Name' field is 'alert - errors mary saw'. The 'Schedule' dropdown is set to 'Trigger in real-time whenever a result matches'. There are 'Cancel' and 'Next »' buttons at the bottom.

This option will create an alert *every time* an event that matches your search occurs. There is an important throttling option in the next step.

- **Run on a schedule once every...:** New options now appear below the menu.



The screenshot shows the 'Create Alert' dialog box with the 'Schedule' step selected. The 'Name' field is 'alert - errors mary saw'. The 'Schedule' dropdown is set to 'Run on a schedule once every...'. Below it, the interval is set to 'Hour'. The 'Trigger if' dropdown is set to 'Number of results', and the condition is 'Is greater than' with a value of '0'. There are 'Cancel' and 'Next »' buttons at the bottom.

- **Schedule:** You can choose to either run your search on a set schedule or run your alert according to a cron schedule. Keep in mind that the time frame selected in the time picker will be used each time the query runs – you probably don't want to run a query looking at 24 hours of data every minute.
- **Trigger if** lets you choose when to trigger the alert.
- **Number of results** lets you build a rule based on the count. **Is greater than 0** is the most commonly used option.
- **A custom condition is met** lets you use a bit of search language to decide whether to fire the alert. If any events pass the search language test then the rule passes and the alert is fired. For example, `search authclass` would test each event for the word `authclass`, which in our example would pass one event. In most cases, you would use a threshold value. The purpose is to test the search results without affecting the search results that are handed along to the defined action.
- **Monitor in real-time over a rolling window of...:** This is a very useful option for generating alerts as soon as some threshold is passed. For instance, you could watch the access logs for a web server, and if the number of events seen in the last minute falls below 100, send an alert.

Working with our example data, let's set an alert to fire any time more than five errors affecting the user `mary` are matched in the last 5 minutes.

The screenshot shows a 'Create Alert' dialog box with three tabs: '1 Schedule', '2 Actions', and '3 Sharing'. The 'Schedule' tab is active. It contains the following fields:

- Name:** alert - errors mary saw
- Schedule:** Monitor in real-time over a rolling window of... (dropdown menu)
- Time:** 5 (input field) minute (dropdown menu)
- Trigger if:** Number of results (dropdown menu)
- Condition:** Is greater than (dropdown menu)
- Value:** 5 (input field)

At the bottom, there are two buttons: 'Cancel' and 'Next »'.

Actions

Once we've set all of our options, we can click on **Next >>** to proceed to **Actions**.

The **Actions** pane is where you decide what you want to do with the results of your alert. There are several options under **Enable actions**, as follows:

- **Send email:** This is the most common action. Simply fill out the list of e-mail addresses. The resulting e-mail will always contain a link back to this Splunk server, directly to the search results. You can customize the **Subject** string and optionally include the results of the search in the e-mail.

- **Run a script:** This will run a script with the results of the search. Any script must be installed by the administrator at `$(SPLUNK_HOME)/bin/scripts/`. This is covered in *Chapter 12, Extending Splunk*.
- **Show triggered alerts in Alert manager:** The alert manager is a listing of alerts populated by saved searches. The alerts window is a convenient way to group all alerts without filling your mailbox. Use the **Alerts** link at the top of the window.

The next two options determine how many alerts to issue:

- **Execute actions on:** Your options are **All results** and **Each result**. In most cases, you will only want one alert per search (**All results**), but you could treat each event independently and issue an alert per event, in special cases. You should be cautious with **Each result**, making sure to limit the number of results returned, most likely by using reporting commands.
- **Throttling:** This allows you to determine how often the same alert will be fired. You may want to search for a particular event every minute, but you probably don't want an e-mail every minute. With throttling, you can tell Splunk to only send you an e-mail every half hour even if the error continues to happen every minute.

If you choose **Execute actions on each result**, another input box appears to let you throttle against specific fields. For instance, if host A has an error, you may not want to know about any other host A errors for another 30 minutes, but if host B has an error in those 30 minutes, you would like to know immediately. Simply entering `host` in this field will compare the values of the `host` field.

The third screen simply lets you choose whether this search is available to other users. Not all users will have permissions to make searches public.

Summary

In this chapter, we covered searching in Splunk and doing a few useful things with those search results. There are lots of little tricks that we will touch upon as we go forward.

In the next chapter, we will start using fields for more than searches; we'll build tables and graphs, and then, we'll learn how to make our own fields.

3

Tables, Charts, and Fields

Up to this point, we have learned how to search for and retrieve raw events, but you will most likely want to create tables and charts to expose useful patterns. Thankfully, the reporting commands in Splunk make short work of most reporting tasks. We will step through a few common use cases in this chapter. Later in the chapter, we will learn how to create custom fields for even more custom reports.

About the pipe symbol

Before we dive into the actual commands, it is important to understand what the pipe symbol (`|`) is used for in Splunk. In a command line, the pipe symbol is used to represent the sending of data from one process to another. For example, in a Unix-style operating system, you might say:

```
grep foo access.log | grep bar
```

The first command finds, in the file `access.log`, lines that contain `foo`. Its output is taken and piped to the input of the next `grep` command, which finds lines that contain `bar`. The final output goes wherever it was destined, usually the terminal window.

The pipe symbol is different in Splunk in a few important ways:

1. Unlike the command line, events are not simply text, but rather each is a set of key/value pairs. You can think of each event as a database row, a Python dictionary, a Javascript object, a Java map, or a Perl associative array. Some fields are hidden from the user but are available for use. Many of these hidden fields are prefixed with an underscore, for instance `_raw`, which contains the original event text, and `_time`, which contains the parsed time in UTC epoch form. Unlike a database, events do not adhere to a schema, and fields are created dynamically.

2. Commands can do anything to the events they are handed. Usually, a command does one of the following:
 - Modifies or creates fields – for example, `eval`, `rex`
 - Filters events – for example, `head`, `where`
 - Replaces events with a report – for example, `top`, `stats`
3. Some commands can act as generators, which produce what you might call "synthetic" events, such as `|metadata` and `|inputcsv`.

We will get to know the pipe symbol very well through examples.

Using `top` to show common field values

A very common question to answer is, "What values are most common?" When looking for errors, you are probably interested in what piece of code has the most errors. The `top` command provides a very simple way to answer this question. Let's step through a few examples.

First, run a search for errors:

```
source="impl_splunk_gen" error
```

Using our sample data, we find events containing the word `error`, a sampling of which is listed here:

```
2012-03-03T19:36:23.138-0600 ERROR Don't worry, be happy.
[logger=AuthClass, user=mary, ip=1.2.3.4]
2012-03-03T19:36:22.244-0600 ERROR error, ERROR, Error!
[logger=LogoutClass, user=mary, ip=3.2.4.5, network=green]
2012-03-03T19:36:21.158-0600 WARN error, ERROR, Error!
[logger=LogoutClass, user=bob, ip=3.2.4.5, network=red]
2012-03-03T19:36:21.103-0600 ERROR Hello world. [logger=AuthClass,
user=jacky, ip=4.3.2.1]
2012-03-03T19:36:19.832-0600 ERROR Nothing happened. This is worthless.
Don't log this. [logger=AuthClass, user=bob, ip=4.3.2.1]
2012-03-03T19:36:18.933-0600 ERROR Hello world. [logger=FooClass,
user=Bobby, ip=1.2.3.4]
2012-03-03T19:36:16.631-0600 ERROR error, ERROR, Error!
[logger=LogoutClass, user=bob, ip=1.2.3.3]
2012-03-03T19:36:13.380-0600 WARN error, ERROR, Error! [logger=FooClass,
user=jacky, ip=4.3.2.1, network=red]
2012-03-03T19:36:12.399-0600 ERROR error, ERROR, Error!
[logger=LogoutClass, user=linda, ip=3.2.4.5, network=green]
```

```
2012-03-03T19:36:11.615-0600 WARN error, ERROR, Error! [logger=FooClass,
user=mary, ip=1.2.3.4]
```

```
2012-03-03T19:36:10.860-0600 ERROR Don't worry, be happy.
[logger=BarClass, user=linda, ip=4.3.2.1, network=green]
```

To find the most common values of `logger`, simply add `| top logger` to our search, like so:

```
source="impl_splunk_gen" error | top logger
```

The results are transformed by `top` into a table like the following one:

	logger ↕	count ↕	percent ↕
1	BarClass	242	63.185379
2	FooClass	49	12.793734
3	AuthClass	47	12.271540
4	LogoutClass	45	11.749347

From these results, we see that **BarClass** is logging significantly more errors than any other logger. We should probably contact the developer of that code.

Next, let's determine whom those errors are happening to. Adding another field name to the end of the command instructs `top` to slice the data again. For example, let's add `user` to the end of our previous query, like so:

```
sourcetype="impl_splunk_gen" error | top logger user
```

The results might look like the following screenshot:

	logger ↕	user ↕	count ↕	percent ↕
1	BarClass	mary	114	14.709677
2	BarClass	Bobby	101	13.032258
3	BarClass	linda	98	12.645161
4	BarClass	jacky	89	11.483871
5	BarClass	bob	83	10.709677
6	FooClass	mary	28	3.612903
7	FooClass	jacky	25	3.225806
8	FooClass	linda	24	3.096774
9	LogoutClass	Bobby	23	2.967742
10	LogoutClass	bob	22	2.838710

In these results, we see that `mary` is logging the most errors from the logger `BarClass`. If we simply wanted to see the distribution of errors by `user`, you could specify only the `user` field, like so:

```
sourcetype="impl_splunk_gen" error | top user
```

Controlling the output of top

The default behavior for `top` is to show the 10 largest counts. The possible row count is the product of all fields specified, in this case `logger` and `user`. In this case, there are 25 possible combinations. If you would like to see more than 10 rows, add the argument `limit`, like so:

```
sourcetype="impl_splunk_gen" error | top limit=100 logger user
```

Arguments change the behavior of a command; they take the form of `name=value`. Many commands require the arguments to immediately follow the command name, so it's a good idea to always follow this structure.

Each command has different arguments, as appropriate. As you type in the search bar, a help drop-down box will appear for the last command in your search, as shown in the following figure:



```
top | Help | More »
Displays the most common values of a field.

Examples
Return the 20 most common values of the "url" field.
... | top limit=20 url

Return top "user" values for each "host".
... | top user by host

Return top URL values.
... | top url
```

Help takes you to the documentation for that command at `splunk.com`.
More >> provides concise documentation in-line.

Let's use a few arguments to make a shorter list but also roll all other results into another line:

```
sourcetype="impl_splunk_gen" error
| top
  limit=5
  useother=true
  otherstr="everything else"
  logger user
```

This produces results like those shown in the following screenshot:

	logger ↕	user ↕	count ↕	percent ↕
1	BarClass	mary	162	19.565217
2	BarClass	linda	102	12.318841
3	BarClass	jacky	97	11.714976
4	BarClass	Bobby	89	10.748792
5	BarClass	bob	72	8.695652
6	everything else	everything else	306	36.956522

The last line represents everything that didn't fit into the top five. `useother` enables this last row, while `otherstr` controls the value printed instead of the default value "other".

For the opposite of `top`, see the `rare` command.

Using stats to aggregate values

While `top` is very convenient, `stats` is extremely versatile. The basic structure of a `stats` statement is:

```
stats functions by fields
```

Many of the functions available in `stats` mimic similar functions in SQL or Excel, but there are many functions unique to Splunk. The simplest `stats` function is `count`. Given the following query, the results will contain exactly one row, with a value for the field `count`:

```
sourcetype="impl_splunk_gen" error | stats count
```

Using the `by` clause, `stats` will produce a row per unique value for each field listed, which is similar to the behavior of `top`. Run the following query:

```
sourcetype="impl_splunk_gen" error | stats count by logger user
```

It will produce a table like that shown in the following screenshot:

	logger ↕	user ↕	count ↕
1	AuthClass	Bobby	877
2	AuthClass	bob	939
3	AuthClass	jacky	851
4	AuthClass	linda	890
5	AuthClass	mary	1809
6	BarClass	Bobby	4470
7	BarClass	bob	4340
8	BarClass	jacky	4558
9	BarClass	linda	4513
10	BarClass	mary	8799
11	FooClass	Bobby	933
12	FooClass	bob	877
13	FooClass	jacky	934
14	FooClass	linda	940
15	FooClass	mary	1737
16	LogoutClass	Bobby	885
17	LogoutClass	bob	834
18	LogoutClass	jacky	944
19	LogoutClass	linda	860
20	LogoutClass	mary	1720

There are a few things to notice about these results:

1. The results are sorted against the values of the "by" fields, in this case `logger` followed by `user`. Unlike `top`, the largest value will not necessarily be at the top of the list. You can sort in the GUI simply by clicking on the field names at the top of the table, or by using the `sort` command.
2. There is no limit to the number of rows produced. The number of rows will equal the possible combinations of field values.
3. The function results are displayed last. In the next example, we will add a few more functions, and this will become more obvious.

Using `stats`, you can add as many "by" fields or functions as you want into a single statement. Let's run this query:

```
sourcetype="impl_splunk_gen" error
| stats
  count avg(req_time) max(req_time) as "Slowest time"
  by logger user
```

The results look like those in the following screenshot:

	logger ↕	user ↕	count ↕	avg(req_time) ↕	Slowest time ↕
1	AuthClass	Bobby	9	7568.000000	10875
2	AuthClass	bob	15	6799.600000	11749
3	AuthClass	jacky	17	4726.714286	9051
4	AuthClass	linda	13	5927.142857	10764
5	AuthClass	mary	39	6029.200000	12108
6	BarClass	Bobby	79	6462.081081	11969
7	BarClass	bob	86	5579.666667	11163
8	BarClass	jacky	99	5647.111111	11688
9	BarClass	linda	100	7122.333333	12071
10	BarClass	mary	142	6100.516667	12187
11	FooClass	Bobby	16	6468.200000	12164
12	FooClass	bob	15	3890.125000	9388
13	FooClass	jacky	20	4502.444444	12128
14	FooClass	linda	19	7087.200000	12151
15	FooClass	mary	36	6375.166667	11421
16	LogoutClass	Bobby	19	6110.666667	11170
17	LogoutClass	bob	25	5784.100000	12169
18	LogoutClass	jacky	17	4448.428571	10820
19	LogoutClass	linda	14	5731.428571	10709
20	LogoutClass	mary	28	5938.600000	10957

Let's step through every part of this query, just to be clear:

- `sourcetype="impl_splunk_gen" error` is the query itself.
- `| stats` starts the `stats` command.
- `count` will return the number of events.

- `avg(req_time)` produces an average value of the `req_time` field.
- `max(req_time)` as "Slowest time" finds the maximum value of the `req_time` field and places the value in a field called `Slowest time`. The quotes are necessary because the field name contains a space.
- `by` indicates that we are done listing functions and want to list the fields to slice the data by. If the data does not need to be sliced, `by` and the fields following it can be omitted.
- `logger` and `user` are our fields for slicing the data. All functions are actually run against each set of data produced per possible combination of `logger` and `user`.



If an event is missing a field that is referenced in a `stats` command, you may not see the results you are expecting. For instance, when computing an average, you may wish for events missing a field to count as zeroes in the average. Also, for events that do not contain a field listed in the `by` fields, the event will simply be ignored.

To deal with both of these cases, you can use the `fillnull` command to make sure that the fields you want exist. We will cover this in *Chapter 5, Advanced Search Examples*.

Let's look at another example, using a time-based function and a little trick. Let's say we wanted to know the most recent time at which each user saw a particular error. We can use the following query:

```
sourcetype="impl_splunk_gen" error logger="FooClass"
| stats count first(ip) max(_time) as _time by user
```

This query produces the following table:

	<code>_time</code> ↕	<code>user</code> ↕	<code>count</code> ↕	<code>first(ip)</code> ↕
1	3/20/12 5:50:00.335 PM	Bobby	115	1.2.3.4
2	3/20/12 5:47:50.467 PM	bob	116	1.2.3.
3	3/20/12 5:48:29.899 PM	extrauser	56	1.2.3.
4	3/20/12 5:49:10.541 PM	jacky	113	1.2.3.4
5	3/20/12 5:44:20.408 PM	linda	120	1.2.3.4
6	3/20/12 5:49:36.602 PM	mary	221	3.2.4.5

Let's step through this example:

- `sourcetype="impl_splunk_gen" error logger="FooClass"` is the query that will find all errors logged by the class `FooClass`.
- `| stats` is our command.
- `count` shows how many times each user saw this error.
- `first(ip)` gives us the IP address that was most recently logged for this user. This will be the most recent event, since results are returned in the order of the most recent first.
- `max(_time)` as `_time` returns the time at which each user most recently saw this error. This takes advantage of three aspects of time in Splunk:
 - `_time` is always present in raw events. As discussed in *Chapter 2, Understanding Search*, the value is the number of seconds since 1970, UTC.
 - `_time` is stored as a number and can be treated as such.
 - If there is a field called `_time` in the results, Splunk will always display the value as the first column of a table in the time zone selected by the user.
- `by user` is our field to split results against.

We have only seen a few functions in `stats`. There are dozens of functions and some advanced syntax that we will touch upon in later chapters. The simplest way to find the full listing is to search with your favorite search engine for `splunk stats functions`.

Using chart to turn data

The `chart` command is useful for "turning" data across two dimensions. It is useful for both tables and charts. Let's start with one of our examples from `stats`:

```
sourcetype="impl_splunk_gen" error | chart count over logger by user
```

The resulting table looks like this:

	logger ↕	Bobby ↕	NULL ↕	bob ↕	extrauser ↕	jacky ↕	linda ↕	mary ↕
1	AuthClass	106	197	114	49	116	119	254
2	BarClass	615	1027	597	238	592	605	1235
3	FooClass	126	164	119	57	131	132	226
4	LogoutClass	123	200	119	49	119	127	261

If you look back at the results from `stats`, the data is presented as one row per combination. Instead of a row per combination, `chart` generates the intersection of the two fields. You can specify multiple functions, but you may only specify one field each for `over` and `by`.

Switching the fields turns the data the other way.

	user ↕	AuthClass ↕	BarClass ↕	FooClass ↕	LogoutClass ↕	NULL ↕
1	Bobby	106	615	126	123	298
2	bob	114	597	119	119	316
3	extrauser	49	238	57	49	0
4	jacky	116	592	131	119	315
5	linda	119	605	132	127	295
6	mary	254	1235	226	261	638

By simply clicking on the chart icon above the table, we can see these results in a chart:



This is a bar chart, with **Stack mode** set to **Stacked**, and `usnull` set to `false`, like so:

```
sourcetype="impl_splunk_gen" error  
| chart usnull=false count over logger by user
```

`chart` can also be used to simply turn data, even if the data is non-numerical. For example, say we enter this query:

```
sourcetype="impl_splunk_gen" error  
| chart usnull=false values(network) over logger by user
```

It will create a table like this:

	logger ↕	Bobby ↕	bob ↕	jacky ↕	linda ↕	mary ↕
1	AuthClass		red	red	red	green
2	BarClass	red	green red	green	green red	green red
3	FooClass			green red	red	
4	LogoutClass		red			green red

Since there are no numbers, this cannot be directly made into an image, but it is still a very useful representation of the data.

Using timechart to show values over time

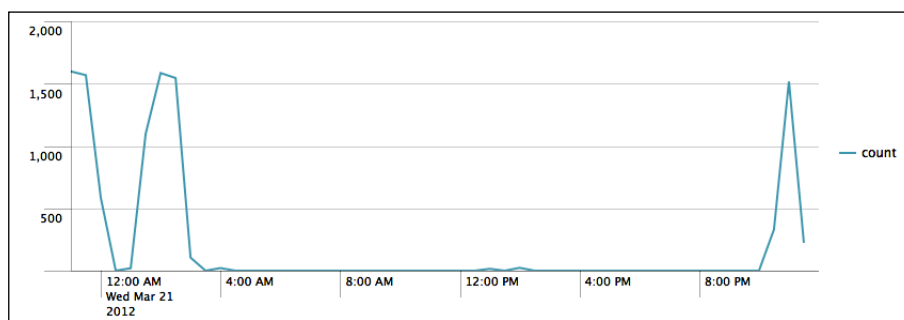
`timechart` lets us show numerical values over time. It is similar to the `chart` command, except that time is always plotted on the x axis. Here are a couple of things to note:

- The events must have an `_time` field. If you are simply sending the results of a search to `timechart`, this will always be true. If you are using interim commands, you will need to be mindful of this requirement.
- Time is always "bucketed", meaning that there is no way to draw a point per event.

Let's see how many errors have been occurring:

```
sourcetype="impl_splunk_gen" error | timechart count
```

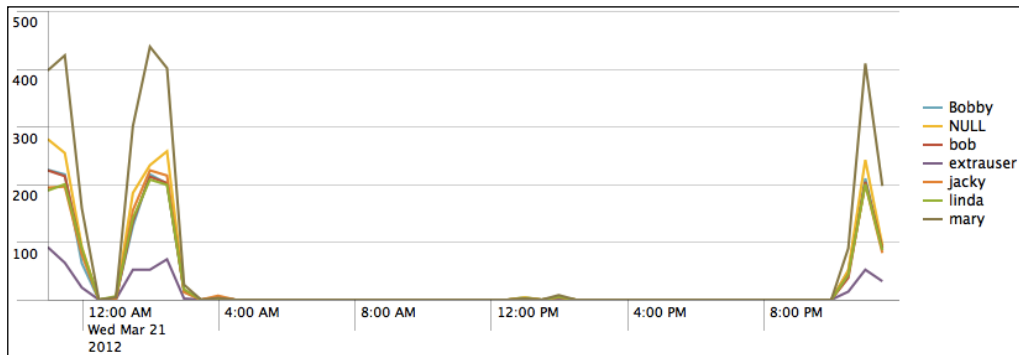
The default chart will look something like this:



Now let's see how many errors have occurred per user over the same time period. We simply need to add `by user` to the query:

```
sourcetype="impl_splunk_gen" error | timechart count by user
```

This produces the following chart:



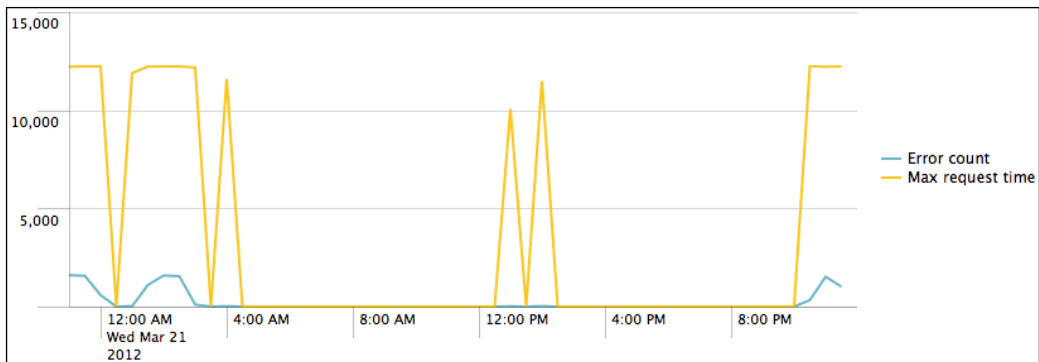
As we stated before, the x axis is always time. The y axis can be:

- One or more functions
- A single function with a `by` clause
- Multiple functions with a `by` clause (a new feature in Splunk 4.3)

An example of a timechart with multiple functions might be:

```
sourcetype="impl_splunk_gen" error  
| timechart  
  count as "Error count"  
  max(req_time) as "Max request time"
```

This would produce a graph like this:



timechart options

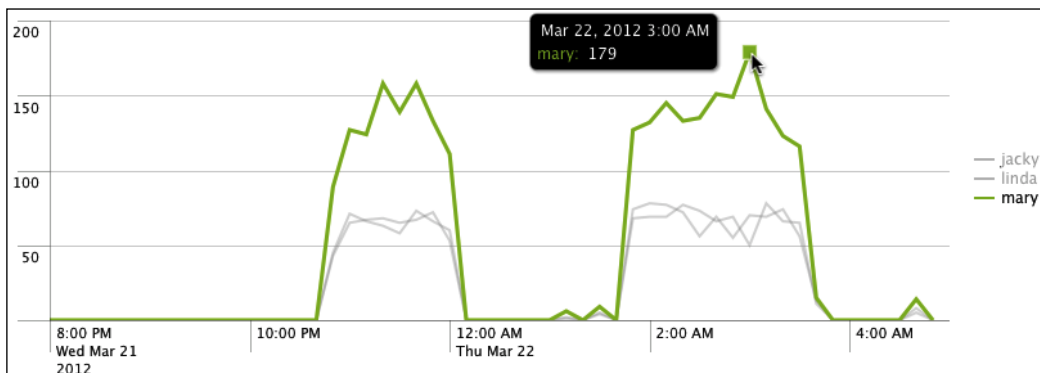
`timechart` has many arguments and formatting options. We'll touch upon a few examples of formatting, but they are too numerous to cover in detail. We will use other chart types in later chapters. Let's throw a few options in and see what they do.

```
timechart bins=100 limit=3 useother=false usenull=false
count as "Error count" by user
```

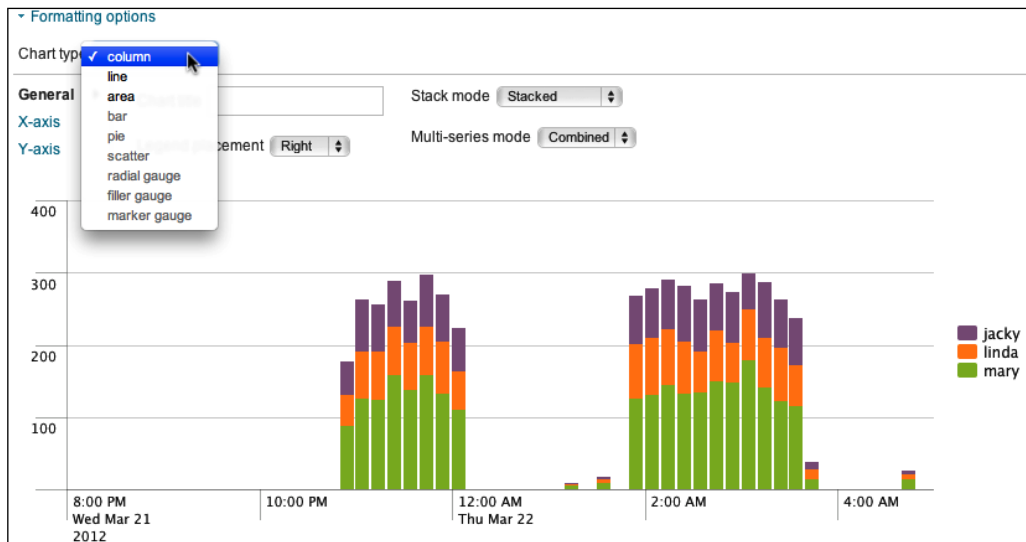
Let's step through each of these arguments:

- `bins` defines how many "bins" to slice time into. The number of bins will probably not be exactly 100 as the time will be sliced into logical units. In our example, this comes to 10 minutes per bin. To be more exact, you can use `span` (for example, `span=1h`) for hourly slices, but note that if your `span` value creates too many time slices, the chart will be truncated.
- `limit` changes the number of series returned. The series with the largest values are returned, much like in `top`. In this case, the most common values of user will be returned.
- `useother` instructs `timechart` whether to group all series beyond the limit into an "other" bucket. The default value is `true`.
- `usenull` instructs `timechart` whether to bucket, into the group `NULL`, events that do not have a value for the fields in the `by` clause. The default value is `true`.

This combination of arguments produces a graph similar to this:



Clicking on **Formatting options** above the graph gives us quite a few options to work with.



This graph shows one of my personal favorite chart styles, the stacked column. This graph is useful for showing how many events of a certain kind occurred, but with colors to give us an idea of distribution. splunk.com has great examples of all of the available chart styles, and we will touch upon more styles in future chapters.

Working with fields

All of the fields we have used so far were either indexed fields (such as `host`, `sourcetype`, and `_time`) or fields that were automatically extracted from `key=value` pairs. Unfortunately, most logs don't follow this format, especially for the first few values in each event. New fields can be created either inline, by using commands, or through configuration.

A regular expression primer

Most of the ways to create new fields in Splunk involve regular expressions. There are many books and sites dedicated to regular expressions, so we will only touch upon the subject here.

Given the log snippet `ip=1.2.3.4`, let's pull out the subnet (1.2.3) into a new field called `subnet`. The simplest pattern would be the literal string:

```
ip=(?P<subnet>1.2.3).4
```

This is not terribly useful as it will only find the subnet of that one IP address. Let's try a slightly more complicated example:

```
ip=(?P<subnet>\d+\.\d+\.\d+)\.\d+
```

Let's step through this pattern:

- `ip=` simply looks for the raw string `ip=`.
- `(` starts a "capture buffer". Everything until the closing parentheses is part of this capture buffer.
- `?P<subnet>` immediately inside a parentheses, says "create a field called `subnet` from the results of this capture buffer".
- `\d` matches any single digit, from 0 to 9.
- `+` says "one or more of the item immediately before".
- `\.` matches a literal period. A period without the backslash matches any character.
- `\d+\.\d+` matches the next two parts of the IP address.
- `)` ends our capture buffer.
- `\.\d+` matches the last part of the IP address. Since it is outside of the capture buffer, it will be discarded.

Now let's step through an overly complicated pattern to illustrate a few more concepts:

```
ip=(?P<subnet>\d+\d*\.[01234-9]+)\.\d+
```

Let's step through this pattern:

- `ip=` simply looks for the raw string `ip=`.
- `(?P<subnet>` starts our capture buffer and defines our field name.
- `\d` means digit. This is one of the many backslash character combinations that represent some sets of characters.
- `+` says "one or more of what came before", in this case `\d`.
- `.` matches a single character. This will match the period after the first set of digits, though it would match any single character.

- `\d*` means zero or more digits.
- `\.` matches a literal period. The backslash negates the special meaning of any special punctuation character. Not all punctuation marks have a special meaning, but so many do that there is no harm adding a backslash before a punctuation mark that you want to literally match.
- `[` starts a character set. Anything inside the brackets will match a single character in the character set.
- `01234-9` means the characters 0, 1, 2, 3, and the range 4-9.
- `]` closes the character set.
- `+` says "one or more of what came before", in this case the character set.
- `)` ends our capture buffer.
- `\.\d+` is the final part of the IP address that we are throwing away. It is not actually necessary to include this, but it ensures that we only match if there were in fact four sets of numbers.

There are a number of different ways to accomplish the task at hand. Here are a few examples that will work:

```
ip=(?P<subnet>\d+\.\d+\.\d+)\.\d+
ip=(?P<subnet>(\d+\.)\{2\}\d+)\.\d+
ip=(?P<subnet>[\d\.]+\)\.\d
ip=(?P<subnet>.*?\.\.*?\.\.*?)\.\
ip=(?P<subnet>\S+)\.
```

For more information about regular expressions, consult the man pages for **Perl Compatible Regular Expressions (PCRE)**, which can be found online at <http://www.pcre.org/pcre.txt>, or one of the many regular expression books or websites dedicated to the subject. We will build more expressions as we work through different configurations and searches, but it's definitely worthwhile to have a reference handy.

Commands that create fields

There are a number of commands that create new fields, but the most commonly used are `eval` and `rex`.

eval

`eval` allows you to use functions to build new fields, much as you would build a formula column in Excel, for example:

```

sourcetype="impl_splunk_gen"
| eval req_time_seconds=req_time/1000
| stats avg(req_time_seconds)

```

This creates a new field called `req_time_seconds` on every event that has a value for `req_time`. Commands after this statement see the field as if it were part of the original event. `stats` then creates a table of the average value of our newly created field.

	avg(req_time_seconds) ↕
1	6.175161

There are a huge number of functions available for use with `eval`. The simplest way to find the full listing is to search google.com for `splunk eval functions`. I would suggest bookmarking this page as you will find yourself referring to it often.

rex

`rex` lets you use regular expressions to create fields. It can work against any existing field but, by default, will use the field `_raw`. Let's try one of the patterns we wrote in our short regular expression primer:

```

sourcetype="impl_splunk_gen"
| rex "ip=(?P<subnet>\d+\.\d+\.\d+)\.\d+"
| chart values(subnet) by user network

```

This would create a table like this:

	user ↕	green ↕	red ↕
1	bob	1.2.3 3.2.4	1.2.3
2	jacky	1.2.3 3.2.4	1.2.3
3	linda	1.2.3	1.2.3 3.2.4
4	mary	1.2.3 4.3.2	1.2.3

With the addition of the `field` argument, we can work against the `ip` field that is already being created automatically from the `name=value` pair in the event.

```
sourcetype="impl_splunk_gen"  
| rex field=ip "(?P<subnet>.*)\."  
| chart values(subnet) by user network
```

This will create exactly the same result as the previous example.

Extracting loglevel

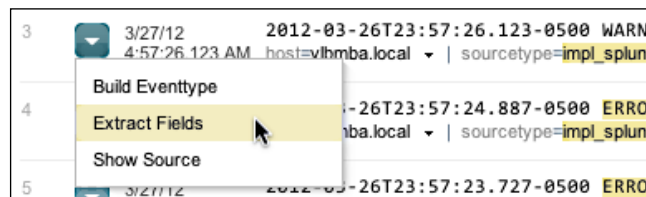
In our examples, we searched for the raw word `error`. You may have noticed that many of the events weren't actually errors, but simply contained the word `error` somewhere in the message. For example, given the following events, we probably only care about the second event:

```
2012-03-21T18:59:55.472-0500 INFO This is not an error  
2012-03-21T18:59:42.907-0500 ERROR Something bad happened
```

Using an extracted field, we can easily create fields in our data, without re-indexing, that allow you to search for values that occur in a specific location in your events.

Using the Extract Fields interface

There are several ways to define a field. Let's start by using the **Extract Fields** interface. To access this interface, choose **Extract Fields** from the workflow actions menu next to any event:



This menu launches the **Extract fields** view:

splunk > Extract fields

A regex has been successfully learned. Validate its correctness by reviewing the Sample extractions, or running Test. To improve results, add more examples and remove incorrect extractions.

Interactive field extractor
Teach Splunk how to extract a field by providing it with example values.

Restrict extraction to:
sourcetype="impl_splunk_gen"

Example values for a field:
ERROR
WARN
INFO
(One example per line. Include multiple examples for best results.)

Generate

Generated pattern (regex)
(?)^[^]* (?P<FIELDNAME>[^\s]+)
Edit Test Save

Sample extractions
Validate the extracted values. To improve results, remove incorrect extractions and add more example values.

- DEBUG
- WARN
- INFO
- ERROR

Sample events
This list is based on the event you selected from your search and the field restriction you specified.

2012-03-27T01:04:50.645-0500	ERROR	error, ERROR, Error! [logger=BarClass,
2012-03-27T01:04:49.885-0500	ERROR	Nothing happened. This is worthless. D
2012-03-27T01:04:49.610-0500	DEBUG	error, ERROR, Error! [logger=BarClass,
2012-03-27T01:04:49.463-0500	ERROR	Don't worry, be happy. [logger=BarClass
2012-03-27T01:04:49.400-0500	WARN	Don't worry, be happy. [logger=BarClass
2012-03-27T01:04:49.235-0500	DEBUG	Nothing happened. This is worthless. D
2012-03-27T01:04:48.296-0500	DEBUG	Nothing happened. This is worthless. D

In this view, you simply provide example values, and Splunk will attempt to build a regular expression that matches. In this case, we specify **ERROR**, **WARN**, and **INFO**.

Under **Sample extractions**, we see that the values **DEBUG**, **WARN**, **INFO**, and **ERROR** were matched. Notice that there are more values than we listed – the pattern is looking for placement, not our sample values.

Under **Sample events**, we get a preview of what data was matched, in context.

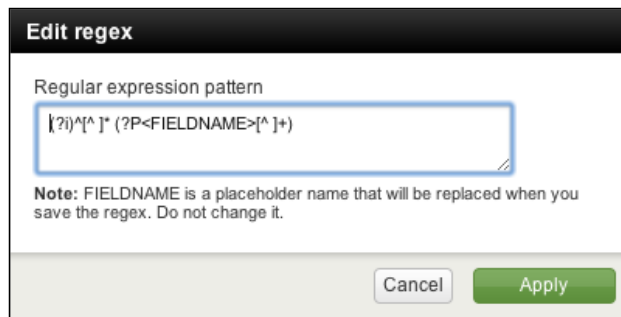
Finally, under **Generated pattern**, we see the regular expression that Splunk generated, which is as follows:

```
(?i)^[^ ]* (?P<FIELDNAME>[^\ ]+)
```

Let's step through the pattern:

- `(?i)` says that this pattern is case insensitive. By default, regular expressions are case sensitive.
- `^` says that this pattern must match at the beginning of the line.
- `[^]*` says "any character but a space, zero or more times".
- The space is literal.
- `(?P<FIELDNAME>[^\]+)` says to match anything that is not a space, and capture it in the field `FIELDNAME`. You will have the opportunity to name the field when you click on **Save**.

Edit presents a dialog to let you modify the pattern manually:



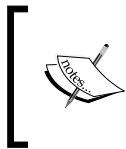
Test will launch a new search window with the pattern loaded into a very useful query that shows the most common values extracted. In this case, it is the following query:

```
index=main sourcetype="impl_splunk_gen"  
| head 10000  
| rex "(?i)^[^ ]* (?P<FIELDNAME>[^\ ]+)"  
| top 50 FIELDNAME
```

Save prompts you for a name for your new field. Let's call this field `loglevel` and save it:

Now that we've defined our field, we can use it in a number of ways, as follows:

- We can search for the value using the fieldname, for instance, `loglevel=error`



When searching for values by fieldname, the fieldname *is* case sensitive, but the value *is not* case sensitive. In this case `loglevel=Error` would work just fine, but `LogLevel=error` would not.

- We can report on the field, whether we searched for it or not. For instance:
`sourcetype="impl_splunk_gen" user=mary | top loglevel`
- We can search for only events that contain our field:
`sourcetype="impl_splunk_gen" user=mary loglevel="*"`

Using rex to prototype a field

When defining fields, it is often convenient to build the pattern directly in the query and then copy the pattern into configuration. You might have noticed that the test in the Extract fields workflow used `rex`.

Let's turn the subnet pattern we built earlier into a field. First, we build the query with the `rex` statement:

```
sourcetype="impl_splunk_gen" ip="*"
| rex "ip=(?P<subnet>\d\.\d\.\d+)\.\d+"
| table ip subnet
```

Since we know there will be an `ip` field in the events we care about, we can use `ip="*"` to limit the results to events that have a value for that field.

`table` takes a list of fields and displays a table, one row per event:

	ip ↕	subnet ↕
1	1.2.3.4	1.2.3
2	1.2.3.4	1.2.3
3	4.31.2.1	
4	4.31.2.1	
5	4.31.2.1	
6	4.31.2.1	
7	1.22.3.3	
8	1.2.3.4	1.2.3
9	1.22.3.3	
10	1.22.3.3	
11	1.22.3.3	

As we can see, the `rex` statement doesn't always work. Looking at the pattern again, you may notice that the first two instances of `\d` are now missing their trailing `+`. Without the plus sign, only addresses with a single digit in both their first and second sections will match. After adding the missing plus signs to our pattern, all rows will have a subnet.

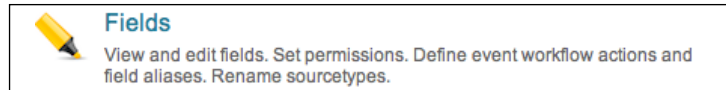
```
sourcetype="impl_splunk_gen" ip="*"
| rex "ip=(?P<subnet>\d+\.\d+\.\d+)\.\d+"
| table ip subnet
```

We can now take the pattern from the `rex` statement and use it to build a configuration.

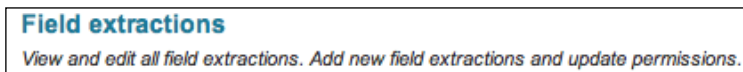
Using the admin interface to build a field

Taking our pattern from the previous example, we can build the configuration to "wire up" this extract.

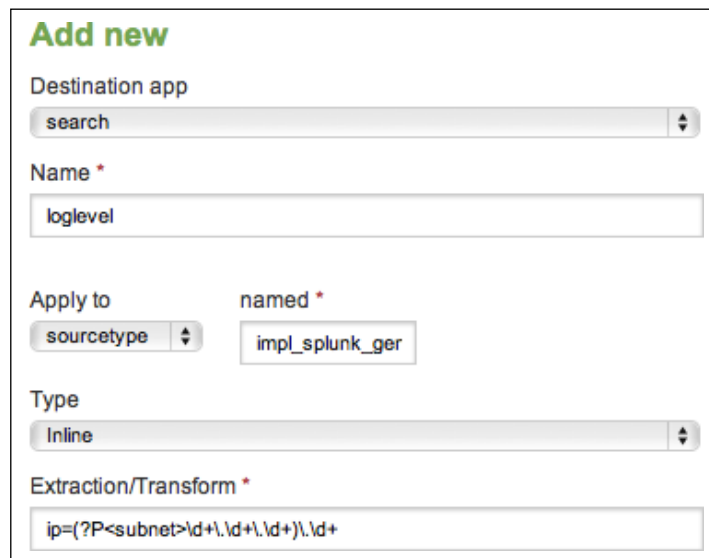
First, click on **Manager** in the upper-right corner. The **Fields** section contains everything, funnily enough, about fields.



There are a number of different things you can do with fields via configuration, but for now, we're interested in **Field extractions**.



After clicking on **Add new** to the right of **Field extractions**, or on the **New** button after clicking on **Field extractions**, we are presented with the interface for creating a new field.



Add new

Destination app
search

Name *
loglevel

Apply to
sourcetype

named *
impl_splunk_ger

Type
Inline

Extraction/Transform *
ip=(?P<subnet>\d+\.\d+\.\d+)\.\d+

Now, we step through the fields:

- **Destination app** lets us choose the app where this extraction will live and by default, where it will take effect. We will discuss the scope of configurations in *Chapter 10, Configuring Splunk*.
- **Name** is simply a display name for the extraction. Make it as descriptive as you like.
- **Apply to** lets you choose what to bind this extraction to. Your choices are **sourcetype**, **source**, and **host**. The usual choice is **sourcetype**.
- **named** is the name of the item we are binding our extraction to.
- **Type** lets you choose **Inline**, which means specifying the regular expression here, or **Uses transform**, which means we will specify a named transform that exists already in configuration.
- **Extraction/Transform** is where we place either our pattern, if we chose a **Type** option of **Inline**, or the name of a **Transform** object.

Once you click on **Save**, you will return to the listing of extractions. By default, your extraction will be private to you and will only function in the application it was created in. If you have rights to do so, you can share the extraction with other users and change the scope of where it runs. Click on **Permissions** in the listing to see the permissions page, which most objects in Splunk use.

Roles	Read	Write
Everyone	<input type="checkbox"/>	<input type="checkbox"/>
admin	<input type="checkbox"/>	<input type="checkbox"/>
can_delete	<input type="checkbox"/>	<input type="checkbox"/>
power	<input type="checkbox"/>	<input type="checkbox"/>
user	<input type="checkbox"/>	<input type="checkbox"/>

The top section controls the context in which this extraction will run. Think about when the field would be useful, and limit the extractions accordingly. An excessive number of extractions can affect performance, so it is a good idea to limit the extracts to a specific app when appropriate. We will talk more about creating apps in *Chapter 7, Working with Apps*.

The second section controls what roles can read or write this configuration. The usual selections are the **Read** option for the **Everyone** parameter and the **Write** option for the **admin** parameter. As you build objects going forward, you will become very familiar with this dialog.

Indexed fields versus extracted fields

When an event is written to an index, the raw text of the event is captured along with a set of indexed fields. The default indexed fields include `host`, `sourcetype`, `source`, and `_time`. There are distinct advantages and a few serious disadvantages to using indexed fields.

First, let's look at the advantages of an indexed field (we will actually discuss configuring indexed fields in *Chapter 10, Configuring Splunk*):

- As an indexed field is stored in the index with the event itself, it is only calculated at index time, and in fact, can only be calculated once at index time.
- It can make finding specific instances of common terms efficient. See use case 1 in the following section, as an example.
- You can create new words to search against that simply don't exist in the raw text or are embedded inside a word. See use cases 2–4 in the following sections.
- You can efficiently search for words in other indexed fields. See the *Indexed field case 3 – application from source* section.

Now for the disadvantages of an indexed field:

- It is not retroactive. This is different from extracted fields, where all events, past and present, will gain the newly defined field if the pattern matches. This is the biggest disadvantage of indexed fields and has a few implications, as follows:
 - Only newly indexed events will gain a newly defined indexed field
 - If the pattern is wrong in certain cases, there is no practical way to apply the field to already indexed events
 - Likewise, if the log format changes, the indexed field may not be generated (or generated incorrectly)
- It adds to the size of your index on disk.
- It counts against your license.

- Any changes usually require a restart to be applied.
- In most cases, the value of the field is already an indexed word, in which case creating an indexed field will likely have no benefit, except in the rare cases where that value is very common.

With the disadvantages out of the way, let's look at a few cases where an indexed field would improve search performance and then at one case where it would probably make no difference.

Indexed field case 1 – rare instances of a common term

Let's say your log captures process exit codes. If a 1 represents a failure, you probably want to be able to search for this efficiently. Consider a log that looks something like this:

```
4/1/12 6:35:50.000 PM process=important_process.sh, exitcode=1
```

It would be easy to search for this log entry using `exitcode=1`. The problem is that, when working with extracted fields, the search is effectively reduced to this:

```
1 | search exitcode="1"
```

Since the date contains a 1, this search would find every event for the entire day and then filter the events to the few that we are looking for. In contrast, if `exitcode` were defined as an indexed field, the query would immediately find the events, only retrieving the appropriate events from the disk.

Indexed field case 2 – splitting words

In some log formats, multiple pieces of information may be encoded into a single word without whitespace or punctuation to separate the useful pieces of information. For instance, consider a log message such as this:

```
4/2/12 6:35:50.000 PM kernel: abc5s2: 0xc014 (UNDEFINED).
```

Let's pretend that `5s2` is an important piece of information that we need to be able to search for efficiently. The query `*5s2` would find the events but would be a very inefficient search (in essence, a full table scan). By defining an indexed field, you can very efficiently search for this instance of the string `5s2`, because in essence, we create a new "word" in the metadata of this event.



Defining an indexed field only makes sense if you know the format of the logs before indexing, if you believe the field will actually make the query more efficient (see previous section), and if you will be searching for the field value. If you will only be reporting on the values of this field, an extracted field will be sufficient, except in the most extreme performance cases.

Indexed field case 3 – application from source

A common requirement is to be able to search for events from a particular web application. Often, the only easy way to determine the application that created the logs is by inspecting the path to the logs, which Splunk stores in the indexed field `source`. For example, given the following path, the application name is `app_one`:

```
/opt/instance19/apps/app_one/logs/important.log
```

You could search for this instance using `source="*/app_one/*"`, but this effectively initiates a full table scan. You could define an extracted field and then search for `app="app_one"`, but unfortunately, this approach will be no more efficient because the word we're looking for is not contained in the field `_raw`. If we define this field as an indexed field, `app="app_one"` will be an efficient search.

Once again, if you only need this field for reporting, the extracted field is just fine.

Indexed field case 4 – slow requests

Consider a web access log with a trailing request time in microseconds:

```
[31/Jan/2012:18:07 +0000] "GET / HTTP/1.1" 200 7918 ""
"Mozilla/5.0..." 11/11033255
```

Let's say we want to find all requests that took longer than 10 seconds. We can easily extract the value into a field, perhaps `request_ms`. We could then run the search `request_ms>10000000`. This query will work, but it requires scanning every event in the given time frame. Whether the field is extracted or indexed, we would face the same problem as Splunk has to convert the field value to a number before it can test the value.

What if we could define a field and instead search for `slow_request=1`? To do this, we can take advantage of the fact that, when defining an indexed field, the value can be a static value. This could be accomplished with a transform, like so:

```
REGEX = .*/(\d{7,})$
FORMAT = slow_request::1
```


We will cover transforms, and the configurations involved, in *Chapter 10, Configuring Splunk*.

Once again, this is only worth the trouble if you need to efficiently search for these events and not simply report on the value of `request_ms`.

Indexed field case 5 – unneeded work

Once you learn to make indexed fields, it may be tempting to convert all of your important fields into indexed fields. In most cases it is essentially a wasted effort and ends up using extra disk space, wasting license, and adding no performance boost.

For example, consider this log message:

```
4/2/12 6:35:50.000 PM [vincentbumgarner] [893783] sudo bash
```

Assuming the layout of this message is as follows, it might be tempting to put both `userid` and `pid` into indexed fields:

```
date [userid] [pid] action
```

Since the values are uncommon, and are unlikely to occur in unrelated locations, defining these fields as indexed fields is most likely wasteful. It is much simpler to define these fields as extracted fields and shield ourselves from the disadvantages of indexed fields.

Summary

This has been a very dense chapter, but we have really just scratched the surface on a number of important topics. In future chapters, we will use these commands and techniques in more and more interesting ways. The possibilities can be a bit dizzying, so we will step through a multitude of examples to illustrate as many scenarios as possible.

In the next chapter, we will build a few dashboards using the wizard-style interfaces provided by Splunk.

4

Simple XML Dashboards

Dashboards are a way for you to capture, group, and automate tables and charts into useful and informative views. We will quickly cover the wizards provided in Splunk 4.3 and then dig into the underlying XML. With that XML, you can easily build interactive forms, further customize panels, and use the same query for multiple panels, among other things. We will also cover how and when to schedule the generation of dashboards to reduce both the wait time for users and the load on the server.

The purpose of dashboards

Any search, table, or chart you create can be saved and made to appear in the menus for other users to see. With that power, why would you bother creating a dashboard? Here are a few reasons:

- A dashboard can contain multiple panels, each running a different query.
- Every dashboard has a unique URL, which is easy to share.
- Dashboards are more customizable than an individual query.
- The search bar is removed, making it less intimidating to many users.
- Forms allow you to present the user with a custom search interface that only requires specific values.
- Dashboards look great. Many organizations place dashboards on projectors and monitors for at-a-glance information about their environment.
- Dashboards can be scheduled for PDF delivery by e-mail. This feature is not the most robust, but with some consideration, it can be used effectively.

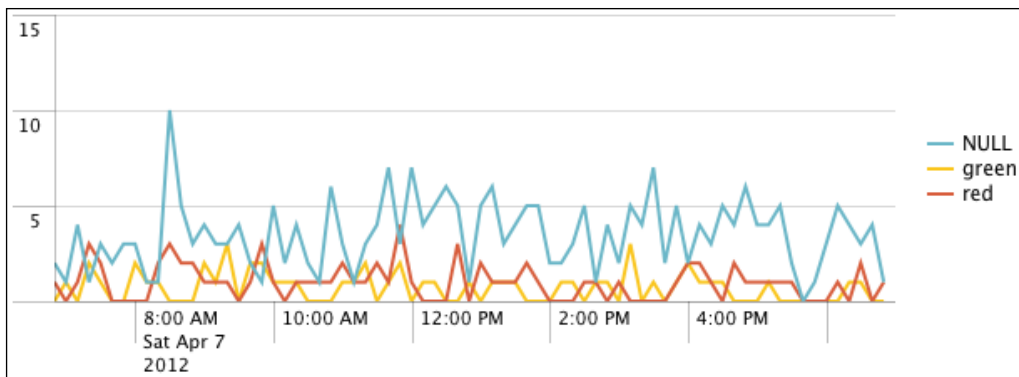
With all of this said, if a saved search is working the way it is, there is no strong reason to turn it into a dashboard.

Using wizards to build dashboards

Using some of the queries from previous chapters, let's make an operational dashboard for errors occurring in our infrastructure. We will start by making a query (note that this query relies on the `loglevel` fields we created in *Chapter 3, Tables, Charts, and Fields*):

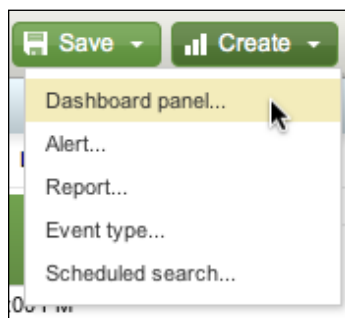
```
sourcetype="impl_splunk_gen" loglevel=error | timechart count as "Error count" by network
```

This will produce a graph like this one:



To add this to a dashboard, we perform the following steps:

1. Choose **Create | Dashboard panel...**



- This opens a wizard interface that guides you through saving the query, adding it to a dashboard, and then scheduling the search. First, we name the search.

Create Dashboard Panel

1 Search — 2 Dashboard — 3 Panel

* Search name

* Search string

Time range to

-1d (one day ago), now (triggering time)
rt-1d (one day ago in real-time), rt(triggering time)
Time specifiers: y, mon, d, h, m, s [Learn more](#)



As you create more dashboards, you will end up creating a lot of searches. A naming convention will help you keep track of what search belongs to what dashboard. Here is one possible approach: Dashboard - [dashboard name] - [search name and panel type]. When the number of dashboards and searches becomes large, apps can be used to group dashboards and searches together, providing yet another way to organize and share assets.

- The next step is to create or choose an existing dashboard.

Create Dashboard Panel

1 Search — 2 Dashboard — 3 Panel

Select New dashboard name

Share dashboard

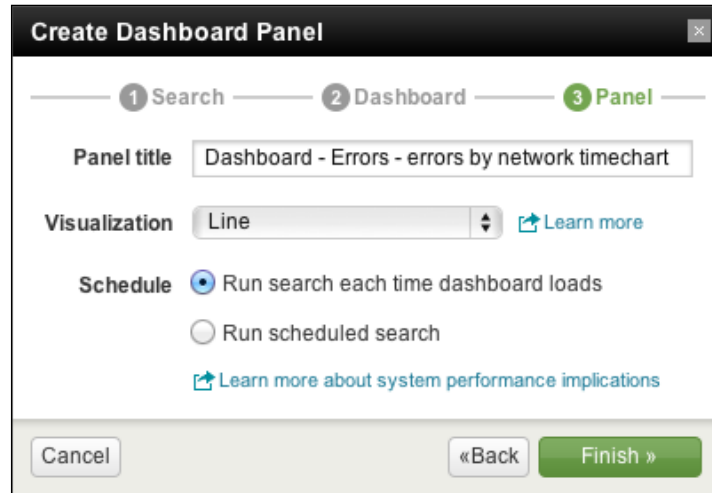
Keep dashboard private

Share as read-only to all users of current app

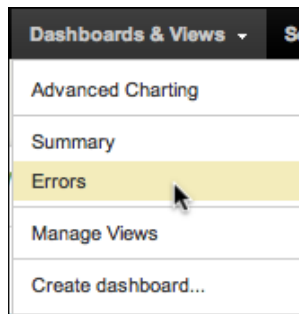
Additional permission settings available in *Manager* » *Searches and reports*

Existing dashboard

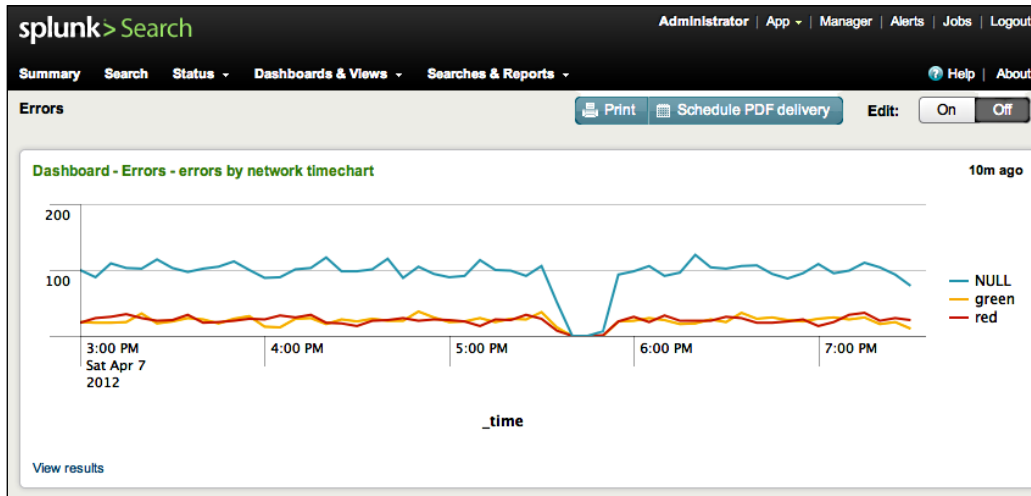
- Let's create a new dashboard called `Errors`. The next step is to add our new saved search to our new dashboard in a new panel.



- Panel title** is the text that will appear above your new panel in the dashboard. **Visualization** lets you choose a chart type and will default to the type of chart you started with. We will discuss **Schedule** in the next section.
- After clicking on **Finish** and saving our dashboard, it will now be available in the **Dashboards & Views** menu.



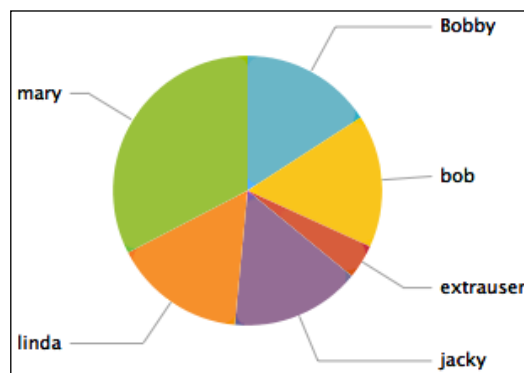
7. The dashboard with our first panel looks as follows:



Following the same steps, let's add a few pie charts showing this information broken down in a few ways.

```
sourcetype="impl_splunk_gen" loglevel=error | stats count by user
```

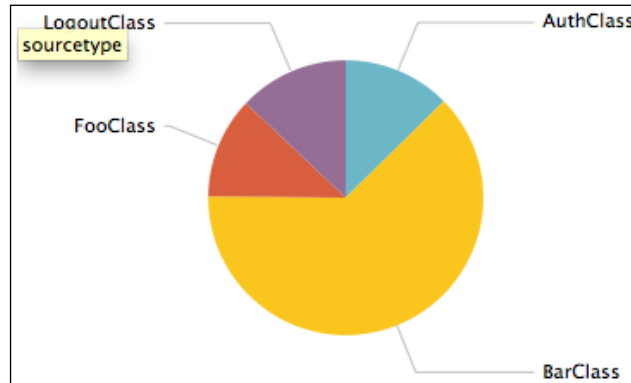
This query produces the following chart:



This gives us a breakdown of errors by user. Next, let's add a breakdown by logger.

```
sourcetype="impl_splunk_gen" loglevel=error | stats count by logger
```

This query produces the following chart:

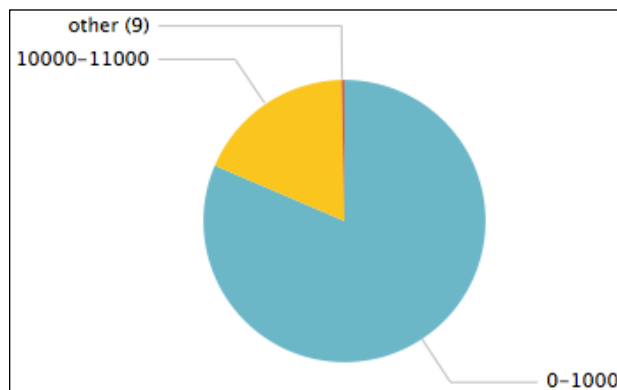


With this breakdown, we can see that the main producer of errors is the logger **BarClass**.

Let's learn another command, `bucket`. The `bucket` command is used to group sets of numeric values and has special capabilities with the `_time` field. This example will group the values of the field `req_time` in up to 10 evenly distributed bins. `bucket` has some other cool tricks we will use later. The following query will group `req_time`:

```
sourcetype="impl_splunk_gen" loglevel=error  
| bucket bins=10 req_time | stats count by req_time
```

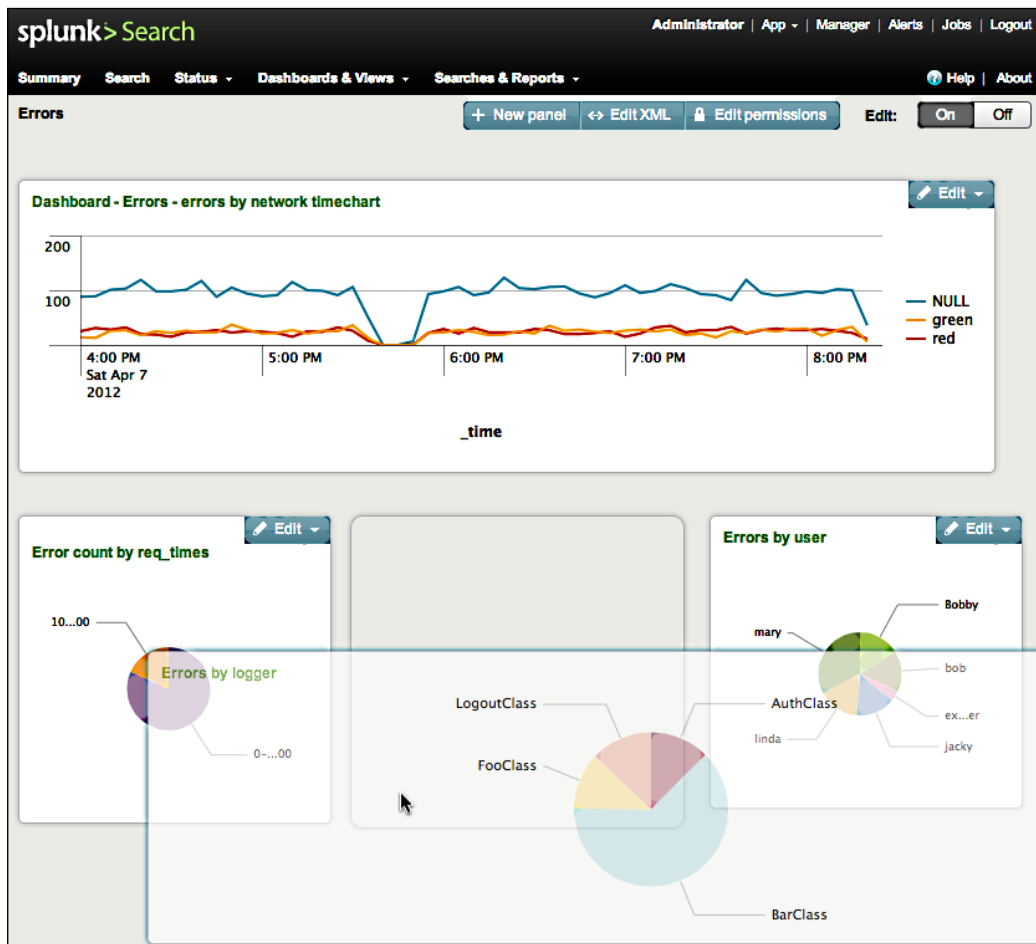
The results produce the following pie chart:



Using the wizard interface, step through the same actions to add these pie charts to our dashboard, this time choosing **Existing dashboard** in step 2.

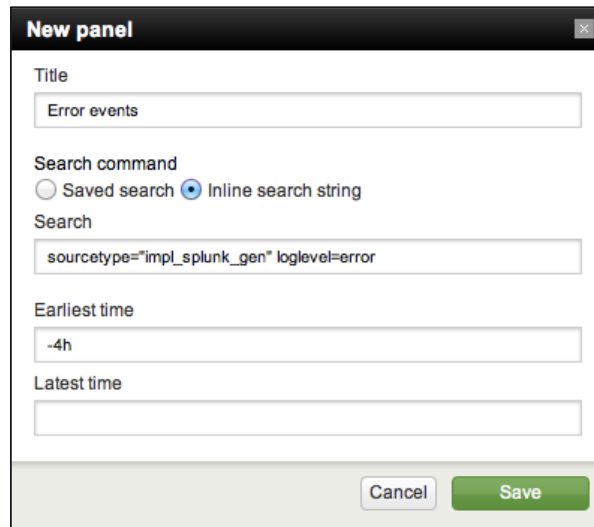
By default, each new panel is added to the bottom of the dashboard. Dashboards allow you to have up to three panels distributed horizontally, which is a great way to show pie charts.

After clicking on the **On** button for **Edit**, near the top of the dashboard, you can drag the panels around the page, like so:



You may have noticed the three new buttons that appeared at the top of the dashboard after we clicked on the **On** button:

- **Edit XML** allows you to directly edit the XML underlying this dashboard. We will use this later in the chapter.
- **Edit permissions** takes you to the standard permissions panel that we have seen before.
- Clicking on **New panel** opens the following dialog to allow us to add new panels directly:



The screenshot shows a 'New panel' dialog box with the following fields and options:

- Title:** Error events
- Search command:** Radio buttons for 'Saved search' and 'Inline search string' (selected).
- Search:** sourcetype=impl_splunk_gen loglevel=error
- Earliest time:** -4h
- Latest time:** (empty)
- Buttons:** Cancel and Save

Saved search lets you choose an existing saved search. This allows you to reuse queries on different dashboards or build queries without a dashboard in mind.

Inline search string lets us build a query directly in the dashboard. This is often convenient as many searches will have no purpose but for a particular dashboard, so there is no reason for these searches to appear in the menus. This also reduces external dependencies, making it easier to move the dashboard to another app. Be sure to either specify an **Earliest time** value, or append `| head` to your query to limit the number of results, or the query will be run over **All time**.

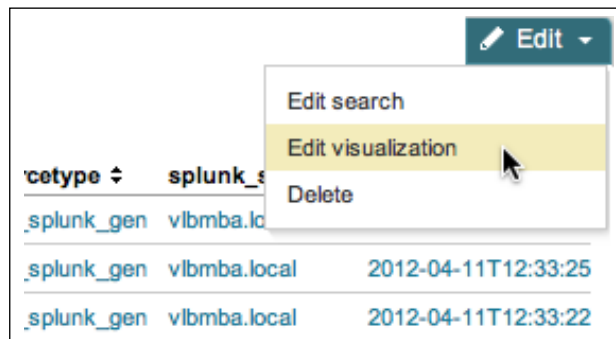
In this case, we want to create an event listing. After clicking on **Save**, this panel is added to our dashboard.

Error events

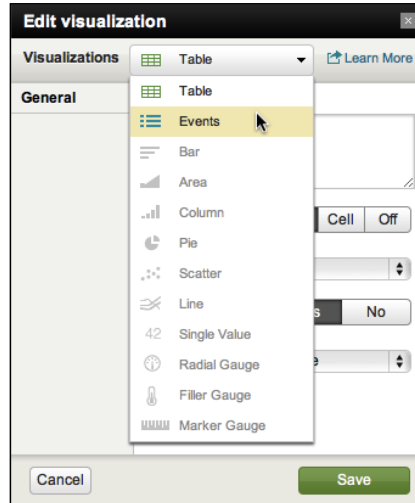
« prev 1 2 3 4 5 6 7 8 9 10 next »

	_time ↕	host ↕	index ↕	linecount ↕	loglevel ↕	source ↕	sourcetype ↕	splunk_server ↕	_raw ↕
1	4/11/12 5:33:25.500 PM	vlbmba.local	main	1	ERROR	impl_splunk_gen	impl_splunk_gen	vlbmba.local	2012-04-11T12:33:25
2	4/11/12 5:33:25.458 PM	vlbmba.local	main	1	ERROR	impl_splunk_gen	impl_splunk_gen	vlbmba.local	2012-04-11T12:33:25
3	4/11/12 5:33:22.883 PM	vlbmba.local	main	1	ERROR	impl_splunk_gen	impl_splunk_gen	vlbmba.local	2012-04-11T12:33:22
4	4/11/12 5:33:22.480 PM	vlbmba.local	main	1	ERROR	impl_splunk_gen	impl_splunk_gen	vlbmba.local	2012-04-11T12:33:22
5	4/11/12 5:33:21.305 PM	vlbmba.local	main	1	ERROR	impl_splunk_gen	impl_splunk_gen	vlbmba.local	2012-04-11T12:33:21
6	4/11/12 5:33:18.818 PM	vlbmba.local	main	1	ERROR	impl_splunk_gen	impl_splunk_gen	vlbmba.local	2012-04-11T12:33:18
7	4/11/12 5:33:17.196 PM	vlbmba.local	main	1	ERROR	impl_splunk_gen	impl_splunk_gen	vlbmba.local	2012-04-11T12:33:17
8	4/11/12 5:33:15.544 PM	vlbmba.local	main	1	ERROR	impl_splunk_gen	impl_splunk_gen	vlbmba.local	2012-04-11T12:33:15
9	4/11/12 5:33:13.277 PM	vlbmba.local	main	1	ERROR	impl_splunk_gen	impl_splunk_gen	vlbmba.local	2012-04-11T12:33:13
10	4/11/12 5:33:11.035 PM	vlbmba.local	main	1	ERROR	impl_splunk_gen	impl_splunk_gen	vlbmba.local	2012-04-11T12:33:11

The default visualization type is **Table**, which is not what we want in this case. To change this, choose **Edit visualization** on the panel.



This presents us with an editor window where we can change the visualization type.



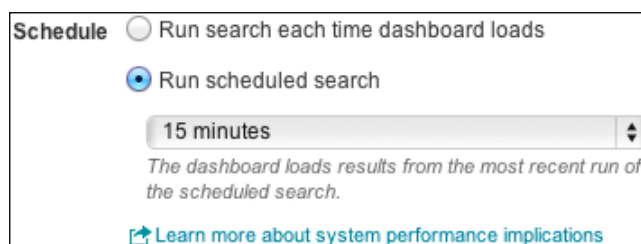
After saving and disabling the **Edit** mode, we see our event listing.

Error events			
« prev 1 2 3 4 5 6 7 8 9 10 next »			
1		4/11/12 5:33:25.500 PM	2012-04-11T12:33:25.500-0500 ERROR Nothing happened. This is worthless. Don't log this. [logger=BarClass, ip=1.2.3., req_time=1308, network=red]
2		4/11/12 5:33:25.458 PM	2012-04-11T12:33:25.458-0500 ERROR Nothing happened. This is worthless. Don't log this. [logger=LogoutClass, user=jacky, ip=4.31.2.1, req_time=6677]
3		4/11/12 5:33:22.883 PM	2012-04-11T12:33:22.883-0500 ERROR Hello world. [user=Bobby, ip=113.2.4.5, req_time=6242, network=red]
4		4/11/12 5:33:22.480 PM	2012-04-11T12:33:22.480-0500 ERROR Hello world. [logger=BarClass, ip=1.2.3.4, req_time=855, session_id="NDA5NDY30Dgy"]
5		4/11/12 5:33:21.305 PM	2012-04-11T12:33:21.305-0500 ERROR Don't worry, be happy. [logger=BarClass, user=mary, ip=1.2.3.4, req_time=7123]
6		4/11/12 5:33:18.818 PM	2012-04-11T12:33:18.818-0500 ERROR Nothing happened. This is worthless. Don't log this. [logger=FooClass, user=linda, ip=4.31.2.1, network=red]
7		4/11/12 5:33:17.196 PM	2012-04-11T12:33:17.196-0500 ERROR Don't worry, be happy. [logger=BarClass, user=mary, req_time=8748, network=red]
8		4/11/12 5:33:15.544 PM	2012-04-11T12:33:15.544-0500 ERROR Nothing happened. This is worthless. Don't log this. [user=Bobby, ip=3.2.4.5, req_time=7527, user=extrauser]
9		4/11/12 5:33:13.277 PM	2012-04-11T12:33:13.277-0500 ERROR Don't worry, be happy. [logger=LogoutClass, user=jacky, ip=1.2.3., req_time=7946]
10		4/11/12 5:33:11.035 PM	2012-04-11T12:33:11.035-0500 ERROR error, ERROR, Error! [logger=LogoutClass, ip=1.22.3.3, req_time=11972, network=green]

This panel is added to the bottom of the dashboard, which is just right in this case.

Scheduling the generation of dashboards

As we stepped through the wizard interface to create panels, we accepted the default value of **Run search each time dashboard loads**. If we instead select **Run scheduled search**, we are given a time picker.



Schedule Run search each time dashboard loads
 Run scheduled search
15 minutes
The dashboard loads results from the most recent run of the scheduled search.
[Learn more about system performance implications](#)

When the dashboard is loaded, the results from the last scheduled run will be used. The dashboard will draw as quickly as the browser can draw the panels. This is particularly useful when multiple users use a dashboard, perhaps in an operations group. If there are no saved results available, the query will simply be run normally.

Be sure to ask yourself just how fresh the data on a dashboard needs to be. If you are looking at a week's worth of data, is up to one-hour-old data acceptable? What about four hours old? 24 hours old? The less often the search is run, the fewer resources you will use, and the more responsive the system will be for everyone else. As your data volume increases, the searches will take more time to complete. If you notice your installation becoming less responsive, check the performance of your scheduled searches in the **Jobs** or the **Status** dashboards in the **Search** app.

For a dashboard that will be constantly monitored, real-time queries are probably more efficient, particularly if multiple people will be using the dashboard. New in Splunk 4.3, real-time queries are first backfilled. For instance, a real-time query watching 24 hours will first run a query against the previous 24 hours and then add new events to the results as they appear. This feature makes real-time queries over fairly long periods practical and useful.

Editing the XML directly

First let me take a moment to tip my hat to Splunk for the new dashboard editor in Splunk 4.3. There are only of a couple of reasons why you would still need to edit simplified XML dashboards: forms and post-processing data. I predict that these reasons will go away in the future as more features are added to the dashboard editor.



The documentation for simplified XML panels can be found by searching `splunk.com` for **Panel reference for simple XML**.

UI Examples app

Before digging into the XML behind dashboards, it's a very good idea to install the app *Splunk UI examples app for 4.1+*, available from Splunkbase (see *Chapter 7, Working with Apps*, for information about Splunkbase). The examples provided in this app give a good overview of the features available in both simplified XML and advanced XML dashboards.

The simplest way to find this app is by searching for `examples` in **App | Find more apps....**

Building forms

Forms allow you to make a template that needs one or more pieces of information supplied to run. You can build these directly using raw XML, but I find it simpler to build a simple dashboard and then modify the XML accordingly. The other option is to copy an example, like those found in the *UI Examples app* (see the *UI Examples app* section, earlier in this chapter). We will touch on a simple use case in the following section.

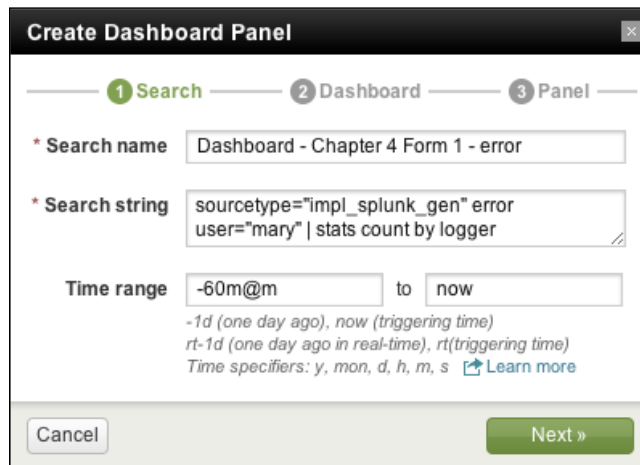
Creating a form from a dashboard

First, let's think of a use case. How about a form that tells us about errors for a particular user? Let's start with a report for a particular user, our friend `mary`:

```
sourcetype="impl_splunk_gen" error user="mary"
| stats count by logger
```

Now let's create a simple dashboard using this query:

1. Quickly create a simple dashboard using the wizard interface that we used before, by selecting **Create | Dashboard Panel**.



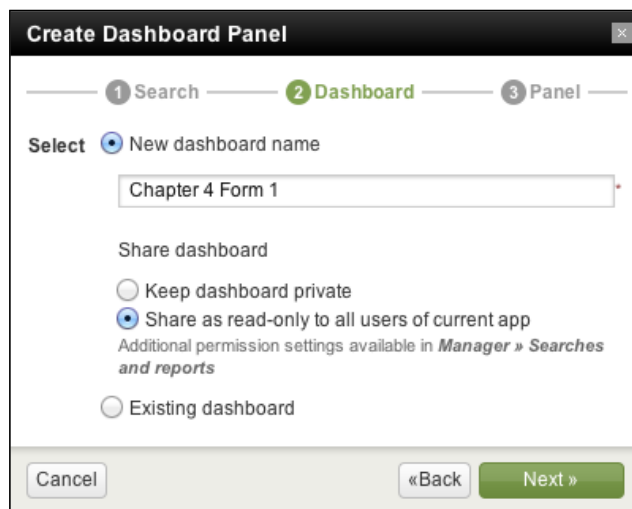
The screenshot shows the 'Create Dashboard Panel' wizard in its first step, 'Search'. The progress bar at the top indicates '1 Search', '2 Dashboard', and '3 Panel'. The form contains the following fields:

- Search name:** Dashboard - Chapter 4 Form 1 - error
- Search string:** sourcetype="impl_splunk_gen" error user="mary" | stats count by logger
- Time range:** -60m@m to now

Below the time range field, there is a legend: *-1d (one day ago), now (triggering time)*, *rt-1d (one day ago in real-time), rt(triggering time)*, and *Time specifiers: y, mon, d, h, m, s* with a [Learn more](#) link.

At the bottom, there are 'Cancel' and 'Next »' buttons.

2. Select a destination for our new panel. In this case, we are making a new dashboard.



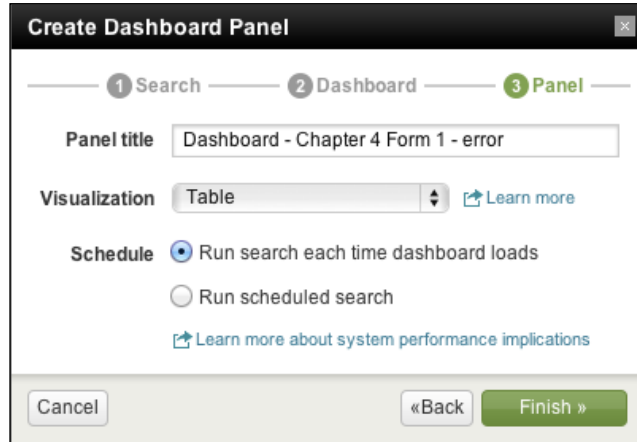
The screenshot shows the 'Create Dashboard Panel' wizard in its second step, 'Dashboard'. The progress bar at the top indicates '1 Search', '2 Dashboard', and '3 Panel'. The form contains the following options:

- Select:** New dashboard name
- Chapter 4 Form 1** (text input field)
- Share dashboard:**
 - Keep dashboard private
 - Share as read-only to all users of current app

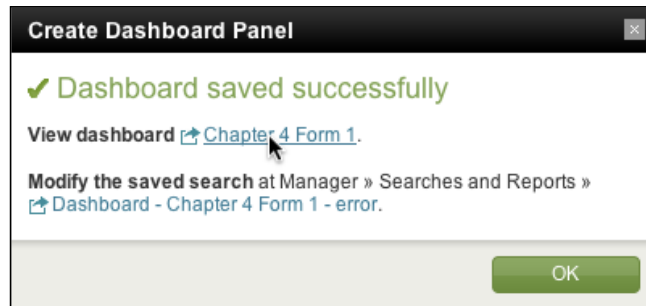
*Additional permission settings available in **Manager » Searches and reports***
- Existing dashboard

At the bottom, there are 'Cancel', «Back', and 'Next » buttons.

3. Select **Table** and give our panel a title.



4. On the final window, click on the title next to **View dashboard**.

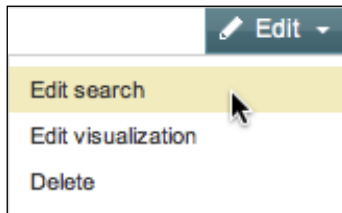


Let's look at the XML for our new dashboard. Click on the **On** button near the **Edit** label, then on **Edit XML**. The XML for our dashboard looks like this:

```
<?xml version='1.0' encoding='utf-8'?>
<dashboard>
  <label>Chapter 4 Form 1</label>
  <row>
    <table>
      <searchName>Dashboard - Chapter 4 Form 1 - error</searchName>
      <title>Dashboard - Chapter 4 Form 1 - error</title>
    </table>
  </row>
</dashboard>
```

That's pretty simple. To convert this dashboard into a form, we have to do the following things:

1. Searches need to be defined directly in the XML so that we can insert variables into the searches. We can use the editor itself to change the XML for us. Choose **Edit search** from the **Edit** menu on our table panel.



2. Then, click on **Edit as an inline search** followed by **Save**. This will convert the XML defining the query for us. The changes are highlighted.

```
<?xml version='1.0' encoding='utf-8'?>
<dashboard>
  <label>Chapter 4 Form 1</label>
  <row>
    <table>
      <searchString>
        sourcetype="impl_splunk_gen" error user="mary"
        | stats count by logger
      </searchString>
      <title>Dashboard - Chapter 4 Form 1 - error</title>
      <earliestTime>-60m@m</earliestTime>
      <latestTime>now</latestTime>
    </table>
  </row>
</dashboard>
```

3. Change `<dashboard>` to `<form>`. Don't forget the closing tag.

```
<form>
  <label>Chapter 4 Form 1</label>
  ...
  </row>
</form>
```


4. Create a `<fieldset>` tag with any form elements.

```
<form>
  <label>Chapter 4 Form 1</label>
  <fieldset>
    <input type="text" token="user">
      <label>User</label>
    </input>
  </fieldset>
</form>
```

5. Add appropriate variables in `<searchString>` to reflect the form values.

```
<searchString>
  sourcetype="impl_splunk_gen" error user="$user$"
  | stats count by logger
</searchString>
```

When we're through, our XML looks like this:

```
<?xml version='1.0' encoding='utf-8'?>
<form>

  <label>Chapter 4 Form 1</label>

  <fieldset>
    <input type="text" token="user">
      <label>User</label>
    </input>
  </fieldset>

  <row>
    <table>
      <searchString>
        sourcetype="impl_splunk_gen" error user="$user$"
        | stats count by logger</searchString>
      <title>Dashboard - Chapter 4 Form 1 - error</title>
      <earliestTime>-60m@m</earliestTime>
      <latestTime>now</latestTime>
    </table>
  </row>

</form>
```

Let's click on **Save** and then search for bobby.

	logger	count
1	AuthClass	45
2	BarClass	173
3	FooClass	41
4	LogoutClass	40

We now have a useful form for seeing errors by logger for a particular user.

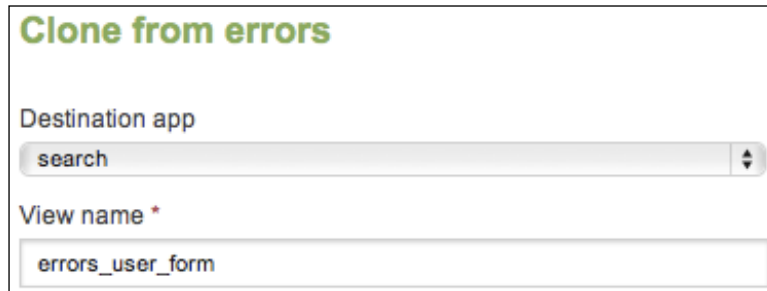
Driving multiple panels from one form

A single form can also be used to drive multiple panels at once. Let's convert a copy of the **Errors** dashboard that we created earlier in the chapter into a form:

1. Choose **Manage Views** from **Dashboards & Views**, or select **Manager | User interface | Views**.
2. To make a copy, click on **Clone** on the same row as **errors**.

View name	Owner	App	Sharing	Status	Actions
errors	admin	search	App Permissions	Enabled	Open Clone Move Delete
form_1	admin	search	App Permissions	Enabled	Open Clone Move Delete

3. In the editor that appears next, the value of **View name** will actually be used as the filename and URL, so it must not contain spaces or special characters. Let's call it `errors_user_form`.



The screenshot shows a web interface for cloning a dashboard. At the top, it says "Clone from errors" in green. Below that, there are two input fields. The first is labeled "Destination app" and has a dropdown menu with "search" selected. The second is labeled "View name *" and has the text "errors_user_form" entered.

4. The name in the menu comes from the `label` tag inside the dashboard's XML. Let's change that to `Errors User Form`:

```
<label>Errors User Form</label>
```
5. Save the new dashboard and click on **Open** next to the dashboard.
6. Next, convert all of the searches to inline using **Edit | Edit search | Edit as inline search**, as we did in the previous example.
7. Change `<dashboard>` to `<form>` and add the same `<fieldset>` block as before.
8. Insert `user="$user$"` into each `<searchString>` tag appropriately.

The XML in the end will be much larger than what we saw before, but hopefully still understandable. Lines changed manually are highlighted in the following code snippet:

```
<?xml version='1.0' encoding='utf-8'?>
<form>

  <label>Errors User Form</label>

  <fieldset>
    <input type="text" token="user">
      <label>User</label>
    </input>
  </fieldset>

</form>

<row>
  <chart>
    <searchString>
      sourcetype="impl_splunk_gen" loglevel=error user="$user$"
      | timechart count as "Error count" by network
    </searchString>
  </chart>
</row>
```

```
</searchString>
<title>Dashboard - Errors - errors by network timechart</title>
<earliestTime>-4h@h</earliestTime>
<latestTime>now</latestTime>
<option name="charting.chart">line</option>
</chart>
</row>

<row>
  <chart>
    <searchString>
      sourcetype="impl_splunk_gen" loglevel=error user="$user$"
      | bucket bins=10 req_time | stats count by req_time
    </searchString>
    <title>Error count by req_times</title>
    <earliestTime>-4h@h</earliestTime>
    <latestTime>now</latestTime>
    <option name="charting.chart">pie</option>
  </chart>
  <chart>
    <searchString>
      sourcetype="impl_splunk_gen" loglevel=error user="$user$"
      | stats count by logger
    </searchString>
    <title>Errors by logger</title>
    <earliestTime>-4h@h</earliestTime>
    <latestTime>now</latestTime>
    <option name="charting.chart">pie</option>
  </chart>
  <chart>
    <searchString>
      sourcetype="impl_splunk_gen" loglevel=error user="$user$"
      | stats count by user
    </searchString>
    <title>Errors by user</title>
    <earliestTime>-4h@h</earliestTime>
    <latestTime>now</latestTime>
    <option name="charting.chart">pie</option>
  </chart>
</row>

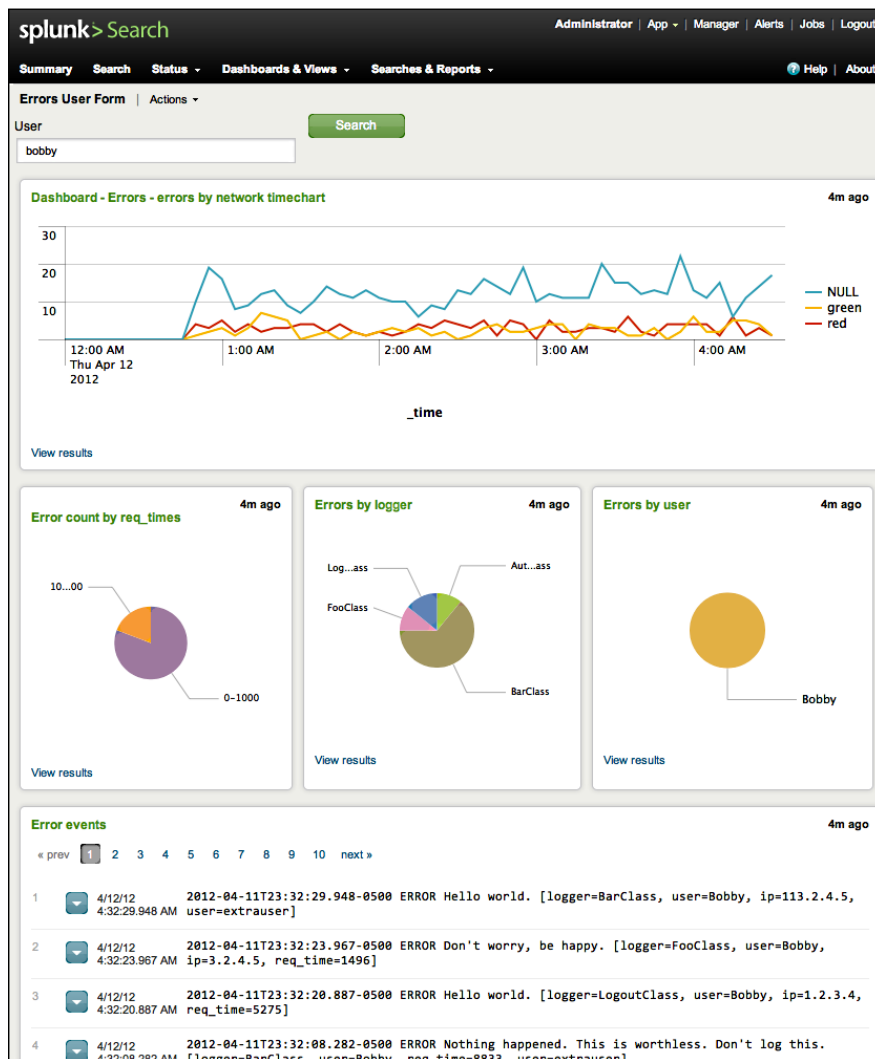
<row>
  <event>
    <searchString>
      sourcetype="impl_splunk_gen" loglevel=error user="$user$"
    </searchString>
    <title>Error events</title>
    <earliestTime>-4h@h</earliestTime>
    <latestTime>now</latestTime>
    <option name="count">10</option>
  </event>
</row>
```

Simple XML Dashboards

```
<option name="displayRowNumbers">true</option>
<option name="maxLines">10</option>
<option name="segmentation">outer</option>
<option name="softWrap">true</option>
</event>
</row>
```

</form>

After clicking on **Save**, we should be back at the dashboard, which is now a form. Searching for bobby renders this:



Let's make a few more changes:

1. Remove the **Errors by user** pie chart.
2. Add a time input to <fieldset>.
3. Remove the earliest and latest times from the queries. If a panel has time specified, it will always take precedence over the time field specified in <fieldset>.

Our XML now looks like this:

```
<?xml version='1.0' encoding='utf-8'?>
<form>

  <label>Errors User Form</label>

  <fieldset>
    <input type="text" token="user">
      <label>User</label>
    </input>
    <input type="time" />
  </fieldset>

  <row>
    <chart>
      <searchString>
        sourcetype="impl_splunk_gen" loglevel=error user="$user$"
        | timechart count as "Error count" by network
      </searchString>
      <title>Dashboard - Errors - errors by network timechart</title>
      <!-- remove time specifier -->
      <option name="charting.chart">line</option>
    </chart>
  </row>

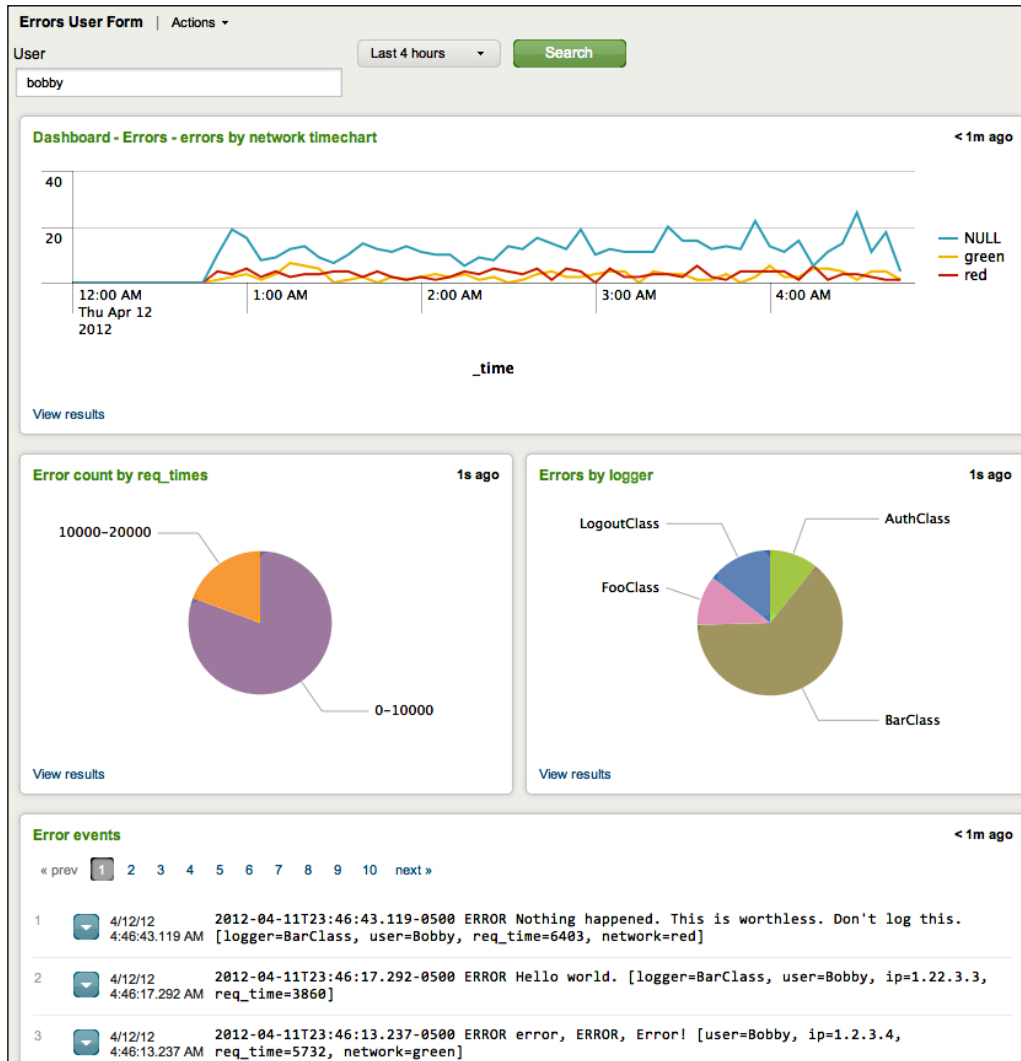
  <row>
    <chart>
      <searchString>
        sourcetype="impl_splunk_gen" loglevel=error user="$user$"
        | bucket bins=10 req_time | stats count by req_time
      </searchString>
      <title>Error count by req_times</title>
      <!-- remove time specifier -->
      <option name="charting.chart">pie</option>
    </chart>
  </row>
</form>
```

```
<chart>
  <searchString>
    sourcetype="impl_splunk_gen" loglevel=error user="$user$"
    | stats count by logger
  </searchString>
  <title>Errors by logger</title>
  <!-- remove time specifier -->
  <option name="charting.chart">pie</option>
</chart>
<!-- errors by user removed -->
</row>

<row>
  <event>
    <searchString>
      sourcetype="impl_splunk_gen" loglevel=error user="$user$"
    </searchString>
    <title>Error events</title>
    <!-- remove time specifier -->
    <option name="count">10</option>
    <option name="displayRowNumbers">true</option>
    <option name="maxLines">10</option>
    <option name="segmentation">outer</option>
    <option name="softWrap">true</option>
  </event>
</row>

</form>
```

Our dashboard now looks like this:



There are several other form elements available, with many options to customize their behavior. To find the official documentation, search `splunk.com` for Build and edit forms with simple XML.

There are also many useful examples in the documentation and in the *UI Examples* app (see the *UI Examples app* section, earlier in this chapter).

Post-processing search results

You may have noticed that, in our previous example, all of our queries started with the same actual query:

```
sourcetype="impl_splunk_gen" loglevel=error user="$user$"
```

It is of course wasteful to run the same query four times. Using `<searchPostProcess>`, we can run the query once and then run commands on the results for each panel.

The first step is to move the initial query out of the panel to the top level of the XML. The results from `<searchTemplate>` will be used by a panel if it has no query of its own or will be used as the source for `<searchPostProcess>`.

One additional piece of information is needed – the fields that are needed by the panels. We can get this by using `table`, like so:

```
<?xml version='1.0' encoding='utf-8'?>
<form>
  <searchTemplate>
    sourcetype="impl_splunk_gen" loglevel=error user="$user$"
    | table _time _raw network req_time logger
  </searchTemplate>
```

`table` mandates what fields will be passed from this query. `_time` is needed by the `timechart` command. `_raw` is used by the events listing panel at the bottom. `network`, `req_time`, and `logger` are used in the `by` clauses of each panel, respectively.

Let's edit our dashboard XML accordingly.

```
<?xml version='1.0' encoding='utf-8'?>
<form>

  <label>Errors User Form PostProcess</label>

  <searchTemplate>
```

```
    sourcetype="impl_splunk_gen" loglevel=error user="$user$"
    | table _time _raw network req_time logger
</searchTemplate>

<fieldset>
  <input type="text" token="user">
    <label>User</label>
  </input>
  <input type="time" />
</fieldset>

<row>
  <chart>
    <searchPostProcess>
      timechart count as "Error count" by network
    </searchPostProcess>
    <title>Dashboard - Errors - errors by network timechart</title>
    <option name="charting.chart">line</option>
  </chart>
</row>

<row>
  <chart>
    <searchPostProcess>
      bucket bins=10 req_time | stats count by req_time
    </searchPostProcess>
    <title>Error count by req_times</title>
    <option name="charting.chart">pie</option>
  </chart>
  <chart>
    <searchPostProcess>
      stats count by logger
    </searchPostProcess>
    <title>Errors by logger</title>
    <option name="charting.chart">pie</option>
  </chart>
</row>

<row>
  <event>
    <!-- remove searchString and use the events from searchTemplate
-->
    <title>Error events</title>
    <option name="count">10</option>
```

```
        <option name="displayRowNumbers">true</option>
        <option name="maxLines">10</option>
        <option name="segmentation">outer</option>
        <option name="softWrap">true</option>
    </event>
</row>

</form>
```

This will work exactly like our previous example but will only run the query once, drawing more quickly, and saving resources for everyone.

Post-processing limitations

When using `<searchPostProcess>`, there is one big limitation and several smaller limitations that often mandate a little extra work:

1. Only the first 10,000 results are passed from a raw query. To deal with this, it is necessary to run events through `stats`, `timechart`, or `table`. Transforming commands such as `stats` will reduce the number of rows produced by the initial query, increasing the performance.
2. Only fields referenced specifically are passed from the original events. This can be dealt with by using `table` (as we did in the previous example) or by aggregating results into fewer rows with `stats`.
3. `<searchPostProcess>` elements cannot use form values. If you need the values of form elements, you need to hand them along from the initial query.
4. Panels cannot use form values in a `<searchString>` element if they are referenced in the top level `<searchTemplate>` element. This can be accomplished in advanced XML, which we will cover in *Chapter 8, Building Advanced Dashboards*.

The first limitation is the most common item to affect users. The usual solution is to pre-aggregate the events into a superset of what is needed by the panels. To accomplish this, our first task is to look at the queries and figure out what fields need to be handed along for all queries to work.

Panel 1

Our first chart applies this post-processing:

```
timechart count as "Error count" by network
```

For this query to work, we need `_time`, `count` and `network`. Since `_time` is the actual time of the event, we need to group the times to reduce the number of rows produced by `stats`. We can use `bucket` for this task. Our initial query will now look like this:

```
sourcetype="impl_splunk_gen" loglevel=error user="$user$"
| bucket span=1h _time
| stats count by network _time
```

This query will produce results such as those shown in the following screenshot:

	<code>_time</code> ↕	<code>network</code> ↕	<code>count</code> ↕
1	4/16/12 2:00:00.000 AM	green	68
2	4/16/12 3:00:00.000 AM	green	55
3	4/16/12 4:00:00.000 AM	green	60
4	4/16/12 5:00:00.000 AM	green	77
5	4/16/12 6:00:00.000 AM	green	54
6	4/16/12 7:00:00.000 AM	green	72
7	4/16/12 8:00:00.000 AM	green	70
8	4/16/12 9:00:00.000 AM	green	69
9	4/16/12 10:00:00.000 AM	green	62
10	4/16/12 11:00:00.000 AM	green	69

To actually use these results in our panel, we need to modify the contents of `<searchPostProcess>` slightly. Since `count` expects to see raw events, the count will not be what we expect. We need instead to apply the `sum` function to the `count` field. We will also set the `span` value to match the span we used in the initial query:

```
timechart span=1h sum(count) as "Error count" by network
```

Panel 2

In the next panel, we currently have:

```
bucket bins=10 req_time | stats count by req_time
```

Since the `bucket` command needs to run against the raw events, we will add the command to the original query and also add `req_time` to `stats`:

```
sourcetype="impl_splunk_gen" loglevel=error user="$user$"
| bucket span=1h _time
| bucket bins=10 req_time
| stats count by network _time req_time
```

Our results will then look like this:

	_time ↕	network ↕	req_time ↕	count ↕
1	4/16/12 3:00:00.000 AM	green	0-10000	29
2	4/16/12 3:00:00.000 AM	green	10000-20000	8
3	4/16/12 4:00:00.000 AM	green	0-10000	32
4	4/16/12 4:00:00.000 AM	green	10000-20000	7
5	4/16/12 5:00:00.000 AM	green	0-10000	50
6	4/16/12 5:00:00.000 AM	green	10000-20000	6
7	4/16/12 6:00:00.000 AM	green	0-10000	26
8	4/16/12 6:00:00.000 AM	green	10000-20000	5
9	4/16/12 7:00:00.000 AM	green	0-10000	34
10	4/16/12 7:00:00.000 AM	green	10000-20000	10

The panel query then becomes:

```
stats sum(count) by req_time
```

Panel 3

The last panel that we can add is the simplest yet.

```
stats count by logger
```

We simply need to add `logger` to the end of our initial query.

```
sourcetype="impl_splunk_gen" loglevel=error user="$user$"
| bucket span=1h _time
| bucket bins=10 req_time
| stats count by network _time req_time logger
```

We will also need to replace `count` with `sum(count)`, thus:

```
stats sum(count) by logger
```

Final XML

What we have built is a query that produces a row for every combination of fields. You can avoid this work by using `table`, but doing this extra work to reduce the rows produced by the initial query can increase performance considerably.

After all of these changes, here is our final XML. The changed lines are highlighted:

```

<?xml version='1.0' encoding='utf-8'?>
<form>

  <label>Errors User Form PostProcess</label>

  <searchTemplate>
    sourcetype="impl_splunk_gen" loglevel=error user="$user$"
    | bucket span=1h _time
    | bucket bins=10 req_time
    | stats count by network _time req_time logger
  </searchTemplate>

  <fieldset>
    <input type="text" token="user">
      <label>User</label>
    </input>
    <input type="time" />
  </fieldset>

  <row>
    <chart>
      <searchPostProcess>
        timechart span=1h sum(count) as "Error count" by network
      </searchPostProcess>
      <title>Dashboard - Errors - errors by network timechart</title>
      <option name="charting.chart">line</option>
    </chart>
  </row>

  <row>
    <chart>
      <searchPostProcess>
        stats sum(count) by req_time
      </searchPostProcess>
      <title>Error count by req_times</title>
      <option name="charting.chart">pie</option>
    </chart>
    <chart>
      <searchPostProcess>
        stats sum(count) by logger
      </searchPostProcess>
      <title>Errors by logger</title>
      <option name="charting.chart">pie</option>
    </chart>
  </row>

  <!-- remove the event listing, as per limitation #4 -->
</form>

```

Summary

Once again, we have really only scratched the surface of what is possible, using simplified XML dashboards. I encourage you to dig into the examples in the *UI Examples* app (see the *UI Examples app* section, earlier in this chapter).

When you are ready to make additional customizations or use some of the cool modules available from Splunk and the community, you can use advanced XML features, which we will look at in *Chapter 8, Building Advanced Dashboards*.

In *Chapter 5, Advanced Search Examples*, we will dive into advanced search examples, which can be a lot of fun. We'll expose some really powerful features of the search language and go over a few tricks that I've learned over the years.

5

Advanced Search Examples

In this chapter, we will work through a few advanced search examples in great detail. The examples and data shown are fictitious, but hopefully will spark some ideas that you can apply to your own data. For a huge collection of examples and help topics, check out Splunk answers at <http://answers.splunk.com>.

Using subsearches to find loosely related events

The number of use cases for subsearches in the real world might be small, but for those situations where they can be applied, subsearches can be a magic bullet. Let's look at an example and then talk about some rules.

Subsearch

Let's start with these events:

```
2012-04-20 13:07:03 msgid=123456 from=mary@companyx.com
2012-04-20 13:07:04 msgid=654321 from=bobby@companyx.com
2012-04-20 13:07:05 msgid=123456 to=bob@vendor1.co.uk
2012-04-20 13:07:06 msgid=234567 from=mary@companyx.com
2012-04-20 13:07:07 msgid=234567 to=larry@vender3.org
2012-04-20 13:07:08 msgid=654321 to=bob@vendor2.co.uk
```

From these events, let's find out who mary has sent messages to. In these events, we see that the `from` and `to` values are in different entries. We could use `stats` to pull these events together, and then filter the resulting rows, like this:

```
sourcetype=mail to OR from
| stats values(from) as from values(to) as to by msgid
| search from=mary@companyx.com
```


The problem is that on a busy mail server, this search might retrieve millions of events and then throw most of the work away. We want to actually use the index efficiently, and a subsearch can help us do that.

Here is how we could tackle this with a subsearch:

```
[search sourcetype=mail from=mary@companyx.com | fields msgid]
sourcetype=mail to
```

Let's step through everything that's happening here:

1. The search inside the brackets is run:

```
sourcetype=mail from=mary@companyx.com
```

Given our sample events, this will locate two events:

```
2012-04-20 13:07:03 msgid=123456 from=mary@companyx.com
2012-04-20 13:07:06 msgid=234567 from=mary@companyx.com
```

2. `| fields msgid` then instructs the subsearch to only return the field `msgid`. Behind the scenes, the subsearch results are essentially added to the outer search as an OR statement, producing this search:

```
( (msgid=123456) OR (msgid=234567) ) sourcetype=mail to
```

This will be a much more efficient search, using the index effectively.

3. This new search returns the answer we're looking for:

```
2012-04-20 13:07:05 msgid=123456 to=bob@vender1.co.uk
2012-04-20 13:07:07 msgid=234567 to=larry@vender3.org
```

Subsearch caveats

To prevent a subsearch from being too expensive, they are limited by a time and event count:

- The default time limit for the subsearch to complete is 60 seconds. If the subsearch is still running at that point, the subsearch is finalized, and only the events located up to that point are added to the outer search.
- Likewise, the default event limit for the subsearch is 1,000. After this point, any further events will be truncated.

If either of these limits is reached, there is probably a better way to accomplish the task at hand.

Another consideration is that the fields returned from the subsearch must be searchable. There is a magical field called "search" that will be added to the query as a raw search term, but you have to do a little more work. See "search context" later in this chapter for an example.

Nested subsearches

Subsearches can also be nested, if needed. With mail server logs, it is sometimes necessary to find all the events related to a particular message. Some fictitious log entries are given, such as:

```
... in=123 msgid=123456 from=mary@companyx.com
... msgid=123456 out=987 subject=Important
... out=987 to=bob@vendor1.co.uk
```

We can see that the first event has the value of `from`, but there is no longer anything in common with the event that contains the `to` field. Luckily, there is an interim event that does contain `out`, and contains `msgid`, which we *do* have in the first event.

We can write a query like this to find our events:

```
[search sourcetype=mail out
  [search sourcetype=mail from=mary@companyx.com | fields msgid]
  | fields out]
sourcetype=mail to
```

Here are the parts of the search, numbered according to the order of execution:

1. [search sourcetype=mail from=mary@companyx.com | fields msgid]
2. [search sourcetype=mail out

| fields out]
3. sourcetype=mail to

Let's step through this example:

1. The innermost nested search (1) is run:

sourcetype=mail from=mary@companyx.com | fields msgid

2. This is attached to the next innermost search (2), like this:

```
sourcetype=mail out
  (msgid=123456)
  | fields out
```

3. The results of this search are attached to the outermost search (3), like this:

```
(out=987)
sourcetype=mail to
```

This is the final query, which returns the answer we are looking for:

```
... out=987 to=bob@vendor1.co.uk
```

Using transaction

The `transaction` command lets you group events based on their proximity to other events. This proximity is determined either by ranges of time, or by specifying the text contained in the first and/or last event in a transaction. This is an expensive process, but is sometimes the best way to group certain events. Unlike other transforming commands, when using `transaction`, the original events are maintained and instead are grouped together into multivalued events.

Some rules of thumb for the usage of `transaction` are as follows:

- If the question can be answered using `stats`, it will almost always be more efficient.
- All of the events needed for the transaction have to be found in one search.
- When grouping is based on field values, and all of the events need at least one field in common with at least one other event, then it can be considered as part of the transaction. This doesn't mean that every event must have the same field, but that all events should have some field from the list of fields specified.
- When grouping is based solely on `startswith` and `endswith`, it is important that transactions do not interleave in the search results.
- Every effort should be made to reduce the number of open transactions, as an inefficient query can use a lot of resources. This is controlled by limiting the scope of time with `maxspan` and `maxpause`, and/or by using `startswith` and `endswith`.

Let's step through a few possible examples of the `transaction` command in use.

Using transaction to determine the session length

Some fictitious events are given as follows. Assuming this is a busy server, there might be a huge number of events occurring between requests from this particular session:

```
2012-04-27T03:14:31 user=mary GET /foo?q=1 uid=abcdefg
```

...hundreds of events...

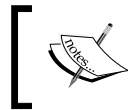
```
2012-04-27T03:14:46 user=mary GET /bar?q=2 uid=abcdefg
```

...hundreds of thousands of events...

```
2012-04-27T06:40:45 user=mary GET /foo?q=3 uid=abcdefg
```

...hundreds of events...

```
2012-04-27T06:41:49 user=mary GET /bar?q=4 uid=abcdefg
```



The definition of "huge" depends on the infrastructure that you have dedicated to Splunk. See *Chapter 11, Advanced Deployments*, for more information about sizing your installation, or contact Splunk support.

Let's build a query to see the transactions belonging to `mary`. We will consider a session complete when there have been no events for five minutes:

```
sourcetype="impl_splunk_web" user=mary
| transaction maxpause=5m user
```

Let's step through everything that's happening here:

1. The initial query is run, simply returning all events for the user `mary`:

```
sourcetype="impl_splunk_web" user=mary
```
2. `| transaction` starts the command.
3. `maxpause=5m` indicates that any transaction that has not seen an event for five minutes will be closed. On a large dataset, this time frame might be too expensive, leaving a huge number of transactions open longer than necessary.
4. `user` is the field to use to link events together. If events have different values of `user`, a new transaction will start with the new value of `user`.

Given our events, we will end up with two groupings of events:

1		4/30/12 5:39:49.000 PM	2012-04-30T22:39:49 user=mary GET /foo?q=3 uid=abcdefg 2012-04-30T22:40:47 user=mary GET /bar?q=4 uid=abcdefg sourcetype=impl_splunk_web duration=58 eventcount=2
2		4/30/12 2:13:43.000 PM	2012-04-30T19:13:43 user=mary GET /foo?q=1 uid=abcdefg 2012-04-30T19:13:45 user=mary GET /bar?q=2 uid=abcdefg sourcetype=impl_splunk_web duration=2 eventcount=2

Each of these groupings can then be treated like a single event.

A transaction command has some interesting properties as follows:

- The field `_time` is assigned the value of `_time` from the first event in the transaction.
- The field `duration` contains the time difference between the first and last event in the transaction.
- The field `eventcount` contains the number of events in the transaction.
- All fields are merged into unique sets. In this case, the field `user` would only ever contain `mary`, but the field `q` would contain the values `[1, 2]`, and `[3, 4]` respectively.

With these extra fields, we can render a nice table of transactions belonging to `mary` like this:

```
sourcetype="impl_splunk_web" user=mary
| transaction maxpause=5m user
| table _time duration eventcount q
```

This will produce a table like this:

	<code>_time</code> ↕	<code>duration</code> ↕	<code>eventcount</code> ↕	<code>q</code> ↕
1	4/30/12 5:39:49.000 PM	58	2	3 4
2	4/30/12 2:13:43.000 PM	2	2	1 2

Combining `transaction` with `stats` or `timechart`, we can generate statistics about the transactions themselves:

```
sourcetype="impl_splunk_web" user=mary
| transaction maxpause=5m user
| stats avg(duration) avg(eventcount)
```

This would give us a table, as shown in the following screenshot:

	avg(duration) ↕	avg(eventcount) ↕
1	30.000000	2.000000

Calculating the aggregate of transaction statistics

Using the values added by `transaction`, we can somewhat naively answer the questions of how long the users spend on a site and how many pages they view per session.

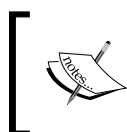
Let's create sessions based on the `uid` field for all events. Using `stats`, we will then calculate the average `duration` value, the average `eventcount` value, and while we're at it, we will determine the distinct number of users and session IDs.

```
sourcetype="impl_splunk_web"
| transaction maxpause=5m uid
| stats avg(duration) avg(eventcount) dc(user) dc(uid)
```

This will give us a table as shown in the following screenshot:

	avg(duration) ↕	avg(eventcount) ↕	dc(user) ↕	dc(uid) ↕
1	892.857143	227.553571	5	55

Transactions have an average length of 892 seconds, and 227 events.



For large amounts of web traffic, you will want to calculate transactions over small slices of time into a summary index. We will cover summary indexes in *Chapter 9, Using Summary Indexes*.

Combining subsearches with transaction

Let's put what we learned about subsearches together with transactions. Let's imagine that `q=1` represents a particular entry point into our site, perhaps a link from an advertisement. We can use `subsearch` to find users that clicked on the advertisement, then use `transaction` to determine how long these users stayed on our site.

To do this, first we need to locate the sessions initiated from this link. The search can be as simple as:

```
sourcetype="impl_splunk_web" q=1
```

This will return events like:

```
2012-04-27T07:13:19 user=user1 GET /foo?q=1 uid=NDQ5NjIzNw
```

In our fictitious logs, the field `uid` represents a session ID. Let's use `stats` to return one row per unique `uid`:

```
sourcetype="impl_splunk_web" q=1
| stats count by uid
```

This will render a table like this (the first 10 rows are shown):

	uid ↕	count ↕
1	MTA4NDI5Nw	2
2	MTAxNzE4NA	10
3	MTE3MDE0NQ	3
4	MTE3MjM5NQ	1
5	MTI4OTc0Ng	1
6	MTM3NTlyNg	2
7	MTM3NjA1Ng	2
8	MTQ4MTEzNQ	3
9	MTY4NzYwMQ	3
10	MTcwNzlyNw	1

We need to add one more command, `fields`, to limit the fields that come out of our subsearch:

```
sourcetype="impl_splunk_web" q=1
| stats count by uid
| fields uid
```

Now we feed this back to our outer search:

```
[search
  sourcetype="impl_splunk_web" q=1
  | stats count by uid
  | fields uid
]
sourcetype="impl_splunk_web"
```

After the subsearch runs, the combined query is essentially as follows:

```
( (uid=MTAyMjQ2OA) OR (uid=MTI2NzEzNg) OR (uid=MTM0MjQ3NA) )
sourcetype="impl_splunk_web"
```

From this combined query, we now have every event from every uid that clicked a link that contained q=1 in our time frame. We can now add transaction as we saw earlier to combine these sessions into groups:

```
[search sourcetype="impl_splunk_web" q=1
  | stats count by uid
  | fields uid]
sourcetype="impl_splunk_web"
| transaction maxpause=5m uid
```


This gives us the following transactions:

1	4/30/12 5:40:59.000 PM	2012-04-30T22:40:59 user=user1 GET /foo?q=10 uid=Mzg2MzE3NQ 2012-04-30T22:41:06 user=user1 GET /foo?q=1 uid=Mzg2MzE3NQ 2012-04-30T22:41:20 user=user1 GET /foo?q=9 uid=Mzg2MzE3NQ 2012-04-30T22:41:21 user=user1 GET /foo?q=14 uid=Mzg2MzE3NQ 2012-04-30T22:41:28 user=user1 GET /foo?q=10 uid=Mzg2MzE3NQ 2012-04-30T22:41:38 user=user1 GET /foo?q=11 uid=Mzg2MzE3NQ 2012-04-30T22:41:40 user=user1 GET /foo?q=5 uid=Mzg2MzE3NQ 2012-04-30T22:41:49 user=user1 GET /foo?q=8 uid=Mzg2MzE3NQ 2012-04-30T22:41:50 user=user1 GET /foo?q=11 uid=Mzg2MzE3NQ 2012-04-30T22:41:53 user=user1 GET /foo?q=3 uid=Mzg2MzE3NQ
Show all 16 lines		
sourcetype=impl_splunk_web duration=79 eventcount=16		
2	4/30/12 5:39:49.000 PM	2012-04-30T22:39:49 user=mary GET /foo?q=3 uid=abcdefg 2012-04-30T22:40:47 user=mary GET /bar?q=4 uid=abcdefg
Show all 2 lines		
sourcetype=impl_splunk_web duration=58 eventcount=2		
3	4/30/12 5:38:13.000 PM	2012-04-30T22:38:13 user=user3 GET /foo?q=1 uid=NDI0NzUzMg 2012-04-30T22:38:19 user=user3 GET /foo?q=12 uid=NDI0NzUzMg 2012-04-30T22:38:19 user=user3 GET /foo?q=3 uid=NDI0NzUzMg 2012-04-30T22:38:22 user=user3 GET /foo?q=3 uid=NDI0NzUzMg 2012-04-30T22:38:25 user=user3 GET /foo?q=6 uid=NDI0NzUzMg 2012-04-30T22:38:29 user=user3 GET /foo?q=13 uid=NDI0NzUzMg 2012-04-30T22:38:31 user=user3 GET /foo?q=11 uid=NDI0NzUzMg 2012-04-30T22:38:37 user=user3 GET /foo?q=9 uid=NDI0NzUzMg 2012-04-30T22:38:39 user=user3 GET /foo?q=9 uid=NDI0NzUzMg 2012-04-30T22:38:40 user=user3 GET /foo?q=3 uid=NDI0NzUzMg
Show all 60 lines		
sourcetype=impl_splunk_web duration=255 eventcount=60		

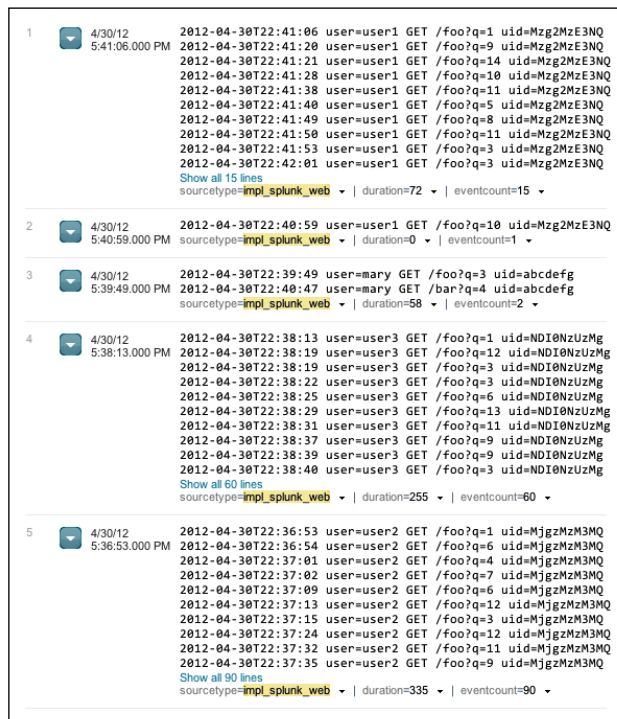
Notice that not all of our transactions start with $q=1$. This means that this transaction did not start when the user clicked the advertisement. Let's make sure our transactions start from the desired entry point of $q=1$:

```
[search sourcetype="impl_splunk_web" q=1
| stats count by uid
| fields uid]
sourcetype="impl_splunk_web"
| transaction maxpause=5m
  startswith="q=1"
  uid
```

startswith indicates that a new transaction should start at the time the search term $q=1$ is found in an event.

 startswith only works on the field `_raw` (the actual event text). In this case, `startswith="q=1"`, is looking for the literal phrase "q=1", not the field `q`.

This will cause any occurrence of $q=1$ to start a new transaction. We still have a few transactions that do not contain $q=1$, which we will eliminate next.



Transaction ID	Time Range	Events
1	4/30/12 5:41:06.000 PM - 5:41:06.000 PM	15 events (all start with q=1)
2	4/30/12 5:40:59.000 PM - 5:40:59.000 PM	1 event (starts with q=10)
3	4/30/12 5:39:49.000 PM - 5:39:49.000 PM	2 events (start with q=3 and q=4)
4	4/30/12 5:38:13.000 PM - 5:38:13.000 PM	60 events (all start with q=1)
5	4/30/12 5:36:53.000 PM - 5:36:53.000 PM	90 events (all start with q=1)

To discard the transactions that do not contain `q=1`, add a search command:

```
[search sourcetype="impl_splunk_web" q=1
 | stats count by uid
 | fields uid]
sourcetype="impl_splunk_web"
 | transaction maxpause=5m startswith="q=1" uid
 | search q=1
```

Finally, let's add `stats` to count the number of transactions, the distinct values of `uid`, the average duration of each transaction, and the average number of clicks per transaction:

```
[search sourcetype="impl_splunk_web" q=1
 | stats count by uid
 | fields uid]
sourcetype="impl_splunk_web"
 | transaction maxpause=5m startswith="q=1" uid
 | search q=1
 | stats count dc(uid) avg(duration) avg(eventcount)
```

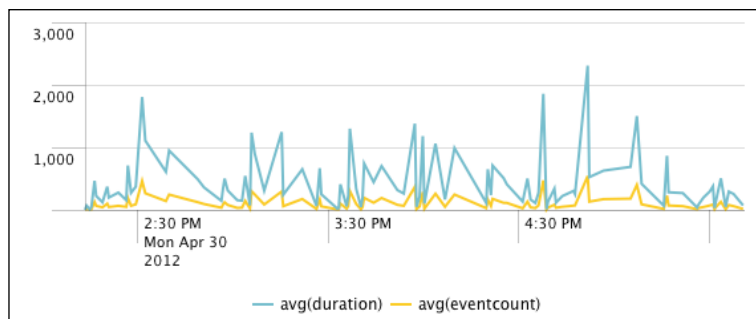
This gives us a table as shown in the following screenshot:

count ↕	dc(uid) ↕	avg(duration) ↕	avg(eventcount) ↕
118	54	409.254237	103.440678

We can swap `timechart` with `stats` to see how these statistics change over time:

```
[search sourcetype="impl_splunk_web" q=1
 | stats count by uid
 | fields uid]
sourcetype="impl_splunk_web"
 | transaction maxpause=5m startswith="q=1" uid
 | search q=1
 | timechart bins=500 avg(duration) avg(eventcount)
```

This produces a graph as shown in the following screenshot:



Determining concurrency

Determining the number of users currently using a system is difficult, particularly if the log does not contain events for both the beginning and end of a transaction. With web server logs in particular, it is not quite possible to know when a user has left a site. Let's investigate a couple of strategies for answering this question.

Using transaction with concurrency

If the question you are trying to answer is "how many transactions were happening at a time?", you can use `transaction` to combine related events and calculate the duration of each transaction. We will then use the `concurrency` command to increase a counter when the events start, and decrease when the time has expired for each transaction. Let's start with our searches from the previous section:

```
sourcetype="impl_splunk_web"
| transaction maxpause=5m uid
```

This will return a transaction for every `uid`, assuming that if no requests were made for five minutes, the session is complete. This provides results as shown in the following screenshot:

Transaction ID	Start Time	End Time	User	Method	URL	UID
1	11/11/12 1:27:41.000 PM	2012-11-11T19:27:41	user=user1	GET	/products/x/?q=6018685	uid=MTA0NjI1Ng
		2012-11-11T19:27:45	user=user1	GET	/bar?q=4533574	uid=MTA0NjI1Ng
		2012-11-11T19:27:49	user=user1	GET	/products/y/?q=4013720	uid=MTA0NjI1Ng
		2012-11-11T19:27:52	user=user1	GET	/about/?q=7566917	uid=MTA0NjI1Ng
		2012-11-11T19:27:58	user=user1	GET	/products/index.html?q=7507807	uid=MTA0NjI1Ng
		2012-11-11T19:27:59	user=user1	GET	/products/index.html?q=3427879	uid=MTA0NjI1Ng
		2012-11-11T19:28:03	user=user1	GET	/products/?q=14744799	uid=MTA0NjI1Ng
		2012-11-11T19:28:04	user=user1	GET	/contact/?q=8369940	uid=MTA0NjI1Ng
		2012-11-11T19:28:08	user=user1	GET	/products/index.html?q=1038064	uid=MTA0NjI1Ng
2	11/11/12 1:26:33.000 PM	2012-11-11T19:26:33	user=mary	GET	/products/?q=3	uid=abcdefg
		2012-11-11T19:27:35	user=mary	GET	/bar?q=4	uid=abcdefg
3	11/11/12 1:22:34.000 PM	2012-11-11T19:22:34	user=user1	GET	/products/y/?q=11925408	uid=NDU0NTI2Ng
		2012-11-11T19:22:36	user=user1	GET	/products/y/?q=1	uid=NDU0NTI2Ng
		2012-11-11T19:22:46	user=user1	GET	/products/y/?q=5262471	uid=NDU0NTI2Ng
		2012-11-11T19:22:50	user=user1	GET	/products/?q=13404385	uid=NDU0NTI2Ng
		2012-11-11T19:23:09	user=user1	GET	/contact/?q=12356362	uid=NDU0NTI2Ng
		2012-11-11T19:23:10	user=user1	GET	/foo?q=5435855	uid=NDU0NTI2Ng
		2012-11-11T19:23:14	user=user1	GET	/contact/?q=7668322	uid=NDU0NTI2Ng
		2012-11-11T19:23:19	user=user1	GET	/products/x/?q=5561275	uid=NDU0NTI2Ng
		2012-11-11T19:23:22	user=user1	GET	/contact/?q=13199295	uid=NDU0NTI2Ng
		2012-11-11T19:23:25	user=user1	GET	/products/x/?q=14941070	uid=NDU0NTI2Ng

By simply adding the `concurrency` command, we can determine the overlap of these transactions, and find out how many transactions were occurring at a time. Let's also add the `table` and `sort` commands to create a table:

```
sourcetype="impl_splunk_web"
| transaction maxpause=5m uid
```

```

| concurrency duration=duration
| table _time concurrency duration eventcount
| sort _time

```

This produces a table as follows:

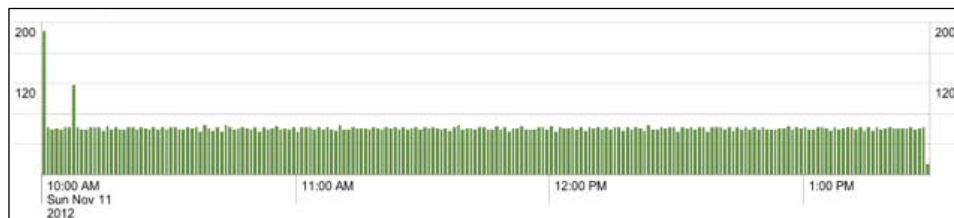
	<u>_time</u> ↕	<u>concurrency</u> ↕	<u>duration</u> ↕	<u>eventcount</u> ↕
1	11/11/12 10:00:24.000 AM	1	1524	360
2	11/11/12 10:00:24.000 AM	2	130	60
3	11/11/12 10:00:24.000 AM	3	507	148
4	11/11/12 10:00:25.000 AM	4	690	187
5	11/11/12 10:00:25.000 AM	5	2033	439
6	11/11/12 10:00:27.000 AM	6	0	2
7	11/11/12 10:02:33.000 AM	6	966	194
8	11/11/12 10:08:56.000 AM	5	1768	365
9	11/11/12 10:11:54.000 AM	6	649	140
10	11/11/12 10:18:38.000 AM	6	1229	243
11	11/11/12 10:22:52.000 AM	5	163	39

From these results, we can see that as transactions begin, concurrency increases and then levels off as transactions expire. In our sample data, the highest value of **concurrency** we see is 6.

Using concurrency to estimate server load

In the previous example, the number of concurrent sessions was quite low, since each transaction is counted as one event, no matter how many events make up that transaction. While this provides an accurate picture of the number of concurrent transactions, it doesn't necessarily provide a clear picture of server load.

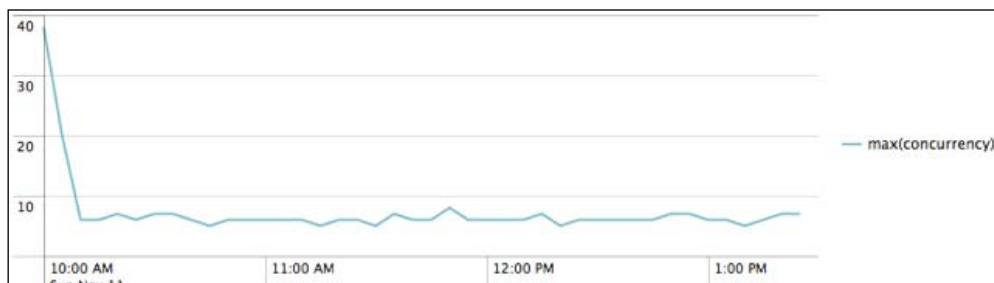
Looking at the timeline of our events, we see a large spike of events at the beginning of our log. This did not affect the previous example, because most of these events belong to a single user session.



Some web logs provide the time it took to serve a request. Our log does not have this duration, so we'll use `eval` to simulate a value for `duration` per request:

```
sourcetype="impl_splunk_web"
| eval duration=2
| concurrency duration=duration
| timechart max(concurrency)
```

Here we have set the duration of each request to 2 seconds. `concurrency` will use the value of `duration`, treating each event as if it were a 2-second long transaction. The timechart looks like this:



As we can see, in our sample data, the large spike of requests at the beginning of our log translates to high concurrency.

Later in this chapter, we will calculate events per some period of time, which will provide a very similar answer more efficiently, but it's not quite the same answer, as the count will be by fixed slices of time instead of a running total that changes with each event.

Calculating concurrency with a `by` clause

One limitation of the `concurrency` command is that there is no way to simultaneously calculate concurrency for multiple sets of data. For instance, what if you wanted to know the concurrency *per host*, as opposed to concurrency across your entire environment?

In our sample set of data, we only have one `host`, but we have multiple values for the `network` field. Let's use that field for our exercise.

Our fake concurrency example from the previous example looks like this:

```
sourcetype=impl_splunk_gen network="*"
| eval d=2
| concurrency duration=d
| timechart max(concurrency)
```

First, let's rebuild this search using the `streamstats` command. This command will calculate rolling statistics and attach the calculated values to the events themselves.

To accommodate `streamstats`, we will need an event representing the start and end of each transaction. We can accomplish this by creating a multivalued field, essentially an array, and then duplicate our events based on the values in this field.

First, let's create our end time. Remember that `_time` is simply the UTC epoch time at which this event happened, so we can treat it as a number.

```
sourcetype=impl_splunk_gen network="*"
| eval endtime=_time+2
```

Piping that through `table _time network endtime`, we see:

	<code>_time</code> ↕	<code>network</code> ↕	<code>endtime</code> ↕
1	11/30/12 1:23:28.046 PM	qa	1354303410.046
2	11/30/12 1:23:25.869 PM	qa	1354303407.869
3	11/30/12 1:23:25.057 PM	qa	1354303407.057
4	11/30/12 1:23:22.736 PM	qa	1354303404.736
5	11/30/12 1:23:20.944 PM	prod	1354303402.944
6	11/30/12 1:23:19.729 PM	prod	1354303401.729
7	11/30/12 1:23:16.351 PM	qa	1354303398.351
8	11/30/12 1:23:15.544 PM	qa	1354303397.544
9	11/30/12 1:23:13.553 PM	prod	1354303395.553
10	11/30/12 1:23:11.155 PM	prod	1354303393.155

Next, we want to combine `_time` and our `endtime` into a multivalued field, which we will call `t`:

```
sourcetype=impl_splunk_gen network="*"
| eval endtime=_time+2
| eval t=mvappend(_time, endtime)
```

Piping that through `table _time network t`, we see:

	<code>_time</code>	<code>network</code>	<code>t</code>
1	11/30/12 1:23:28.046 PM	qa	1354303408.046 1354303410.046
2	11/30/12 1:23:25.869 PM	qa	1354303405.869 1354303407.869
3	11/30/12 1:23:25.057 PM	qa	1354303405.057 1354303407.057
4	11/30/12 1:23:22.736 PM	qa	1354303402.736 1354303404.736
5	11/30/12 1:23:20.944 PM	prod	1354303400.944 1354303402.944
6	11/30/12 1:23:19.729 PM	prod	1354303399.729 1354303401.729
7	11/30/12 1:23:16.351 PM	qa	1354303396.351 1354303398.351
8	11/30/12 1:23:15.544 PM	qa	1354303395.544 1354303397.544
9	11/30/12 1:23:13.553 PM	prod	1354303393.553 1354303395.553
10	11/30/12 1:23:11.155 PM	prod	1354303391.155 1354303393.155

As you can see, we have our actual `_time`, which Splunk always draws according to the user's preferences, then our `network` value, and then the two values for `t` created using `mvappend`. Now we can expand each event into two events, so that we have a start and end event:

```
sourcetype=impl_splunk_gen network="*"
| eval endtime=_time+2
| eval t=mvappend(_time, endtime)
| mvexpand t
```

`mvexpand` replicates each event for each value in the field specified. In our case, each event will create two events, as `t` always contains two values. All other fields are copied into the new event. With the addition of `table _time network t`, our events now look like this:

	<code>_time</code>	<code>network</code>	<code>t</code>
1	11/30/12 1:23:28.046 PM	qa	1354303408.046
2	11/30/12 1:23:28.046 PM	qa	1354303410.046
3	11/30/12 1:23:25.869 PM	qa	1354303405.869
4	11/30/12 1:23:25.869 PM	qa	1354303407.869
5	11/30/12 1:23:25.057 PM	qa	1354303405.057
6	11/30/12 1:23:25.057 PM	qa	1354303407.057
7	11/30/12 1:23:22.736 PM	qa	1354303402.736
8	11/30/12 1:23:22.736 PM	qa	1354303404.736
9	11/30/12 1:23:20.944 PM	prod	1354303400.944
10	11/30/12 1:23:20.944 PM	prod	1354303402.944

Now that we have a start and end event, we need to mark the events as such. We will create a field named `increment` that we can use to create a running total. Start events will be positive, while end events will be negative. As the events stream through `streamstats`, the positive value will increment our counter, and the negative value will decrement our counter.

Our start events will have the value of `_time` replicated in `t`, so we will use `eval` to test this and set the value of `increment` accordingly. After setting `increment`, we will reset the value of `_time` to the value of `t`, so that our end events appear to have happened in the future.

```
sourcetype=impl_splunk_gen network="*"
| eval endtime=_time+2
| eval t=mvappend(_time,endtime)
| mvexpand t
| eval increment=if(_time=t,1,-1)
| eval _time=t
```

With the addition of `table _time network increment`, this gives us results as shown in the following screenshot:

	<code>_time</code> ↕	<code>network</code> ↕	<code>increment</code> ↕
1	11/30/12 1:23:28.046 PM	qa	1
2	11/30/12 1:23:30.046 PM	qa	-1
3	11/30/12 1:23:25.869 PM	qa	1
4	11/30/12 1:23:27.869 PM	qa	-1
5	11/30/12 1:23:25.057 PM	qa	1
6	11/30/12 1:23:27.057 PM	qa	-1
7	11/30/12 1:23:22.736 PM	qa	1
8	11/30/12 1:23:24.736 PM	qa	-1
9	11/30/12 1:23:20.944 PM	prod	1
10	11/30/12 1:23:22.944 PM	prod	-1

`streamstats` expects events to be in the order that you want to calculate your statistics. Currently, our fictitious end events are sitting right next to the start events, but we want to calculate a running total of `increment`, based on the order of `_time`. The `sort` command will take care of this for us. The `0` value before the field list defeats the default limit of 10,000 rows.

```
sourcetype=impl_splunk_gen network="*"
| eval endtime=_time+2
| eval t=mvappend(_time,endtime)
| mvexpand t
```



```
| eval increment=if(_time=t,1,-1)
| eval _time=t
| sort 0 _time network increment
```



One thing to note at this point is that we have reset several values in this query using commands. We have changed `_time`, and now we have changed `increment`. A field can be changed as many times as is needed, and the last assignment in the chain wins.

Now that our events are sorted by `_time`, we are finally ready for `streamstats`. This command calculates statistics over a rolling set of events, in the order the events are seen. In combination with our `increment` field, this command will act just like `concurrency`, but will keep separate running totals for each of the fields listed after by:

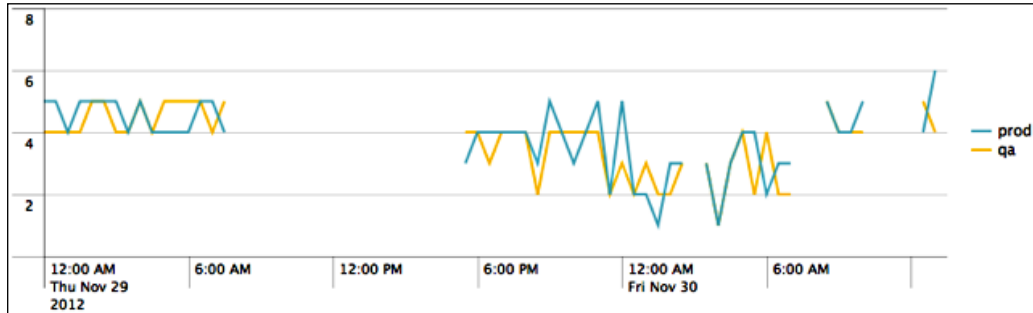
```
sourcetype=impl_splunk_gen network="*"
| eval endtime=_time+2
| eval t=mvappend(_time,endtime)
| mvexpand t
| eval increment=if(_time=t,1,-1)
| eval _time=t
| sort 0 _time network increment
| streamstats sum(increment) as concurrency by network
| search increment="1"
```

The last `search` statement will eliminate our synthetic end events.

Piping the results through `table _time network increment concurrency`, we get these results:

	<u>_time</u> ↕	<u>network</u> ↕	<u>increment</u> ↕	<u>concurrency</u> ↕
1	11/29/12 12:00:00.788 AM	qa	1	1
2	11/29/12 12:00:01.005 AM	prod	1	1
3	11/29/12 12:00:03.551 AM	qa	1	1
4	11/29/12 12:00:03.981 AM	qa	1	2
5	11/29/12 12:00:04.671 AM	qa	1	3
6	11/29/12 12:00:05.573 AM	qa	1	3
7	11/29/12 12:00:06.013 AM	qa	1	3
8	11/29/12 12:00:07.694 AM	qa	1	2
9	11/29/12 12:00:08.212 AM	qa	1	2
10	11/29/12 12:00:10.714 AM	prod	1	1

With the addition of `timechart max(concurrency) by network`, we see:



While this has been an interesting exercise, in the real world, you probably wouldn't calculate web server utilization in such a manner. The number of events is often quite large, and the time each event takes is normally negligible. This approach would be more interesting for longer running processes, such as batch or database processes.

The more common approach for web logs is to simply count events over time. We'll look at several ways to accomplish this next.

Calculating events per slice of time

There are a number of ways to calculate events per some period of time. All of these techniques rely on rounding `_time` down to some period of time, and then grouping the results by the rounded "buckets" of `_time`.

Using timechart


The simplest approach to count events over time is simply to use `timechart`, like this:

```
sourcetype=impl_splunk_gen
| timechart span=1m count
```

In table view, we see:

	_time ↕	count ↕
1	5/3/12 12:00:00.000 AM	109
2	5/3/12 12:01:00.000 AM	122
3	5/3/12 12:02:00.000 AM	122
4	5/3/12 12:03:00.000 AM	119
5	5/3/12 12:04:00.000 AM	125
6	5/3/12 12:05:00.000 AM	126
7	5/3/12 12:06:00.000 AM	119
8	5/3/12 12:07:00.000 AM	126
9	5/3/12 12:08:00.000 AM	120
10	5/3/12 12:09:00.000 AM	129
11	5/3/12 12:10:00.000 AM	114
12	5/3/12 12:11:00.000 AM	126
13	5/3/12 12:12:00.000 AM	120

Looking at a 24-hour period, we are presented with 1,440 rows, one per minute.

 Charts in Splunk do not attempt to show more points than the pixels present on the screen. The user is instead expected to change the number of points to graph, using the `bins` or `span` attributes. *Calculating average events per minute, per hour* shows another way of dealing with this behavior.

If we only wanted to know about minutes that actually had events, instead of every minute of the day, we could use `bucket` and `stats`, like this:

```
sourcetype=impl_splunk_gen
| bucket span=1m _time
| stats count by _time
```

`bucket` rounds the `_time` field of each event *down* to the minute in which it occurred, which is exactly what `timechart` does internally. This data will look the same, but any minutes with out events will not be included. Another way to accomplish the same thing would be as follows:

```
sourcetype=impl_splunk_gen
| timechart span=1m count
| where count>0
```

Calculating average requests per minute

If we take our previous queries and send the results through `stats`, we can calculate the average events per minute, like this:

```
sourcetype=impl_splunk_gen
| timechart span=1m count
| stats avg(count) as "Average events per minute"
```

This gives us exactly one row:

Average events per minute ↕	
1	61.240972

Alternatively, we can use `bucket` to group events by minute, and `stats` to count by each minute that has values, as shown in the following code:

```
sourcetype=impl_splunk_gen
| bucket span=1m _time
| stats count by _time
| stats avg(count) as "Average events per minute"
```

We are now presented with a much higher number:

Average events per minute ↕	
1	118.690444

Why? In this case, our fictitious server was down for about 10 hours. In our second example, only minutes that actually had events were included in the results, because `stats` does not produce an event for every slice of time, as `timechart` does. To illustrate this difference, look at the results of two queries:

```
sourcetype=impl_splunk_gen
| timechart span=1h count
```

This query produces the following table:

	<u>_time</u> ↕	<u>count</u> ↕
1	5/8/12 7:00:00.000 AM	3362
2	5/8/12 8:00:00.000 AM	498
3	5/8/12 9:00:00.000 AM	0
4	5/8/12 10:00:00.000 AM	38

Using bucket and stats, like this:

```
sourcetype=impl_splunk_gen
| bucket span=1h _time
| stats count by _time
```

We then get this table:

	_time ↕	count ↕
1	5/8/12 7:00:00.000 AM	3362
2	5/8/12 8:00:00.000 AM	498
3	5/8/12 10:00:00.000 AM	38

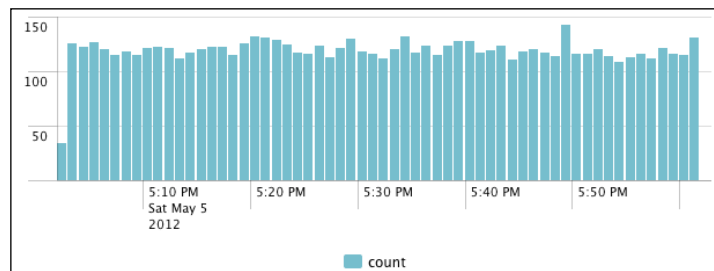
In this case, there are no results for the 9:00 AM to 10:00 AM time slot.

Calculating average events per minute, per hour

One limitation of graphing in Splunk is that only a certain number of events can be drawn, as there are only so many pixels available to draw. When counting or adding values over varying periods of time, it can be difficult to know what timescale is being represented. For example, given the following query:

```
earliest=-1h sourcetype=impl_splunk_gen
| timechart count
```

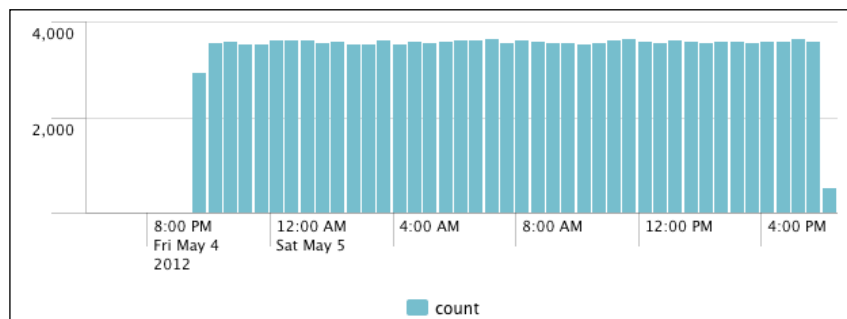
Splunk will produce this graph:



Each of these bars represent one minute. If we change the time frame to 24 hours:

```
earliest=-24h sourcetype=impl_splunk_gen
| timechart count
```

We are presented with this graph:

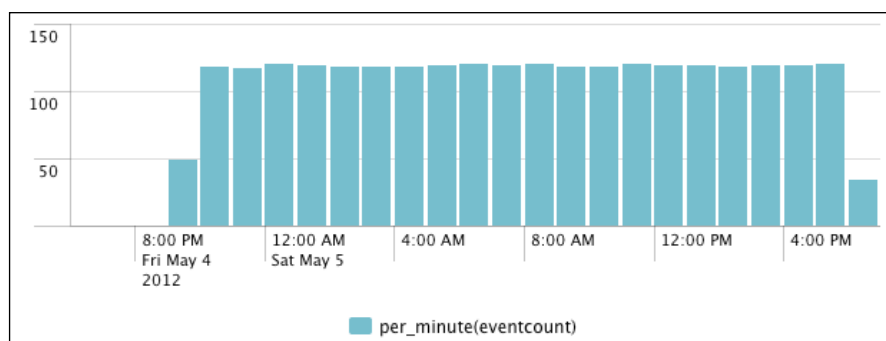


There is no indication of what period of time is represented by each bar unless you roll over the chart. In this case, each bar represents 30 minutes. This makes the significance of the y axis difficult to judge. In both cases, we can add `span=1m` to `timechart`, and we would know that each bar represents one minute. This would be fine for a chart representing one hour, but a query for 24 hours would produce too many points, and we would see a truncated chart.

Another approach would be to calculate the average events per minute, and then calculate that value over whatever time frame we are looking at. `timechart` provides a convenient function to accomplish this, but we have to do a little extra work.

```
earliest=-24h sourcetype=impl_splunk_gen
| eval eventcount=1
| timechart span=1h per_minute(eventcount)
```

`per_minute` calculates the sum of `eventcount` per minute, then finds the average value for the slice of time each bar represents. In this case, we are seeing the average number of events per hour.

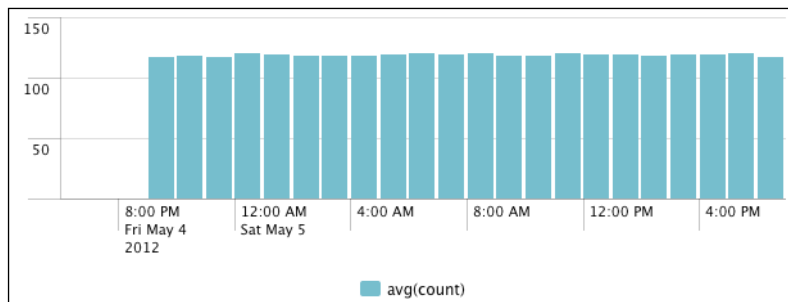


This scale looks in line with our one-hour query, as we are now looking at the event count per minute.

Like in the *Calculating average requests per minute* section, we could also ignore minutes that had no data. We could accomplish that as shown in the following code:

```
earliest=-24h sourcetype=impl_splunk_gen
| bucket span=1m _time
| stats count by _time
| timechart span=1h avg(count)
```

This approach does not penalize incomplete hours, for instance, the current hour. The graph looks like this:



This gives us a better understanding of events for the *current* hour, but is arguably not entirely truthful about the *first* hour in the graph.

Rebuilding top

The `top` command is very simple to use, but is actually doing a fair amount of interesting work. I often start with `top`, then switch to `stats count`, but then wish for something that `top` provides automatically. This exercise will show you how to recreate all of the elements, so that you might pick and choose what you need.

Let's recreate the `top` command by using other commands.

Here is the query that we will replicate:

```
sourcetype="impl_splunk_gen" error
| top useother=t limit=5 logger user
```

The output looks like this:

	logger ↕	user ↕	count ↕	percent ↕
1	BarClass	mary	773	18.812363
2	BarClass	jacky	422	10.270139
3	BarClass	bob	394	9.588708
4	BarClass	linda	391	9.515697
5	BarClass	Bobby	381	9.272329
6	OTHER	OTHER	1748	42.540764

To build count, we can use `stats` like this:

```
sourcetype="impl_splunk_gen" error
| stats count by logger user
```

This gets us most of the way to our end goal:

	logger ↕	user ↕	count ↕
1	AuthClass	Bobby	103
2	AuthClass	bob	68
3	AuthClass	extrauser	33
4	AuthClass	jacky	81
5	AuthClass	linda	79
6	AuthClass	mary	148
7	BarClass	Bobby	381
8	BarClass	bob	394
9	BarClass	extrauser	162
10	BarClass	jacky	422
11	BarClass	linda	391
12	BarClass	mary	773
13	FooClass	Bobby	104
14	FooClass	bob	87
15	FooClass	extrauser	43
16	FooClass	jacky	90
17	FooClass	linda	81
18	FooClass	mary	146
19	LogoutClass	Bobby	88
20	LogoutClass	bob	74
21	LogoutClass	extrauser	28
22	LogoutClass	jacky	82
23	LogoutClass	linda	80
24	LogoutClass	mary	171

To calculate the percentage that `top` includes, we will first need the total number of events. The `eventstats` command lets us add statistics to every row, without replacing the rows.

```
sourcetype="impl_splunk_gen" error
| stats count by logger user
| eventstats sum(count) as totalcount
```

This adds our `totalcount` column in the result:

	logger ↕	user ↕	count ↕	totalcount ↕
1	AuthClass	Bobby	103	4109
2	AuthClass	bob	68	4109
3	AuthClass	extrauser	33	4109
4	AuthClass	jacky	81	4109
5	AuthClass	linda	79	4109
6	AuthClass	mary	148	4109
7	BarClass	Bobby	381	4109
8	BarClass	bob	394	4109
9	BarClass	extrauser	162	4109
10	BarClass	jacky	422	4109
11	BarClass	linda	391	4109
12	BarClass	mary	773	4109
13	FooClass	Bobby	104	4109
14	FooClass	bob	87	4109
15	FooClass	extrauser	43	4109
16	FooClass	jacky	90	4109
17	FooClass	linda	81	4109
18	FooClass	mary	146	4109
19	LogoutClass	Bobby	88	4109
20	LogoutClass	bob	74	4109
21	LogoutClass	extrauser	28	4109
22	LogoutClass	jacky	82	4109
23	LogoutClass	linda	80	4109
24	LogoutClass	mary	171	4109

Now that we have our total, we can calculate the percentage for each row. While we're at it, let's sort the results in descending order by count:

```
sourcetype="impl_splunk_gen" error
| stats count by logger user
| eventstats sum(count) as totalcount
| eval percent=count/totalcount*100
| sort -count
```

This gives us:

	logger ↕	user ↕	count ↕	percent ↕	totalcount ↕
1	BarClass	mary	773	18.812363	4109
2	BarClass	jacky	422	10.270139	4109
3	BarClass	bob	394	9.588708	4109
4	BarClass	linda	391	9.515697	4109
5	BarClass	Bobby	381	9.272329	4109
6	LogoutClass	mary	171	4.161596	4109
7	BarClass	extrauser	162	3.942565	4109
8	AuthClass	mary	148	3.601850	4109
9	FooClass	mary	146	3.553176	4109
10	FooClass	Bobby	104	2.531029	4109
11	AuthClass	Bobby	103	2.506693	4109
12	FooClass	jacky	90	2.190314	4109
13	LogoutClass	Bobby	88	2.141640	4109
14	FooClass	bob	87	2.117303	4109
15	LogoutClass	jacky	82	1.995619	4109
16	AuthClass	jacky	81	1.971283	4109
17	FooClass	linda	81	1.971283	4109
18	LogoutClass	linda	80	1.946946	4109
19	AuthClass	linda	79	1.922609	4109
20	LogoutClass	bob	74	1.800925	4109
21	AuthClass	bob	68	1.654904	4109
22	FooClass	extrauser	43	1.046483	4109
23	AuthClass	extrauser	33	0.803115	4109
24	LogoutClass	extrauser	28	0.681431	4109

If not for `useother=t`, we could simply end our query with `head 5`, which would return the first five rows. To accomplish the "other" row, we will have to label everything beyond row **5** with a common value, and collapse the rows using `stats`. This will take a few steps.

First, we need to create a counter field, which we will call `rownum`:

```
sourcetype="impl_splunk_gen" error
| stats count by logger user
| eventstats sum(count) as totalcount
| eval percent=count/totalcount*100
| sort -count
| eval rownum=1
```

This gives us (only the first 10 rows are shown):

	logger ↕	user ↕	count ↕	percent ↕	rownum ↕	totalcount ↕
1	BarClass	mary	773	18.812363	1	4109
2	BarClass	jacky	422	10.270139	1	4109
3	BarClass	bob	394	9.588708	1	4109
4	BarClass	linda	391	9.515697	1	4109
5	BarClass	Bobby	381	9.272329	1	4109
6	LogoutClass	mary	171	4.161596	1	4109
7	BarClass	extrauser	162	3.942565	1	4109
8	AuthClass	mary	148	3.601850	1	4109
9	FooClass	mary	146	3.553176	1	4109
10	FooClass	Bobby	104	2.531029	1	4109

Next, using `accum`, we will increment the value of `rownum`:

```
sourcetype="impl_splunk_gen" error
| stats count by logger user
| eventstats sum(count) as totalcount
| eval percent=count/totalcount*100
| sort -count
| eval rownum=1
| accum rownum
```

This gives us (only the first 10 rows are shown):

	logger ↕	user ↕	count ↕	percent ↕	rownum ↕	totalcount ↕
1	BarClass	mary	773	18.812363	1	4109
2	BarClass	jacky	422	10.270139	2	4109
3	BarClass	bob	394	9.588708	3	4109
4	BarClass	linda	391	9.515697	4	4109
5	BarClass	Bobby	381	9.272329	5	4109
6	LogoutClass	mary	171	4.161596	6	4109
7	BarClass	extrauser	162	3.942565	7	4109
8	AuthClass	mary	148	3.601850	8	4109
9	FooClass	mary	146	3.553176	9	4109
10	FooClass	Bobby	104	2.531029	10	4109

Now using `eval`, we can label everything beyond row 5 as `OTHER`, and flatten `rownum` beyond 5:

```
sourcetype="impl_splunk_gen" error
| stats count by logger user
| eventstats sum(count) as totalcount
| eval percent=count/totalcount*100
| sort -count
| eval rownum=1
| accum rownum
| eval logger=if(rownum>5,"OTHER",logger)
| eval user=if(rownum>5,"OTHER",user)
| eval rownum=if(rownum>5,6,rownum)
```

This gives us (only the first 10 rows are shown):

	logger ↕	user ↕	count ↕	percent ↕	rownum ↕	totalcount ↕
1	BarClass	mary	773	18.812363	1	4109
2	BarClass	jacky	422	10.270139	2	4109
3	BarClass	bob	394	9.588708	3	4109
4	BarClass	linda	391	9.515697	4	4109
5	BarClass	Bobby	381	9.272329	5	4109
6	OTHER	OTHER	171	4.161596	6	4109
7	OTHER	OTHER	162	3.942565	6	4109
8	OTHER	OTHER	148	3.601850	6	4109
9	OTHER	OTHER	146	3.553176	6	4109
10	OTHER	OTHER	104	2.531029	6	4109

Next, we will recombine the values using `stats`. Events are sorted by the fields listed after `by`, which will maintain our original order:

```
sourcetype="impl_splunk_gen" error
| stats count by logger user
| eventstats sum(count) as totalcount
| eval percent=count/totalcount*100
| sort -count
| eval rownum=1
| accum rownum
| eval logger=if(rownum>5,"OTHER",logger)
| eval user=if(rownum>5,"OTHER",user)
| eval rownum=if(rownum>5,6,rownum)
| stats
  sum(count) as count
  sum(percent) as percent
  by rownum logger user
```

This gives us:

	rownum ↕	logger ↕	user ↕	count ↕	percent ↕
1	1	BarClass	mary	773	18.812363
2	2	BarClass	jacky	422	10.270139
3	3	BarClass	bob	394	9.588708
4	4	BarClass	linda	391	9.515697
5	5	BarClass	Bobby	381	9.272329
6	6	OTHER	OTHER	1748	42.540764

We're almost done! All that's left to do is hide the `rownum` column. We can use `fields` for this purpose:

```
sourcetype="impl_splunk_gen" error
| stats count by logger user
| eventstats sum(count) as totalcount
| eval percent=count/totalcount*100
| sort -count
| eval rownum=1
| accum rownum
| eval logger=if(rownum>5,"OTHER",logger)
| eval user=if(rownum>5,"OTHER",user)
| eval rownum=if(rownum>5,6,rownum)
| stats
  sum(count) as count
  sum(percent) as percent
  by rownum logger user
| fields - rownum
```

This finally gives us what we are after:

	logger ↕	user ↕	count ↕	percent ↕
1	BarClass	mary	773	18.812363
2	BarClass	jacky	422	10.270139
3	BarClass	bob	394	9.588708
4	BarClass	linda	391	9.515697
5	BarClass	Bobby	381	9.272329
6	OTHER	OTHER	1748	42.540764

And we're done. Just a reminder of what we were reproducing:

```
top useother=t limit=5 logger user
```

That is a pretty long query to replicate a one liner! While completely recreating `top` is not something practically needed, hopefully this example sheds some light on how to combine commands in interesting ways.

Summary

I hope this chapter was enlightening, and has sparked some ideas that you can apply to your own data. As stated in the introduction, Splunk Answers (<http://answers.splunk.com>) is a fantastic place to find examples and general help. You can ask your questions there, and contribute answers back to the community.

In the next chapter, we will use more advanced features of Splunk to help extend the search language, and enrich data at search time.

6

Extending Search

In this chapter, we will look at some of the features that Splunk provides to go beyond its already powerful search language. We will cover the following with the help of examples:

- Tags and event types that help you categorize events, both for search and reporting
- Lookups that allow you to add external fields to events as though they were part of the original data
- Macros that let you reuse snippets of search in powerful ways
- Workflow actions that let you build searches and links based on field values in an event
- External commands that allow you to use Python code to work with search results

In this chapter, we will investigate a few of the many commands included in Splunk. We will write our own commands in *Chapter 12, Extending Splunk*.

Using tags to simplify search

Tags allow you to attach a "marker" to fields and event types in Splunk. You can then search and report on these tags later. Let's attach a tag to a couple of users who are administrators. Start with the following search:

```
sourcetype="impl_splunk_gen"  
| top user
```


This search gives us a list of our users such as **mary**, **linda**, **Bobby**, **jacky**, **bob**, and **extrauser**:

	user ↕	count ↕	percent ↕
1	mary	39044	31.695417
2	linda	19641	15.944311
3	Bobby	19593	15.905346
4	jacky	19503	15.832285
5	bob	19460	15.797378
6	extrauser	5944	4.825263

Let's say that in our group, linda and jacky are administrators. Using a standard search, we can simply search for these two users like this:

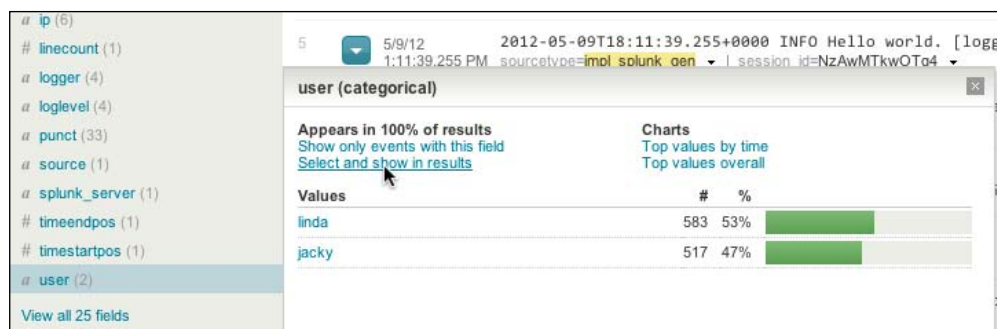
```
sourcetype="impl_splunk_gen" (user=linda OR user=jacky)
```

Searching for these two users while going forward will still work, but instead if we search for the tag value, we can avoid being forced to update multiple saved queries in the future.

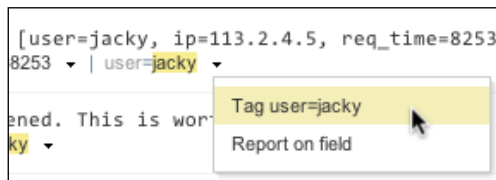
To create a tag, first we need to locate the field.



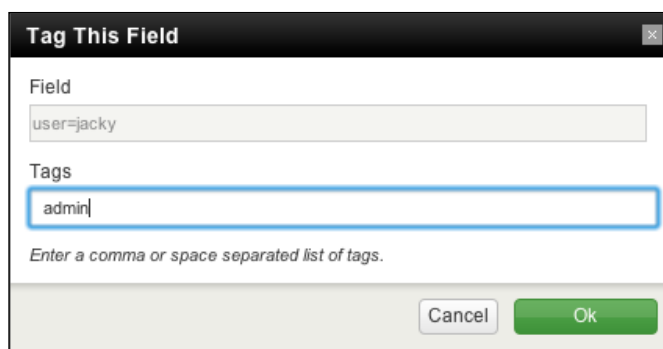
If the **user** field isn't already visible, click on it in the field picker, and then click on **Select and show in results**:



With the menu now visible, we can tag this value of the **user** field:



We are presented with the **Tag This Field** dialog as shown in the following screenshot. Let's tag user=jacky with admin:



We now see our tag next to this field:



Once this is done, follow the steps used for user=jacky for user=linda.

With these two users tagged, we can search for the tag value instead of the actual usernames:

```
sourcetype="impl_splunk_gen" tag::user="admin"
```

Under the covers, this query is unrolled into exactly the same query we started with. The advantage is that if this tag is added to new values or removed from existing ones, no queries have to be updated.

Some other interesting features of tags are as follows:

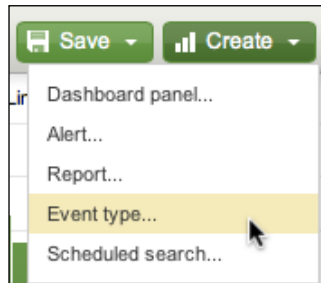
- Tags can be searched globally simply by using `tag=tag_name`; in this case `tag=admin`. Using this capability, you can apply any tag to any field or event type, and simply search for the tag. This is commonly used in security applications to tag hosts, users, and event types that need special monitoring.
- Any field or event type can have any number of tags. Simply choose the tag editor and enter multiple tag values separated by spaces.
- To remove a tag, simply edit the tags again and remove the value(s) you want to remove.
- Tags can also be edited in **Manager** at **Manager | Tags**.

Using event types to categorize results

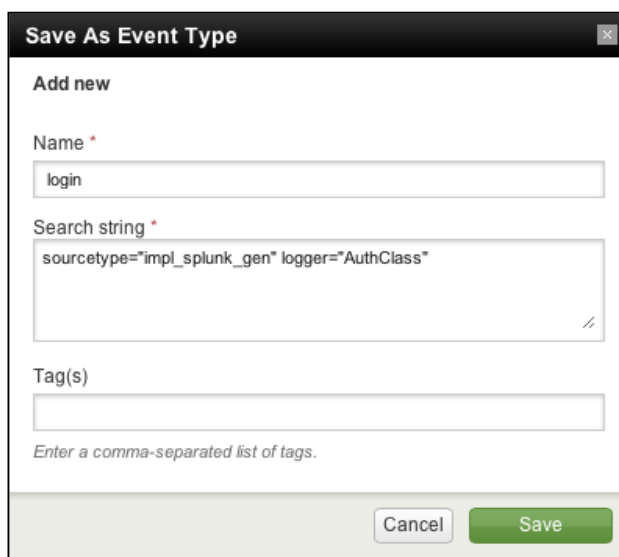
An event type is essentially a simple search definition, with no pipes or commands. To define an event type, first make a search. Let's search for:

```
sourcetype="impl_splunk_gen" logger="AuthClass"
```

Let's say these events are login events. To make an event type, choose **Event type...** from the **Create** menu, as shown here:



This presents us with a dialog, where we can assign a **Name** string and optionally any **Tags(s)** to this event type, as shown in the following screenshot:



Let's name our event type `login`.

We can now search for the same events using the event type:

```
eventtype=login
```

Event types can be used as part of another search, as follows:

```
eventtype=login loglevel=error
```

Event type definitions can also refer to other event types. For example, let's assume that all `login` events that have a `loglevel` value of `ERROR` are in fact failed logins.

We can now save this into another event type using the same steps as mentioned previously. Let's call it `failed_login`. We can now search for these events using the following:

```
eventtype="failed_login"
```

Now, let's combine this event type with the users that we tagged as `admin` in the previous section:

```
eventtype="failed_login" tag::user="admin"
```

This will find all failed logins for administrators. Let's now save this as yet another event type, `failed_admin_login`. We can now search for these events, as follows:

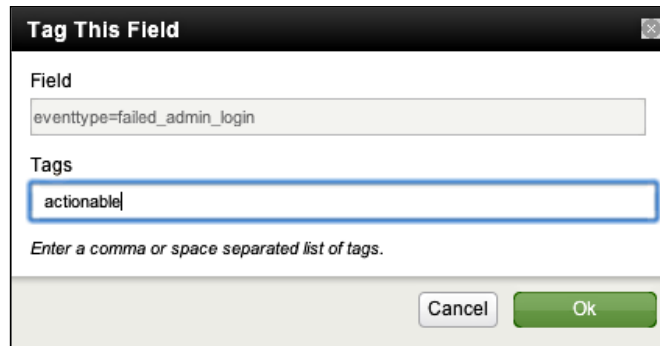
```
eventtype="failed_admin_login"
```

As a final step, let's tag this event type. First, make sure the field **eventtype** is visible. Your events should look like this:



Notice the three values of **eventtype** in this case. We are searching only for `eventtype=failed_admin_login`, but this event also matches the definitions of `eventtype=failed_login` and `eventtype=login`. Also notice our tagged user. We are not searching for the `admin` tag, but `jacky` matches `tag::user=admin`, so the value is tagged accordingly.

Following the steps in the previous section, tag `eventtype=failed_admin_login` with the value `actionable`:



We can now search for these events with the following query:

```
tag::eventtype="actionable"
```

This technique is very useful for building up definitions of events that should appear in alerts and reports. For example, consider the following query:

```
tag::eventtype="actionable"  
| table _time eventtype user
```

This will now give us a very useful report, shown as follows:

	<u>time</u> ↕	<u>eventtype</u> ↕	<u>user</u> ↕
1	5/14/12 12:43:02.202 PM	failed_admin_login failed_login login	jacky
2	5/14/12 12:42:16.394 PM	failed_admin_login failed_login login	jacky
3	5/14/12 12:40:17.947 PM	failed_admin_login failed_login login	jacky
4	5/14/12 12:39:30.712 PM	failed_admin_login failed_login login	linda
5	5/14/12 12:39:17.054 PM	failed_admin_login failed_login login	linda

Think about the ways that these event types are being used in this seemingly simple query:

- **Search:** An event type definition is defined as a search, so it seems only natural that you can search for events that match an event type definition.
- **Categorization:** As events are retrieved, if the events match the definition of *any* event type, those events will have that event type's name added to the `eventtype` field.
- **Tagging:** Since event types can also be tagged, tag values assigned to certain event types can be used for both search and categorization. This is extremely powerful for assigning common tags to varied sets of results; for instance, events that belong in a report or should cause an alert.

For clarity, let's unroll this query to see what Splunk is essentially doing under the covers. The query is expanded from the tag and event type definitions, as follows:

- `tag::eventtype="actionable"`
- `eventtype="failed_admin_login"`
- `eventtype="failed_login" tag::user="admin"`
- `(eventtype=login loglevel=error) tag::user="admin"`
- `((sourcetype="impl_splunk_gen" logger="AuthClass") loglevel=error) tag::user="admin"`
- `((sourcetype="impl_splunk_gen" logger="AuthClass") loglevel=error) (user=linda OR user=jacky)`

Let's explain what happens at each step:

1. The initial search.
2. All event types that are tagged `actionable` are substituted. In this case, we only have one, but if there were multiple, they would be combined with `OR`.
3. The definition of the event type `failed_admin_login` is expanded.
4. The definition of `failed_login` is expanded.
5. The definition of `login` is expanded.
6. All values of `user` with the tag `admin` are substituted, separated by `OR`.

Any changes to tagged values or event type definitions will be reflected the next time they are used in any search or report.

Using lookups to enrich data

Sometimes, information that would be useful for reporting and searching is not located in the logs themselves, but is available elsewhere. Lookups allow us to enrich data, and even search against the fields in the lookup as if they were part of the original events.

The source of data for a lookup can be either a **Comma Separated Values (CSV)** file or a script. We will cover the most common use of a CSV lookup in the next section. We will cover scripted lookups in *Chapter 12, Extending Splunk*.

There are three steps for fully defining a lookup: creating the file, defining the lookup definition, and optionally wiring the lookup to run automatically.

Defining a lookup table file

A lookup table file is simply a CSV file. The first line is treated as a list of field names for all other lines.

Lookup table files are managed at **Manager | Lookups | Lookup table files**. Simply upload a new file and give it a filename, preferably ending in `.csv`.

The lookup file `users.csv` is included in `ImplementingSplunkDataGenerator`:

```
user,city,department,state
mary,Dallas,HR,TX
jacky,Dallas,IT,TX
linda,Houston,HR,TX
Bobby,Houston,IT,TX
bob,Chicago,HR,IL
```

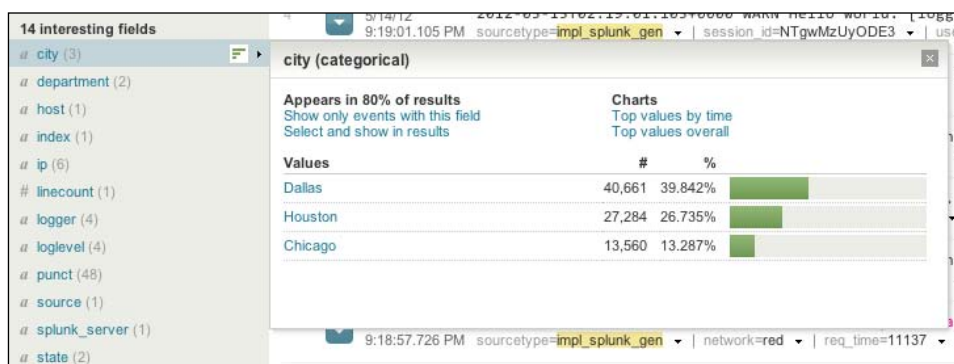
With this file uploaded, we can immediately use it with the `lookup` command. In the simplest case, the format of the `lookup` command is as follows:

```
lookup [lookup definition or file name] [matching field]
```

An example of its usage is as follows:

```
sourcetype="impl_splunk_gen"
| lookup users.csv user
```

We can now see all of the fields from the lookup file as if they were in the events:



We can use these fields in reports:

```
sourcetype="impl_splunk_gen"
| lookup users.csv user
| stats count by user city state department
```

This will produce results as shown in the following screenshot:

	user ↕	city ↕	state ↕	department ↕	count ↕
1	Bobby	Houston	TX	IT	13632
2	bob	Chicago	IL	HR	13560
3	jacky	Dallas	TX	IT	13411
4	linda	Houston	TX	HR	13652
5	mary	Dallas	TX	HR	27250

This is all that is required to use a CSV lookup to enrich data, but if we do a little more configuration work, we can make the lookup even more useful.

Defining a lookup definition

Though you can access a lookup immediately by the filename, defining the lookup allows you to set other options, reuse the same file, and later make the lookup run automatically. Creating a definition also eliminates a warning message that appears when simply using the filename.

Navigate to **Manager | Lookups | Lookup definitions** and click on the **New** button.

Add new

Destination app
search

Name *
userlookup

Type
File-based

Lookup file
users.csv

Create and manage lookup table files.

Configure time-based lookup

Advanced options

Minimum matches
1

The minimum number of matches for each input lookup value. Default is 0.

Maximum matches

The maximum number of matches for each input lookup value. If time-based, default is 1; otherwise, default is 100.

Default matches
unknown

If fewer than the minimum number of matches are present for any given input, write out this value one or more times such that the minimum is reached

Stepping through these fields, we have:

- **Destination app:** This is where the lookup definition will be stored. This matters because you may want to limit the scope of a lookup to a particular application for performance reasons.
- **Name:** This is the name that you will use in search strings.
- **Type:** The options here are **File-based** or **External**. We will cover **External**, or scripted, in *Chapter 12, Extending Splunk*.

- **Lookup file:** We have chosen `users.csv` in this case.
- **Configure time-based lookup:** Using a time-based lookup, you can have a value that changes at certain points in time while going forward. For instance, if you built a lookup of what versions of software were deployed to what hosts at what time, you could generate a report on errors or response times by the software version.
- **Advanced options:** This simply exposes the remaining fields.
- **Minimum matches:** This defines the number of items in the lookup that must be matched. With a value of 1, the value of **Default matches** will be used if no match is found.
- **Maximum matches:** This defines the maximum number of matches before stopping. For instance, if there were multiple entries for each user in our lookup file, this value would limit the number of rows that would be applied to each event.
- **Default matches:** This value will be used to populate all fields from the lookup when no match is found, and **Minimum matches** is greater than 0.

After clicking on **Save**, we can use our new lookup in the following manner:

```
sourcetype="impl_splunk_gen"
| lookup userslookup user
| stats count by user city state department
```

This will produce results as shown in the following screenshot:

	user ↕	city ↕	state ↕	department ↕	count ↕
1	Bobby	Houston	TX	IT	13436
2	bob	Chicago	IL	HR	13315
3	extrauser	unknown	unknown	unknown	4171
4	jacky	Dallas	TX	IT	13184
5	linda	Houston	TX	HR	13443
6	mary	Dallas	TX	HR	26819

Notice that `extrauser` now appears in the table since it has values for `city`, `state`, and `department`.

Lookup tables have other features, including wildcard lookups, CIDR lookups, and temporal lookups. We will use those features in later chapters.

Defining an automatic lookup

Automatic lookups are, in this author's opinion, one of the coolest features in Splunk. Not only are the contents of the lookup added to events as if they were always there, but you can also search against the fields in the lookup file as if they were part of the original event.

To define the automatic lookup, navigate to **Manager | Lookups | Automatic lookups** and click on the **New** button:

The screenshot shows the 'Add new' configuration form for an automatic lookup. The form is titled 'Add new' in green. It contains the following fields and options:

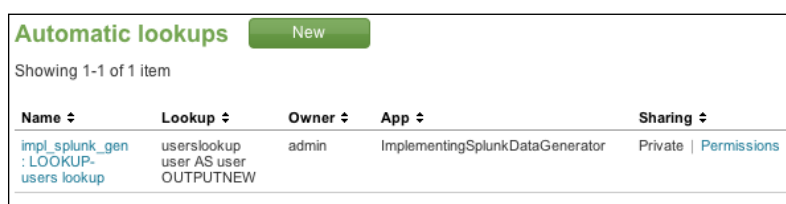
- Destination app:** A dropdown menu with 'ImplementingSplunkDataGenerator' selected.
- Name:** A text input field containing 'lookupusers'.
- Lookup table:** A dropdown menu with 'userslookup' selected.
- Apply to:** A dropdown menu with 'sourcetype' selected.
- named:** A text input field containing 'impl_splunk_ger'.
- Lookup input fields:** A section with one field mapping 'user' to 'user', with a 'Delete' button to the right.
- Lookup output fields:** A section with two empty input fields separated by an equals sign, with a 'Delete' button to the right.
- Overwrite field values:** A checkbox that is currently unchecked.
- Buttons:** 'Cancel' and 'Save' buttons at the bottom.

Let's step through the fields in this definition:

- **Destination app:** This is the application where the definition will live. We'll discuss the implications of this choice in *Chapter 7, Working with Apps*.
- **Name:** This name is used in the configuration. It should not contain spaces or special characters. We will discuss its significance in *Chapter 10, Configuring Splunk*.
- **Lookup table:** This is the name of the lookup definition.
- **Apply to:** This lets us choose which events are acted upon. The usual case is **sourcetype**, which must match a sourcetype name exactly. Alternatively, you can specify **source** or **host**, with or without wildcards.

- **Lookup input fields:** This defines the fields that will be queried in the lookup file. One field must be specified, but multiple fields can be specified. Think of this as a join in a database. The left side is the name of the field in the lookup file. The right side is the name of the existing field in our events.
- **Lookup output fields:** This section lets you decide what columns to include from the lookup file and optionally overrides the names of those fields. The left side is the name of the field in the lookup file. The right side is the field to be created in the events. If left blank, the default behavior is to include all fields from the lookup, using the names defined in the lookup file.
- **Overwrite field values:** If this option is selected, any existing field values in an event will be overwritten by a value with the same name from the lookup file.

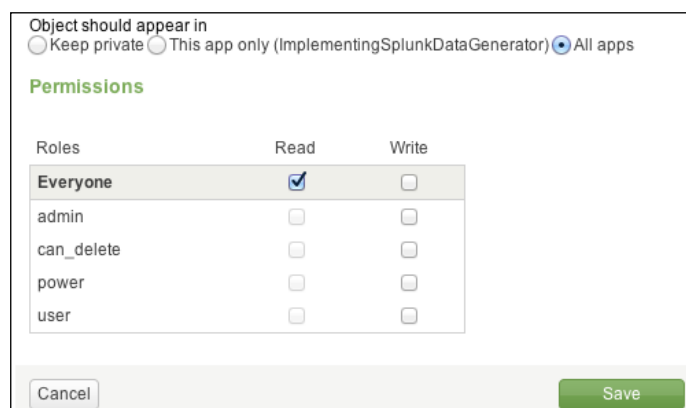
After clicking on **Save**, we see the listing of **Automatic lookups**. Initially, the **Sharing** option is **Private**, which will cause problems if you want to share searches with others. To share the lookup, first click on **Permissions**.



The screenshot shows the 'Automatic lookups' page with a 'New' button and a table listing one item. The table has columns for Name, Lookup, Owner, App, and Sharing. The item listed is 'impl_splunk_gen : LOOKUP-users lookup' with owner 'admin' and app 'ImplementingSplunkDataGenerator'. The sharing status is 'Private' with a 'Permissions' link.

Name	Lookup	Owner	App	Sharing
impl_splunk_gen : LOOKUP-users lookup	userslookup user AS user OUTPUTNEW	admin	ImplementingSplunkDataGenerator	Private Permissions

This presents us with the **Permissions** page. Change the value of **Object should appear in** to **All apps**. We will discuss these permissions in greater detail in *Chapter 10, Configuring Splunk*.



The screenshot shows the 'Permissions' configuration page. At the top, there are radio buttons for 'Object should appear in': 'Keep private', 'This app only (ImplementingSplunkDataGenerator)', and 'All apps' (which is selected). Below this is a table for 'Permissions' with columns for 'Roles', 'Read', and 'Write'. The 'Everyone' role has 'Read' checked and 'Write' unchecked. Other roles listed are 'admin', 'can_delete', 'power', and 'user', all with 'Read' and 'Write' unchecked. At the bottom, there are 'Cancel' and 'Save' buttons.

Object should appear in
 Keep private This app only (ImplementingSplunkDataGenerator) All apps

Permissions

Roles	Read	Write
Everyone	<input checked="" type="checkbox"/>	<input type="checkbox"/>
admin	<input type="checkbox"/>	<input type="checkbox"/>
can_delete	<input type="checkbox"/>	<input type="checkbox"/>
power	<input type="checkbox"/>	<input type="checkbox"/>
user	<input type="checkbox"/>	<input type="checkbox"/>

Cancel Save

We now have a fully automatic lookup, enriching the source type `impl_splunk_gen` based on the value of `user` in each event. To show the power of this lookup, let's search for a field in the lookup file, as if it were part of the events:

```
source="impl_splunk_gen" department="HR" | top user
```

Even though `department` isn't in our events at all, Splunk will reverse the lookup, find the values of `user` that are in `department`, and run the search for those users. This returns the following result:

	user ↕	count ↕	percent ↕
1	mary	6376	49.894358
2	bob	3206	25.088035
3	linda	3197	25.017607

Let's combine this search with an event type that we defined earlier. To find the most recent failed login for each member of HR, we can run:

```
source="impl_splunk_gen" department="HR" eventtype="failed_login"  
| dedup user  
| table _time user department city state
```

This returns:

	_time ↕	user ↕	department ↕	city ↕	state ↕
1	5/14/12 11:07:48.570 PM	bob	HR	Chicago	IL
2	5/14/12 11:04:53.993 PM	linda	HR	Houston	TX
3	5/14/12 11:04:49.755 PM	mary	HR	Dallas	TX

The `dedup` command simply says to keep only one event for each value of `user`. As events are returned in the "most recent first" order, this query will return the most recent login for each `user`.

We will configure more advanced lookups in later chapters.

Troubleshooting lookups

If you are having problems with a lookup, very often the problem is with permissions. Check permissions at all three of these paths:

- **Manager | Lookups | Lookup table files**
- **Manager | Lookups | Lookup definitions**
- **Manager | Lookups | Automatic lookups**

Once permissions are squared away, be sure to keep the following points in mind:

- Check your spelling of the fields.
- By default, lookup values are case sensitive.
- If your installation is using multiple indexers, it may take some time for the lookup files and definitions to be distributed to your indexers, particularly if the lookup files are large or you have installed many apps that have assets to be distributed.
- A rule of thumb is that a lookup file should not have more than two million rows. If a lookup is too large, an external lookup script may be required.

Using macros to reuse logic

A **macro** serves the purpose of replacing bits of search language with expanded phrases. Using macros can help you reuse logic and greatly reduce the length of queries.

Let's use one of our examples from *Chapter 5, Advanced Search Examples*, as our example case:

```
sourcetype="impl_splunk_web" user=mary
| transaction maxpause=5m user
| stats avg(duration) avg(eventcount)
```

Creating a simple macro

Let's take the last two lines of our query and convert them to a macro. First, navigate to [Manager](#) | [Advanced search](#) | [Advanced search](#) | [Search macros](#) and click on **New**.

The screenshot shows a web form titled "Add new" for creating a search macro. The form contains the following fields and options:

- Destination app:** A dropdown menu with "search" selected.
- Name *:** A text input field containing "webtransactions". Below it is a small instruction: "Enter the name of the macro. If the search macro takes an argument, indicate this by appending the number of arguments." (partially visible).
- Definition *:** A text area containing the query: `| transaction maxpause=5m user | stats avg(duration) avg(eventcount)`. Below it is a small instruction: "Enter the string the search macro expands to when it is referenced in another search. If arguments are included, enclose them in curly braces." (partially visible).
- Use eval-based definition?:** An unchecked checkbox.
- Arguments:** A text input field. Below it is a small instruction: "Enter a comma-delimited string of argument names. Argument names may only contain alphanumeric, '_' and '.' characters." (partially visible).
- Validation Expression:** A text input field. Below it is a small instruction: "Enter an eval or boolean expression that runs over macro arguments." (partially visible).
- Validation Error Message:** A text input field. Below it is a small instruction: "Enter a message to display when the validation expression returns 'false'." (partially visible).

At the bottom of the form are two buttons: "Cancel" and "Save".

Walking through our fields, we have:

- **Destination app:** This is where the macro will live.
- **Name:** This is the name we will use in our searches.
- **Definition:** This is the text that will be placed in our search.
- **Use eval-based definition?:** If checked, the **Definition** string is treated as an `eval` statement instead of raw text. We'll use this option later.
- The remaining fields are used if arguments are specified. We will use these in our next example.

After clicking on **Save**, our macro is now available for use. We can use it like this:

```
sourcetype="impl_splunk_web" user=mary `webtransactions`
```

`webtransactions` is enclosed by backticks. This is similar to the usage of backticks on a Unix command line, where a program can be executed to generate an argument. In this case, ``webtransactions`` is simply replaced with the raw text defined in the macro, recreating the query we started with.

Creating a macro with arguments

Let's collapse the entire search into a macro that takes two arguments, the `user` and a value for `maxpause`.

Add new

Destination app

Name *
 Enter the name of the macro. If the search macro takes an argument, indicate this by appending the number of arguments to the

Definition *
 Enter the string the search macro expands to when it is referenced in another search. If arguments are included, enclose them in

```
sourcetype="impl_splunk_web" user=$user$
| transaction maxpause=$maxpause$ user
| stats avg(duration) avg(eventcount)
```

Use eval-based definition?

Arguments
 Enter a comma-delimited string of argument names. Argument names may only contain alphanumeric, '_' and '-' characters.

Validation Expression
 Enter an eval or boolean expression that runs over macro arguments.

Validation Error Message
 Enter a message to display when the validation expression returns 'false'.



Be sure to remove newlines from your search definition. Macros do not appear to work with embedded newlines.

Walking through our fields, we have:

- **Name:** This is the name we will use in our searches. The parentheses and integer (2) specify how many arguments this macro expects.
- **Definition:** We have defined the entire query in this case. The variables are defined as `$user$` and `$maxpause$`. We can use these names because we have defined the variables under **Arguments**.
- **Arguments:** This list assigns variable names to the values handed in to the macro.

After clicking on **Save**, our macro is now available for use. We can use it like this:

```
webtransactions_user_maxpause(mary,5m)
```

or

```
`webtransactions_user_maxpause("mary","5m")`
```

Using eval to build a macro

We will use this feature in conjunction with a workflow action later in this chapter. See the *Building a workflow action to show field context* section later in this chapter.

Creating workflow actions

Workflow actions allow us to create custom actions based on the values in search results. The two supported actions either run a search or link to a URL.

Running a new search using values from an event

To build a workflow action, navigate to **Manager | Fields | Workflow actions** and click on **New**. You are presented with this form:

Add new

Destination app

Name *

Enter a unique name without spaces or special characters. This is used for identifying your workflow action later on within Splunk Manager.

Label *

Enter the label that appears for this action. Optionally, incorporate a field's value by enclosing the field name in dollar signs, e.g. 'Search for ticket number : \$ticketnum\$'.

Apply only to the following fields

Specify a comma-separated list of fields that must be present in an event for the workflow action to apply to it. When fields are specified, the workflow action only appears in the field menus for those fields; otherwise it appears in all field menus.

Apply only to the following event types

Specify a comma-separated list of event types that an event must be associated with for the workflow action to apply to it.

Show action in

Action type

Search configuration

Search string *

Enter the search for this action. Optionally, specify fields as \$fieldname\$, e.g. sourcetype=rails controller=\$controller\$ error=".

Run in app

Choose an app for the search to run in. Defaults to the current app.

Open in view

Enter the name of a view for the search to open in. Defaults to the current view.

Run search in

Time range

Earliest time Latest time

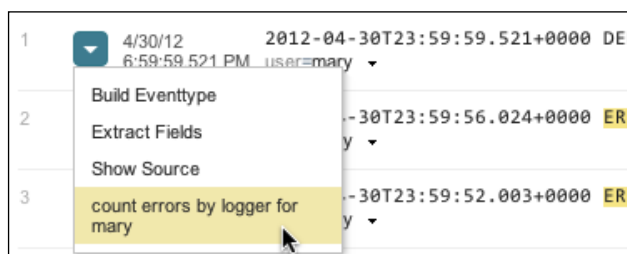
Use the same time range as the search that created the field listing

Let's walk through the following fields:

- **Destination app:** This is the app where the workflow action definition will live.
- **Name:** This is the name used in configuration files. This name cannot contain spaces, but underscores are fine.
- **Label:** This is what will appear in the menu. It can contain variables. In this case, we have included `$user$`, which will be populated with the value of the `user` field from the event.
- **Apply only to the following fields:** This workflow action will only appear on an event if all fields specified in this list have a value. **Show action in** will determine which menus can contain the workflow action.
- **Apply only to the following event types:** Only show this workflow action for events that match a particular event type. For instance, if you defined an event type called `login`, you might want a custom workflow action to search for all logins for this particular user over the last week.
- **Show action in:** The three options are **Event menu**, **Fields menus**, and **Both**.
 - The **Event menu** option is to the left of the event. If **Apply only to the following fields** is not empty, the workflow action will only be present if all of the fields specified are present in the event.
 - The **Fields menus** option falls to the right of each field under the events. If **Apply only to the following fields** is not empty, only the fields listed will contain the workflow action.
 - **Both** will show the workflow action in both places, following the same rules.
- **Action type:** The choices here are **search** or **link**. We have chosen **search**. We will try **link** in the next section.
- **Search string:** This is the search template to run. You will probably use field values here, but it is not required.
- **Run in app:** If left blank, the current app will be used, otherwise the search will be run in the app that is specified. You would usually want to leave this blank.
- **Open in view:** If left blank, the current view will be used. If you expect to use an events listing panel on dashboards, you probably want to set this to `flashtimeline`.

- **Run search in:** The choices here are **New window** or **Current window**.
- **Time range:** You *can* specify a specific time range here, either in epoch time or relative time. Leaving **Latest time** empty will search to the latest data available.
- **Use the same time range as the search that created the field listing:** In most cases, you will either check this checkbox or provide a value in at least **Earliest time**. If you do not, the query will run over all time, which is not usually what you want. It is also possible to specify the time frame in our query.

After we click on **Save**, we now see our action in the event workflow action menu like this:




After we choose the option, a new window appears with our results, like this:

	user ↕	logger ↕	count ↕	values(sourcetype) ↕
1	mary	0	131	impl_splunk_gen
2	mary	AuthClass	64	impl_splunk_gen
3	mary	BarClass	251	impl_splunk_gen
4	mary	FooClass	46	impl_splunk_gen
5	mary	LogoutClass	55	impl_splunk_gen

Linking to an external site

A workflow action can also link to an external site, using information from an event. Let's imagine that your organization has some other web-based tool. If that tool can accept arguments via `GET` or `POST` requests, then we can link directly to it from the Splunk results.

Create a new workflow action as we did in the previous example, but change **Action type** to **link**. The options change to those shown in the following screenshot:



Link configuration

URI *

http://webserver/toolx/search?ip=\$ip&user=\$user\$

Enter the location to link to. Optionally, specify fields by enclosing the field name in dollar signs, e.g. http://www.google.com/search?q=\$host\$.

Open link in

New window

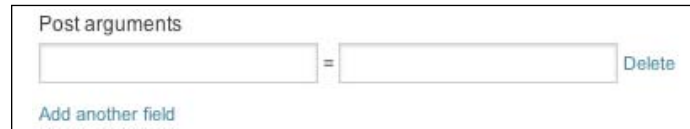
Link method

get

Splunk will encode any variables in the URL so that special characters survive. If you need a variable to not be encoded – for instance, if the value is actually part of the URL – add an exclamation point before the variable name, like this:

`$!user$`

If **Link method** is set to **post**, then more input fields appear, allowing you to specify post arguments like this:





Post arguments

[] = [] Delete

[Add another field](#)

Choosing this workflow action will open a new window with the URL we specified, either in the current window or in a new window according to the value of **Open link in**.

 The fields used by a workflow action can also come from automatic lookups. This is useful in cases where the external tool needs some piece of information that is not in your events, but can be derived from your events. 

Building a workflow action to show field context

Show Source is available as a workflow action on all events. When chosen, it runs a query that finds events around the current event for the same `source` and `host`. While this is very useful, sometimes it would be nice to see events that have something else in common besides `source`, and to see those events in the regular search interface, complete with the timeline and field picker.

To accomplish this, we will make a workflow action and macro that work in tandem to build the appropriate query. This example is fairly advanced, so don't be alarmed if it doesn't make a lot of sense.

Building the context workflow action

First, let's build our workflow action. As before, make a workflow action with **Action type** set to **search**.

Name *

Enter a unique name without spaces or special characters. This is used for identifying your workflow action later on within Splunk Manager.

Label *

Enter the label that appears for this action. Optionally, incorporate a field's value by enclosing the field name in dollar signs, e.g. 'Search for ticket number : \$ticketnum\$'.

Apply only to the following fields

Specify a comma-separated list of fields that must be present in an event for the workflow action to apply to it. When fields are specified, the workflow action only appears in the field menus for those fields; otherwise it appears in all field menus.

Apply only to the following event types

Specify a comma-separated list of event types that an event must be associated with for the workflow action to apply to it.

Show action in

Action type

Search configuration

Search string *

Enter the search for this action. Optionally, specify fields as \$fieldname\$, e.g. sourcetype=trails controller=\$controller\$ error=*

Run in app

Choose an app for the search to run in. Defaults to the current app.

Open in view

Enter the name of a view for the search to open in. Defaults to the current view.

Run search in


Time range

Earliest time Latest time

Use the same time range as the search that created the field listing

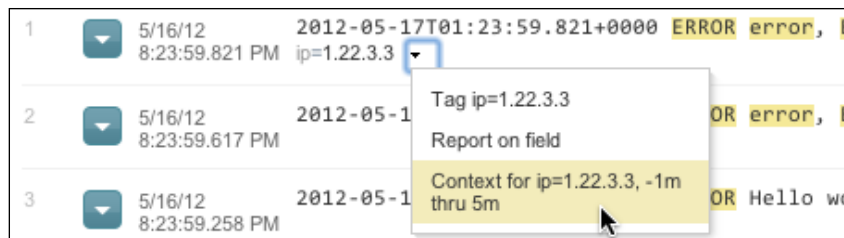
Let's step through our values, as follows:

- **Name:** This can be anything. Let's name it after our time frame.
- **Label:** This is what will appear in the menu. You may notice two special fields, @field_name and @field_value. These two fields only make sense when **Show action in** is set to **Fields menus**.

 There are a number of @variables available to workflow actions. Search <http://docs.splunk.com/> for Create workflow actions in Splunk to find complete documentation.

- **Apply only to the following fields:** This can be blank or * to indicate all fields.
- **Show action in:** We have chosen **Fields menus** in this case.
- **Action type:** We are running a search. It's a fairly strange search, as we are using a macro, but it is still technically a search.
- **Search string:** The fact that this query is a macro doesn't matter to the workflow action, ``context("$@field_name$", "$@field_value$", "$_time$", "-1m", "+5m")``. We will create the `context` macro next.
- **Run in app:** With nothing chosen, this macro will execute the search in the current app.
- **Open in view:** We want to make sure that our query executes in `flashtimeline`, so we explicitly set it.
- **Run search in:** We choose **New window**.
- **Time:** Contrary to the previous advice, we have left the time frame unspecified. We will be overriding the search times in the search itself. Anything specified here will be replaced.

After clicking on **Save**, the workflow action is available on all the field menus.



Choosing this menu item generates this search:

```
'context("ip", "1.22.3.3", "2012-05-16T20:23:59-0500", "-1m", "+5m")'
```

Let us consider our query definition:

```
'context("${@field_name}", "${@field_value}", "${_time}", "-1m", "+5m")'
```

We can see that the variables were simply replaced, and the rest of the query was left unchanged. `_time` is not in the format I would expect (I would have expected the epoch value), but we can work with it.

Building the context macro

When searching, you can specify the time ranges in the query itself. There are several fields that allow us to specify the time. They are as follows:

- `earliest`: This is the earliest time, inclusive. It can be specified as either a relative time or an epoch time in seconds.
- `latest`: This is the latest time, exclusive. Only events with a date *before* this time will be returned. This value can be specified as either a relative time or an epoch time in seconds.
- `now`: Using this field, you can redefine what relative values in `earliest` and `latest` are calculated against. It must be defined as epoch time in seconds.

Now, given our inputs, let's define our variable names:

- `field_name = ip`
- `field_value = 1.22.3.3`
- `event_time = 2012-05-16T20:23:59-0500`
- `earliest_relative = -1m`
- `latest_relative = +5m`

The query we want to run looks like this:


```
earliest=-1m latest=+5m now=[epoch event time] ip=1.22.3.3
```

The only value we don't have is `now`. To calculate this, there is a function available to `eval` called `strptime`. To test this function, let's use `|stats` to create an event, build an `event_time` field, and parse the value. Consider the following code:

```
|stats count
| eval event_time="2012-05-16T20:23:59-0500"
| eval now=strptime(event_time,"%Y-%m-%dT%H:%M:%S%z")
```


This gives us the following table:

	count ↕	event_time ↕	now ↕
1	0	2012-05-16T20:23:59-0500	1337217839.000000


 Good references for `strftime` formats can be found on modern Linux systems by running `man strftime` or `man date`, or by searching `google.com`. Splunk has several special extensions to `strftime` that can be found by searching for Enhanced `strftime()` support at <http://docs.splunk.com/>.

Now that we have our epoch value for `now`, we can build and test our query like this:

```
earliest=-1m latest=+5m now=1337217839 ip=1.22.3.3
```

This gives us a normal event listing, from one minute before our event to five minutes after our selected event, only showing events that have the field `ip` in common.

1	5/19/12 3:32:59.872 PM	2012-05-19T20:32:59.872+0000	DEBUG Hello world. [logger=LogoutClass, user=Bobby, ip=1.22.3.3]
2	5/19/12 3:32:58.715 PM	2012-05-19T20:32:58.715+0000	INFO Nothing happened. This is worthless. Don't log this. [l...
3	5/19/12 3:32:55.201 PM	2012-05-19T20:32:55.201+0000	INFO Nothing happened. This is worthless. Don't log this. [l...
4	5/19/12 3:32:54.444 PM	2012-05-19T20:32:54.444+0000	ERROR error, ERROR, Error! [user=jacky, ip=1.22.3.3, req_time...

Now that we have our search, and our `eval` statement for converting the value of `now`, we can actually build our macro in **Manager | Advanced search | Search macros | Add new**.

Name *
Enter the name of the macro. If the search macro takes an argument, indicate this by appending the number of arguments to the name. For example: `mymacro(2)`

Definition *
Enter the string the search macro expands to when it is referenced in another search. If arguments are included, enclose them in dollar signs. For example: `$arg1$`

Use eval-based definition?

Arguments
Enter a comma-delimited string of argument names. Argument names may only contain alphanumeric, `'` and `\"` characters.

This macro is using a few interesting features, as follows:

- Macros can take arguments. The number of arguments is specified in the name of the macro by appending ([argument count]) to the name of the macro. In this case, we are expecting five arguments.
- The definition of a macro can actually be an `eval` statement. This means we can use `eval` functions to build our query based on some value handed to the macro. In this case, we are using `strptime`. Things to note about this feature are as follows:
 - The `eval` statement is expected to return a string. If your statement fails, for some reason, to return a string, the user will see an error.
 - The variable names specified are replaced before the `eval` statement is executed. This means that there may be issues with escaping the values in the variables, so some care is required to make sure whether your value contains quotes or not as is expected.
- **Use eval-based definition?** is checked to indicate that this macro is expected to be parsed as an `eval` statement.
- In the **Arguments** field, we specify names for the arguments handed in. These are the names we refer to in the **Definition** field.

After clicking on **Save**, we have a working macro. You might make adjustments to this workflow action to better suit your needs. Let's change the definition to sort events by ascending time, and prevent searching across indexes. Change the workflow action definition **Search string** to:

```
'context("$@field_name$", "$@field_value$", "$_time$", "-1m", "+5m")'
  index=$index$ | reverse
```

Let's expand this just for clarity, like this:

```
'context("$@field_name$", "$@field_value$", "$_time$", "-1m", "+5m")'
  index=$index$ | reverse
'context("ip", "1.22.3.3", "2012-05-16T20:23:59-0500", "-1m", "+5m")'
  index=implsplunk | reverse
earliest=-1m latest=+5m now=1337217839 ip=1.22.3.3
  index=implsplunk | reverse
```

You can create multiple workflow actions that specify different time frames, or include other fields, for instance `host`.

Using external commands

The Splunk search language is extremely powerful, but at times, it may be either difficult or impossible to accomplish some piece of logic by using nothing but the search language. To deal with this, Splunk allows external commands to be written in Python. A number of commands ship with the product, and a number of commands are available in apps at <http://splunk-base.splunk.com/>.

Let's try out a few of the included commands. The documentation for the commands is included with other search commands at <http://docs.splunk.com/>. You can find a list of all included commands, both internal and external, by searching for `All search commands`. We will write our own commands in *Chapter 12, Extending Splunk*.

Extracting values from XML

Fairly often, machine data is written in XML format. Splunk will index this data without any issue, but it has no native support for XML. Though XML is not an ideal logging format, it can usually be parsed simply enough. Two commands are included in the search app that can help us pull fields out of XML.

xmlkv

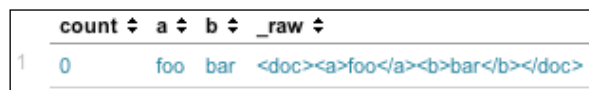
`xmlkv` uses regular expressions to create fields from tag names. For instance, given the following XML:

```
<doc><a>foo</a><b>bar</b></doc>
```

`xmlkv` will produce the fields `a=foo` and `b=bar`. To test, try this:

```
|stats count
| eval _raw="<doc><a>foo</a><b>bar</b></doc>"
| xmlkv
```

This produces a table, as shown in the following screenshot:



	count	a	b	_raw
1	0	foo	bar	<doc><a>foobar</doc>

As this command is using regular expressions its advantage is that malformed or incomplete XML statements will still produce results.



Using an external command is significantly slower than using the native search language, particularly if you are dealing with large sets of data. If it is possible to build the required fields using `rex` or `eval`, it will execute faster and it will introduce a smaller load on your Splunk servers. For instance, in the previous example, the fields could be extracted using:

```
| rex "<a.*?>(?!<a>.*?)" | rex "<b.*?>(?!<b>.*?)"
```

XPath

XPath is a powerful language for selecting values from an XML document. Unlike `xmlkv`, which uses regular expressions, XPath uses an XML parser. This means that the event must actually contain a valid XML document.

For example, consider the following XML document:

```
<d>
  <a x="1">foo</a>
  <a x="2">foo2</a>
  <b>bar</b>
</d>
```

If we wanted the value for the `a` tag whose `x` attribute equals 2, the XPath code would look like this:

```
//d/a[@x='2']
```

To test this, let's use our `|stats` trick to generate a single event and execute the `xpath` statement:

```
|stats count
| eval _raw="<d><a x='1'>foo</a><a x='2'>foo2</a><b>bar</b></d>"
| xpath outfield=a "//d/a[@x='2']"
```

This generates an output, as shown in the following screenshot:

count	a	_raw
1	foo2	<d>foofoo2bar</d>

`xpath` will also retrieve multivalued fields. For instance, this `xpath` statement simply says to find any `a` field:

```
|stats count
| eval _raw="<d><a x='1'>foo</a><a x='2'>foo2</a><b>bar</b></d>"
| xpath outfield=a "//a"
```

The result of this query is as shown:

	count	a	_raw
1	0	foo	<d>foofoo2bar</d>
		foo2	

There are many XPath references available online. My favorite quick reference is at the Mulberry Technologies website: <http://www.mulberrytech.com/quickref/xpath2.pdf>.

Using Google to generate results

External commands can also act as data generators, similar to the `stats` command that we used to create test events. There are a number of these commands, but let's try a fun example, `google`. This command takes one argument, a search string, and returns the results as a set of events. Let's execute a search for `splunk`:

```
|google "splunk"
```

This produces a table, as shown in the following screenshot:

	_time	url	title	description
1	5/19/12 10:42:37.000 PM	http://www.splunk.com/	Splunk	Splunk indexes and m
2	5/19/12 10:42:37.000 PM	http://www.splunk.com/download	Download Splunk for free on your operating system	Index up to 500/mb a
3	5/19/12 10:42:37.000 PM	http://www.splunk.com/view/careers/SP-CAAAAGG	Splunk Jobs at Splunk	Careers. We're
4	5/19/12 10:42:37.000 PM	http://en.wikipedia.org/wiki/Splunk	Splunk - Wikipedia, the free encyclopedia	Splunk is a software t
5	5/19/12 10:42:37.000 PM	http://www.splunk.com/company	Splunk A Different Kind of Software Company	Every Company is a

This example may not be terribly useful, but you can probably think of external sources that you would like to query as a starting point, or even to populate a subsearch for another Splunk query. We'll write an example data generator in *Chapter 12, Extending Splunk*.

Summary

In this chapter, we quickly covered tags, event types, lookups, macros, workflow actions, and external commands. I hope these examples and discussions will serve as starting points for your apps. More examples can be found in the official Splunk documentation at <http://docs.splunk.com/> and at <http://splunk-base.splunk.com/>.

In the next chapter, we will dive into creating and customizing our own apps.

7

Working with Apps

In this chapter, we will explore what makes up a Splunk app. We will:

- Inspect included apps
- Install apps from Splunkbase
- Build our own app
- Customize app navigation
- Customize app look and feel

Defining an app

In the strictest sense, an **app** is a directory of configurations and, sometimes, code. The directories and files inside have a particular naming convention and structure. All configurations are in plain text, and can be edited using your choice of text editor.

Apps generally serve one or more of the following purposes:

1. **A container for searches, dashboards, and related configurations:** This is what most users will do with apps. This is not only useful for logical grouping, but also for limiting what configurations are applied and at what time. This kind of app usually does not affect other apps.
2. **Providing extra functionality:** Many objects can be provided in an app for use by other apps. These include field extractions, lookups, external commands, saved searches, workflow actions, and even dashboards. These apps often have no user interface at all; instead they add functionality to other apps.

3. **Configuring a Splunk installation for a specific purpose:** In a distributed deployment, there are several different purposes that are served by the multiple installations of Splunk. The behavior of each installation is controlled by its configuration, and it is convenient to wrap those configurations into one or more apps. These apps completely change the behavior of a particular installation.

Included apps

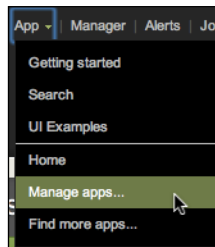
Without apps, Splunk has no user interface, rendering it essentially useless. Luckily, Splunk comes with a few apps to get us started. Let's look at a few of these apps:

- **gettingstarted:** This app provides the help screens that you can access from the launcher. There are no searches, only a single dashboard that simply includes an HTML page.
- **search:** This is the app where users spend most of their time. It contains the main search dashboard that can be used from any app, external search commands that can be used from any app, admin dashboards, custom navigation, custom css, a custom app icon, a custom app logo, and many other useful elements.
- **splunk_datapreview:** This app provides the data preview functionality in the admin interface. It is built entirely using JavaScript and custom REST endpoints.
- **SplunkDeploymentMonitor:** This app provides searches and dashboards to help you keep track of your data usage and the health of your Splunk deployment. It also defines indexes, saved searches, and summary indexes. It is a good source for more advanced search examples.
- **SplunkForwarder** and **SplunkLightForwarder:** These apps, which are disabled by default, simply disable portions of a Splunk installation so that the installation is lighter in weight. We will discuss these in greater detail in *Chapter 11, Advanced Deployments*.

If you never create or install another app, and instead simply create saved searches and dashboards in the app `search`, you can still be quite successful with Splunk. Installing and creating more apps, however, allows you to take advantage of others' work, organize your own work, and ultimately share your work with others.

Installing apps

Apps can either be installed from Splunkbase or uploaded through the admin interface. To get started, let's navigate to **Manager | Apps**, or choose **Manage apps...** from the **App** menu as shown in the following screenshot:




Installing apps from Splunkbase

If your Splunk server has direct access to the Internet, you can install apps from Splunkbase with just a few clicks. Navigate to **Manager | Apps** and click on **Find more apps online**. The most popular apps will be listed as follows:


Find more apps
Browse to find more apps to get the most out of your Splunk experience.

1 2 3 4 5 6 7 8 9 10 next»

 **Splunk App for Windows**
The Splunk App for Windows provides examples of pre-built data inputs, searches, reports, alerts, and dashboards for Windows server and desktop management... [Read more](#)


Author: Splunk Version: 4.5.2 Last updated: 05/22/12 Downloads: 45467 License: Splunk Master Software License Agreement

[Install free](#)

 **Splunk for Unix and Linux**
Splunk for *nix provides pre-built data inputs, searches, reports, alerts and dashboards for Linux and Unix management... [Read more](#)


Author: Splunk Version: 4.5 Last updated: 12/09/11 Downloads: 35457 License: Splunk Master Software License Agreement

[Install free](#)

 **Google Maps**
Google Maps for Splunk adds a geo-visualization module based on the Google Maps API and allows you to quickly plot geographical information on a map... [Read more](#)


Author: ziegfried Version: 1.1 Last updated: 06/14/11 Downloads: 14190 License: Creative Commons BY-NC-SA 2.5

[Install free](#)

 **PDF Report Server - install on Linux only**
The PDF Report Server add-on enables your Linux-based Splunk instance to generate emailed reports in PDF format... [Read more](#)

Author: Splunk Version: 1.3 Last updated: 01/04/12 Downloads: 10826 License: Splunk Master Software License Agreement

[Install free](#)

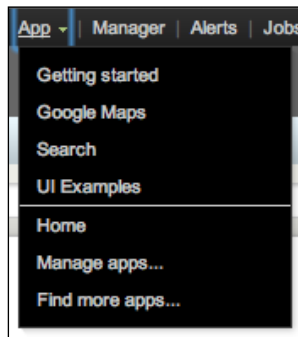
 **Geo Location Lookup Script (powered by MAXMIND)**
Splunk for Use with MAXMIND is an application that provides geo_ip information on any public IP in your Splunk DB in a scalable fashion... [Read more](#)

Author: will Version: 1.0.6 Last updated: 06/10/11 Downloads: 10436 License: Creative Commons BY-NC-SA 2.5

Let's install a pair of apps and have a little fun. First, install **Geo Location Lookup Script (powered by MAXMIND)** by clicking on the **Install free** button. You will be prompted for your `splunk.com` login. This is the same login that you created when you downloaded Splunk. If you don't have an account, you will need to create one.

Next, install the **Google Maps** app. This app was built by a Splunk customer and contributed back to the Splunk community. This app will prompt you to restart Splunk.

Once you have restarted and logged back in, check the **App** menu.

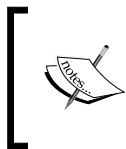


Google Maps is now visible, but where is **Geo Location Lookup Script**? Remember that not all apps have dashboards; nor do they necessarily have any visible components at all.

Using Geo Location Lookup Script

Geo Location Lookup Script provides a lookup script to provide geolocation information for IP addresses. Looking at the documentation, we see this example:


```
eventtype=firewall_event | lookup geoip clientip as src_ip
```



You can find the documentation for any Splunkbase app by searching for it at `splunkbase.com`, or by clicking on **Read more** next to any installed app by navigating to **Manager | Apps | Browse more apps**.

Let's read through the arguments of the `lookup` command:

- `geoip`: This is the name of the lookup provided by **Geo Location Lookup Script**.

 You can see the available lookups by going to **Manager | Lookups | Lookup definitions.**

- `clientip`: This is the name of the field in the lookup that we are matching against.
- `as src_ip`: This says to use the value of `src_ip` to populate the field before it; in this case, `clientip`. I personally find this wording confusing. In my mind, I read this as "using" instead of "as".

Included in the *ImplementingSplunkDataGenerator* app (available at <http://packtpub.com/>) is a `sourcetype` instance named `impl_splunk_ips`, which looks like this:

```
2012-05-26T18:23:44 ip=64.134.155.137
```

The IP addresses in this fictitious log are from one of my websites. Let's see some information about these addresses:

```
sourcetype="impl_splunk_ips"
| lookup geoip clientip AS ip
| top client_country
```

This gives us a table similar to the one shown in the following screenshot:

	client_country ↕	count ↕	percent ↕
1	United States	447	71.634615
2	China	90	14.423077
3	Russian Federation	39	6.250000
4	Slovenia	15	2.403846
5	United Kingdom	14	2.243590
6	Ukraine	9	1.442308
7	South Africa	3	0.480769
8	Germany	2	0.320513
9	United Arab Emirates	1	0.160256
10	Turkey	1	0.160256

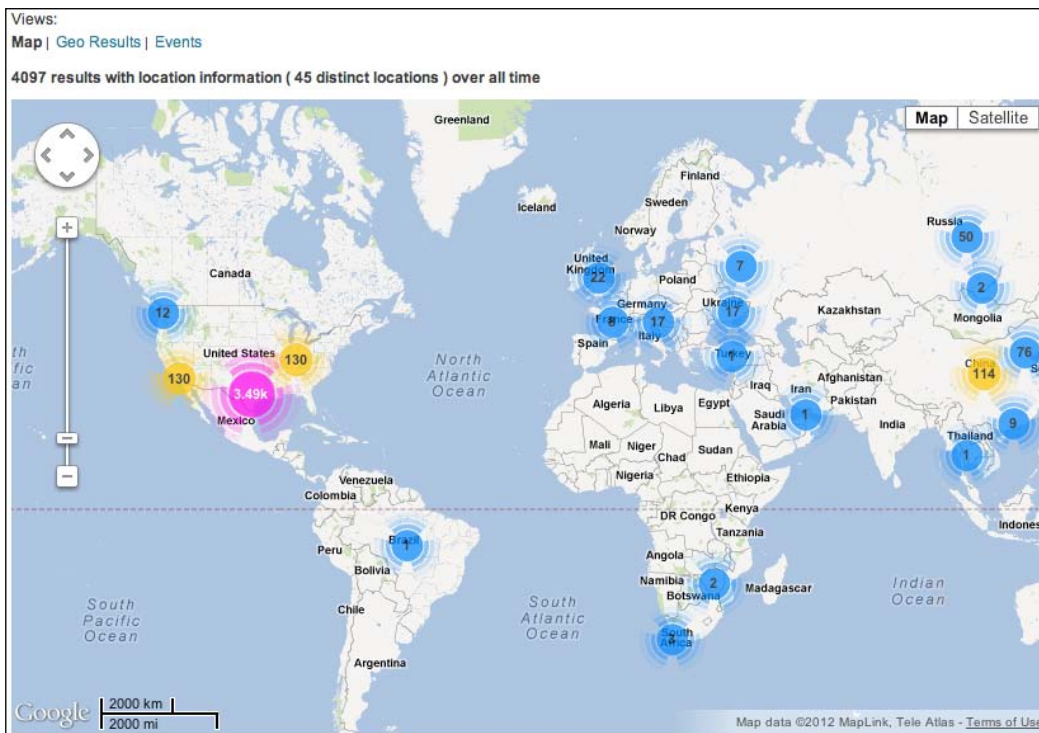
That's interesting. I wonder who is visiting my site from Slovenia!

Using Google Maps

Now let's do a similar search in the Google Maps app. Choose **Google Maps** from the **App** menu. The interface looks like the standard search interface, but with a map instead of an event listing. Let's try this remarkably similar (but not identical) query using a lookup provided in the **Google Maps** app:

```
sourcetype="impl_splunk_ips"  
| lookup geo ip
```

The map generated looks like this:



Unsurprisingly, most of the traffic to this little site came from my house in Austin, Texas. We'll use the Google Maps app for something more interesting in *Chapter 8, Building Advanced Dashboards*.

Installing apps from a file

It is not uncommon for Splunk servers to not have access to the Internet, particularly in a datacenter. In this case, follow these steps:

1. Download the app from `splunkbase.com`. The file will have a `.spl` or `.tgz` extension.
2. Navigate to **Manager | Apps**.
3. Click on **Install app from file**.
4. Upload the downloaded file using the form provided.
5. Restart if the app requires it.
6. Configure the app if required.

That's it. Some apps have a configuration form. If this is the case, you will see a **Set up** link next to the app when you go to **Manager | Apps**. If something goes wrong, contact the author of the app.



If you have a distributed environment, in most cases the app only needs to be installed on your search head. The components that your indexers need will be distributed automatically by the search head. Check the documentation for the app.

Building your first app

For our first app, we will use one of the templates provided with Splunk. To get started, navigate to **Manager | Apps** and then click on **Create app**. The following page will open:

Add new

Name

Give your app a friendly name for display in Splunk Web.

Folder name *

This name maps to the app's directory in \$SPLUNK_HOME/etc/apps/.

Visible
 No Yes
Only apps with views should be made visible.

Author

Name of the app's owner.

Description

Enter a description for your app.

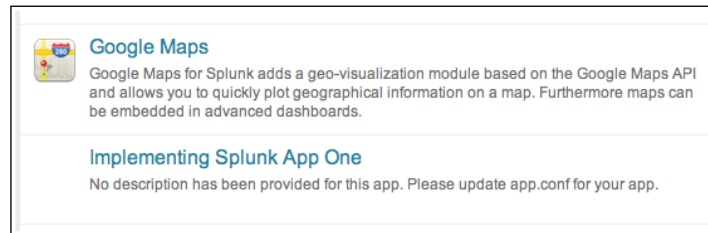
Template

This template contains example views and searches.

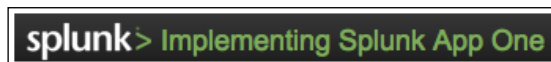
Set the fields as follows:

- Set **Name** to `Implementing Splunk App One`. This name will be visible on the home screen, in the **App** menu, and in the app banner in the upper left of the window.
- Set **Folder name** to `is_app_one`. This value will be the name of the app directory on the filesystem, so you should limit your name to letters, numbers, and underscores.
- Set **Visible** to **Yes**. If your app simply provides resources for other apps to use, there may be no reason for it to be visible.
- Set **Template** to **barebones**. The **barebones** template contains sample navigation and the minimal configuration required by an app. The **sample_app** template contains many example dashboards and configurations.

After clicking on **Save**, we can now visit our app by going to **Manager | Apps**, in the **App** menu, and in the **Home** app.



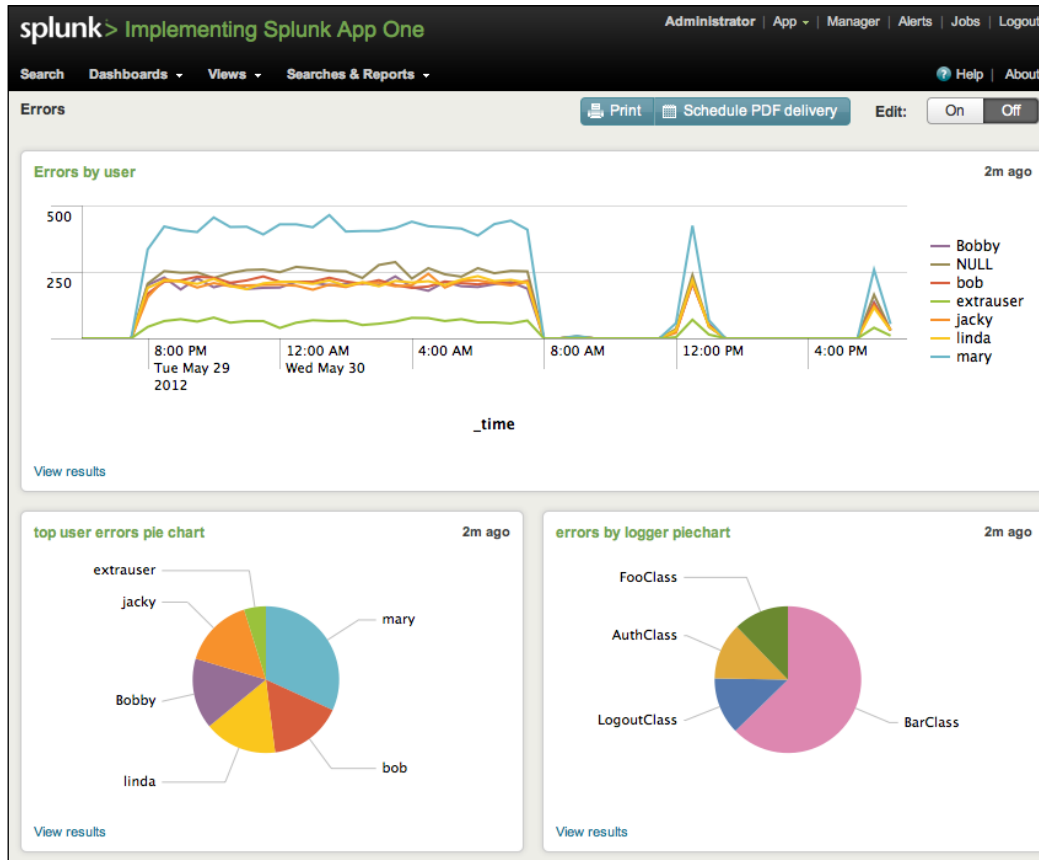
Now that we have our app, we can create searches and dashboards, and maintain them in our app. The simplest way to ensure that your objects end up in your app is to verify that the app banner is correct before creating objects or before entering the Splunk Manager. Our app banner looks like this:



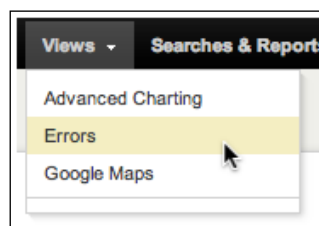
Create a dashboard called `Errors` using the following searches (refer back to *Chapter 4, Simple XML Dashboards*, for detailed instructions):

```
error sourcetype="impl_splunk_gen" | timechart count by user
error sourcetype="impl_splunk_gen" | top user
error sourcetype="impl_splunk_gen" | top logger
```

This produces the following result:



The searches appear under **Searches & Reports**, and our new dashboard appears in the navigation menu under **Views**:



Editing navigation

Navigation is controlled by an XML file that can be accessed by going to **Manager | User interface | Navigation menus**.

Nav name ↕	Owner ↕	App ↕	Sharing ↕	Status ↕
default	No owner	is_app_one	App Permissions	Enabled

There can only be one active navigation file per app, and it is always called default. After clicking on the name, we see the XML provided by the **barebones** template:

```
<nav>
  <view name="flashtimeline" default='true' />
  <collection label="Dashboards">
    <view source="unclassified" match="dashboard"/>
    <divider />
  </collection>
  <collection label="Views">
    <view source="unclassified" />
    <divider />
  </collection>
  <collection label="Searches & Reports">
    <collection label="Reports">
      <saved source="unclassified" match="report" />
    </collection>
    <divider />
    <saved source="unclassified" />
  </collection>
</nav>
```

The structure of the XML is essentially the following:

```
nav
  view
  saved
  collection
    view
    a href
    saved
    divider
  collection
  ...
```

The logic of navigation is probably best absorbed by simply editing it and seeing what happens. You should keep a backup, as this XML is somewhat fragile and Splunk does not provide any kind of version control. Here are some general details about `nav`:

- Children of `nav` appear in the navigation bar.
- `collection`: Children of `collection` tags appear in a menu or submenu. If the child tags do not produce any results, the menu will not appear. The `divider` tag always produces a result, so it can be used to ensure that a menu appears.
- `view`: This tag represents a dashboard, with the following attributes:
 - `name` is the name of the dashboard filename, without `.xml`.
 - The first `view` element with the attribute `default='true'` will load automatically when the app is selected.
 - The label of each `view` is based on the contents of the `label` tag in the dashboard XML, not the name of the dashboard filename.
 - `match="dashboard"` selects all dashboards whose filename contains `dashboard`. If you want to group dashboards, you may want to follow a naming convention to make grouping more predictable.
 - `source="unclassified"` essentially means "all views that have not been previously associated to a menu". In other words, this will match dashboards that were not explicitly referenced by `name` or matched using the `match` attribute or a different `view` tag.
- `a href`: You can include standard HTML links of the form ``. The link is untouched and passed along as written.
- `saved`: This tag represents a saved search, with the following attributes:
 - `name` is equal to the name of a saved search.
 - `match="report"` selects all saved searches that have `report` in their names.
 - `source="unclassified"` essentially means "all searches that have not yet been previously associated to a menu". In other words, this will match searches that were not explicitly referenced by `name` or matched using the `match` attribute or a different `saved` tag.

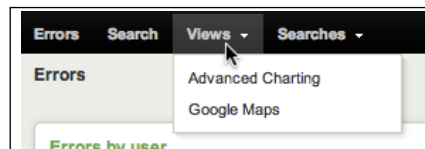
Let's customize our navigation. We'll make a few changes like these:

- Create an entry specifically for our `errors` dashboard
- Add `default='true'` so that this dashboard loads by default
- Simplify the `Views` and `Searches` collections

These changes are reflected in the following code:

```
<nav>
  <view name="errors" default='true' />
  <view name="flashtimeline" />
  <collection label="Views">
    <view source="unclassified" />
  </collection>
  <collection label="Searches">
    <saved source="unclassified" />
  </collection>
</nav>
```

Our navigation now looks like this screenshot:



With this navigation in place, all new dashboards will appear under **Views**, and all new saved searches will appear under **Searches**.

Notice that **Advanced Charting** and **Google Maps** appear under **Views**. Neither of these dashboards are part of our app, but are visible because of the permissions in their respective apps. We will discuss permissions in more detail in the *Object permissions* section.

Customizing the appearance of your app

It is helpful to further customize the appearance of your application, if for no other reason than to make it more obvious which app is currently active.

Customizing the launcher icon

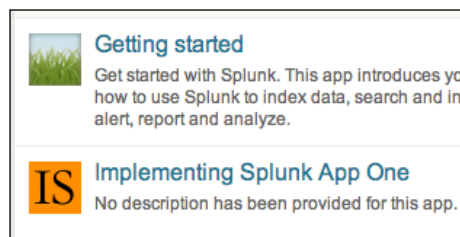
The launcher icon is seen both in the Home app and in Splunkbase, if you decide to share your app. The icon is a 36 x 36 PNG file named `appIcon.png`. I have created a simple icon for our sample app (please don't judge my art skills):



To use the icon follow these steps:

1. Navigate to **Manager | Apps**.
2. Click on **Edit properties** next to our app, **Implementing Splunk App One**.
3. Click on **Upload asset** and select the file.
4. Click on **Save**.

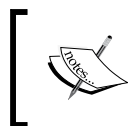
Our icon will now appear on the launcher screen, like in the following screenshot:



Using custom CSS

The look of the Splunk application is controlled via CSS. One common element to change is the application icon in the application bar. Follow these steps to do just that:

1. First, create a file named `application.css`. This file will be loaded on every dashboard of the application containing it. The CSS is listed later in this section.



As of Splunk Version 4.3.2, the first time `application.css` is added to an app of Version 4.3.2, a restart is required before the file is served to the users. Subsequent updates do not require a restart.

2. Next, create a file named `appLogo.png`. This file can be called anything, as we will reference it explicitly in our CSS file. Borrowing CSS from the search app, we will make our file 155 x 43 pixels:



3. For each file, follow the same steps as for uploading the launcher icon:
 1. Navigate to **Manager | Apps**.
 2. Click on **Edit properties** next to our app, **Implementing Splunk App One**.
 3. Click on **Upload asset** and select the file.
 4. Click on **Save**.

Our CSS references a few classes in the application header bar:

```
.appHeaderWrapper h1 {
  display: none;
}

.appLogo {
  height: 43px;
  width: 155px;
  padding-right: 5px;
  float: left;
  background: url(appLogo.png) no-repeat 0 0;
}

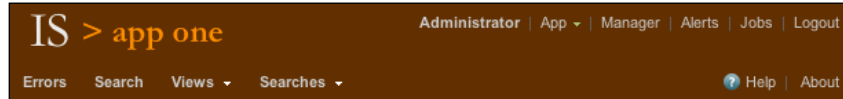
.appHeaderWrapper {
  background: #612f00;
}
```

Let's step through these classes:

- `.appHeaderWrapper h1`: By default, the name of the app appears as text in the upper-left corner. This definition hides that text.

- `.appLogo`: This sets the background of the upper-left block to our custom file. The height and width should match the dimensions of our logo.
- `.appHeaderWrapper`: This sets the background color of the top bar.

With everything in place, our top bar now looks like this:



Using custom HTML

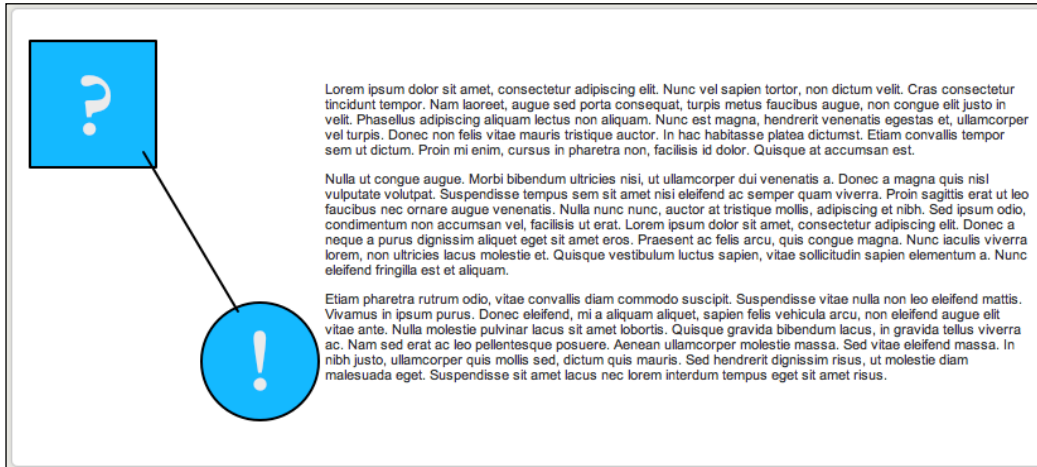
In some apps, you will see static HTML blocks. This can be accomplished using both simple and complex dashboards.

Custom HTML in a simple dashboard

In a simple dashboard, you can simply insert an `<html>` element inside a `<row>` element, and include static HTML inline. For example, after uploading an image named `graph.png`, the following block can be added to any dashboard:

```
<row>
  <html>
    <table>
      <tr>
        <td></td>
        <td>
          <p>Lorem ipsum ...</p>
          <p>Nulla ut congue ...</p>
          <p>Etiam pharetra ...</p>
        </td>
      </tr>
    </table>
  </html>
</row>
```

The XML would render this panel:



This approach has the advantage that no other files are needed. The disadvantage, however, is that you cannot build the HTML document in an external program and upload it untouched.

You could also reference custom CSS using this method by adding classes to `application.css` and then referencing those classes in your HTML block.

Using ServerSideInclude in a complex dashboard

You can also develop static pages as HTML documents, referencing other files in the same directory. Let's build a slightly more complicated page using `graph.png`, but also a style from `application.css` as follows:

1. Place `graph.png` and `application.css` into a directory.
2. Create a new HTML file. Let's name it `intro.html`.
3. Add any styles for your page to `application.css`.
4. Upload the new HTML file and modified CSS file.
5. Create the dashboard referencing the HTML file.

Starting with the HTML from our previous example, let's make it a complete document: move the image to a CSS style and add a class to our text, like this:

```
<html>
<head>
  <link rel="stylesheet" type="text/css"
    href="application.css" />
```

```
</head>
<body>
  <table>
    <tr>
      <td class="graph_image"></td>
      <td>
        <p class="lorem">Lorem ipsum ...</p>
        <p class="lorem">Nulla ut congue ...</p>
        <p class="lorem">Etiam pharetra ...</p>
      </td>
    </tr>
  </table>
</body>
</html>
```

Maintaining the classes for the navigation bar, add our page classes to application.css, like this:

```
.appHeaderWrapper h1 {
  display: none;
}

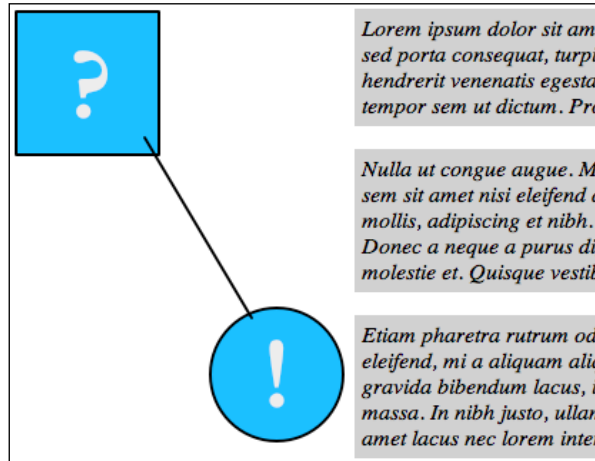
.appLogo {
  height: 43px;
  width: 155px;
  padding-right: 5px;
  float: left;
  background: url(appLogo.png) no-repeat 0 0;
}

.appHeaderWrapper {
  background: #612f00;
}

.lorem {
  font-style: italic;
  background: #CCCCCC;
  padding: 5px;
}

.graph_image {
  height: 306px;
  width: 235px;
  background: url(graph.png) no-repeat 0 0;
}
```

We can now open this file in a browser. Clipped for brevity, the page looks like this:



To include this external HTML document, we have to use advanced XML. We will cover advanced XML more thoroughly in *Chapter 8, Building Advanced Dashboards*.

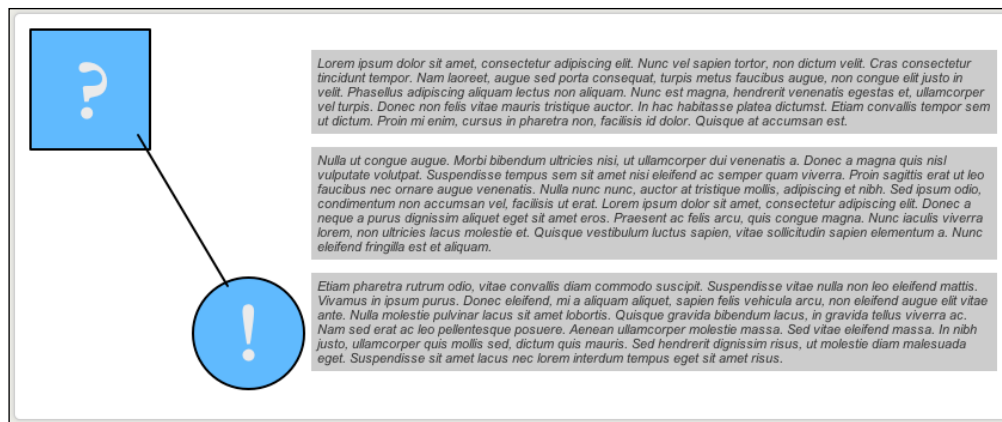
First, build a minimal dashboard like this:

```
<view template="dashboard.html">
  <label>Included</label>
  <!-- chrome here -->
  <module
    name="ServerSideInclude"
    layoutPanel="panel_row1_col1">
    <param name="src">intro.html</param>
  </module>
</view>
```



All "simple" XML dashboards are converted to "advanced" XML behind the scenes. We will take advantage of this later.

Now upload our files as we did before under the *Customizing the launcher icon* section. The page should render nearly identically as the file did in the browser, with the addition of the border around the panel:



A few things to note from this overly simplified example are as follows:

1. Your CSS classes may end up merging with styles included by Splunk in unexpected ways. Using the developer tools in any modern browser will help greatly.
2. The navigation and dashboard title were excluded for brevity. They would normally go where we see `<!-- chrome here -->`. This is interesting because there are cases where you would want to exclude the navigation; something that cannot be done with simple XML.
3. The static files, such as `application.css`, can be edited directly on the filesystem, and the changes will be seen immediately. This is not true of the dashboard XML file. We will cover these locations later in the *App directory structure* section.

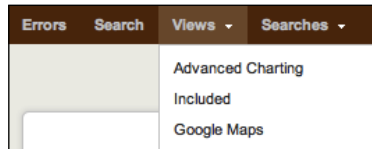
Object permissions

Almost all objects in Splunk have permissions associated with them. The permissions essentially have the following three options:

- **Private:** Only the user that created the search can see or use the object, and only in the app where it was created
- **App:** All users that have permission to read an object may use that object in the context of the app that contains that object
- **Global:** All users that have permission to read an object may use that object in any app

How permissions affect navigation

To see a visible instance of permissions in action, let's look at our navigation. In our application, **Implementing Splunk App One**, our navigation looks like this:



If you recall the navigation XML we built before, this menu is controlled by the following XML:

```
<collection label="Views">
  <view source="unclassified" />
</collection>
```

There is no mention of any of these dashboards. Here is where they are coming from:

- **Advanced Charting** is inherited from the **Search** app. Its permissions are set to **Global**.
- **Included** is from this app. Its permissions are set to **App**.
- **Google Maps** is inherited from the **Google Maps** app. Its permissions are set to **Global**.



If the permissions of a dashboard or search are set to **Private**, a green dot appears next to the name in the navigation.

Dashboards or searches shared from other apps can also be referenced by name. For example, most apps, including ours, will include a link to `flashtimeline`, which appears as **Search**, the label in that dashboard's XML:

```
<view name="flashtimeline" />
```

This allows us to use this dashboard in the context of our app so that all of the other objects that are scoped solely to our app will be available.

How permissions affect other objects

Almost everything you create in Splunk has permissions. To see all objects, navigate to **Manager** | **All configurations**.

splunk> Manager » All configurations

App context: Search (search) Owner: Any

Showing 1-25 of 253 items

Name	Config type	Owner	App	Sharing	Status	Actions
_admin	views	No owner	system	Global Permissions	Enabled	
access-extractions	transforms-extract	No owner	system	Global Permissions	Enabled Disable	
access-request	transforms-extract	No owner	system	Global Permissions	Enabled Disable	
access_combined : REPORT-access	props-extract	No owner	system	Global Permissions	Enabled	
access_combined_wcookie : REPORT-access	props-extract	No owner	system	Global Permissions	Enabled	
access_common : REPORT-access	props-extract	No owner	system	Global Permissions	Enabled	
ActiveDirectory : EXTRACT-GUID	props-extract	No owner	system	Global Permissions	Enabled	
ActiveDirectory : EXTRACT-SID	props-extract	No owner	system	Global Permissions	Enabled	
ActiveDirectory : REPORT-MESSAGE	props-extract	No owner	system	Global Permissions	Enabled	
ad-kv	transforms-extract	No owner	system	Global Permissions	Enabled Disable	

Everything with the value **system** in the **App** column ships with Splunk. These items live in `$SPLUNK_HOME/etc/system`. We will cover these different configuration types in *Chapter 10, Configuring Splunk*, but the important takeaway is that the **Sharing** settings affect nearly everything.

When you create new objects and configurations, it is important to share all related objects. For instance, in *Chapter 6, Extending Search*, we created lookups. It is important that all three parts of the lookup definition are shared appropriately, or users will be presented with error messages.

Correcting permission problems

If you see errors about permissions, it is more than likely that some object still has **Sharing** set to **Private**, or is shared at the **App** level but needs to be **Global**. Follow these steps to find the object:

1. Navigate to **Manager | All configurations**.
2. Change **App context** to **All**.
3. Sort by using the **Sharing** status. Click twice so that **Private** objects come to the top.
4. If there are too many items to look through, filter the list by adding terms to the search field in the upper-right corner, or changing the **App context** value.

5. Fix the permissions appropriately. In most cases, the permissions you want will look like this:

Saved eventtype should appear in
 Keep private This app only (search) All apps

Permissions

Roles	Read	Write
Everyone	<input checked="" type="checkbox"/>	<input type="checkbox"/>
admin	<input type="checkbox"/>	<input checked="" type="checkbox"/>
can_delete	<input type="checkbox"/>	<input type="checkbox"/>
power	<input type="checkbox"/>	<input type="checkbox"/>
user	<input type="checkbox"/>	<input type="checkbox"/>



You should choose **All apps** with care. For instance, when building a lookup, it is common to share the lookup table file and lookup definition across all apps. This allows the lookup to be used in searches by other apps. It is less common to share the Automatic lookup, as this can affect performance in other apps in unforeseen ways.

App directory structure

If you do much beyond building searches and dashboards, sooner or later you will need to edit files in the filesystem directly. All apps live in `$$SPLUNK_HOME/etc/apps/`. On Unix systems, the default installation directory is `/opt/splunk`. On Windows, the default installation directory is `c:\Program Files\Splunk`. This is the value that `$$SPLUNK_HOME` will inherit on startup.

Stepping through the most common directories, we have:

- `appserver`: This directory contains files that are served by the Splunk web app. The files that we uploaded in earlier sections of this chapter are stored in `appserver/static`.
- `bin`: This is where command scripts belong. These scripts are then referenced in `commands.conf`. This is also a common location for scripted inputs to live, though they can live anywhere.

- `default` and `local`: These two directories contain the vast majority of the configurations that make up an app. We will discuss these configurations and how they merge in *Chapter 10, Configuring Splunk*. Here is a brief look:
 - Newly created, unshared objects live in `$SPLUNK_HOME/etc/users/USERNAME/APPNAME/local`.
 - Once an object is shared at the App or Global level, the object is moved to `$SPLUNK_HOME/etc/APPNAME/local`.
 - Files in `local` take precedence over its equivalent value in `default`.
 - Dashboards live in `(default|local)/data/ui/views`.
 - Navigations lives in `(default|local)/data/ui/nav`.
 - When editing files by hand, my general rule of thumb is to place configurations in `local` unless the app will be redistributed. We'll discuss this in more detail in the *Adding your app to Splunkbase* section.
- `lookups`: Lookup files belong in this directory. They are then referenced in `(default|local)/transforms.conf`.
- `metadata`: The files `default.meta` and `local.meta` in this directory tell Splunk how configurations in this app should be shared. It is generally much easier to edit these settings through the **Manager** interface.

Let's look at the contents of our `is_app_one` app, which we created earlier:

```

appserver/static/appIcon.png
appserver/static/application.css
appserver/static/appLogo.png
appserver/static/graph.png
appserver/static/intro.html
bin/README
default/app.conf
default/data/ui/nav/default.xml
default/data/ui/views/README
local/app.conf
local/data/ui/nav/default.xml
local/data/ui/views/errors.xml
local/data/ui/views/included.xml
local/savedsearches.conf
local/viewstates.conf
metadata/default.meta
metadata/local.meta

```

The file `metadata/default.meta`, and all files in `default/`, were provided in the template app. We created all of the other files. With the exception of the `png` files, all files are plain text.

Adding your app to Splunkbase

Splunkbase (splunkbase.com) is a wonderful community-supported site that Splunk put together for users and Splunk employees alike to share Splunk apps. The apps on Splunkbase are a mix of fully realized apps, add-ons of various sorts, and just example code. Splunk has good documentation for sharing apps at the following URL:

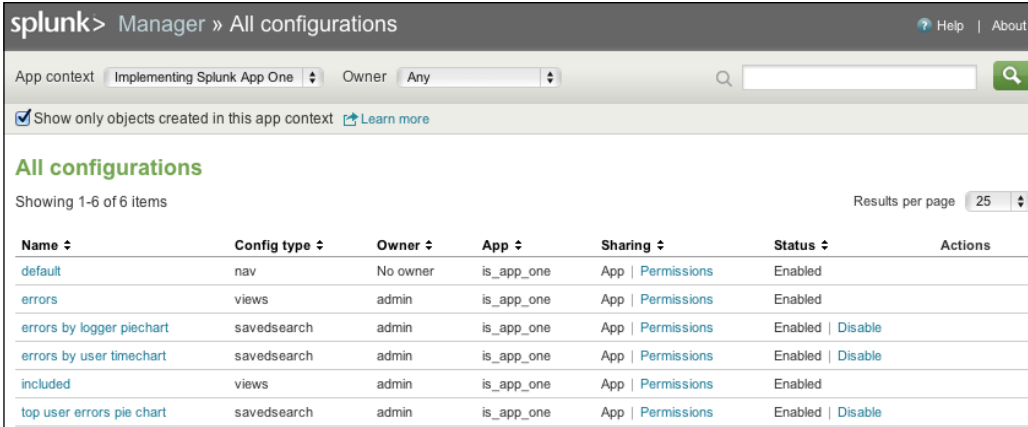
<http://docs.splunk.com/Documentation/Splunk/latest/Developer/ShareYourWork>

Preparing your app

Before we upload our app, we need to make sure all of our objects are shared properly, move our files to `default`, and configure `app.conf`.

Confirming sharing settings

To see sharing settings for all our objects, navigate to **Manager | All configurations** and set the **App** context option:



Name	Config type	Owner	App	Sharing	Status	Actions
default	nav	No owner	is_app_one	App Permissions	Enabled	
errors	views	admin	is_app_one	App Permissions	Enabled	
errors by logger piechart	savedsearch	admin	is_app_one	App Permissions	Enabled Disable	
errors by user timechart	savedsearch	admin	is_app_one	App Permissions	Enabled Disable	
included	views	admin	is_app_one	App Permissions	Enabled	
top user errors pie chart	savedsearch	admin	is_app_one	App Permissions	Enabled Disable	

In the case of a self-contained app like ours, all objects should probably be set to **App** under **Sharing**. If you are building an app to share lookups or commands, the value should be **Global**.

Cleaning up our directories

When you upload an app, you should move everything out of `local` and into `default`. This is important because all changes a user makes will be stored in `local`. When your app is upgraded, all files in the app will be replaced, and the user's changes will be lost. The following Unix commands illustrate what needs to be done:

1. First, let's copy our app to another location, perhaps `/tmp`:

```
cp -r $SPLUNK_HOME/etc/apps/is_app_one /tmp/
```
2. Next, let's move everything from `local` to `default`. In the case of `.xml` files, we can simply move the files; but `.conf` files are a little more complicated, and we need to merge them manually. The following code does this:

```
cd /tmp/is_app_one
mv local/data/ui/nav/*.xml default/data/ui/nav/
mv local/data/ui/views/*.xml default/data/ui/views/
#move conf files, but don't replace conf files in default
mv -n local/*conf default/
```
3. Now we need to merge any `.conf` files that remain in `local`. The only configuration we have left is `app.conf`;

local/app.conf	default/app.conf
[ui]	[install]
	is_configured = 0
[launcher]	
	[ui]
[package]	is_visible = 1
check_for_updates = 1	label = Implementing Splunk App One
	[launcher]
	author =
	description =
	version = 1.0

Configuration merging is additive, with any values from `local` added to the values in `default`. In this case, the merged configuration would be as follows:

```
[install]
is_configured = 0

[ui]
```

```
is_visible = 1
label = Implementing Splunk App One
```

```
[launcher]
author =
description =
version = 1.0
```

```
[package]
check_for_updates = 1
```

4. Place this merged configuration in `default/app.conf` and delete `local/app.conf`.

We will cover configuration merging extensively in *Chapter 10, Configuring Splunk*.

Packaging your app

To package an app, we need to be sure that there are a few values in `default/app.conf`, and only then build the archive.

First, edit `default/app.conf` like this:

```
[install]
is_configured = 0
build = 1

[ui]
is_visible = 1
label = Implementing Splunk App One

[launcher]
author = My name
description = My great app!
version = 1.0

[package]
check_for_updates = 1
id = is_app_one
```

`build` is used in all URLs, so it should be incremented to defeat browser caching. `id` should be a unique value in Splunkbase—you will be alerted if the value is not unique.

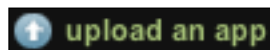
Next, we need to build a `tar` file compressed with `gzip`. With a modern version of `tar`, the command is simply the following:

```
cd /tmp
tar -czvf is_app_one.tgz is_app_one
#optionally rename as spl
mv is_app_one.tgz is_app_one.spl
```

The Splunk documentation (<http://docs.splunk.com/Documentation/Splunk/latest/AdvancedDev/PackageApp>) covers this extensively, including Mac and Windows procedures.

Uploading your app

Now that we have our archive, all we have to do is send it up to Splunkbase. First, click on the **upload an app** button.



Then fill out the form shown in the following screenshot:

Upload Your App or Add-on to Splunkbase

Name

Description

Tags (separated by spaces, e.g. 'windows ad performance')

Splunk Compatibility
Which version of Splunk is your app compatible with?

4.x
 4.3
 4.2
 4.1
 4.0
 3.x

Type

- **Apps** package together Splunk features like saved searches, dashboards and inputs into their own GUI.
- **Add-ons** are smaller components that don't have their own GUI and may need some extra configuration.
- **Suites** are robust collections of apps built to provide comprehensive IT solutions on top of Splunk. Suites may require professional services to support and install and are usually built by Splunk or our Partners.

More than likely your app will not be compatible with Splunk 3.x, so uncheck the **3.x** checkbox.

Once Splunk personnel approve your app, it will appear in Splunkbase, ready for others to download.

Summary

In this chapter, we covered installing, building, customizing, and sharing apps. Apps are a loose concept in Splunk, with many different purposes served by a simple directory of files. Hopefully we have covered the basics well enough for you to get started on your own great apps. In later chapters, we will build even more complicated object types, as well as custom code to extend Splunk in unique ways.

In the next chapter, we will dig into advanced dashboards, both covering what can be done with Splunk alone, and what can be done with the help of a few popular apps.

8

Building Advanced Dashboards

In *Chapter 4, Simple XML Dashboards*, we covered building dashboards using simple XML. We first used the wizards provided in Splunk, and then edited the resultant XML. When you reach the limits of what can be accomplished with simple XML, one option is to dive into Splunk's advanced XML.

Reasons for working with advanced XML

Here are a few reasons to use advanced XML:

1. **More control over layout:** With advanced XML, you have better control over where form elements and chrome appear, and somewhat improved control over the placement of the output.
2. **Custom drilldowns:** It is only possible to create custom drilldowns from tables and charts using advanced XML.
3. **Access to more parameters:** The modules in simple XML actually use advanced XML modules, but many parameters are not exposed. Sometimes the desire is actually to disable features, and this is only possible by using advanced XML.
4. **Access to more modules:** There are many modules *not* available when using simple XML, for example the search bar itself. All extra modules provided by the apps at Splunkbase, for example *Google Maps*, are for use in advanced XML.

Reasons for not working with advanced XML

There are also a number of reasons to not work with advanced XML:

1. **Steep learning curve:** Depending on what technologies you are comfortable working with, and possibly on how well the rest of this chapter is written, the learning curve for advanced XML can be steep.
2. **No direct control over HTML:** If there is a particular HTML you want to produce from search results, this may not be as simple as you had hoped. Short of writing your own module, you must work within the bounds of the options provided to the existing modules, modify CSS with `application.css`, or modify the HTML using JavaScript.
3. **No direct control over logic:** If you need specific things to happen when you click on specific table cells, particularly based on other values in the same row, this can only be accomplished by modifying the document using JavaScript. This is possible, but it is not well documented. Examples can be found at <http://splunkbase.com> both in answers posts and sample applications. Check out `customBehaviors` in the third-party app *Sideview Utils* for an alternative.



If you have specific layout or logic requirements, you may be better served using one of the Splunk APIs available at <http://dev.splunk.com> and writing applications in your favorite language.

Development process

When building dashboards, my approach is generally as follows:

1. Create the needed queries.
2. Add the queries to a simple XML dashboard. Use the GUI tools to tweak the dashboard as much as possible. Finish all graphical changes at this stage, if possible.
3. Convert the simple XML dashboard to a form if form elements are needed. Make all logic work with simple XML if possible.
4. Convert the simple XML dashboard to an advanced XML dashboard. There is no reverse conversion possible, so this should be done as late as possible, and only if needed.
5. Edit the advanced XML dashboard accordingly.

The idea is to take advantage of the Splunk GUI tools as much as possible, letting the simple XML conversion process add all of the advanced XML that you would have to otherwise find yourself. We covered steps 1-3 in the previous chapters. Step 4 is covered in the *Converting simple XML to advanced XML* section.

Advanced XML structure

Before we dig into the modules provided, let's look at the structure of the XML itself and cover a couple of concepts.

The tag structure of an advanced XML document is essentially:

```
view
  module
    param
    ...
  module
  ...
```

The main concept of Splunk's XML structure is that the effects of modules flow downstream to child modules. This is a vital concept to understand. The XML structure has almost nothing to do with layout, and everything to do with the flow of data.

Let's look at a simple example like this:

```
<view
  template="dashboard.html">

  <label>Chapter 8, Example 1</label>

  <module
    name="HiddenSearch"
    layoutPanel="panel_row1_col1"
    autoRun="True">
    <param name="earliest">-1d</param>
    <param name="search">error | top user</param>

    <module name="SimpleResultsTable"></module>

  </module>


</view>
```

This document produces a sparse dashboard with one panel like this:

	user ↕	count ↕	percent ↕
1	mary	1668	31.376975
2	Bobby	871	16.384500
3	jacky	851	16.008277
4	bob	844	15.876599
5	linda	828	15.575621
6	extrauser	254	4.778029

Let's step through this example line by line.

- `<view>`: Open the outer tag. This tag begins all advanced XML dashboards.
- `template="dashboard.html">`: Set the base HTML template. Dashboard layout templates are stored in `$SPLUNK_HOME/share/splunk/search_mrsparkle/templates/view/`. Among other things, the templates define the panels available for use in `layoutPanel`.
- `<label>Chapter 8, Example 1</label>`: Set the label used by navigation.
- `<module>`: Begin our first module declaration.
- `name="HiddenSearch"`: The name of the module to use. `HiddenSearch` runs a search but displays nothing, relying instead on child modules to render the output.
- `layoutPanel="panel_row1_col1"`: This states where in the dashboard to display our panel. It seems strange to put this attribute on a module that displays nothing, but `layoutPanel` must be specified on every immediate child of `view`. See the *Understanding layoutPanel* section later for more details.
- `autoRun="True">`: Without this attribute, the search does not run when the dashboard loads, and instead waits for user interaction from form elements. Since we have no form elements, we need this attribute to see the results.
- `<param name="earliest">-1d</param>`: It is very important to specify a value for `earliest`, as the query will by default run over All time.

 param values affect only the module tag they are nested directly inside.

- `<param name="search">error | top user</param>`: The actual query to run.
- `<module name="SimpleResultsTable"></module>`: This module simply displays a table of the events produced by a parent module. Since there are no param tags specified, the defaults for this module will be used.
- `</module>`: Close the `HiddenSearch` module. This is required for valid XML, but it also implies that the scope of influence for this module is closed. To reiterate, only the downstream modules of the `HiddenSearch` module will receive the events it produces.
- `</view>`: Close the document.

This is a very simple dashboard. It lacks navigation, form elements, job status, and drilldowns. Adding all of these things is initially somewhat complicated to understand. Luckily, you can build a dashboard in simple XML, convert it to advanced XML, and then modify the provided XML as needed.

Converting simple XML to advanced XML

Let's go back to one of the dashboards we created in *Chapter 4, Simple XML Dashboards*, `errors_user_form`. We built this before our app, so it still lives in the Search app. In my instance, this URL is `http://mysplunkserver:8000/en-US/app/search/errors_user_form`.

Just to refresh, the simple XML behind this dashboard looks like:

```
<?xml version='1.0' encoding='utf-8'?>
<form>

  <fieldset>
    <input type="text" token="user">
      <label>User</label>
    </input>
    <input type="time" />
  </fieldset>

  <label>Errors User Form</label>

  <row>
    <chart>
      <searchString>
        sourcetype="impl_splunk_gen" loglevel=error user="$user$"
        | timechart count as "Error count" by network
```

```
    </searchString>
    <title>
      Dashboard - Errors - errors by network timechart
    </title>
    <option name="charting.chart">line</option>
  </chart>
</row>

<row>
  <chart>
    <searchString>
      sourcetype="impl_splunk_gen" loglevel=error user="$user$"
      | bucket bins=10 req_time | stats count by req_time
    </searchString>
    <title>
      Error count by req_times
    </title>
    <option name="charting.chart">pie</option>
  </chart>
  <chart>
    <searchString>
      sourcetype="impl_splunk_gen" loglevel=error user="$user$"
      | stats count by logger
    </searchString>
    <title>Errors by logger</title>
    <option name="charting.chart">pie</option>
  </chart>
</row>

<row>
  <event>
    <searchString>
      sourcetype="impl_splunk_gen" loglevel=error user="$user$"
    </searchString>
    <title>Error events</title>
    <option name="count">10</option>
    <option name="displayRowNumbers">>true</option>
    <option name="maxLines">10</option>
    <option name="segmentation">outer</option>
    <option name="softWrap">>true</option>
  </event>
</row>

</form>
```

In the simple XML, the layout and logic flow are tied together.

Before this simple XML is rendered to the user, Splunk first dynamically converts it to advanced XML in memory. We can access that advanced XML by appending `?showsource=1` to any URL, like this:

```
http://mysplunkserver:8000/en-US/app/search/errors_user_form?showsource=1
```

This produces a page with a tree view of the module structure like this:

View source: errors_user_form (Errors User Form)

Properties

- `template`: dashboard.html
- `autoCancelInterval`: 90
- `objectMode`: simpleForm
- `label`: Errors User Form
- `stylesheet`: None
- `onunloadCancelJobs`: True
- `isVisible`: True

Module tree

[Collapse all](#) | [Expand all](#) | [Toggle all](#)

- [-] AccountBar_0_0_0 *appHeader*
 - [-] AccountBar config
- [-] AppBar_0_0_1 *navigationHeader*
 - [-] AppBar config
- [-] Message_0_0_2 *messaging*
 - [-] Message config
- [-] Message_1_0_3 *messaging*
 - [-] Message config
- [-] TitleBar_0_0_4 *viewHeader*
 - [-] TitleBar config
- [-] ExtendedFieldSearch_0_0_5 *viewHeader*
 - [-] ExtendedFieldSearch config
 - [-] TimeRangePicker_0_1_0
 - [-] TimeRangePicker config
 - [-] SubmitButton_0_2_0
 - [-] SubmitButton config
 - [-] HiddenSearch_0_3_0 *panel_row1_col1*
 - [-] HiddenSearch config
 - [-] ViewstateAdapter_0_4_0
 - [-] ViewstateAdapter config
 - [-] HiddenFieldPicker_0_5_0
 - [-] HiddenFieldPicker config
 - [-] JobProgressIndicator_0_6_0
 - [-] JobProgressIndicator config
 - [-] EnablePreview_0_7_0
 - [-] EnablePreview config
 - [-] HiddenChartFormatter_0_8_0
 - [-] HiddenChartFormatter config

This is followed by a textbox containing the raw XML like this:

```
XML source
<view autoCancelInterval="90" isVisible="true" objectMode="SimpleForm" onunloadCancelJobs="true"
template="dashboard.html">
<label>Errors User Form</label>
<module name="AccountBar" layoutPanel="appHeader"/>
<module name="AppBar" layoutPanel="navigationHeader"/>
<module name="Message" layoutPanel="messaging">
<param name="filter">*</param>
<param name="clearOnJobDispatch">False</param>
<param name="maxSize">1</param>
</module>
<module name="Message" layoutPanel="messaging">
<param name="filter">splunk.search.job</param>
<param name="clearOnJobDispatch">True</param>
<param name="maxSize">1</param>
</module>
<module name="TitleBar" layoutPanel="viewHeader">
<param name="actionsMenuFilter">dashboard</param>
</module>
<module name="ExtendedFieldSearch" layoutPanel="viewHeader">
<param name="replacementMap">
<param name="arg">
<param name="user"/>
</param>
</param>
</module>
```

An abbreviated version of the advanced XML version of errors_user_form follows:

```
<view
... template="dashboard.html">
<label>Errors User Form</label>
<module name="AccountBar" layoutPanel="appHeader"/>
<module name="AppBar" layoutPanel="navigationHeader"/>
<module name="Message" layoutPanel="messaging">
...<module name="Message" layoutPanel="messaging">
...<module name="TitleBar" layoutPanel="viewHeader">
...<module name="ExtendedFieldSearch" layoutPanel="viewHeader">
<param name="replacementMap">
<param name="arg">
<param name="user"/>
</param>
</param>
<param name="field">User</param>
<param name="intention">
... <module name="TimeRangePicker">
<param name="searchWhenChanged">False</param>
<module name="SubmitButton">
<param name="allowSoftSubmit">True</param>
<param name="label">Search</param>
<module
name="HiddenSearch"
```

```

        layoutPanel="panel_row1_col1"
        group="Dashboard - Errors - errors by network timechart"
        autoRun="False">
<param name="search">
    sourcetype="impl_splunk_gen"
    loglevel=error user="$user$"
    | timechart count as "Error count" by network
</param>
<param name="groupLabel">
    Dashboard - Errors - errors by network timechart
</param>
<module name="ViewstateAdapter">
    <param name="suppressionList">
        <item>charting.chart</item>
    </param>
    <module name="HiddenFieldPicker">
        <param name="strictMode">True</param>
        <module name="JobProgressIndicator">
            <module name="EnablePreview">
                <param name="enable">True</param>
                <param name="display">False</param>
                <module name="HiddenChartFormatter">
                    <param name="charting.chart">line</param>
                    <module name="JSChart">
                        <param name="width">100%</param>
                        <module name="Gimp"/>
                        <module name="ConvertToDrilldownSearch">
                            <module name="ViewRedirector">
                                ...
                                </module>
                            <module name="ViewRedirectorLink">
                                ...
                                </module>
                            </module>
                        <module
                            name="HiddenSearch"
                            layoutPanel="panel_row2_col1"
                            group="Error count by req_times"
                            autoRun="False">
<param name="search">
    sourcetype="impl_splunk_gen" loglevel=error
    user="$user$"
    | bucket bins=10 req_time | stats count by req_time
</param>
    <param name="groupLabel">Error count by req_times</param>
    ...
</module>

```

```
    <module
      name="HiddenSearch"
      layoutPanel="panel_row2_col2"
      group="Errors by logger"
      autoRun="False">
    <param name="search">
      sourcetype="impl_splunk_gen"
      loglevel=error user="$user$"
      | stats count by logger
    </param>
    <param name="groupLabel">Errors by logger</param>
  ... </module>
  <module
    name="HiddenSearch"
    layoutPanel="panel_row3_col1"
    group="Error events"
    autoRun="False">
  <param name="search">
    sourcetype="impl_splunk_gen"
    loglevel=error
    user="$user$"
  </param>
  <param name="groupLabel">Error events</param>
  <module name="ViewstateAdapter">
  ...   <module name="HiddenFieldPicker">
  ...     <module name="JobProgressIndicator"/>
  ...     <module name="Paginator">
  ...       <param name="count">10</param>
  ...     <module name="EventsViewer">
  ...       <module name="Gimp"/>
  ...     </module>
  ...   </module>
  ... </view>
```

This XML is more verbose than we actually need, but luckily it is easier to delete code than to create it.

Module logic flow

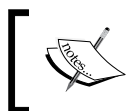
The main concept of nested modules is that parent (upstream) modules affect child (downstream) modules. Looking at the first panel, the full module flow is:

```
<module name="ExtendedFieldSearch">
  <module name="TimeRangePicker">
    <module name="SubmitButton">
      <module name="HiddenSearch">
```

```

<module name="ViewstateAdapter">
  <module name="HiddenFieldPicker">
    <module name="JobProgressIndicator">
      <module name="EnablePreview">
        <module name="HiddenChartFormatter">
          <module name="JSChart">
            <module name="ConvertToDrilldownSearch">
              <module name="ViewRedirector">
                <module name="ViewRedirectorLink">

```



A reference for the modules installed in your instance of Splunk is available at /modules. In my case, the full URL is <http://mysplunkserver:8000/modules>.

Let's step through these modules in turn and discuss what they are each accomplishing:

- **ExtendedFieldSearch**: This provides a textbox for entry. The parameters for this module are complicated, and represent arguably the most complicated aspect of advanced XML – **intentions**. Intentions affect child modules, specifically `HiddenSearch`. We will cover intentions later.
- **TimeRangePicker**: This provides the standard time picker. It affects child `HiddenSearch` modules that do not have times specified either using `param` values or in the query itself. The precedence of times used in a query are:
 - Times specified in the query itself
 - Times specified via `earliest` and `latest` `param` values to the search module
 - A value provided by `TimeRangePicker`
- **SubmitButton**: This draws the **Search** button and fires off any child search modules when clicked.
- **HiddenSearch**: As we saw before, this runs a query and produces events for downstream modules. In this case, `autoRun` is set to `false`, so that the query waits for the user.
- **ViewstateAdapter**: A **viewstate** describes what settings a user has selected in the GUI, for instance, sort order, page size, or chart type. Any time you change a chart setting or pick a time range, you create a viewstate that is saved by Splunk. This module is used to access an existing viewstate, or to suppress specific viewstate settings. By suppressing specific settings, the default or specified values of child modules will be used instead. This module is rarely needed unless you are using a saved search with an associated viewstate.

- `HiddenFieldPicker`: This module limits what fields are accessible by downstream modules. This is useful when running a query that produces many fields, but only certain fields are needed. This would affect the fields shown below events in an events listing, or the columns displayed in a table view. This module is rarely needed.
- `JobProgressIndicator`: This module displays a progress bar until the job is completed. In this case, because of the placement of the module in the XML, it will appear above the results. This module does not affect downstream modules, so it can be listed on its own.
- `EnablePreview`: This module allows you to specify whether searches should refresh with incomplete results while the query is running. The default appears to be true for Splunk-provided modules, but this module allows you to control this behavior. This module does not affect downstream modules, so could be listed on its own.



Disabling preview can improve the performance dramatically, but provides no information until the query is complete, which is less visually appealing, particularly during a long-running query.

- `HiddenChartFormatter`: This module is where the chart settings are specified. These settings affect any child modules that draw charts.
- `JSChart`: This draws a chart using JavaScript. Prior to Splunk 4.3, all charts were drawn using Flash. The `FlashChart` module is still included, for backward compatibility.
- `ConvertToDrilldownSearch`: This module takes the values from a click on a parent module and produces a query based on the query that produced the results. This usually works, but not always, depending on the complexity of the query. We will build a custom drilldown search later.
- `ViewRedirector`: This module accepts the query from its upstream module and redirects the user to `viewTarget`, with the query specified in the URL. Usually, `flashtimeline` is specified as the `viewTarget` param, but it could be any dashboard. The query will affect a `HiddenSearch` or `SearchBar` module.
- `ViewRedirectorLink`: This module sends the user to a new search page with the search results for this module.

Thinking about what we have seen in this flow, we could say that modules can:

- Generate events
- Modify a query

- Modify the behavior of a downstream module
- Display an element on the dashboard
- Handle actions produced by clicks

It is also possible for a module to:

- Post process the events produced by a query
- Add custom JavaScript to the dashboard

Understanding layoutPanel

In an advanced XML dashboard, which panel a module is drawn to is determined by the value of the `layoutPanel` attribute. This separation of logic and layout can be useful – for instance, allowing you to reuse data generated by a query with multiple modules – but displays the results on different parts of the page.

A few rules about this attribute are as follows:

- The `layoutPanel` attribute *must* appear on all *immediate* children of `<view>`.
- The `layoutPanel` attribute *can* appear on descendant child module tags.
- If a module does not have a `layoutPanel` attribute, it will inherit the value from the closest upstream module that does.
- Modules that have visible output are added to their respective `layoutPanel` attribute in the order they appear in the XML.
- Modules "flow" in the panel they are placed. Most modules take the entire width of the panel, but some do not, and flow left to right before wrapping.

Looking through our XML, we find these elements with the `layoutPanel` attribute like this:

```
<module name="AccountBar" layoutPanel="appHeader"/>
<module name="AppBar" layoutPanel="navigationHeader"/>
<module name="Message" layoutPanel="messaging">

<module name="TitleBar" layoutPanel="viewHeader">
<module name="ExtendedFieldSearch" layoutPanel="viewHeader">
  <module name="TimeRangePicker">
    <module name="SubmitButton">

    <module name="HiddenSearch" layoutPanel="panel_row1_col1">
      ...
```

```
<module name="HiddenSearch" layoutPanel="panel_row2_col1">
  ...
<module name="HiddenSearch" layoutPanel="panel_row2_col2">
  ...
<module name="HiddenSearch" layoutPanel="panel_row3_col1">
  ...
```

The first set of the `layoutPanel` values are panels included in the "chrome" of the page. This displays the account information, the navigation, and any messages to the user. The second set of modules make up the title and form elements. Notice that `TimeRangePicker` and `SubmitButton` have no `layoutPanel` value, but will inherit from `ExtendedFieldSearch`.

The results panels all begin with a `HiddenSearch` module. All of the children of each of these modules inherit this `layoutPanel` value.

Panel placement

For your dashboard panels, you will almost always use a `layoutPanel` value of the form `panel_rowX_colY`.

A simple visualization of the layout produced by our modules would look like:



In our simple XML version of this dashboard, the layout was tied directly to the order of the XML, like this:

```
<row>
  <chart></chart>
</row>

<row>
  <chart></chart>
  <chart></chart>
</row>

<row>
  <event></event>
</row>
```

Just to reiterate, the simple XML structure translates to:

```
<row>
  <chart></chart> == panel_row1_col1
</row>

<row>
  <chart></chart> == panel_row2_col1
  <chart></chart> == panel_row2_col2
</row>

<row>
  <event></event> == panel_row3_col1
</row>
```

There is another extension available, `_grp1`, which allows you to make columns inside a panel. We will try that out in the *Creating a custom drilldown* section later.

Reusing a query

One example of separating layout from data would be using a single query to populate both a table and a chart. The advanced XML for this could look like the following:

```
<view template="dashboard.html">
  <label>Chapter 8 - Reusing a query</label>

  <module
    name="StaticContentSample"
    layoutPanel="panel_row1_col1">
    <param name="text">Text above</param>
  </module>

  <module
    name="HiddenSearch"
    layoutPanel="panel_row1_col1"
    autoRun="True">
    <param name="search">
      sourcetype="impl_splunk_gen" loglevel=error | top user
    </param>
    <param name="earliest">-24h</param>

    <module name="HiddenChartFormatter">
      <param name="charting.chart">pie</param>
```



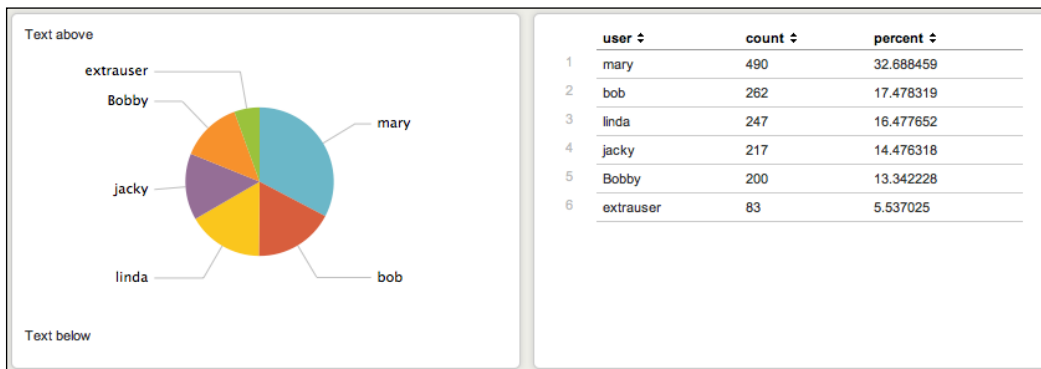
```
<module name="JSChart"></module>

<module
  name="StaticContentSample"
  layoutPanel="panel_row1_col1">
  <!-- this layoutPanel is unneeded, but harmless -->
  <param name="text">Text below</param>
</module>
</module>

<module name="SimpleResultsTable"
  layoutPanel="panel_row1_col2"></module>

</module>
</view>
```

This XML will render a dashboard like the following screenshot:



There are some things to notice in this XML:

- The data produced by `HiddenSearch` is used by both child modules.
- `JSChart` inherits `layoutPanel="panel_row1_col1"` from `HiddenSearch`.
- `SimpleResultsTable` has its own `layoutPanel` attribute set to `panel_row1_col2`, so the table draws to the right.
- Both `StaticContentSample` modules specify `layoutPanel="panel_row1_col1"`, and therefore appear in the same panel as the chart. Though they are at different depths in the XML, the order drawn follows the order seen in the XML.

Using intentions

Intentions allow you to affect downstream searches, using values provided by other modules, for instance, form fields or the results of a click. There are a number of available intention types, but we will cover the two most common, `stringreplace` and `addterm`. You can see examples of other types of intentions in the *UI Examples* app available at <http://splunkbase.com>.

stringreplace

This is the most common intention to use, and maps directly to the only available action in simple XML – variable replacement. Let's look at our search field from our advanced XML example:

```
<module name="ExtendedFieldSearch" layoutPanel="viewHeader">
  <param name="replacementMap">
    <param name="arg">
      <param name="user"/>
    </param>
  </param>
  <param name="field">User</param>
  <param name="intention">
    <param name="name">stringreplace</param>
    <param name="arg">
      <param name="user">
        <param name="fillOnEmpty">True</param>
      </param>
    </param>
  </param>
</module>
```

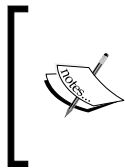
Stepping through the params we have:

- `field`: This is the label for the field displayed in the dashboard.
- `replacementMap`: This parameter names the variable that the `ExtendedFieldSearch` module is creating. I have been told that the nested nature means nothing, and we should simply copy and paste the entire block of XML, changing nothing but the value of the deepest param, in this case to `user`.
- `intention`: Intentions have specific structures that build blocks of query from a structured XML. In the case of `stringreplace` (which is the most common use case), we can essentially copy the entire XML and once again change nothing but the value of the third-level param, which is currently `user`. `fillOnEmpty` determines whether to make the substitution when the user variable is empty.

All of this code simply says to replace `$user$` in any searches with the value of the input field. Our first `HiddenSearch` looks like the following:

```
<module name="HiddenSearch" ...
  <param name="search">
    sourcetype="impl_splunk_gen"
    loglevel=error user="$user$"
    | timechart count as "Error count" by network
  </param>
```

The value of `$user$` will be replaced and the query will be run.



If you want to see exactly what is happening, you can insert a `SearchBar` module as a child of the form elements, and it will render the resulting query. For an example, see the code of the dashboard `drilldown_chart1` in the *UI Examples* app available at <http://splunkbase.com>.

addterm

This intention is useful for adding search terms to a query, with or without user interaction. For example, let's say you always want to ensure that a particular value of the field `source` is queried. You can then modify the query that will be run, appending a search term. Here is an example from the dashboard `advanced_lister_with_searchbar` in the *UI Examples* app available at <http://splunkbase.com>:

```
<module name="HiddenIntention">
  <param name="intention">
    <param name="name">addterm</param>
    <param name="arg">
      <param name="source">*metrics.log</param>
    </param>
    <!-- tells the addterm intention to put our
         term in the first search clause no matter what. -->
    <param name="flags"><list>indexed</list></param>
  </param>
```

Stepping through the params:

- `name`: This parameter sets the type of intention, in this case `addterm`.
- `arg`: This is used to set the field to add to the query.
 - The nested `param` tag sets the fieldname and value to use in the query. In this case, `source="*metrics.log"` will be added to the query.

- Variables can be used in either the name attribute or body of this nested param tag. We will see an example of this under the *Creating a custom drilldown* section.
- flags: Every example of `addterm` that I can find includes this attribute, exactly as written. It essentially says that the term to be added to the search should be added before the first pipe symbol, not at the end of the full query. For example, consider the following query:

```
error | top logger
```

This param would amend our query like this:

```
error source="*metrics.log" | top logger
```

Creating a custom drilldown

A drilldown is a query built using values from a previous query. The module `ConvertToDrilldownSearch` will build a query automatically from the table or graph that it is nested inside. Unfortunately, this only works well when the query is fairly simple, and when you want to see raw events. To build a custom drilldown, we combine intentions and the nested nature of modules.

Building a drilldown to a custom query

Looking back at our chart in the *Reusing a query* section, let's build a custom drilldown that shows the top instances of another field when it is clicked on.

Here is an example dashboard that draws a chart and then runs a custom query when clicked on:

```
<view template="dashboard.html">
  <label>Chapter 8 - Drilldown to custom query</label>
  <!-- chrome -->
  <module
    name="HiddenSearch"
    layoutPanel="panel_row1_col1"
    autoRun="True"
    group="Errors by user">
    <param name="search">
      sourcetype="impl_splunk_gen" loglevel=error | top user
    </param>
    <param name="earliest">-24h</param>
```

```
<!-- draw the chart -->
<module name="HiddenChartFormatter">
  <param name="charting.chart">pie</param>
</module name="JSChart">

  <!-- nested modules are invoked on click -->
  <!-- create a new query -->
  <module name="HiddenSearch">
    <param name="search">
      sourcetype="impl_splunk_gen" loglevel=error
      | top logger
    </param>

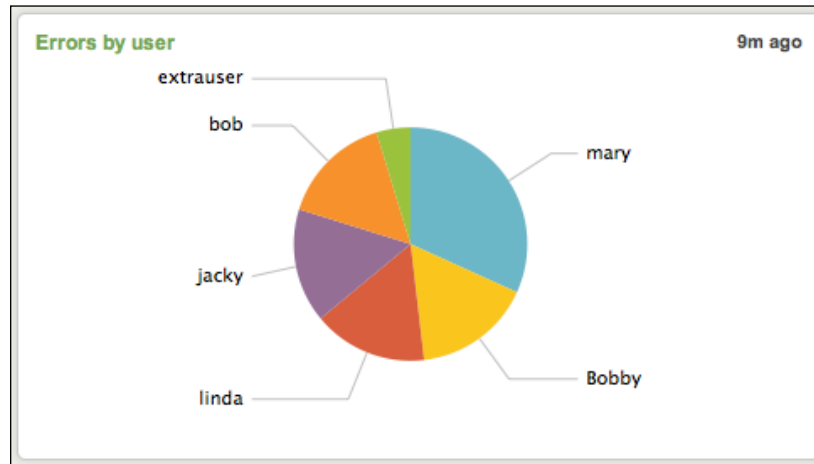
    <!-- create an intention using the value from the chart.
      Use addterm to add a user field to the query. -->
    <module name="ConvertToIntention">
      <param name="intention">
        <param name="name">addterm</param>
        <param name="arg">
          <param name="user">${click.value}</param>
        </param>
        <param name="flags">
          <item>indexed</item>
        </param>
      </param>
    </module>

    <!-- Send the user to flashtimeline
      with the new query. -->
    <module name="ViewRedirector">
      <param name="viewTarget">flashtimeline</param>
    </module>
  </module>
</module>
</module>
</module>
</module>
</view>
```

Everything should look very similar up until the JSChart module. Inside this module we find a HiddenSearch module. The idea is that the downstream modules of display modules are not invoked until the display module is clicked. HiddenSearch in this case is used to build a query, but instead of the query being handed to a display module, it is handed to the ViewRedirector module.

The "magical" field in all of this is `click.value`. This field contains the value that was clicked on in the chart.

Let's look at what this dashboard renders:



The resulting query when we click on the slice of the pie for the user **bob** looks like:

Search

sourcetype="impl_splunk_gen" loglevel=error user="bob" | top logger since 1:45:07 PM July 19, 2012

2,632 matching events

Show timeline

4 results since 1:45:07 PM July 19, 2012

Overlay: None

	logger	count	percent
1	BarClass	1267	64.022233
2	FooClass	257	12.986357
3	AuthClass	228	11.520970
4	LogoutClass	227	11.470440

Look back to the *addterm* section for more details on how this intention works.

Building a drilldown to another panel

Another option for a drilldown is to draw a new panel on the same dashboard. This lets you create various drilldowns without redrawing the screen, which might be less jarring to the user. Here is the XML:

```
<?xml version="1.0"?>
<view template="dashboard.html">
  <label>Chapter 8 - Drilldown to new graph</label>
  <!-- chrome should go here -->
  <module
    name="HiddenSearch"
    layoutPanel="panel_row1_col1"
    autoRun="True"
    group="Errors by user">
    <param name="search">
      sourcetype="impl_splunk_gen" loglevel=error | top user
    </param>
    <param name="earliest">-24h</param>
    <module name="HiddenChartFormatter">
      <param name="charting.chart">pie</param>

    <!-- draw the first chart -->
    <module name="JSChart">

    <!-- the modules inside the chart will wait for
      interaction from the user -->
    <module name="HiddenSearch">
      <param name="earliest">-24h</param>
      <param name="search">
        sourcetype="impl_splunk_gen" loglevel=error
        user="$user$" | timechart count by logger
      </param>
      <module name="ConvertToIntention">
        <param name="intention">
          <param name="name">stringreplace</param>
          <param name="arg">
            <param name="user">
              <param name="value">${click.value}</param>
            </param>
          </param>
        </param>
      </param>

    <!-- print a header above the new chart -->
```

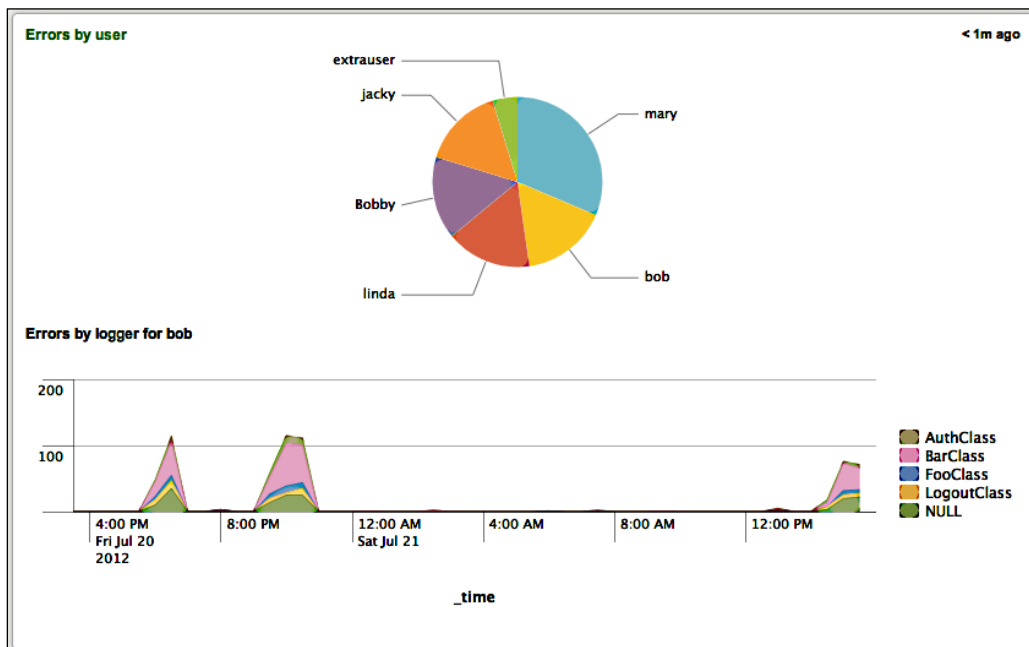
```

<module name="SimpleResultsHeader">
  <param name="entityName">results</param>
  <param name="headerFormat">
    Errors by logger for $click.value$
  </param>
</module>

<!-- draw the chart. We have not specified another
      layoutPanel, so it will appear below the first
      chart -->
<module name="HiddenChartFormatter">
  <param name="charting.chart">area</param>
  <param name="chart.stackMode">stacked</param>
  <module name="JSChart"/>
</module>
</module>
</module>
</module>
</module>
</module>
</view>

```

Here's what the dashboard looks like after clicking on **bob** in the pie chart:



Building a drilldown to multiple panels using HiddenPostProcess

Taking the last dashboard further, let's build a number of panels from a single custom drilldown query. As we covered in *Chapter 4, Simple XML Dashboards*, search results can be post processed, allowing you to use the same query results multiple ways. In advanced XML, this is accomplished using the `HiddenPostProcess` module. We will also add the chrome for our first complete dashboard. Here is an abbreviated example. The complete dashboard is in the `Chapter8_drilldown_to_new_graph_with_postprocess.xml` file in the *Implementing Splunk App One* app:

```
<view template="dashboard.html">
  <label>Chapter 8 - Drilldown to new graph with postprocess</label>

  <!-- The chrome at the top of the dashboard
        containing navigation and the app header -->
  <module name="AccountBar" layoutPanel="appHeader"/>
  <module name="AppBar" layoutPanel="navigationHeader"/>
  <module name="Message" layoutPanel="messaging">
    <param name="filter">*</param>
    <param name="clearOnJobDispatch">False</param>
    <param name="maxSize">1</param>
  </module>
  <module name="DashboardTitleBar" layoutPanel="viewHeader"/>
  <module name="Message" layoutPanel="navigationHeader">
    <param name="filter">splunk.search.job</param>
    <param name="clearOnJobDispatch">True</param>
    <param name="maxSize">1</param>
    <param name="level">warn</param>
  </module>

  !-- Begin our initial search
        which will populate our pie chart -->
  <module
    name="HiddenSearch" layoutPanel="panel_row1_col1"
    autoRun="True" group="Errors by user">
    <param name="search">
      sourcetype="impl_splunk_gen" loglevel=error | top user
    </param>
    <param name="earliest">-24h</param>

    <module name="HiddenChartFormatter">
      <param name="charting.chart">pie</param>
      <module name="JSChart">

  <!-- Initially, only the pie chart will be drawn
        After a click on a user wedge, this nested query will run -->
    <module name="HiddenSearch">
```

```

<param name="earliest">-24h</param>
<param name="search">
  sourcetype="impl_splunk_gen" loglevel=error
  user="$user$" | bucket span=30m _time
  | stats count by logger _time
</param>
<module name="ConvertToIntention">
  <param name="intention">
    <param name="name">stringreplace</param>
    <param name="arg">
      <param name="user">
        <param name="value">${click.value}</param>
      </param>
    </param>
  </param>
  ...
<!-- The remaining modules are downstream from the pie chart
and are invoked when a pie wedge is clicked -->
  <module name="SimpleResultsHeader">
    layoutPanel="panel_row2_col1">
      <param name="entityName">results</param>
      <param name="headerFormat">
        Errors by logger for ${click.value}
      </param>
    </module>

<!-- The SingleValue modules -->
  <module name="HiddenPostProcess">
    <param name="search">
      stats sum(count) as count by logger
      | sort -count | head 1
      | eval f=logger + " is most common (" + count + ")" |
table f </param>
    <module name="SingleValue">
      layoutPanel="panel_row2_col1"></module>
    </module>
  ...
<!-- The chart -->
  <module name="HiddenPostProcess">
    <param name="search">
      timechart span=30m sum(count) by logger
    </param>
    <module name="HiddenChartFormatter">
      <param name="charting.chart">area</param>
      <param name="chart.stackMode">stacked</param>
    <module
      name="JSChart"
      layoutPanel="panel_row4_col1_grp1"/>
    </module>
  </module>

<!-- The table -->

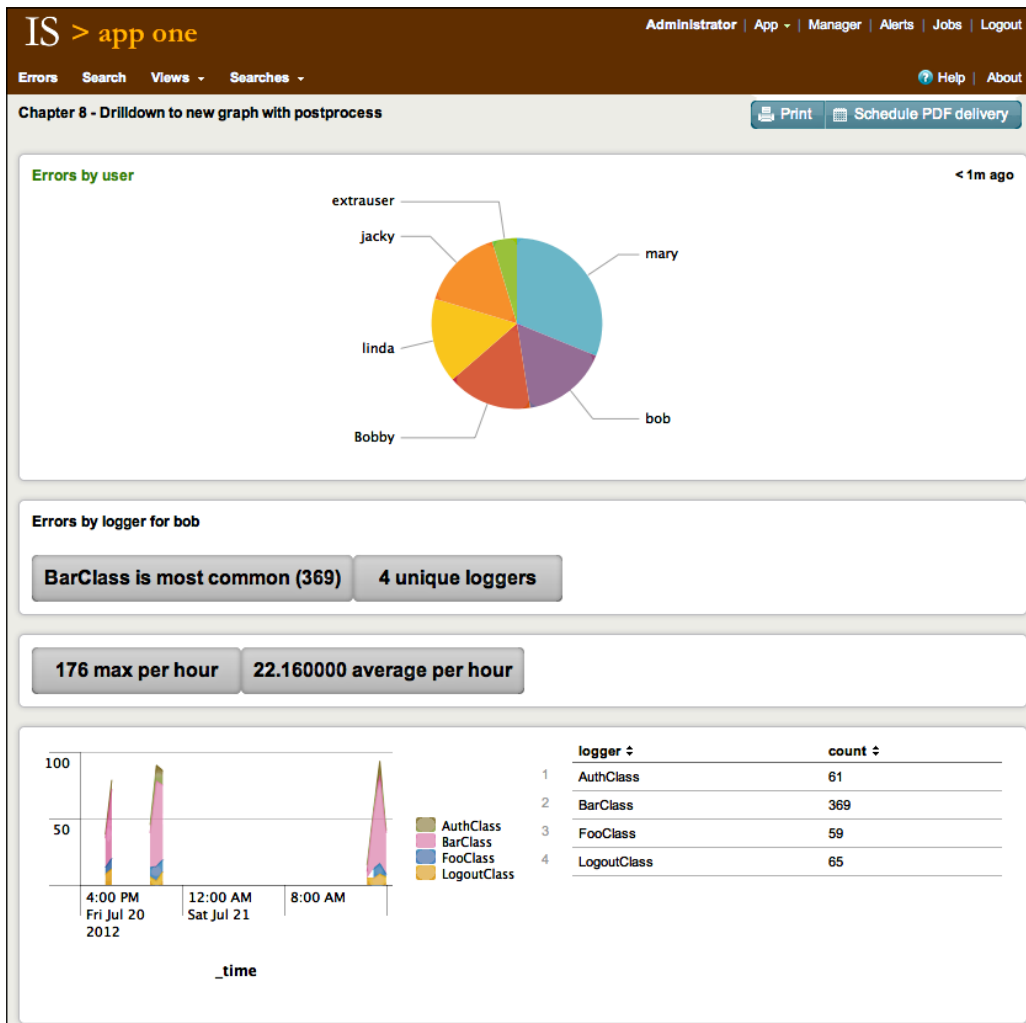
```

```

<module name="HiddenPostProcess">
  <param name="search">
    stats sum(count) as count by logger
  </param>
  <module name="SimpleResultsTable"
    layoutPanel="panel_row4_col1_grp2"/>
</module>
...
</module>
</view>

```

This dashboard contains the chrome, which is very useful as it displays the errors in your intentions and query statements. After clicking on **bob**, this is what we see:



Let's step through the new queries. The initial query is the same:

```
sourcetype="impl_splunk_gen" loglevel=error | top user
```

The next query may seem strange, but there's a good reason for this:

```
sourcetype="impl_splunk_gen" loglevel=error user="$user$"
| bucket span=30m _time
| stats count by logger _time
```

If you look back to *Chapter 5, Advanced Search Examples*, we used `bucket` and `stats` to slice events by `_time` and other fields. This is a convenient way to break down events for post processing, where one or more of the post-process queries uses `timechart`. This query produces a row with the field `count` for every unique value of `logger` in each 30-minute period.



Post processing has a limit of 10,000 events. To accommodate this limit, all aggregation possible should be done in the initial query. Ideally, only what is needed by all child queries should be produced by the initial query. It is also important to note that all fields needed by post-process queries must be returned by the initial query.

The first `HiddenPostProcess` builds a field for a module we haven't used yet, `SingleValue`, which takes the first value it sees and renders that value in a rounded rectangle.

```
stats sum(count) as count by logger
| sort -count
| head 1
| eval f=logger + " is most common (" + count + ")"
| table f
```

The query is additive, so the full query for this module is essentially:

```
sourcetype="impl_splunk_gen" loglevel=error user="bob"
| bucket span=30m _time
| stats count by logger _time
| stats sum(count) as count by logger
| sort -count
| head 1
| eval f=logger + " is most common (" + count + ")"
| table f
```

The remaining `SingleValue` modules do similar work to find the count of unique loggers, the max errors per hour, and the average errors per hour. To step through these queries, simply copy each piece and add it to a query in search.

Other things to notice in this dashboard are:

- `grp` builds columns inside a single panel, for instance, in `layoutPanel="panel_row4_col1_grp2"`
- `SingleValue` modules do not stack vertically, but rather flow horizontally, overflowing onto the next line when the window width is reached
- `span` used in the `bucket` statement is the minimum needed by any post-process statements, but as large as possible to minimize the number of events returned

Third-party add-ons

There are many excellent apps available at <http://splunkbase.com>, a number of which provide custom modules. We will cover two of the most popular, *Google Maps* and *Sideview Utils*.

Google Maps

As we saw in *Chapter 7, Working with Apps*, the *Google Maps* app provides a dashboard and lookup for drawing results on a map. The underlying module is also available to use in your own dashboards.

Here is a very simple dashboard that uses the `GoogleMaps` module:

```
<?xml version="1.0"?>
<view template="search.html">

  <!-- chrome -->
  <label>Chapter 8 - Google Maps Search</label>
  <module name="AccountBar" layoutPanel="appHeader"/>
  <module name="AppBar" layoutPanel="navigationHeader"/>
  <module name="Message" layoutPanel="messaging">
    <param name="filter">*</param>
    <param name="clearOnJobDispatch">False</param>
    <param name="maxSize">1</param>
  </module>

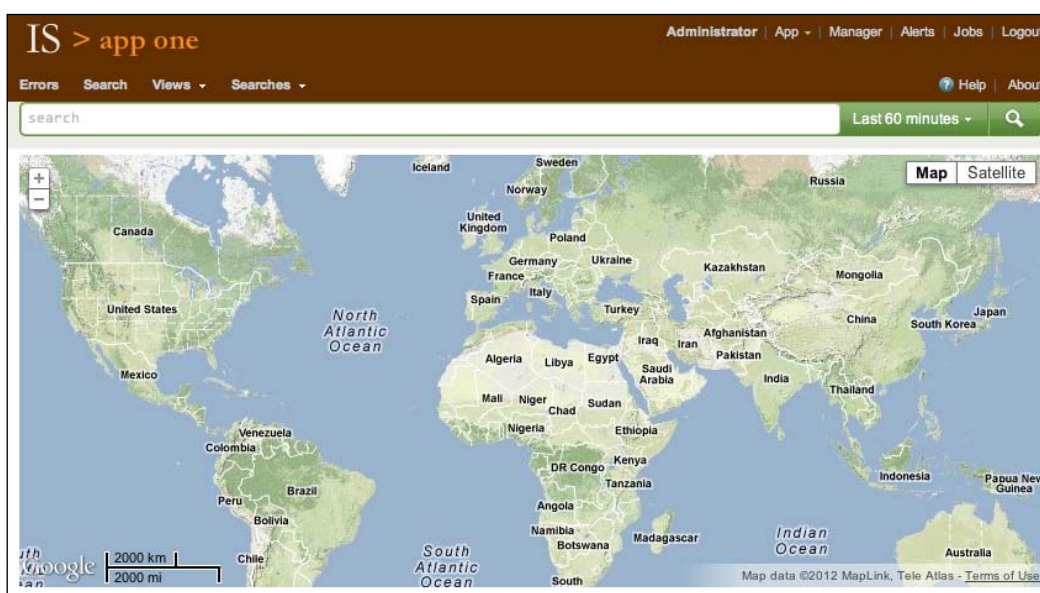
  <!-- search -->
  <module name="SearchBar" layoutPanel="splSearchControls-inline">
    <param name="useOwnSubmitButton">False</param>
    <module name="TimeRangePicker">
      <param name="selected">Last 60 minutes</param>
    <module name="SubmitButton">
```

```

<!-- map -->
<module
  name="GoogleMaps"
  layoutPanel="resultsAreaLeft"
  group="Map" />
</module>
</module>
</module>
</view>

```

This code produces a search bar with a map under it, as seen here in the following screenshot:



When using the `GoogleMaps` module, you would usually convert some set of values to geographic coordinates. This is usually accomplished using the `geoip` lookup (see *Chapter 7, Working with Apps*, for examples) to convert IP addresses to locations, or by using a custom lookup of some sort.

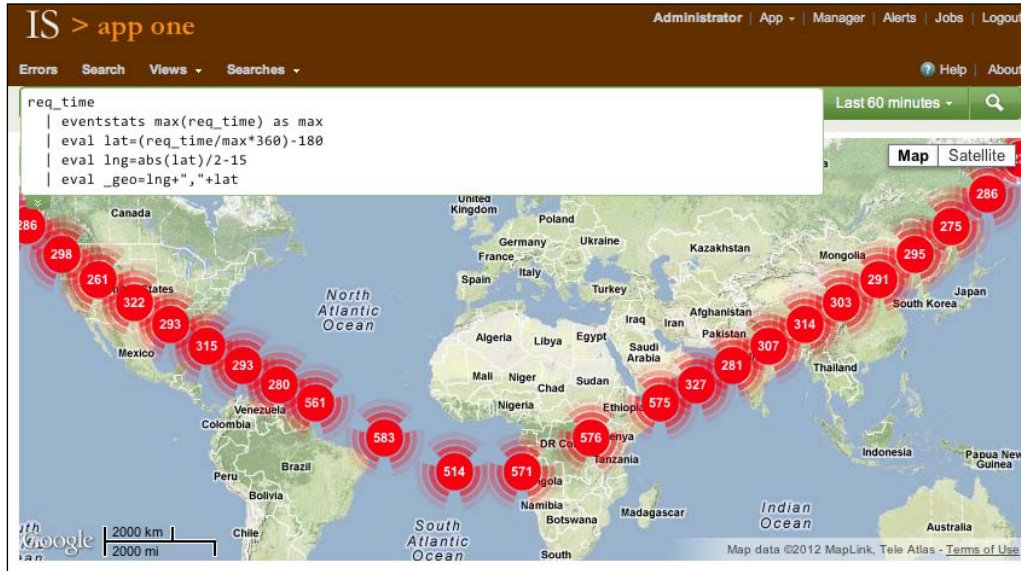
Just to show that the data can come from anywhere, let's make a graph by setting the `_geo` field on events from one of our example source types:

```

sourcetype="impl_splunk_gen" req_time
| eventstats max(req_time) as max
| eval lat=(req_time/max*360)-180
| eval lng=abs(lat)/2-15
| eval _geo=lng+", "+lat

```

This query will produce a "V" from our random `req_time` field, as shown in the following screenshot. See the maps documentation at splunkbase.com for more information about the `_geo` field:



This is a very simplistic example, using the default settings for nearly everything. For a more complete example, see the **Google Maps** dashboard included with the **Google Maps** app. You can see the source code in the manager, or by using the `showsourc` attribute. On my server, that URL would be <http://mysplunkserver:8000/en-US/app/maps/maps?showsourc=1>.

Sideview Utils

Sideview Utils is a third-party app for Splunk that provides an alternative set of modules for most of what you need to build an interactive Splunk dashboard. These modules remove the complexity of intentions, make it much easier to build forms, make it possible to use variables in HTML, and make it much simpler to hand values between panels and dashboards.

We will use a few of the modules to build forms and link multiple dashboards together based on URL values.

An older but still functional version of `SideviewUtils` is available through Splunkbase. You can download the latest version from <http://sideviewapps.com/>, which adds a number of features, including a visual editor for assembling dashboards.

The Sideview Search module

Let's start with a simple search:

```
<?xml version="1.0"?>
<view template="dashboard.html">

  <!-- add sideview -->
  <module layoutPanel="appHeader" name="SideviewUtils"/>

  <!-- chrome -->
  <label>Chapter 8 - Sideview One</label>
  <module name="AccountBar" layoutPanel="appHeader"/>
  <module name="AppBar" layoutPanel="navigationHeader"/>
  <module name="Message" layoutPanel="messaging">
    <param name="filter">*</param>
    <param name="clearOnJobDispatch">False</param>
    <param name="maxSize">1</param>
  </module>

  <!-- search -->
  <module
    name="Search"
    autoRun="True"
    group="Chapter 8 - Sideview One"
    layoutPanel="panel_row1_col1">
    <param name="earliest">-1h</param>
    <param name="search">source="impl_splunk_gen" | top user</param>

    <!-- chart -->
    <module name="HiddenChartFormatter">
      <param name="charting.chart">pie</param>
      <module name="JSChart"/>
    </module>
  </module>
</view>
```

This dashboard renders identically to the first panel, previously described in the *Building a drilldown to a custom query* section. There are two things to notice in this example:

1. The `SideviewUtils` module is needed to include the code needed by all *Sideview Utils* apps.
2. We use the alternative `Search` module as a replacement for the `HiddenSearch` module to illustrate our first `SideviewUtils` module. In this simplistic example, `HiddenSearch` would still work.

Linking views with Sideview

Starting from our simple dashboard, let's use the `Redirector` module to build a link. This link could be to anything, but we will link to another Splunk dashboard, which we will build next. Here's the XML:

```
...
<module name="JSChart">
  <module name="Redirector">
    <param name="arg.user">${click.value}</param>
    <param name="url">chapter_8_sideview_2</param>
  </module>
</module>
...
```

After clicking on **mary**, a new URL is built using the user value. In my case, the URL is:

```
http://mysplunkserver:8000/en-US/app/is_app_one/chapter_8_
sideview_2?user=mary
```

The dashboard referenced does not exist yet, so this URL will return an error. Let's create the second dashboard now.

Sideview URLLoader

The `URLLoader` module provides the ability to set variables from the query string of a URL, a very useful feature. For our next dashboard, we will draw a table showing the error counts for the `user` value provided in the URL:

```
<view template="dashboard.html">

  <!-- add sideview -->
  <module name="SideviewUtils" layoutPanel="appHeader"/>

  <!-- chrome -->
  <label>Chapter 8 - Sideview Two</label>
  <module name="AccountBar" layoutPanel="appHeader"/>
  <module name="AppBar" layoutPanel="navigationHeader"/>
  <module name="Message" layoutPanel="messaging">
    <param name="filter">*</param>
    <param name="clearOnJobDispatch">False</param>
    <param name="maxSize">1</param>
  </module>
</view>
```

```

</module>

<!-- search -->
<module
  name="URLLoader"
  layoutPanel="panel_row1_col1"
  autoRun="True">
  <module name="HTML">
    <param name="html"><![CDATA[
      <h2>Errors by logger for $user$.</h2>
    ]]>
    </param>
  </module>
  <module name="Search" group="Chapter 8 - Sideview Two">
    <param name="earliest">-1h</param>
    <param name="search">
      source="impl_splunk_gen" user=$user$
      | top logger
    </param>

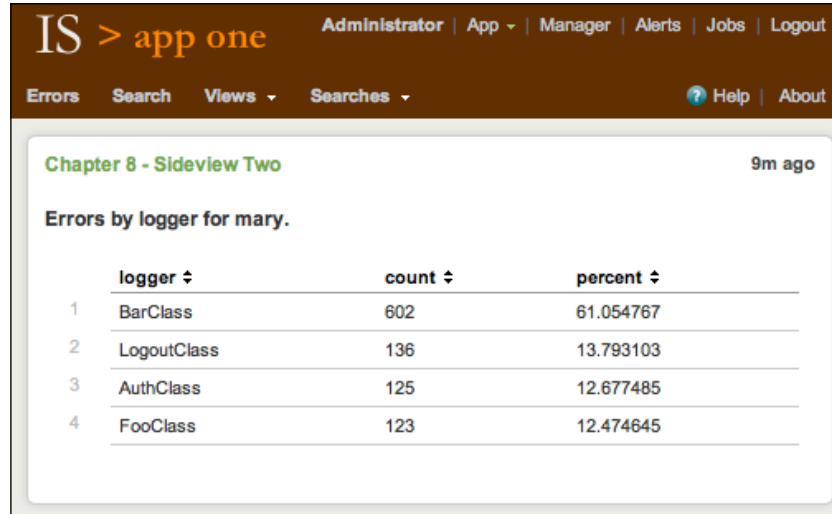
    <!-- table -->
    <module name="SimpleResultsTable">
      <param name="drilldown">row</param>
      <module name="Redirector">
        <param name="url">chapter_8_sideview_3</param>
        <param name="arg.logger">
          $click.fields.logger.rawValue$
        </param>
        <param name="arg.user">$user$</param>
        <param name="arg.earliest">
          $search.timeRange.earliest$
        </param>
      </module>
    </module>
  </module>
</module>
</view>

```



It is very important that `autoRun="true"` be placed in one module, most likely `URLLoader`, and that it exists only in a single module.

With the value of *user* as *mary* in our URL, this dashboard creates the simple view:



The screenshot shows a dashboard interface with a dark header. The header contains the text 'IS > app one' on the left, and navigation links 'Administrator | App | Manager | Alerts | Jobs | Logout' on the right. Below the header is a secondary navigation bar with 'Errors', 'Search', 'Views', 'Searches', 'Help', and 'About'. The main content area is titled 'Chapter 8 - Sideview Two' and '9m ago'. Below this is a section titled 'Errors by logger for mary.' containing a table with three columns: 'logger', 'count', and 'percent'. The table lists four entries: BarClass (602, 61.054767%), LogoutClass (136, 13.793103%), AuthClass (125, 12.677485%), and FooClass (123, 12.474645%).


	logger ↕	count ↕	percent ↕
1	BarClass	602	61.054767
2	LogoutClass	136	13.793103
3	AuthClass	125	12.677485
4	FooClass	123	12.474645

Looking at the modules in this example that are of interest, we see:

- **SideviewUtils:** This module is required to use any of the other Sideview modules. It is invisible to the user, but is still required.
- **URLLoader:** This module takes any values specified in the URL query string and turns them into variables to be used by the descendant modules. Our URL contains `user=mary`, so `$user$` will be replaced with the value `mary`.
- **HTML:** This module draws a snippet of HTML inline. Variables from `URLLoader` and from form elements are replaced.
- **Search:** This replacement for `HiddenSearch` understands variables from `URLLoader` and form elements. This completely obviates the need for intentions. In our case, `$user$` will be replaced.
- **Redirector:** In this example, we are going to hand along two values to the next dashboard—`user` from `URLLoader`, and `logger` from the table itself.

A few things to notice:

- `logger` will be populated with `$click.fields.logger.rawValue$`.
- When a table is clicked on, the variable `click.fields` contains all fields from the row of the table clicked on.
- `rawValue` makes sure the unescaped value is returned. As the Sideview docs say: *Rule of Thumb - for displaying in headers and sending via redirects, use `$foo.rawValue$`. For searches, use `foo`.*

 This rule applies to values in `Redirector`, not in `display`.

- `search.timeRange` contains information about the times used by this search, whether it comes from the URL, a `TimeRangePicker`, or params to the `Search` module. `arg.earliest` will add the value to the URL.

With a click on the table row for **LogoutClass**, we are taken to the following URL:

```
http://mysplunkserver:8000/en-US/app/is_app_one/chapter_8_sideview_3?
user=mar&ylogger=LogoutClass&earliest=1344188377
```

We will create the dashboard at this URL in the next section.

Sideview forms

For our final dashboard using `Sideview` modules, we will build a dashboard with a form that can be prefilled from a URL, and allows changing the time range. The advantage of this dashboard is that it can be used as a destination of a click without being linked to from elsewhere. If the user accesses this dashboard directly, the default values specified in the dashboard will be used instead. Let's look at the code:

```
<?xml version="1.0"?>
<view template="dashboard.html">

  <!-- add sideview -->
  <module name="SideviewUtils" layoutPanel="appHeader"/>

  <!-- chrome -->
  <label>Chapter 8 - Sideview Three</label>
  <module name="AccountBar" layoutPanel="appHeader"/>
  <module name="AppBar" layoutPanel="navigationHeader"/>
  <module name="Message" layoutPanel="messaging">
    <param name="filter">*</param>
    <param name="clearOnJobDispatch">False</param>
    <param name="maxSize">1</param>
  </module>

  <!-- URLLoader -->
  <module
    name="URLLoader"
    layoutPanel="panel_row1_col1"
```

```
        autoRun="True">

<!-- form -->

<!-- user dropdown -->
<module name="Search" layoutPanel="panel_row1_col1">
  <param name="search">
    source="impl_splunk_gen" user user="*"
    | top user
  </param>
  <param name="earliest">-24h</param>
  <param name="latest">now</param>

  <module name="Pulldown">
    <param name="name">user</param>
    <!-- use valueField in SideView 2.0 -->
    <param name="searchFieldsToDisplay">
      <list>
        <param name="value">user</param>
        <param name="label">user</param>
      </list>
    </param>
    <param name="label">User</param>
    <param name="float">left</param>

<!-- logger textfield -->
<module name="TextField">
  <param name="name">logger</param>
  <param name="default">*</param>
  <param name="label">Logger:</param>
  <param name="float">left</param>
  <module name="TimeRangePicker">
    <param name="searchWhenChanged">True</param>
    <param name="default">Last 24 hours</param>

<!-- submit button -->
<module name="SubmitButton">
  <param name="allowSoftSubmit">True</param>

<!-- html -->
<module name="HTML">

  <param name="html"><![CDATA[
    <h2>Info for user $user$, logger $logger$.</h2>
  ]]></param>
</module>
```

```
<!-- search 1 -->
<module
  name="Search"
  group="Chapter 8 - Sideview Three">
  <param name="search">
    source="impl_splunk_gen" user="$user$"
    logger="$logger$"
    | fillnull value="unknown" network
    | timechart count by network
  </param>

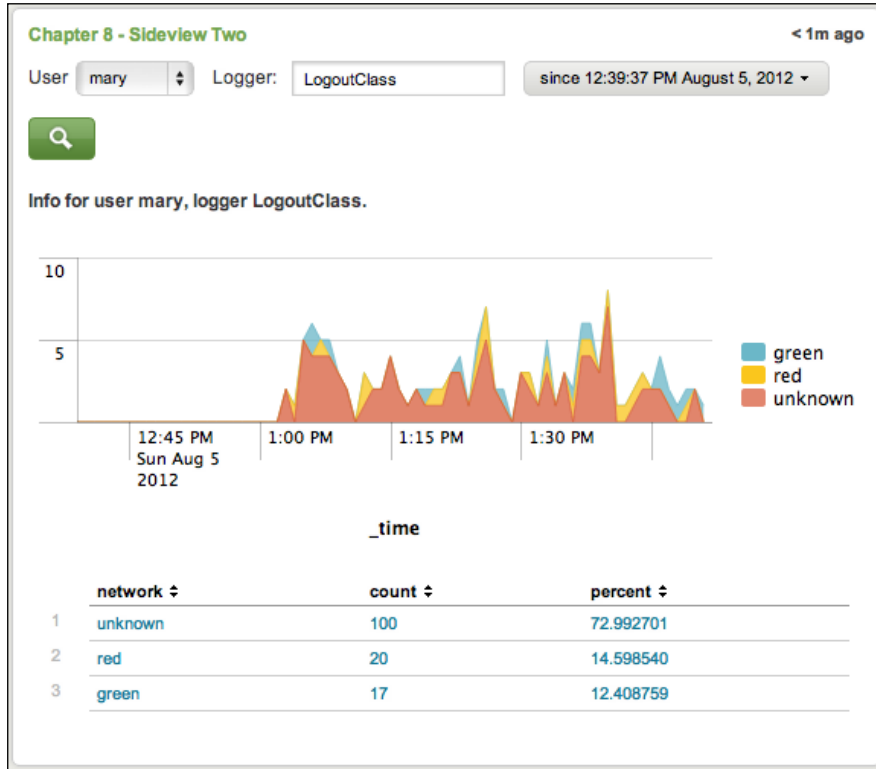
  <!-- JobProgressIndicator -->
  <module name="JobProgressIndicator"/>

  <!-- chart -->
  <module name="HiddenChartFormatter">
    <param name="charting.chart">area</param>
    <param name="charting.chart.stackMode">
      stacked
    </param>
    <module name="JSChart"/>
  </module>
</module>

<!-- search 2 -->
<module
  name="Search"
  group="Chapter 8 - Sideview Three">
  <param name="search">
    source="impl_splunk_gen" user="$user$"
    logger="$logger$"
    | fillnull value="unknown" network
    | top network
  </param>

  <!-- table -->
  <module name="SimpleResultsTable"/>
</module>
</module>
</module>
</module>
</module>
</module>
</view>
```

This draws a dashboard like this:



There are quite a few things to cover in this example, so let's step through portions of the XML.

Include SideviewUtils to enable the other Sideview modules. In this case, URLLoader, HTML, Pulldown, Search, and TextField are Sideview modules.

```
<module layoutPanel="appHeader" name="SideviewUtils"/>
```

Wrap everything in URLLoader so that we get values from the URL:

```
<module  
  name="URLLoader"  
  layoutPanel="panel_row1_col1"  
  autoRun="True">
```

Start a search to populate the user dropdown. This query will find all users in the last 24 hours:

```
<module name="Search" layoutPanel="panel_row1_col1">
  <param name="search">
    source="impl_splunk_gen" user user="*"
    | top user
  </param>
  <param name="earliest">-24h</param>
  <param name="latest">now</param>
```



Using a query to populate a dropdown can be very expensive, particularly as your data volumes increase. You may need to precalculate these values, either storing the values in a CSV using `outputcsv` and `inputcsv`, or using a summary index. See *Chapter 9, Summary Indexes and CSV Files*, for examples of summary indexing and using CSV files for transient data.

This module draws the user selector. The menu is filled by the `Search` module previously, but notice that the value selected is from our URL value:

```
<module name="Pulldown">
  <!-- use valueField in SideView 2.0 -->
  <param name="searchFieldsToDisplay">
    <list>
      <param name="value">user</param>
      <param name="label">user</param>
    </list>
  </param>
  <param name="name">user</param>
  <param name="label">User</param>
  <param name="float">left</param>
```

Next is a text field for our logger. This is a Sideview version of `ExtendedFieldSearch`. It will prepopulate using upstream variables:

```
<module name="TextField">
  <param name="name">logger</param>
  <param name="default">*</param>
  <param name="label">Logger:</param>
  <param name="float">left</param>
```


The `TimeRangePicker` module will honor the values earliest and latest in the URL. Note that `searchWhenChanged` must be `True` to work properly in this case. As a rule of thumb, `searchWhenChanged` should always be `True`.

```
<module name="TimeRangePicker">
  <param name="searchWhenChanged">True</param>
  <param name="default">Last 24 hours</param>
```

The `SubmitButton` module will kick off a search when values are changed. `allowSoftSubmit` allows outer modules to start the query, either by choosing a value or hitting return in a text field.

```
<module name="SubmitButton">
  <param name="allowSoftSubmit">True</param>
```

Next are two `Search` modules, each containing an output module:

```
<module
  name="Search"
  group="Chapter 8 - Sideview Three">
  <param name="search">
    source="impl_splunk_gen" user="$user$"
    logger="$logger$"
    | fillnull value="unknown" network
    | timechart count by network
  </param>

  <!-- JobProgressIndicator -->
  <module name="JobProgressIndicator"/>

  <!-- chart -->
  <module name="HiddenChartFormatter">
    <param name="charting.chart">area</param>
    <param name="charting.chart.stackMode">
      stacked
    </param>
    <module name="JSChart"/>
  </module>
</module>

<!-- search 2 -->
<module
```

```
        group="Chapter 8 - Sideview Three"
        name="Search">
<param name="search">
    source="impl_splunk_gen" user="$user$"
    logger="$logger$"
    | fillnull value="unknown" network
    | top network
</param>

<!-- table -->
<module name="SimpleResultsTable">
    <param name="drilldown">row</param>
</module>
```

...

For greater efficiency, these two searches could be combined into one query and the `PostProcess` module used.

Summary

We have covered an enormous amount of ground in this chapter. The toughest concepts we touched on were module nesting, the meaning of `layoutPanel`, intentions, and an alternative to intentions with *SideView Utils*. As with many skills, the best way to become proficient is to dig in, and hopefully have some fun along the way! The examples in this chapter should give you a head start.

In the next chapter, we will cover summary indexing, a powerful part of Splunk that can improve the efficiency of your queries greatly.

9

Summary Indexes and CSV Files

As the number of events retrieved by a query increases, performance decreases linearly. Summary indexing allows you to calculate statistics in advance, then run reports against these "roll ups", dramatically increasing performance.

Understanding summary indexes

A **summary index** is a place to store events calculated by Splunk. Usually, these events are aggregates of raw events broken up over time, for instance, how many errors occurred per hour. By calculating this information on an hourly basis, it is cheap and fast to run a query over a longer period of time, for instance, days, weeks, or months.

A summary index is usually populated from a saved search with **Summary indexing** enabled as an action. This is not the only way, but is certainly the most common.

On disk, a summary index is identical to any other Splunk index. The difference is solely the source of data. We create the index through configuration or through the GUI like any other index, and we manage the index size in the same way.



Think of an index like a table, or possibly a tablespace in a typical SQL database. Indexes are capped by size and/or time, much like a tablespace, but all the data is stored together, much like a table. We will discuss index management in *Chapter 10, Configuring Splunk*.

Creating a summary index

To create an index, navigate to **Manager** | **Indexes** | **Add new**.

Add new

Index settings

Index name *

Set index name (e.g., INDEX_NAME). Search using index=INDEX_NAME.

Home path

Hot/warm db path. Leave blank for default (\$SPLUNK_DB/INDEX_NAME/db).

Cold path

Cold db path. Leave blank for default (\$SPLUNK_DB/INDEX_NAME/colddb).

Thawed path

Thawed/resurrected db path. Leave blank for default (\$SPLUNK_DB/INDEX_NAME/thaweddb).

Max size (MB) of entire index

Maximum target size of entire index.

Max size (MB) of hot/warm/cold bucket

Maximum target size of buckets. Enter 'auto_high_volume' for high-volume indexes.

Frozen archive path

Frozen bucket archive path. Set this if you want Splunk to automatically archive frozen buckets.

For now, let's simply give our new index a name and accept the default values. We will discuss these settings under the *indexes.conf* section in *Chapter 10, Configuring Splunk*. I like to put the word `summary` at the beginning of any summary index, but the name does not matter. I would suggest you follow some naming convention that makes sense to you.

Now that we have an index to store events in, let's do something with it.

When to use a summary index

When the question you want to answer requires looking at all or most events for a given source type, very quickly the number of events can become huge. This is what is generally referred to as a "dense search".

For example, if you want to know how many page views happened on your website, the query to answer this question must inspect every event. Since each query uses a processor, we are essentially timing how fast our disk can retrieve the raw data and how fast a single processor can decompress that data. Doing a little math:

1,000,000 hits per day /
10,000 events processed per second =
100 seconds

If we use multiple indexers, or possibly buy much faster disks, we can cut this time, but only linearly. For instance, if the data is evenly split across four indexers, without changing disks, this query will take roughly 25 seconds.

If we use summary indexing, we should be able to improve our times dramatically. Let's assume we have calculated hit counts per five minutes. Now doing the math:

*24 hours * 60 minutes per hour / 5 minute slices =*
288 summary events

If we then use those summary events in a query, the math looks like:

288 summary events /
10,000 events processed per second =
.0288 seconds

This is a significant increase in performance. In reality, we would probably store more than 288 events. For instance, let's say we want to count events by their HTTP response code. Assuming there are 10 different status codes we see on a regular basis, we have:

*24 hours * 60 minutes per hour / 5 minute slices * 10 codes =*
2880 events

The math then looks like:

$$\frac{2,880 \text{ summary events}}{10,000 \text{ events processed per second}} = .288 \text{ seconds}$$

That's still a significant improvement over 100 seconds.

When to not use a summary index

There are several cases where summary indexes are either inappropriate or inefficient. Consider the following:

- **When you need to see the original events:** In most cases, summary indexes are used to store aggregate values. A summary index could be used to store a separate copy of events, but this is not usually the case. The more events you have in your summary index, the less advantage it has over the original index.
- **When the possible number of categories of data is huge:** For example, if you want to know the top IP addresses seen per day, it may be tempting to simply capture a count of every IP address seen. This can still be a huge amount of data, and may not save you a lot of search time, if any. Likewise, simply storing the top 10 addresses per slice of time may not give an accurate picture over a long period of time. We will discuss this scenario under the *Calculating top for a large time frame* section.
- **When it is impractical to slice the data across sufficient dimensions:** If your data has a large number of dimensions or attributes, and it is useful to slice the data across a large number of these dimensions, then the resulting summary index may not be sufficiently smaller than your original index to bother with.
- **When it is difficult to know the acceptable time slice:** As we set up a few summary indexes, we have to pick the slice of time to which we aggregate. If you think 1 hour is an acceptable time slice, and you find out later that you really need 10 minutes of resolution, it is not the easiest task to recalculate the old data into these 10-minute slices. It is, however, very simple to later change your 10-minute search to one hour, as the 10-minute slices should still work for your hourly reports.

Populating summary indexes with saved searches

A search to populate a summary index is much like any other saved search (see *Chapter 2, Understand Search*, for more detail on creating saved searches). The differences are that this search will run periodically and the results will be stored in the summary index. Let's build our first summary search by following these steps:

1. Start with a search that produces some statistic:
`source="impl_splunk_gen" | stats count by user`
2. Save this search as `summary - count by user`.
3. Edit the search in **Manager** by navigating to **Manager | Searches and reports | summary - count by user**. The **Save search...** wizard provides a link to the manager on the last dialog in the wizard.
4. Set the appropriate times. This is a somewhat complicated discussion. See the *How latency affects summary queries* section discussed later.

splunk>
 Manager » Searches and reports » summary - count by user

summary - count by user

Search *

source="impl_splunk_gen" | stats count by user

Description

Time range

Start time: -62m@m Finish time: -2m@m

Time specifiers: y, mon, d, h, m, s
[Learn more](#)

Schedule and alert

Schedule this search

Schedule type: Cron

Cron schedule: 2 * * * *

Enter a cron-style schedule.
 For example "5 * * * *" (every 5 minutes) or "0 21 * * *" (every day at 9 PM).

Summary Indexing

Enable

Select the summary index: summary_impl_splunk

Only indexes that you can write to are listed.

Add fields: [] = [] [Delete](#)

[Add another field](#)

[Cancel](#) [Save](#)

Let's look at the following fields:

- **Search:** `source="impl_splunk_gen" | stats count by user`
This is our query. Later we will use `sistats`, a special summary index version of `stats`.
- **Start time:** `-62m@m`
It may seem strange that we didn't simply say `-60m@m`, but we need to take latency into account. See the *How latency affects summary queries* section discussed later for more details.
- **Finish time:** `-2m@m`
- **Schedule and Alert | Schedule this search:** This checkbox needs to be checked for the query to run on a schedule.
- **Schedule type:** **Cron**
- **Cron schedule:** `2 * * * *`
This indicates that the query runs on minute 2 of every hour, every day. To accommodate for latency, **Cron schedule** is shifted after the beginning of the hour along with the start and finish times. See the *How latency affects summary queries* section discussed later for more details.
- **Summary indexing | Enable:** This checkbox enables writing the output to another index.
- **Select the summary index:** `summary_impl_splunk`
This is the index to write our events to.



Non-admin users are only allowed to write to the index summary. This ability is controlled by the `indexes_edit` capability, which only the admin role has enabled by default. See *Chapter 10, Configuring Splunk*, for a discussion on roles and capabilities.

- **Add fields:** Using these fields, you can store extra pieces of information in your summary index. This can be used to group results from multiple summary results, or to tag results.

Using summary index events in a query

After the query to populate the summary index has run for some time, we can use the results in other queries.

If you're in a hurry, or need to report against slices of time before the query was created, you will need to "backfill" your summary index. See the *How and when to backfill summary data* section for details about calculating summary values for past events.

First, let's look at what actually goes into the summary index:

```
08/15/2012 10:00:00, search_name="summary - count by user",
search_now=1345046520.000, info_min_time=1345042800.000, info_max_
time=1345046400.000, info_search_time=1345050512.340, count=17,
user=mary
```

Breaking this event down, we have:

- 08/15/2012 10:00:00: This is the time at the beginning of this block of data. This is consistent with how `timechart` and `bucket` work.
- `search_name="summary - count by user"`: This is the name of the search. This is usually the easiest way to find the results you are interested in.
- `search_now ... info_search_time`: These are informational fields about the summary entry, and are generally not useful to users.
- `count=17, user=mary`: The rest of the entry will be whatever fields were produced by the populating query. There will be one summary event per row produced by the populating query.

Now let's build a query against this data. To start the query, we need to specify the name of the index and the name of the search:

```
index="summary_impl_splunk" search_name="summary - count by user"
```

On my machine, this query loads 48 events, compared to the 22,477 original events.

Using `stats`, we can quickly find the statistics by user:

```
index="summary_impl_splunk" | stats sum(count) count by user
```

This produces a very simple table, as shown in the following screenshot:

	user ↕	sum(count) ↕	count ↕
1	Bobby	12113	16
2	bob	11845	16
3	extrauser	3612	16
4	jacky	12158	16
5	linda	12057	16
6	mary	24092	16

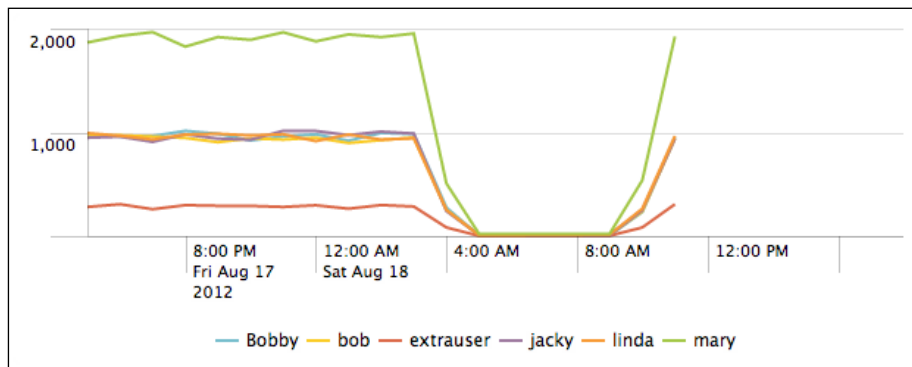
We are calculating `sum(count)` and `count` in this query, which you might expect to produce the same number, but they are doing very different things:

- `sum(count)`: If you look back at our raw event, `count` contains the number of times that user appeared in that slice of time. We are storing the raw value in this `count` field. See the *Using sistats, sitop, and sitimechart* section for a completely different approach.
- `count`: This actually represents the number of events in the summary index. The generator that is producing these events is not very random, so all users produce at least one event per hour.

Producing a timechart is no more complicated:

```
index="summary_impl_splunk" | timechart span=1h sum(count) by user
```

This produces our graph as shown in the following screenshot:



The main thing to remember here is that *we cannot make a graph more detailed than the schedule of our populating query*. In this case, the populating query uses a span of one hour. 1 hour is granular enough for most daily reports, and certainly fine for weekly or monthly reports, but it may not be granular enough for an operations dashboard.

The following are a few other interesting queries you could make with this simple set of data:

```
index="summary_impl_splunk" search_name="summary - count by user"
  | stats avg(count) as "Average events per hour"
```

The previous code snippet tells us the average number of events per slice of time, which we know is an hour. Adding `bucket` and another `stats` command, we can calculate for a custom period of time, as follows:

```
index="summary_impl_splunk" search_name="summary - count by user"
  | bucket span=4h _time
  | stats sum(count) as count by _time
  | stats avg(count) as "Average events per 4 hours"
```

This query would give us the user with the maximum number of events in a given hour, and the hour it happened in:

```
index="summary_impl_splunk" search_name="summary - count by user"
  | stats first(_time) as _time max(count) as max by user
  | sort -max
  | head 1
  | rename max as "Maximum events per hour"
```

Using `sistats`, `sitop`, and `sitimechart`

So far we have used the `stats` command to populate our summary index. While this works perfectly well, the `si*` variants have a couple of advantages:

- The remaining portion of the query does not have to be rewritten. For instance, `stats count` still works as if you were counting the raw events.
- `stats` functions that require more data than what happened in that slice of time will still work. For example, if your time slices each represent an hour, it is not possible to calculate the average value for a day using nothing but the average of each hour. `sistats` keeps enough information to make this work.

There are a few fairly serious disadvantages to be aware of:

- The query using the summary index *must* use a subset of the functions and split fields that were in the original populating query. If the subsequent query strays from what is in the original `sistats` data, the results may be unexpected and difficult to debug. For example:

- The following code works fine:

```
source="impl_splunk_gen"
| sitimechart span=1h avg(req_time) by user
| stats avg(req_time)
```

- The following code returns unpredictable and wildly incorrect values:

```
source="impl_splunk_gen"
| sitimechart span=1h avg(req_time) by user
| stats max(req_time)
```

Notice that `avg` went into `sistats`, but we tried to calculate `max` from the results.

- Using `dc` (distinct count) with `sistats` can produce huge events. This happens because to accurately determine unique values over slices of time, all original values must be kept. One common use case is to find the top IP addresses that hit a public facing server. See the *Calculating top for a large time frame* section for alternate approaches to this problem.
- The contents of the summary index are quite difficult to read as they are not meant to be used by humans.

To see how all of this works, let's build a few queries. We start with a simple `stats` query as follows:

```
sourcetype=impl_splunk_gen
| stats count max(req_time) avg(req_time) min(req_time) by user
```

This produces results like you would expect:

	user ↕	count ↕	max(req_time) ↕	avg(req_time) ↕	min(req_time) ↕
1	Bobby	11459	12239	6136.885004	1
2	bob	11294	12237	6107.613410	2
3	extrauser	3464	12239	6128.628753	6
4	jacky	11473	12236	6142.702905	1
5	linda	11375	12237	6107.128160	2
6	mary	23098	12239	6131.918991	1

Now, we could save this and send it straight to the summary index, but the results are not terribly nice to use, and the average of the average would not be accurate. On the other hand, we can use the `sistats` variant as follows:

```
sourcetype=impl_splunk_gen
| sistats count max(req_time) avg(req_time) min(req_time) by user
```

The results have a lot of extra information not meant for humans as shown in the following screenshot:

	psrsvd_ct_req_time ↕	psrsvd_gc ↕	psrsvd_nc_req_time ↕	psrsvd_nn_req_time ↕	psrsvd_nx_req_time ↕	psrsvd...
1	8609	11459	8609	1	12239	528324
2	8531	11294	8531	2	12237	521040
3	3464	3464	3464	6	12239	212295
4	8674	11473	8674	1	12236	532816
5	8505	11375	8505	2	12237	519411
6	17282	23098	17282	1	12239	105971

Splunk knows how to deal with these results, and can use them in combination with the `stats` functions as if they were the original results. You can see how `sistats` and `stats` work together by chaining them together, as follows:

```
sourcetype=impl_splunk_gen
| sistats
  count max(req_time) avg(req_time) min(req_time)
  by user
| stats count max(req_time) avg(req_time) min(req_time) by user
```

Even though the `stats` function is not receiving the original events, it knows how to work with these `sistats` summary events. We are presented with exactly the same results as the original query, as shown in the following screenshot:

	user ↕	count ↕	max(req_time) ↕	avg(req_time) ↕	min(req_time) ↕
1	Bobby	11459	12239	6136.885004	1
2	bob	11294	12237	6107.613410	2
3	extrauser	3464	12239	6128.628753	6
4	jacky	11473	12236	6142.702905	1
5	linda	11375	12237	6107.128160	2
6	mary	23098	12239	6131.918991	1

`sitop` and `sitimechart` work in the same fashion.

Let's step through the procedure to set up summary searches as follows:

1. Save the query using `sistats`.

```
sourcetype=impl_splunk_gen  
| sistats count max(req_time) avg(req_time) min(req_time) by  
user
```
2. Set the times accordingly, as we saw previously in the *Populating summary indexes with saved searches* section. See the *How latency affects summary queries* section for more information.
3. Build a query that queries the summary index, as we saw previously in the *Using summary index events in a query* section. Assuming we saved this query as `testing sistats`, the query would be: `index="summary_impl_splunk" search_name="testing sistats"`.
4. Use the original `stats` function against the results, as follows:

```
index="summary_impl_splunk" search_name="testing sistats"  
| stats count max(req_time) avg(req_time) min(req_time) by user
```

This should produce exactly the same results as the original query.

The `si*` variants still seem somewhat magical to me, but they work so well that it is in your own best interest to dive in and trust the magic. Be very sure that your functions and fields are a subset of the original!

How latency affects summary queries

Latency is the difference between the time assigned to an event (usually parsed from the text) and the time it was written to the index. Both times are captured, in `_time` and `_indextime`, respectively.

This query will show us what our latency is:

```
sourcetype=impl_splunk_gen  
| eval latency = _indextime - _time  
| stats min(latency) avg(latency) max(latency)
```

In my case, these statistics look as shown in the following screenshot:

	min(latency) ↕	avg(latency) ↕	max(latency) ↕
↑	-0.465	31.603530	72.390

The latency in this case is exaggerated, because the script behind `impl_splunk_gen` is creating events in chunks. In most production Splunk instances, the latency is usually just a few seconds. If there is any slowdown, perhaps because of network issues, the latency may increase dramatically, and so it should be accounted for.

This query will produce a table showing the time for every event:

```
sourcetype=impl_splunk_gen
| eval latency = _indextime - _time
| eval time=strftime(_time,"%Y-%m-%d %H:%M:%S.%3N")
| eval indextime=strftime(_indextime,"%Y-%m-%d %H:%M:%S.%3N")
| table time indextime latency
```

The previous query produces the following table:

	time ↕	indextime ↕	latency ↕
51	2012-08-22 21:38:11.107	2012-08-22 21:38:33.000	21.893
52	2012-08-22 21:38:11.011	2012-08-22 21:38:33.000	21.989
53	2012-08-22 21:38:10.546	2012-08-22 21:38:33.000	22.454
54	2012-08-22 21:38:10.433	2012-08-22 21:38:33.000	22.567
55	2012-08-22 21:38:10.419	2012-08-22 21:38:33.000	22.581
56	2012-08-22 21:38:09.588	2012-08-22 21:38:33.000	23.412
57	2012-08-22 21:38:08.955	2012-08-22 21:38:33.000	24.045
58	2012-08-22 21:38:08.502	2012-08-22 21:38:33.000	24.498
59	2012-08-22 21:38:07.867	2012-08-22 21:38:33.000	25.133

To deal with this latency, you should add enough delay in your query that populates the summary index. The following are a few examples:

Confidence	Time slice	Earliest	Latest	cron
2 minutes	1 hour	-62m@m	-2m@m	2 * * * *
15 minutes	1 hour	-1h@h	-0h@h	15 * * * *
5 minutes	5 minutes	-10m@m	-5m@m	*/5 * * * *
1 hour	15 minutes	-75m@m	-60m@m	*/15 * * * *
1 hour	24 hours	-1d@d	-0d@d	0 1 * * * *



Sometimes you have no idea when your logs will be indexed, as when they are delivered in batches on unreliable networks. This is what I would call "unpredictable latency". For one possible solution, take a look at the app *indextime search* available at <http://splunkbase.com>.

How and when to backfill summary data

If you are building reports against summary data, you of course need enough time represented in your summary index. If your report represents only a day or two, then you can probably just wait for the summary to have enough information. If you need the report to work sooner rather than later, or the time frame is longer, then you can backfill your summary index.

Using `fill_summary_index.py` to backfill

The `fill_summary_index.py` script allows you to backfill the summary index for any time period you like. It does this by running the saved searches you have defined to populate your summary indexes, but for the time periods you specify.

To use the script, follow the given procedure:

1. Create your scheduled search, as detailed previously in the *Populating summary indexes with saved searches* section.
2. Log in to the shell on your Splunk instance. If you are running a distributed environment, log in to the search head.
3. Change directories to the Splunk bin directory. `cd $SPLUNK_HOME/bin`.
`$SPLUNK_HOME` is the root of your Splunk installation. The default installation directory is `/opt/splunk` on Unix operating systems, and `c:\Program Files\Splunk` on Windows.
4. Run the `fill_summary_index` command. An example from inside the script is as follows:

```
./splunk cmd python fill_summary_index.py -app is_app_one -name  
"summary - count by user" -et -30d -lt now -j 8 -dedup true -auth  
admin:changeme
```

Let's break down these arguments in the following manner:

- `./splunk cmd`: This essentially sets environment variables so that whatever runs next has the appropriate settings to find Splunk's libraries and included Python modules.
- `python fill_summary_index.py`: This runs the script itself using the Python executable and modules included with the Splunk distribution.
- `-app is_app_one`: This is the name of the app that contains the summary populating queries in question.
- `-name "summary - count by user"`: The name of the query to run. `*` will run all summary queries contained in the app specified.
- `-et -30d`: This is the earliest time to consider. The appropriate times are determined and used to populate the summary index.
- `-lt now`: This is the latest time to consider.
- `-j 8`: This determines how many queries to run simultaneously.
- `-dedup true`: This is used to determine whether there are no results already for each slice of time. Without this flag, you could end up with duplicate entries in your summary index. For some statistics this wouldn't matter, but for most it would.



If you are concerned that you have summary data that is incomplete, perhaps because summary events were produced while an indexer was unavailable, you should investigate the `delete` command to remove these events first. The `delete` command is not efficient, and should be used sparingly, if at all.

- `-auth admin:changeme`: The auth to run the query.

When you run this script, it will run the query with the appropriate times, as if the query had been run at those times in the past. This can be a very slow process, particularly if the number of slices is large. For instance, slices every 5 minutes for a month would be $30 * 24 * (60/5) = 8,640$ queries.

Using collect to produce custom summary indexes

If the number of events destined for your summary index could be represented in a single report, we can use the `collect` function to create our own summary index entries directly. This has the advantage that we can build our index in one shot, which could be much faster than running the backfill script, which must run one search per slice of time. For instance, if you want to calculate 15-minute slices over a month, the script will fire off 2,880 queries.

If you dig into the code that actually produces summary indexes, you will find that it uses the `collect` command to store events into the specified index. The `collect` command is available to us, and with a little knowledge, we can use it directly.

First, we need to build a query that slices our data by buckets of time as follows:

```
source="impl_splunk_gen"
  | bucket span=1h _time
  | stats count by _time user
```

This gives us a simple table as shown in the following screenshot:

	<u>_time</u> ↕	<u>user</u> ↕	<u>count</u> ↕
1	8/22/12 8:00:00.000 PM	Bobby	549
2	8/22/12 8:00:00.000 PM	bob	565
3	8/22/12 8:00:00.000 PM	extrauser	168
4	8/22/12 8:00:00.000 PM	jacky	551
5	8/22/12 8:00:00.000 PM	linda	588
6	8/22/12 8:00:00.000 PM	mary	1115
7	8/22/12 9:00:00.000 PM	Bobby	960
8	8/22/12 9:00:00.000 PM	bob	979
9	8/22/12 9:00:00.000 PM	extrauser	294
10	8/22/12 9:00:00.000 PM	jacky	942

Notice that there is a row per slice of time, and each user that produced events during that slice of time.

Let's add a few more fields for interest:

```
source="impl_splunk_gen"
  | bucket span=1h _time
  | eval error=if(loglevel="ERROR",1,0)
  | stats count avg(req_time) dc(ip) sum(error) by _time user
```

This gives us a table as shown in the following screenshot:

	_time	user	count	avg(req_time)	dc(ip)	sum(error)
1	8/22/12 8:00:00.000 PM	Bobby	549	5918.018913	6	144
2	8/22/12 8:00:00.000 PM	bob	565	6002.448357	6	117
3	8/22/12 8:00:00.000 PM	extrauser	168	6125.517857	6	40
4	8/22/12 8:00:00.000 PM	jacky	551	6005.267123	6	143
5	8/22/12 8:00:00.000 PM	linda	588	6215.339326	6	130
6	8/22/12 8:00:00.000 PM	mary	1115	6039.061078	6	292
7	8/22/12 9:00:00.000 PM	Bobby	960	6144.366255	6	227
8	8/22/12 9:00:00.000 PM	bob	979	6413.421622	6	229
9	8/22/12 9:00:00.000 PM	extrauser	294	6129.421769	6	88
10	8/22/12 9:00:00.000 PM	jacky	942	6115.462518	6	227

Now, to get ready for our summary index, we switch to `sistats`, and add a `search_name` field as the saved search would. Use `testmode` to make sure everything is working as expected, as follows:

```
source="impl_splunk_gen"
| bucket span=1h _time
| eval error=if(loglevel="ERROR",1,0)
| sistats count avg(req_time) dc(ip) sum(error) by _time user
| eval search_name="summary - user stats"
| collect index=summary_impl_splunk testmode=true
```

The results of this query show us what will actually be written to the summary index, but as this is not designed for humans, let's simply test the round trip by adding the original `stats` statement to the end, as follows:


```
source="impl_splunk_gen"
| bucket span=1h _time
| eval error=if(loglevel="ERROR",1,0)
| sistats count avg(req_time) dc(ip) sum(error) by _time user
| eval search_name="summary - hourly user stats - collect test"
| collect index=summary_impl_splunk testmode=true
| stats count avg(req_time) dc(ip) sum(error) by _time user
```

If we have done everything correctly, the results should be identical to the original table:

	_time	user	count	avg(req_time)	dc(ip)	sum(error)
1	8/22/12 8:00:00.000 PM	Bobby	549	5918.018913	6	144
2	8/22/12 8:00:00.000 PM	bob	565	6002.448357	6	117
3	8/22/12 8:00:00.000 PM	extrauser	168	6125.517857	6	40
4	8/22/12 8:00:00.000 PM	jacky	551	6005.267123	6	143
5	8/22/12 8:00:00.000 PM	linda	588	6215.339326	6	130
6	8/22/12 8:00:00.000 PM	mary	1115	6039.061078	6	292
7	8/22/12 9:00:00.000 PM	Bobby	960	6144.366255	6	227
8	8/22/12 9:00:00.000 PM	bob	979	6413.421622	6	229
9	8/22/12 9:00:00.000 PM	extrauser	294	6129.421769	6	88
10	8/22/12 9:00:00.000 PM	jacky	942	6115.462518	6	227

To actually run this query, we simply remove `testmode` from `collect`, as follows:

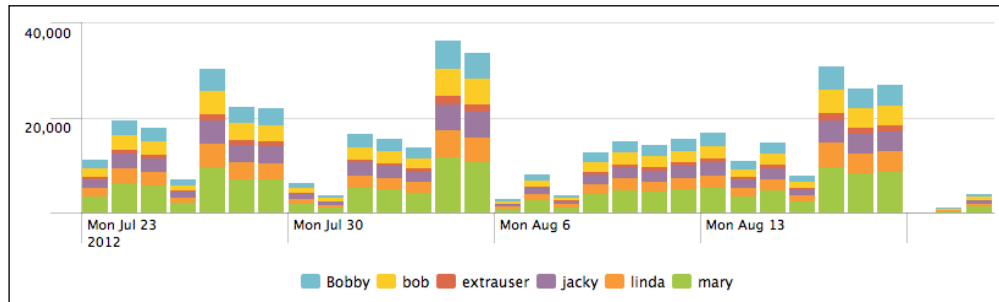
```
source="impl_splunk_gen"
| bucket span=1h _time
| eval error=if(loglevel="ERROR",1,0)
| sistats count avg(req_time) dc(ip) sum(error) by _time user
| eval search_name="summary - user stats"
| collect index=summary_impl_splunk
```

 Beware that you will end up with duplicate values if you use the `collect` command over a time frame that already has results in the summary index. Either use a custom time frame to ensure you do not produce duplicates, or investigate the `delete` command, which as mentioned earlier, is not efficient, and should be avoided if possible.

No results will be available until the query is complete and the file created behind the scenes is indexed. On my installation, querying one month of data, the query inspected 2.2 million events in 173 seconds, producing 2,619 summary events. Let's use the summary data now:

```
index=summary_impl_splunk
search_name="summary - hourly user stats - collect test"
| timechart sum(error) by user
```

This will give us a neat graph as shown in the following screenshot:



Because this is created from the summary, instead of three minutes, this query completes in 1.5 seconds.

In this specific case, using `collect` was four times faster than using the `fill_summary_index.py` script. That said, it is much easier to make a mistake, so be very careful. Rehearse with `collect testmode=true` and a trailing `stats` or `timechart` command.

Reducing summary index size

If the saved search populating a summary index produces too many results, the summary index is less effective at speeding up searches. This usually occurs because one or more of the fields used for grouping has more unique values than is expected.

One common example of a field that can have many unique values is the URL in a web access log. The number of URL values might increase in instances where:

- The URL contains a session ID
- The URL contains search terms
- Hackers are throwing URLs at your site trying to break in
- Your security team runs tools looking for vulnerabilities

On top of this, multiple URLs can represent exactly the same resource, as follows:

- `/home/index.html`
- `/home/`
- `/home/index.html?a=b`
- `/home/?a=b`

We will cover a few approaches to flatten these values. These are just examples and ideas, as your particular case may require a different approach.

Using eval and rex to define grouping fields

One way to tackle this problem is to make up a new field from the URL using `rex`. Perhaps you only really care about hits by directories. We can accomplish this with `rex`, or if needed, multiple `rex` statements.

Looking at the fictional source type `impl_splunk_web`, we see results that look like the following:

```
2012-08-25T20:18:01 user=bobby GET /products/x/?q=10471480 uid=Mzg2NDc00A
2012-08-25T20:18:03 user=user3 GET /bar?q=923891 uid=MjY1NDI5MA
2012-08-25T20:18:05 user=user3 GET /products/index.html?q=9029891
uid=MjY1NDI5MA
2012-08-25T20:18:08 user=user2 GET /about/?q=9376559 uid=MzA4MTc5OA
```

URLs are tricky, as they might or might not contain certain parts of the URL. For instance, the URL may or may not have a query string, may or may not have a page, and may or may not have a trailing slash. To deal with this, instead of trying to make an all encompassing regular expression, we will take advantage of the behavior of `rex`, which is to make no changes to the event if the pattern does not match. Consider the following query:

```
sourcetype="impl_splunk_web"
| rex "\s[A-Z]+\s(?:P<url>.*?)\s"
| rex field=url "(?P<url>.*)\?"
| rex field=url "(?P<url>.*\/)"
| stats count by url
```

In our case, this will produce the following report:

	url ↕	count ↕
1	/	5741
2	/about/	2822
3	/contact/	2847
4	/products/	5653
5	/products/x/	5637
6	/products/y/	2786

Stepping through these `rex` statements we have:

- `rex "\s[A-Z]+\s(?:P<url>.*?)\s"`: This pattern matches a space followed by uppercase letters, followed by a space, and then captures all characters until a space into the field `url`. The `field` attribute is not defined, so the `rex` statement matches against the `_raw` field. The values extracted look like the following:
 - `/products/x/?q=10471480`
 - `/bar?q=923891`
 - `/products/index.html?q=9029891`
 - `/about/?q=9376559`
- `rex field=url "(?:P<url>.*)\?"`: Searching the field `url`, this pattern matches all characters until a question mark. If the pattern matches, the result replaces the contents of the field `url`. If the pattern doesn't match, `url` stays the same. The values of `url` become:
 - `/products/x/`
 - `/bar`
 - `/products/index.html`
 - `/about/`
- `rex field=url "(?:P<url>.*\/)"`: Once again, while searching the field `url`, this pattern matches all characters until and including the last slash. The values of `url` are then:
 - `/products/x/`
 - `/`
 - `/products/`
 - `/about/`

This should effectively reduce the number of possible URLs, and hopefully make our summary index more useful and efficient. It may be that you only want to capture up to three levels of depth. You could accomplish that with this `rex` statement:

```
rex field=url "(?:P<url>/(?:[^\/]*){,3})"
```

The possibilities are endless. Be sure to test as much data as you can when building your summary indexes.

Using a lookup with wildcards

Splunk lookups also support wildcards, which we can use in this case. One advantage is that we can define arbitrary fields for grouping, independent of the values of `url`.

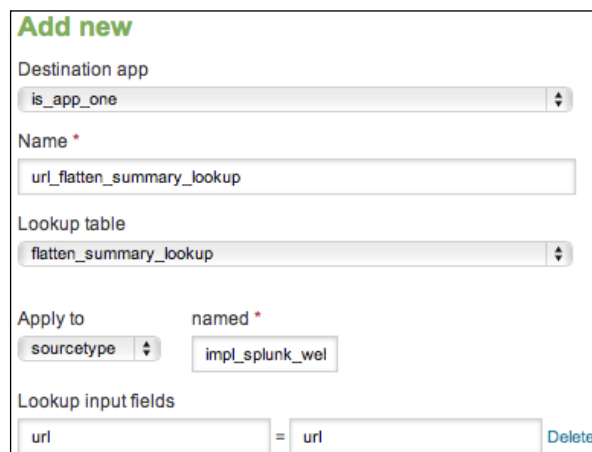
For a lookup wildcard to work, first we need to set up our `url` field and the lookup:

1. Extract the `url` field. The `rex` pattern we used before should work: `\s[A-Z]+\s(?:P<url>.*?)\s`. See *Chapter 3, Tables, Charts, and Fields*, for detailed instructions on setting up a field extraction. Don't forget to set permissions on the extraction.
2. Create our lookup file. Let's call the lookup file `flatten_summary_lookup.csv`. Use the following contents for our example log:

```
url,section
/about/*,about
/contact/*,contact
/*/*,unknown_non_root
/*,root
*,nomatch
```

 If you create your lookup file in Excel on a Mac, be sure to save the file using the format Windows Comma Separated (.csv).

3. Upload the lookup table file, create our lookup definition, and automatic lookup. See the *Using lookups to enrich data* section in *Chapter 6, Extending Search*, for detailed instructions. The automatic lookup definition should look like the following screenshot (the value of **Name** doesn't matter):



The screenshot shows the 'Add new' configuration window in Splunk. It includes the following fields and values:

- Destination app:** is_app_one
- Name:** url_flatten_summary_lookup
- Lookup table:** flatten_summary_lookup
- Apply to:** sourcetype (with a sub-label 'named *') and impl_splunk_wel
- Lookup input fields:** url = url (with a 'Delete' button next to the second 'url')

4. Set the permissions on all of the objects. I usually opt for **All Apps** for **Lookup table files** and **Lookup definitions**, and **This app only** for **Automatic lookups**. See *Chapter 6, Extending Search*, for details.
5. Edit `transforms.conf`. As of Splunk 4.3, not all features of lookups can be defined through the admin interface. To access these features, the configuration files that actually drive Splunk must be edited manually. We will cover configuration files in great detail in *Chapter 10, Configuring Splunk*, but for now, let's add two lines to one file and move on:
 1. Edit `$SPLUNK_HOME/etc/apps/is_app_one/local/transforms.conf`. The name of the directory `is_app_one` may be different depending on what app was active when you created your lookup definition. If you can't find this file, check your permissions and the **App** column in the admin interface.
 2. You should see these two lines, or something similar, depending on what you named your **Lookup table file** and **Lookup definition** instances:

```
[flatten_summary_lookup]
filename = flatten_summary_lookup.csv
```

If you do not see these lines in this file, check your permissions.

1. Add two more lines below `filename`:

```
match_type = WILDCARD(url)
max_matches = 1
```

These two lines effectively say:

- `match_type = WILDCARD(url)`: When evaluating the field `url`, honor wildcard characters. Without this setting, matches are always exact.
- `max_matches = 1`: Stop searching after the first match. By default, up to 10 matches are allowed. We want to match only the first line that matches, effectively using the lookup like a `case` statement.

If everything is wired up properly, we should now be able to run the search:

```
sourcetype=impl_splunk_web | stats count by section
```

This should give us the following simple report:

	section ↕	count ↕
1	about	2822
2	contact	2847
3	root	5741
4	unknown_non_root	14076

To see in greater detail what is really going on, let's try the following search:

```
sourcetype=impl_splunk_web
| rex field=url "(?P<url>.*)\?"
| stats count by section url
```

The rex statement is included to remove the query string from the value of url created by our extracted field. This gives us the following report:

	section ↕	url ↕	count ↕
1	about	/about/	2822
2	contact	/contact/	2847
3	root	/bar	2847
4	root	/foo	2894
5	unknown_non_root	/products/	5653
6	unknown_non_root	/products/x/	5637
7	unknown_non_root	/products/y/	2786

Looking back at our lookup file, our matches appear to be as follows:

url	pattern	section
/about/	/about/*	about
/contact/	/contact/*	contact
/bar	/*	root
/foo	/*	root
/products/	/*/*	unknown_non_root
/products/x/	/*/*	unknown_non_root
/products/y/	/*/*	unknown_non_root

If you read the lookup file from top to bottom, the first pattern that matches wins.

Using event types to group results

Another approach for grouping results to reduce summary index size would be to use event types in creative ways. For a refresher on event types, see *Chapter 6, Extending Search*.

This approach has the following advantages:

- All definitions are defined through the web interface
- It is possible to create arbitrarily complex definitions
- You can easily search for only those events that have defined section names
- You can place events in multiple groups if desired

The disadvantages to this approach are as follows:

- This is a non-obvious approach.
- It is not simple to not place events in multiple groups if more than one event type matches. For instance, if you want a page to match `/product/x/*` but not `/product/*`, this is not convenient to do.

The following is the procedure to create these event types:

1. For each section, create an event type, as follows:

Add new

Destination App

Name *

Search string *

Tag(s)

Enter a comma-separated list of tags.

2. Set the permissions to either **This app only** or **Global**, depending on the scope.
3. Repeat this for each section you want to summarize. The **Clone** link in **Manager** makes this process much faster.

With our event types in place, we can now build queries. The **Tag** value that we included means we can search easily for only those events that match a section, like the following:

```
tag::eventtype="summary_url" | top eventtype
```

The previous code returns a table as shown in the following screenshot:

	eventtype ↕	count ↕	percent ↕
1	bogus	19745	100.000000
2	url_products	14076	71.288934
3	url_contact	2847	14.418840
4	url_about	2822	14.292226

Our results contain the new event types that we created, along with an unwanted event type, **bogus**. Remember that all event type definitions that match an event are attached. This is very powerful, but sometimes is not what you expect. The **bogus** event type definition is `*`, which means it matches everything. The **bogus** event type was added purely to illustrate the point and has no practical use.

Let's create a new field from our summary event type name, then aggregate based on the new field:

```
tag::eventtype="summary_url"
| rex field=eventtype "url_(?P<section>.*)"
| stats count by section
```

The previous code gives us the results we are looking for, as shown in the following screenshot:

	section ↕	count ↕
1	about	2822
2	contact	2847
3	products	14076

This search finds only events that have defined event types, which may be what you want. To group all other results into an "other" group, we instead need to search for all events in the following manner:

```
sourcetype=impl_splunk_web
| rex field=eventtype "url_(?P<section>.*)"
| fillnull value="other" section
| stats count by section
```

The previous code then produces the following report:

	section ↕	count ↕
1	about	2822
2	contact	2847
3	other	5741
4	products	14076

Hopefully these examples will be food for thought when it comes to collapsing your results into more efficient summary results.

Calculating top for a large time frame

One common problem is to find the top contributors out of some huge set of unique values. For instance, if you want to know what IP addresses are using the most bandwidth in a given day or week, you may have to keep track of the total of request sizes across millions of unique hosts to definitively answer this question. When using summary indexes, this means storing millions of events in the summary index, quickly defeating the point of summary indexes.

Just to illustrate, let's look at a simple set of data:

Time	1.1.1.1	2.2.2.2	3.3.3.3	4.4.4.4	5.5.5.5	6.6.6.6
12:00	99	100	100	100		
13:00	99		100	100	100	
14:00	99	100		101	100	
15:00	99		99	100	100	
16:00	99	100			100	100
total	495	300	299	401	400	100

If we only stored the top three IPs per hour, our data set would look like the following:

Time	1.1.1.1	2.2.2.2	3.3.3.3	4.4.4.4	5.5.5.5	6.6.6.6
12:00		100	100	100		
13:00			100	100	100	
14:00		100		101	100	
15:00			99	100	100	
16:00		100			100	100
total		300	299	401	400	100

According to this data set, our top three IP addresses are 4.4.4.4, 5.5.5.5, and 2.2.2.2. The actual largest value was for 1.1.1.1, but it was missed because it was never in the top three.

To tackle this problem, we need to keep track of more data points for each slice of time. But how many?

Using our generator data, let's count a random number and see what kind of results we see. In my data set, it is the following query:

```
source="impl_splunk_gen" | top req_time
```

When run over a week, this query gives me the following results:

	req_time ↕	count ↕	percent ↕
1	10	402	0.072102
2	34	383	0.068694
3	15	377	0.067618
4	118	374	0.067080
5	26	373	0.066901
6	21	370	0.066362
7	18	366	0.065645
8	46	365	0.065466
9	140	365	0.065466
10	291	363	0.065107

How many unique values were there? The following query will tell us that:

```
source="impl_splunk_gen" | stats dc(req_time)
```

This tells us there are 12,239 unique values of req_time. How many different values are there per hour? The following query will calculate the average unique values per hour:

```
source="impl_splunk_gen"  
| bucket span=1h _time  
| stats dc(req_time) as dc by _time  
| stats avg(dc)
```

This tells us that each hour there are an average of 3,367 unique values of `req_time`. So, if we stored every count of every `req_time` for a week, we will store $3,367 * 24 * 7 = 565,656$ values. How many values would we have to store per hour to get the same answer we received before?

The following is a query that attempts to answer that question:

```
source="impl_splunk_gen"
| bucket span=1h _time
| stats count by _time req_time
| sort 0 _time -count
| streamstats count as place by _time
| where place<50
| stats sum(count) as count by req_time
| sort 0 -count
| head 10
```

Breaking this query down we have:

- `source="impl_splunk_gen"`: This finds the events.
- `| bucket span=1h _time`: This floors our `_time` field to the beginning of the hour. We will use this to simulate hourly summary queries.
- `| stats count by _time req_time`: This generates a count per `req_time` per hour.
- `| sort 0 _time -count`: This sorts and keeps all events (that's what 0 means), first ascending by `_time` then descending by `count`.
- `| streamstats count as place by _time`: This loops over the events, incrementing `place`, and starting the count over when `_time` changes. Remember that we flattened `_time` to the beginning of each hour.
- `| where place<50`: This keeps the first 50 events per hour. These will be the largest 50 values of `count` per hour, since we sorted descending by `count`.
- `| stats sum(count) as count by req_time`: This adds up what we have left across all hours.
- `| sort 0 -count`: This sorts the events in descending order by `count`.
- `| head 10`: This shows the first 10 results.

How did we do? Keeping the top 50 results per hour, my results look as shown in the following screenshot:

	req_time ↕	count ▼
1	10	139
2	257	125
3	101	109
4	103	109
5	140	107
6	46	107
7	15	98
8	21	98
9	24	97
10	211	96

That really isn't close. Let's try this again. We'll try where `place<1000`. This gives us the following results:

	req_time ↕	count ▼
1	10	401
2	34	367
3	15	361
4	26	356
5	101	354
6	118	351
7	18	350
8	21	349
9	46	345
10	291	344

That is much closer, but we're still not quite there. After experimenting a little more, `place<2000` was enough to get the expected top 10. This is better than storing 3,367 rows per hour. This may not seem like a big enough difference to bother, but increase the number of events by 10 or 100, and it can make a huge difference.

To use these results in a summary index, you would simply eliminate results going into your data set. One way to accomplish this might be:

```
source="impl_splunk_gen"
| sitop req_time
| streamstats count as place
| where place<2001
```

The first row produced by `sitop` contains the total value.

Another approach, using a combination of `eventstats` and `sistats`, is as follows:

```
source="impl_splunk_gen"
| eventstats count by req_time
| sort 0 -req_time
| streamstats count as place
| where place<2001
| sistats count by req_time
```

Luckily, this is not a terribly common problem, so most of this complexity can be avoided. For another option, see the *Storing a running calculation* section.

Storing raw events in a summary index

Sometimes it is desirable to copy events to another index. I have seen a couple of reasons for doing this, namely:

- **Differing retention:** If some special events need to be kept indefinitely, but the index where they are initially captured rolls off after some period of time, they can be captured into a summary index
- **Enrichment:** Sometimes the enrichment of data is too expensive to happen with every query, or it is important to capture events with the values from a lookup as the values existed at a particular point in time

The process is essentially the same as creating any summary index events. Follow these steps:

1. Create a populating query.
2. Add interesting fields using the `fields` command.
3. Add a `search_name` field to the search definition.
4. Include `_time`, but rename `_raw` to `raw`.

Let's capture all errors that mary sees, enriched with some extra data. First, create the query:

```
sourcetype=impl_splunk_gen mary error  
| eval raw=_raw  
| table _time raw department city
```

Save the query and edit the summary info:

summary - mary errors

Search *

Description

Time range

Start time Finish time

Time specifiers: y, mon, d, h, m, s
[Learn more](#)

Schedule and alert

Schedule this search

Schedule type

Run every

Summary indexing

Enable

Select the summary index

Only indexes that you can write to are listed.

Add fields

<input type="text" value="search_name"/>	=	<input type="text" value="summary - mary errors"/>	Delete
<input type="text"/>	=	<input type="text"/>	Delete

[Add another field](#)

You can then search against the summary index using the `search_name` value you provided:

```
index=summary_impl_splunk search_name="summary - mary errors"
```

The events in the summary index look almost identical to the original event, with the addition of the fields specified:

```
09/03/2012 203:11:59 -0600, search_name="summary - mary errors",
search_now=1346641919.000, info_min_time=1346641919.000, info_max_
time=1346641919.000, info_search_time=1346641919.588, city=Dallas,
department=HR, raw="2012-09-03T03:11:59.107+0000 DEBUG error, ERROR,
Error! [logger=LogoutClass, user=mary, ip=3.2.4.5, req_time=1414]"
```

With the addition of the `table` command, we can see the extra fields that were added using the `fields` command:

```
index=summary_impl_splunk search_name="summary - mary errors"
| table _time department city search_name
```

The previous search renders the following table:

	<code>_time</code> ↕	<code>department</code> ↕	<code>city</code> ↕	<code>search_name</code> ↕
1	9/2/12 10:08:54.946 PM	HR	Dallas	summary - mary errors
2	9/2/12 10:08:50.093 PM	HR	Dallas	summary - mary errors
3	9/2/12 10:08:47.942 PM			summary - mary errors
4	9/2/12 10:08:46.304 PM	HR	Dallas	summary - mary errors
5	9/2/12 10:08:41.989 PM	HR	Dallas	summary - mary errors
6	9/2/12 10:08:35.099 PM	HR	Dallas	summary - mary errors
7	9/2/12 10:08:25.719 PM	HR	Dallas	summary - mary errors
8	9/2/12 10:08:24.545 PM			summary - mary errors
9	9/2/12 10:08:24.211 PM	HR	Dallas	summary - mary errors
10	9/2/12 10:08:22.395 PM	HR	Dallas	summary - mary errors

This process is fairly complicated, so luckily adding events to a summary index is not commonly needed.

Using CSV files to store transient data

Sometimes it is useful to store small amounts of data outside of a Splunk index. Using the `inputcsv` and `outputcsv` commands, we can store tabular data in CSV files on the filesystem.

Pre-populating a dropdown

If a dashboard contains a dynamic dropdown, you must use a search to populate the dropdown. As the amount of data increases, the query to populate the dropdown will run more and more slowly, even from a summary index. We can use a CSV file to store just the information needed, simply adding new values when they occur.

First, we build a query to generate the CSV file. This query should be run over as much data as possible:

```
source="impl_splunk_gen"
| stats count by user
| outputcsv user_list.csv
```

Next, we need a query to run periodically that will append any new entries to the file. Schedule this query to run periodically as a saved search:

```
source="impl_splunk_gen"
| stats count by user
| append [inputcsv user_list.csv]
| stats sum(count) as count by user
| outputcsv user_list.csv
```

To then use this in our dashboard, our populating query will simply be:

```
|inputcsv user_list.csv
```

Simple dashboard XML using this query would look like the following:

```
<input type="dropdown" token="sourcetype">
  <label>User</label>
  <populatingSearch fieldForValue="user" fieldForLabel="user">
    |inputcsv user_list.csv
  </populatingSearch>
</input>
```

Creating a running calculation for a day

If the number of events per day is in the millions or tens of millions, querying all events for that day can be extremely expensive. For that reason, it makes sense to do part of the work on smaller periods of time.

Using a summary index to store these interim values can sometimes be overkill if those values are not needed for long. In the *Calculating top for a large time frame* section, we ended up storing thousands of values every few minutes. If we simply wanted to know the top 10 per day, this might be seen as a waste. To cut down on the noise in our summary index, we can use a CSV as cheap interim storage.

The steps are essentially to:

1. Periodically query recent data and update the CSV.
2. Capture top values in summary at the end of the day.
3. Empty the CSV file.

Our periodic query looks like the following:

```
source="impl_splunk_gen"
| stats count by req_time
| append [inputcsv top_req_time.csv]
| stats sum(count) as count by req_time
| sort 10000 -count
| outputcsv top_req_time.csv
```

Let's break the query down line by line:

- `source="impl_splunk_gen"`: This is the query to find the events for this slice of time.
- `| stats count by req_time`: This helps calculate the count by `req_time`.
- `| append [inputcsv top_req_time.csv]`: This loads the results generated so far from the CSV file, and adds the events to the end of our current results.
- `| stats sum(count) as count by req_time`: This uses `stats` to combine the results from our current time slice and the previous results.
- `| sort 10000 -count`: This sorts the results descending by `count`. The second word, `10000`, specifies that we want to keep the first 10,000 results.
- `| outputcsv top_req_time.csv`: This overwrites the CSV file.

Schedule the query to run periodically, perhaps every 15 minutes. Follow the same rules about latency as discussed in the *How latency affects summary queries* section.

When the rollup is expected, perhaps each night at midnight, schedule two more queries a few minutes apart, as follows:

- `| inputcsv top_req_time.csv | head 100`: Save this as a query adding to a summary index, as in the *Populating summary indexes with saved searches* section
- `| stats count | outputcsv top_req_time.csv`: This query will simply overwrite the CSV file with a single line

Summary

In this chapter, we have explored the use of summary indexes and the commands surrounding them. While summary indexes are not always the answer, they can be very useful for particular problems. We also explored alternative approaches using CSV files for interim storage.

Summary indexes have long been a hotbed of development at Splunk, and I know there has been major work done for Splunk 5, increasing the speed of some summary queries dramatically.

In our next chapter we will dive into the configuration files that drive Splunk.

10

Configuring Splunk

Everything that controls Splunk lives in configuration files sitting in the filesystem of each instance of Splunk. These files are unencrypted, easily readable, and easily editable. Almost all of the work that we have done so far has been accomplished through the web interface, but everything actually ends up in these configuration files.

While the web interface does a lot, there are many options that are not represented in the admin interface. There are also some things that are simply easier to accomplish by editing the files directly.

In this chapter, we will cover:

- Locating configuration files
- Merging configurations
- Debugging configurations
- Common configurations and their parameters

Locating Splunk configuration files

Splunk's configuration files live in `$SPLUNK_HOME/etc`. This is reminiscent of Unix's `/etc` directory but is instead contained within Splunk's directory structure. This has the advantage that the files don't have to be owned by `root`. In fact, the entire Splunk installation can run as an unprivileged user (assuming you don't need to open a port below 1024 or read files only readable by another user).

The directories that contain configurations are:

- `$SPLUNK_HOME/etc/system/default`: The default configuration files that ship with Splunk. Never edit these files as they will be overwritten each time you upgrade.

- `$(SPLUNK_HOME)/etc/system/local`: This is the location of global configuration overrides specific to this host. There are very few configurations that need to live here – most configurations that do live here are created by Splunk itself. In almost all cases, you should make your configuration files inside of an app.
- `$(SPLUNK_HOME)/etc/apps/$app_name/default`: This is the proper location for configurations in an app that will be shared either through Splunkbase or otherwise.
- `$(SPLUNK_HOME)/etc/apps/$app_name/local`: This is where most configurations should live and where all non-private configurations created through the web interface will be placed.
- `$(SPLUNK_HOME)/etc/users/$user_name/$app_name/local`: When a search configuration is created through the web interface, it will have a permission setting of **Private** and will be created in a user-/app-specific configuration file. Once permissions are changed, the configuration will move to the corresponding directory named `$app_name/local`.

There are a few more directories that contain files that are not `.conf` files. We'll talk about those later in this chapter, under the *User interface resources* section.

The structure of a Splunk configuration file

The `.conf` files used by Splunk look very similar to `.ini` files. A simple configuration looks like this:

```
#settings for foo
[foo]
bar=1
la = 2
```

Let's look at the following couple of definitions:

- **stanza**: A stanza is used to group attributes. Our stanza in this example is `[foo]`. A common synonym for this is **section**. Keep in mind the following key points:
 - A stanza name must be unique in a single file
 - Order does not matter

- **attribute:** An attribute is a name-value pair. Our attributes in this example are `bar` and `1a`. A common synonym is **parameter**. Keep in mind the following key points:
 - The attribute name must not contain whitespace or the equals sign
 - Each attribute belongs to the stanza defined above; if the attribute appears above all stanzas, the attribute belongs to the stanza `[default]`
 - The attribute name must be unique in a single stanza but not in a configuration
 - Each attribute must have its own line and can only use one line
 - Spaces around the equal sign do not matter

These are a few rules that may not apply in other implementations:

- Stanza and property names are *case sensitive*
- The comment character is `#`
- Bare attributes at the top of a file are added to the `[default]` stanza
- Any attributes in the stanza `[default]` are added to all stanzas that do not have an attribute with that name already

Configuration merging logic

Configurations in different locations merge behind the scenes into one "super" configuration. Luckily, the merging happens in a predictable way and is fairly easy to learn, and there is a tool to help us preview this merging.


Merging order

Merging order is slightly different depending on whether the configuration is being used by the search engine or another part of Splunk. The difference is whether there is an active user and app.


Merging order outside of search

Configurations being used outside of search are merged in a fairly simple order. These configurations include what files to read, what indexed fields to create, what indexes exist, and deployment server and client configurations as well as other settings. These configurations merge in this order:

1. `$SPLUNK_HOME/etc/system/default`: This directory contains the base configurations that ship with Splunk.

 Never make changes in `$(SPLUNK_HOME)/etc/system/default` as your changes will be lost when you upgrade Splunk.

2. `$(SPLUNK_HOME)/etc/apps/*/default`: Configurations are "overlaid" in reverse ASCII order by app directory name. a beats z.
3. `$(SPLUNK_HOME)/etc/apps/*/local`
4. `$(SPLUNK_HOME)/etc/system/local`
 - The configurations in this directory are applied last.
 - Outside of search, these configurations cannot be overridden by an app configuration. Apps are a very convenient way to compartmentalize control and distribute configurations. This is particularly relevant if you use the deployment server, which we will cover in *Chapter 11, Advanced Deployments*.

 Do *not* edit configurations in `$(SPLUNK_HOME)/etc/system/local` unless you have a very specific reason. An app is almost always the correct place for configuration.

A little pseudo code to describe this process might look like this:

```
$conf = new Configuration('$(SPLUNK_HOME)/etc/')

$conf.merge( 'system/default/$conf_name' )

for $this_app in reverse(sort(@all_apps)):
    $conf.merge( 'apps/$this_app/default/$conf_name' )

for $this_app in reverse(sort(@all_apps)):
    $conf.merge( 'apps/$this_app/local/$conf_name' )

$conf.merge( 'system/local/$conf_name' )
```

Merging order when searching

When searching, configuration merging is slightly more complicated. When running a search, there is always an active user and app, and they come into play. The logical order looks like this:

1. `$(SPLUNK_HOME)/etc/system/default`
2. `$(SPLUNK_HOME)/etc/system/local`

3. `$(SPLUNK_HOME)/etc/apps/not app`
 - Each app, other than the current app, is looped through in ASCII order of the directory name (not the visible app name). Unlike merging outside of search, *z* beats *a*.
 - All configuration attributes that are shared globally are applied, first from `default` and then from `local`.
4. `$(SPLUNK_HOME)/etc/apps/app`
 - All configurations from `default` and then `local` are merged.
5. `$(SPLUNK_HOME)/etc/users/user/app/local`

Maybe a little pseudo code would be clearer:

```

$conf = new Configuration('$(SPLUNK_HOME)/etc/')

$conf.merge( 'system/default/$conf_name' )
$conf.merge( 'system/local/$conf_name' )

for $this_app in sort(@all_apps):
  if $this_app != $current_app:
    $conf.merge_shared( 'apps/$this_app/default/$conf_name' )
    $conf.merge_shared( 'apps/$this_app/local/$conf_name' )

$conf.merge( 'apps/$current_app/default/$conf_name' )
$conf.merge( 'apps/$current_app/local/$conf_name' )

$conf.merge( 'users/$current_user/$current_app/local/$conf_name' )

```

Configuration merging logic

Now that we know what configurations will merge in what order, let's cover the logic for how they actually merge. The logic is fairly simple.

- The configuration name, stanza name, and attribute name must match exactly
- The last configuration added wins

The best way to understand configuration merging is through examples.

Configuration merging example 1

Say we have the base configuration `default/sample1.conf`:

```
[foo]
bar=10
la=20
```

And say we merge a second configuration, `local/sample1.conf`:

```
[foo]
bar=15
```

The resulting configuration would be:

```
[foo]
bar=15
la=20
```

The things to notice are as follows:

- The second configuration does not simply replace the prior configuration
- The value of `bar` is taken from the second configuration
- The lack of a `la` property in the second configuration does not remove the value from the final configuration

Configuration merging example 2

Say we have the base configuration `default/sample2.conf`:

```
[foo]
bar = 10
la=20

[pets]
cat = red
Dog=rex
```

And say we merge a second configuration, `local/sample2.conf`:

```
[pets]
cat=blue
dog=fido
fish = bubbles
```

```
[foo]
bar= 15

[cars]
ferrari =0
```

The resulting configuration would be:

```
[foo]
bar=15
la=20

[pets]
cat=blue
dog=rex
Dog=fido
fish=bubbles

[cars]
ferrari=0
```

Things to notice in this example:

- The order of the stanzas does not matter
- The spaces around the equal signs do not matter
- Dog does not override dog as all stanza names and property names are case sensitive
- The cars stanza is added fully

Configuration merging example 3

Let's do a little exercise, merging four configurations from different locations. In this case, we are not in search, so we will use the rules from the *Merging order outside of search* section. Let's step through a few sample configurations:

- For `$(SPLUNK_HOME)/etc/apps/d/default/props.conf` we have:

```
[web_access]
MAX_TIMESTAMP_LOOKAHEAD = 25
TIME_PREFIX = ^\[

[source::*.log]
BREAK_ONLY_BEFORE_DATE = true
```

- For `$SPLUNK_HOME/etc/system/local/props.conf` we have:

```
BREAK_ONLY_BEFORE_DATE = false
```

```
[web_access]
```

```
TZ = CST
```

- For `$SPLUNK_HOME/etc/apps/d/local/props.conf` we have:

```
[web_access]
```

```
TZ = UTC
```

```
[security_log]
```

```
EXTRACT-<name> = \[(?P<user>.*?)\]
```

- For `$SPLUNK_HOME/etc/apps/b/default/props.conf` we have:

```
[web_access]
```

```
MAX_TIMESTAMP_LOOKAHEAD = 20
```

```
TIME_FORMAT = %Y-%m-%d $H:%M:%S
```

```
[source:*/access.log]
```

```
BREAK_ONLY_BEFORE_DATE = false
```

I've thrown a bit of a curveball here by placing the files out of merging order. These configurations would actually merge in this order:

```
$SPLUNK_HOME/etc/apps/d/default/props.conf
$SPLUNK_HOME/etc/apps/b/default/props.conf
$SPLUNK_HOME/etc/apps/d/local/props.conf
$SPLUNK_HOME/etc/system/local/props.conf
```

Stepping through each merge, the configuration would look like this:

1. We start with `$SPLUNK_HOME/etc/apps/d/default/props.conf`:

```
[web_access]
```

```
MAX_TIMESTAMP_LOOKAHEAD = 25
```

```
TIME_PREFIX = ^\[
```

```
[source:*.log]
```

```
BREAK_ONLY_BEFORE_DATE = true
```

2. We then merge `$SPLUNK_HOME/etc/apps/b/default/props.conf`:

```
[web_access]
```

```
MAX_TIMESTAMP_LOOKAHEAD = 30
```

```
TIME_PREFIX = ^\[
```

```
TIME_FORMAT = %Y-%m-%d $H:%M:%S
```

```
[source:*.log]
BREAK_ONLY_BEFORE_DATE = true
```

```
[source:*/access.log]
BREAK_ONLY_BEFORE_DATE = false
```

Even though `[source:*.log]` and `[source:*/access.log]` both match a file called `access.log`, they will not merge in the configuration because the stanza names do not match exactly. This logic is covered later under **An overview of Splunk .conf files | props.conf | Stanza types**.

3. We then merge `$SPLUNK_HOME/etc/apps/d/local/props.conf`:

```
[web_access]
MAX_TIMESTAMP_LOOKAHEAD = 30
TIME_PREFIX = ^\[
TIME_FORMAT = %Y-%m-%d $H:%M:%S
TZ = UTC
```

```
[source:*.log]
BREAK_ONLY_BEFORE_DATE = true
```

```
[source:*/access.log]
BREAK_ONLY_BEFORE_DATE = false
```

```
[security_log]
EXTRACT-<name> = \[(?P<user>.*?)\]
```

4. We finally merge the globally overriding `$SPLUNK_HOME/etc/system/local/props.conf` file:

```
[default]
BREAK_ONLY_BEFORE_DATE = false
```

```
[web_access]
MAX_TIMESTAMP_LOOKAHEAD = 25
TIME_PREFIX = ^\[
TIME_FORMAT = %Y-%m-%d $H:%M:%S
TZ = CST
BREAK_ONLY_BEFORE_DATE = false
```

```
[source:*.log]
BREAK_ONLY_BEFORE_DATE = true
```

```
[source:*/access.log]
```



```
BREAK_ONLY_BEFORE_DATE = false

[security_log]
EXTRACT-<name> = \[(?P<user>.*?)\]
BREAK_ONLY_BEFORE_DATE = false
```

The setting with the biggest impact here is the bare attribute `BREAK_ONLY_BEFORE_DATE = false`. It is first added to the `[default]` stanza and then is added to *all* stanzas that do not already have any value.



As a general rule, avoid using the `[default]` stanza or bare word attributes. The final impact may not be what you expect.

Configuration merging example 4 (search)

In this case, we *are* in search, so we will use the more complicated merging order. Assuming that we are currently working in the app `d`, let's merge the same configurations again. For simplicity, we are assuming that all attributes are shared globally. We will merge the same configurations listed previously in example 3.

With `d` as our current app, we will now merge in this order:

```
$SPLUNK_HOME/etc/system/local/props.conf
$SPLUNK_HOME/etc/apps/b/default/props.conf
$SPLUNK_HOME/etc/apps/d/default/props.conf
$SPLUNK_HOME/etc/apps/d/local/props.conf
```

Stepping through each merge, the configuration will look like this:

1. We start with `$SPLUNK_HOME/etc/system/local/props.conf`:

```
BREAK_ONLY_BEFORE_DATE = false

[web_access]
TZ = CST
```

2. Now, we merge the default for apps other than our current app (which, in this case, is only one configuration) `$SPLUNK_HOME/etc/apps/b/default/props.conf`:

```
BREAK_ONLY_BEFORE_DATE = false

[web_access]
MAX_TIMESTAMP_LOOKAHEAD = 20
TIME_FORMAT = %Y-%m-%d %H:%M:%S
TZ = CST
```

```
[source:*/access.log]
BREAK_ONLY_BEFORE_DATE = false
```

- Next, we merge our current app default `$SPLUNK_HOME/etc/apps/d/default/props.conf`:

```
BREAK_ONLY_BEFORE_DATE = false
```

```
[web_access]
MAX_TIMESTAMP_LOOKAHEAD = 25
TIME_PREFIX = ^\[
TIME_FORMAT = %Y-%m-%d $H:%M:%S
TZ = CST
```

```
[source:*/access.log]
BREAK_ONLY_BEFORE_DATE = false
```

```
[source:*.log]
BREAK_ONLY_BEFORE_DATE = true
```

- Now we merge our current app local `$SPLUNK_HOME/etc/apps/d/local/props.conf`:

```
BREAK_ONLY_BEFORE_DATE = false
```

```
[web_access]
MAX_TIMESTAMP_LOOKAHEAD = 25
TIME_PREFIX = ^\[
TIME_FORMAT = %Y-%m-%d $H:%M:%S
TZ = UTC
```

```
[source:*/access.log]
BREAK_ONLY_BEFORE_DATE = false
```

```
[source:*.log]
BREAK_ONLY_BEFORE_DATE = true
```

```
[security_log]
EXTRACT-<name> = \[(?P<user>.*?)\]
```

- And finally, we apply our default stanza to stanzas that don't already have the attribute:

```
BREAK_ONLY_BEFORE_DATE = false
```

```
[web_access]
MAX_TIMESTAMP_LOOKAHEAD = 25
TIME_PREFIX = ^\[
TIME_FORMAT = %Y-%m-%d %H:%M:%S
TZ = UTC
BREAK_ONLY_BEFORE_DATE = false

[source:*/access.log]
BREAK_ONLY_BEFORE_DATE = false

[source:*.log]
BREAK_ONLY_BEFORE_DATE = true

[security_log]
EXTRACT-<name> = \[(?P<user>.*?)\]
BREAK_ONLY_BEFORE_DATE = false
```

I know this is fairly confusing, but with practice, it will make sense. Luckily, `btool`, which we will cover next, makes it easier to see.

Using `btool`

To help preview merged configurations, we call on `btool`, a command-line tool that prints the merged version of configurations. The Splunk site has one of my favorite documentation notes of all time, as follows:

Note: `btool` is not tested by Splunk and is not officially supported or guaranteed. That said, it's what our Support team uses when trying to troubleshoot your issues.

With that warning in mind, `btool` has never steered me wrong. The tool has a number of functions, but the only one I have ever used is `list`, like so:

```
$SPLUNK_HOME/bin/splunk cmd btool props list
```

This produces 5,277 lines of output, which I won't list here. Let's list the stanza `impl_splunk_gen` by adding it to the end of the command line, thus:

```
/opt/splunk/bin/splunk cmd btool props list impl_splunk_gen
```

This will produce an output such as this:

```
[impl_splunk_gen]
ANNOTATE_PUNCT = True
BREAK_ONLY_BEFORE =
BREAK_ONLY_BEFORE_DATE = True
```

```

... truncated ...
LINE_BREAKER_LOOKBEHIND = 100
LOOKUP-lookupusers = userslookup user AS user OUTPUTNEW
MAX_DAYS_AGO = 2000
... truncated ...
TRUNCATE = 10000
TZ = UTC
maxDist = 100

```

Our configuration file at `$(SPLUNK_HOME)/etc/apps/ImplementingSplunkDataGenerator/local/props.conf` contains only the following lines:

```

[impl_splunk_web]
LOOKUP-web_section = flatten_summary_lookup url AS url OUTPUTNEW
EXTRACT-url = \s[A-Z]+\s(?P<url_from_app_local>.*?)\s
EXTRACT-foo = \s[A-Z]+\s(?P<url_from_app>.*?)\s

```

So where did the rest of this configuration come from? With the use of the `--debug` flag, we can get more details.

```

/opt/splunk/bin/splunk cmd btool props list impl_splunk_gen --debug

```

This produces the following query:

```

Implementi [impl_splunk_gen]
system      ANNOTATE_PUNCT = True
system      BREAK_ONLY_BEFORE =
system      BREAK_ONLY_BEFORE_DATE = True
... truncated ...
system      LINE_BREAKER_LOOKBEHIND = 100
Implementi LOOKUP-lookupusers = userslookup user AS user OUTPUTNEW
system      MAX_DAYS_AGO = 2000
... truncated ...
system      TRUNCATE = 10000
Implementi  TZ = UTC
system      maxDist = 100

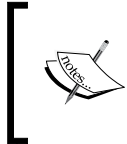
```

The first column, though truncated, tells us what we need to know. The vast majority of these lines are defined in `system`, most likely in `system/default/props.conf`. The remaining items from our file are labeled `Implementi`, which is the beginning of our app directory, `ImplementingSplunkDataGenerator`.

If you ever have a question about where some setting is coming from, `btool` will save you a lot of time. Also, check out the app *Splunk on Splunk* at Splunkbase for access to `btool` from the Splunk web interface.

An overview of Splunk .conf files

If you have spent any time in the filesystem investigating Splunk, you have seen many different files ending in `.conf`. In this section, we will give a quick overview of the most common `.conf` files. The official documentation is the best place to look for a complete reference of files and attributes.



The quickest way to find the official documentation is with your favorite search engine by searching for `splunk filename.conf`. For example, a search for `splunk props.conf` pulls up the Splunk documentation for `props.conf` first in every search engine I tested.

props.conf

The stanzas in `props.conf` define which events to match based on `host`, `source`, and `sourcetype`. These stanzas are merged into the master configuration based on the uniqueness of stanza and attribute names, as with any other configuration, but there are specific rules governing when each stanza is applied to an event and in what order. Stated as simply as possible, attributes are sorted by type, then by priority, and then by ASCII value.

We'll cover those rules under the *Stanza types* section. First, let's look at common attributes.

Common attributes

The full set of attributes allowed in `props.conf` is vast. Let's look at the most common attributes and try to break them down by the time when they are applied.

Search-time attributes

The most common attributes that users will make in `props.conf` are field extractions. When a user defines an extraction through the web interface, it ends up in `props.conf`, like so:

```
[my_source_type]
EXTRACT-foo = \s(?:<bar>\d+)ms
EXTRACT-cat = \s(?:<dog>\d+)s
```

This configuration defines the fields `bar` and `dog` for the source type `my_source_type`. Extracts are the most common search-time configurations. Any of the stanza types listed under the *Stanza types* section can be used, but source type is definitely the most common.

Other common search time attributes include:

- `REPORT-foo = bar`: This attribute is a way to reference stanzas in `transforms.conf` but apply them at search time instead of index time. This approach predates `EXTRACT` and is still useful for a few special cases. We will cover this case later under the *transforms.conf* section.
- `KV_MODE = auto`: This attribute allows you to specify whether Splunk should automatically extract fields in the form `key=value` from events. The default value is `auto`. The most common change is to disable automatic field extraction for performance reasons by setting the value to `none`. Other possibilities are `multi`, `json`, and `xml`.
- `LOOKUP-foo = mylookup barfield`: This attribute lets you wire up a lookup to automatically run for some set of events. The lookup itself is defined in `transforms.conf`.

Index-time attributes

As discussed in *Chapter 3, Indexed fields versus extracted fields*, it is possible to add fields to the metadata of events. This is accomplished by specifying a transform in `transforms.conf`, and an attribute in `props.conf`, to tie the transformation to specific events.

The attribute in `props.conf` looks like this: `TRANSFORMS-foo = bar1,bar2`. This attribute references stanzas in `transforms.conf` by name, in this case, `bar1` and `bar2`. These transform stanzas are then applied to the events matched by the stanza in `props.conf`.

Parse-time attributes

Most of the attributes in `props.conf` actually have to do with parsing events. To successfully parse events, a few questions need to be answered, such as these:

- When does a new event begin? Are events multiline? Splunk will make fairly intelligent guesses, but it is best to specify an exact setting. Attributes that help with this include:
 - `SHOULD_LINEMERGE = false`: If you know your events will never contain the newline character, setting this to `false` will eliminate a lot of processing.
 - `BREAK_ONLY_BEFORE = ^\d\d\d\d-\d\d-\d\d`: If you know that new events always start with a particular pattern, you can specify it using this attribute.

- `TRUNCATE = 1024`: If you are certain you only care about the first n characters of an event, you can instruct Splunk to truncate each line. What is considered a line can be changed with the next attribute.
- `LINE_BREAKER = ([\r\n]+)(?=\d{4}-\d\d-\d\d)`: The most efficient approach to multiline events is to redefine what Splunk considers a line. This example says that a line is broken on any number of newlines followed by a date of the form `1111-11-11`. The big disadvantage to this approach is that, if your log changes, you will end up with garbage in your index until you update your configuration. Try the *props helper* app available at Splunkbase for help making this kind of configuration.
- Where is the date? If there is no date, see `DATETIME_CONFIG` further down this bullet list. The relevant attributes are:
 - `TIME_PREFIX = ^\[:` By default, dates are assumed to fall at the beginning of the line. If this is not true, give Splunk some help and move the cursor past the characters preceding the date. This pattern is applied to each line, so if you have redefined `LINE_BREAKER` correctly, you can be sure only the beginnings of actual multiline events are being tested.
 - `MAX_TIMESTAMP_LOOKAHEAD = 30`: *Even if you change no other setting, you should change this one.* This setting says how far after `TIME_PREFIX` to test for dates. With no help, Splunk will take the first 150 characters of each line and then test regular expressions to find anything that looks like a date. The default regular expressions are pretty lax, so what it finds may look more like a date than the actual date. If you know your date is never more than n characters long, set this value to n or $n+2$. Remember that the characters retrieved come *after* `TIME_PREFIX`.
- What does the date look like? These attributes will be of assistance here:
 - `TIME_FORMAT = %Y-%m-%d %H:%M:%S.%3N %:z`: If this attribute is specified, Splunk will apply `strptime` to the characters immediately following `TIME_PREFIX`. If this matches, then you're done. This is by far the most efficient and least error-prone approach. Without this attribute, Splunk actually applies a series of regular expressions until it finds something that looks like a date.

- `DATETIME_CONFIG = /etc/apps/a/custom_datetime.xml`: As mentioned, Splunk uses a set of regular expressions to determine the date. If `TIME_FORMAT` is not specified, or won't work for some strange reason, you can specify a different set of regular expressions or disable time extraction completely by setting this attribute to `CURRENT` (the indexer clock time) or `NONE` (file modification time, or if there is no file, clock time). I personally have never had to resort to a custom `datetime.xml` file, though I have heard of it being done.
- The **Data preview** function available when adding data through the manager interface builds a good, usable configuration. The generated configuration does not use `LINE_BREAKER`, which is definitely safer but less efficient. Here is a sample stanza using `LINE_BREAKER` for efficiency:

```
[mysourcetype]
TIME_FORMAT = %Y-%m-%d %H:%M:%S.%3N %:z
MAX_TIMESTAMP_LOOKAHEAD = 32
TIME_PREFIX = ^\[
SHOULD_LINEMERGE = False
LINE_BREAKER = ([\r\n]+)(?=[\d{4}-\d{1,2}-\d{1,2}\s+\d{1,2}:\d{1,2}:\d{1,2})
TRUNCATE = 1024000
```

This configuration would apply to log messages that looked like this:

```
[2011-10-13 13:55:36.132 -07:00] ERROR Interesting message.
```

More information.

And another line.

```
[2011-10-13 13:55:36.138 -07:00] INFO All better.
```

```
[2011-10-13 13:55:37.010 -07:00] INFO More data
```

and another line.

Let's step through how these settings affect the first line of this sample configuration:

- `LINE_BREAKER` states that a new event starts when one or more newline characters is followed by a bracket and series of numbers and dashes, in the pattern `[1111-11-11 11:11:11]`.
- `SHOULD_LINEMERGE=False` tells Splunk to not bother trying to recombine multiple lines.
- `TIME_PREFIX` moves the cursor to the character after the `[` character.

- `TIME_FORMAT` is tested against the characters at the current cursor location. If it succeeds, we are done.
- If `TIME_FORMAT` fails, `MAX_TIMESTAMP_LOOKAHEAD` characters are read from the cursor position (after `TIME_PREFIX`) and the regular expressions from `DATE_CONFIG` are tested.
- If the regular expressions fail against the characters returned, the time last parsed from an event is used. If there is no last time parsed, the modification date from the file would be used, if known; otherwise, the current time would be used.

This is the most efficient and precise way to parse events in Splunk, but also the most brittle. If your date format changes, you will almost certainly have junk data in your index. Only use this approach if you are confident the format of your logs will not change without your knowledge.

Input time attributes

There are only a couple of attributes in `props.conf` that matter at the input stage, but they are generally not needed:

- `CHARSET = UTF-16LE`: When reading data, Splunk has to know the character set used in the log. Most applications write their logs using either `ISO-8859-1` or `UTF-8`, which the default settings handle just fine. Some Windows applications write logs in 2-byte Little Endian, which is indexed as garbage. Setting `CHARSET = UTF-16LE` takes care of the problem. Check the official documentation for a list of supported encodings.
- `NO_BINARY_CHECK = true`: If Splunk believes that a file is binary, it will not index the file at all. If you find that you have to change this setting to convince Splunk to read your files, it is likely that the file is in an unexpected character set. You might try other `CHARSET` settings before enabling this setting.

Stanza types

Now that we have looked at common attributes, let's talk about the different types of stanzas in `props.conf`. Stanza definitions can take the three following forms:

- `[foo]`
 - This is the exact name of a source type and is the most common type of stanza to be used; the source type of an event is usually defined in `inputs.conf`
 - Wildcards are not allowed

- `[source::/logs/.../*.log]`
 - This matches the `source` attribute, which is usually the path to the log where the event came from
 - `*` matches a file or directory name
 - `...` matches any part of a path
- `[host::*nyc*]`
 - This matches the `host` attribute, which is usually the value of `hostname` on a machine running Splunk Forwarder
 - `*` is allowed

Precedence across types follows this order:

1. Source.
2. Host.
3. Source type.

For instance, say an event has the following fields:

```
sourcetype=foo_type
source=/logs/abc/def/gh.log
host=dns4.nyc.mycompany.com
```

Given this configuration snippet and our preceding event:

```
[foo_type]
TZ = UTC

[source::/logs/.../*.log]
TZ = MST

[host::*nyc*]
TZ = EDT
```

`TZ = MST` would be used during parsing, because the source stanza takes precedence.

To extend this example, say we have this snippet:

```
[foo_type]
TZ = UTC
TRANSFORMS-a = from_sourcetype

[source::/logs/.../*.log]
```

```
TZ = MST
BREAK_ONLY_BEFORE_DATE = True
TRANSFORMS-b = from_source

[host::*nyc*]
TZ = EDT
BREAK_ONLY_BEFORE_DATE = False
TRANSFORMS-c = from_host
```

The attributes applied to our event would therefore be:

```
TZ = MST
BREAK_ONLY_BEFORE_DATE = True
TRANSFORMS-a = from_sourcetype
TRANSFORMS-b = from_source
TRANSFORMS-c = from_host
```

Priorities inside a type

If there are multiple `source` or `host` stanzas that match a given event, the order in which settings are applied also comes into play. A stanza with a pattern has a priority of 0, while an exact stanza has a priority of 100. Higher priorities win. For instance, say we have the following stanza:

```
[source::/logs/abc/def/gh.log]
TZ = UTC

[source::/logs/.../*.log]
TZ = CDT
```

Our `TZ` value will be `UTC` since the exact match of `source::/logs/abc/def/gh.log` has a higher priority.

When priorities are identical, stanzas are applied by ASCII order. For instance, say we have this configuration snippet:

```
[source::/logs/abc/.../*.log]
TZ = MST

[source::/logs/.../*.log]
TZ = CDT
```

The attribute `TZ=CDT` would win because `/logs/.../*.log` is first in ASCII order. This may seem counterintuitive since `/logs/abc/.../*.log` is arguably a better match. The logic for determining what makes a better match, however, can quickly become fantastically complex, so ASCII order is a reasonable approach.

You can also set your own value of `priority`, but luckily, it is rarely needed.

Attributes with class

As you dig into configurations, you will see attribute names of the form FOO-bar. The word after the dash is generally referred to as the class. These attributes are special in a few ways:

- Attributes merge across files like any other attribute
- Only one instance of each class will be applied, according to the rules described previously
- The final set of attributes is applied in ASCII order by the value of class

Once again, say we are presented with an event with the following fields:

```
sourcetype=foo_type
source=/logs/abc/def/gh.log
host=dns4.nyc.mycompany.com
```

And say that this is the configuration snippet:

```
[foo_type]
TRANSFORMS-a = from_sourcetype1, from_sourcetype2

[source::/logs/.../*.log]
TRANSFORMS-c = from_source_b

[source::/logs/abc/.../*.log]
TRANSFORMS-b = from_source_c

[host::*nyc*]
TRANSFORMS-c = from_host
```

The surviving transforms would then be:

```
TRANSFORMS-c = from_source_b
TRANSFORMS-b = from_source_c
TRANSFORMS-a = from_sourcetype1, from_sourcetype2
```

To determine the order in which the transforms are applied to our event, we sort the stanzas according to the values of their classes, in this case, c, b, and a. This gives us:

```
TRANSFORMS-a = from_sourcetype1, from_sourcetype2
TRANSFORMS-b = from_source_c
TRANSFORMS-c = from_source_b
```

The transforms are then combined into a single list and executed in this order:

```
from_sourcetype1, from_sourcetype2, from_source_c, from_source_b
```

The order of transforms usually doesn't matter but is important to understand if you want to chain transforms and create one field from another. We'll try this later, in the *transforms.conf* section.

inputs.conf

This configuration, as you might guess, controls how data makes it into Splunk. By the time this data leaves the input stage, it still isn't an event but has some base metadata associated with it: *host*, *source*, *sourcetype*, and optionally *index*. This base metadata is then used by the parsing stage to break the data into events according to the rules defined in *props.conf*:

Input types can be broken down into files, network ports, and scripts. First, we will look at attributes that are common to all inputs.

Common input attributes

These common bits of metadata are used in the parsing stage to pick the appropriate stanzas in *props.conf*.

- *host*: By default, *host* will be set to the hostname of the machine producing the event. This is usually the correct value, but it can be overridden when appropriate.
- *source*: This field is usually set to the path to the file or network port that an event came from, but this value can be hardcoded.
- *sourcetype*: This field is almost always set in *inputs.conf* and is the primary field for determining which set of parsing rules in *props.conf* to apply to these events.



It is very important to set *sourcetype*. In the absence of a value, Splunk will create automatic values based on *source*, which can easily result in an explosion of *sourcetype* values.

- *index*: This field says what index to write events to. If it is omitted, the default index will be used.

All of these values can be modified using transforms, the only caveat being that these transforms are applied *after* the parsing step. The practical consequence of this is that you cannot apply different parsing rules to different events in the same file, for instance, different time formats on different lines.

Files as inputs

The vast majority of events in Splunk come from files. Usually, these events are read from the machine where they are produced and as the logs are written. Very often, the entire input's stanza will look like this:

```
[monitor:///logs/interesting.log*]
sourcetype=interesting
```

This is often all that is needed. This stanza is saying:

- Read all logs that match the pattern `/logs/interesting.log*`, and going forward, watch them for new data
- Name the source type `interesting`
- Set the source to the name of the file in which the log entry was found
- Default the host to the machine where the logs originated
- Write the events to the default index

These are usually perfectly acceptable defaults. If `sourcetype` is omitted, Splunk will pick a default source type based on the filename, which you don't want – your source type list will get very messy very fast.

Using patterns to select rolled logs

You may notice that the previous stanza ended in `*`. This is important because it gives Splunk a chance to find events that were written to a log that has recently rolled. If we simply watch `/logs/interesting.log`, it is likely that events will be missed at the end of the log when it rolls, particularly on a busy server.

Will we end up with duplicate events after the log rolls to `interesting.log.1` or `interesting.log.2012-09-17`? The answer is "almost certainly not". This is because Splunk does not use filenames to determine what files have been read but instead does so by using checksums on the contents of the files. This means that logs can be renamed or even moved to a different filesystem on the same server, and they will still be recognized as the same file.

There are specific cases where Splunk can get confused, but in the vast majority of cases, the default mechanisms do exactly what you would hope. See the *When to use crcSalt* section further on for a discussion about special cases.

Using blacklist and whitelist

It is also possible to use a blacklist and whitelist pattern for more complicated patterns. The most common use case is to blacklist files that should not be indexed, for instance, gz and zip files. It can be done as follows:

```
[monitor:///opt/B/logs/access.log*]
sourcetype=access
blacklist=*.gz
```

This stanza would still match `access.log.2012-08-30`, but if we had a script that compressed older logs, Splunk would not try to read `access.log.2012-07-30.gz`.

Conversely, you can use a whitelist to apply very specific patterns, like so:

```
[monitor:///opt/applicationserver/logs]
sourcetype=application_logs
whitelist=(app|application|legacy|foo)\.log(\.\d{4})?
blacklist=*.gz
```

This whitelist would match `app.log`, `application.log`, `legacy.log.2012-08-13`, and `foo.log`, among others. The blacklist will negate any gz files.

Since `logs` is a directory, the default behavior will be to recursively scan that directory.


Selecting files recursively

The layout of your logs or your application may dictate a recursive approach. For instance, say we have these stanzas:

```
[monitor:///opt/*/logs/access.log*]
sourcetype=access

[monitor:///opt/.../important.log*]
sourcetype=important
```

The character `*` will match a single file or directory, while `...` will match any depth. This will match the files you want, with the caveat that all of `/opt` will continually be scanned.

 Splunk will continually scan all directories from the first wildcard in a monitor path!

If `/opt` contains many files and directories, which it almost certainly does, Splunk will use an unfortunate amount of resources scanning all directories for matching files, constantly using memory and CPU. I have seen a single Splunk process watching a large directory structure use 2 gigabytes of memory. A little creativity can take care of this, but it is something to be aware of.

The takeaway is that if you know the possible values for `*`, you are better off writing multiple stanzas. For instance, assuming our directories in `/opt` are `A` and `B`, the following stanzas will be far more efficient:

```
[monitor:///opt/A/logs/access.log*]
sourcetype=access

[monitor:///opt/B/logs/access.log*]
sourcetype=access
```

It is also perfectly acceptable to have stanzas matching files and directories that simply don't exist. This causes no errors, but be careful to not include patterns that are so broad that they match unintended files.

Following symbolic links

When scanning directories recursively, the default behavior is to follow symbolic links. Often this is very useful, but it can cause problems if a symbolic link points to a large or slow file system. To control this behavior, simply set:

```
followSymlink = false
```

It's probably a good idea to put this on all of your monitor stanzas until you know you need to follow a symbolic link.

Setting the value of host from source

The default behavior of using the hostname from the machine forwarding the logs is almost always what you want. If, however, you are reading logs for a number of hosts, you can extract the hostname from `source` using `host_regex` or `host_segment`. For instance, say we have the path:

```
/nfs/logs/webserver1/access.log
```


To set `host` to `webserver1`, you could use either:

```
[monitor:///nfs/logs/*/access.log*]
sourcetype=access
host_segment=3
```

Or:

```
[monitor:///nfs/logs/*/access.log*]
sourcetype=access
host_regex=(.*)/access\.log
```

`host_regex` could also be used to extract the value of `host` from the filename.

It is also possible to reset `host` using a transform, with the caveat that this will occur after parsing, which means any settings in `props.conf` that rely on matching `host` will already have been applied.

Ignoring old data at installation

It is often the case that when Splunk is installed, months or years of logs are sitting in a directory where logs are currently being written. Logs that are appended to infrequently may also have months or years of events that are no longer interesting and would be wasteful to index.

The best solution is to set up archive scripts to compress any logs older than a few days, but in a large environment, this may be difficult to do. Splunk has two settings that help ignore older data, but be forewarned: once these files have been ignored, there is no simple way to change your mind later. If, instead, you compress older logs and blacklist the compressed files as explained in the *Using blacklist and whitelist* section, you can simply decompress at a later stage, any files you would like to index. Let's look at a sample stanza:

```
[monitor:///opt/B/logs/access.log*]
sourcetype = access
ignoreOlderThan = 14d
```

In this case, `ignoreOlderThan` says to ignore, forever, all events in any files whose modification date is older than 14 days. If the file is updated in the future, any *new* events will be indexed.

The `followTail` attribute lets us ignore all events written so far, instead starting at the end of each file. Let's look at an example:

```
[monitor:///opt/B/logs/access.log*]
sourcetype = access
followTail = 1
```

Splunk will note the length of files matching the pattern, but `followTail` instructs Splunk to ignore everything currently in these files. Any new events written to the files will be indexed. Remember that there is no easy way to alter this if you change your mind later.

It is not currently possible to say "ignore all *events* older than X", but since most logs roll on a daily basis, this is not commonly a problem.

When to use `crcSalt`

To keep track of what files have been seen before, Splunk stores a checksum of the first 256 bytes of each file it sees. This is usually plenty, as most files start with a log message, which is *almost* guaranteed to be unique.

This breaks down when the first 256 bytes are not unique on the same server. I have seen two cases where this happens, as follows:

1. Logs that start with a common header containing product version information, for instance:

```
=====
== Great product version 1.2 brought to you by Great company ==
== Server kernel version 3.2.1 ==
```

2. A server writing many thousands of files with low time resolution, for instance:

```
12:13:12 Session created
12:13:12 Starting session
```

To deal with these cases, we can add the path to the log to the checksum, or "salt our `crc`". This is accomplished like so:

```
[monitor:///opt/B/logs/application.log*]
sourcetype = access
crcSalt = <SOURCE>
```

It says to include the full path to this log in the checksum.

This method will only work if your logs have a unique name. The easiest way to accomplish this is to include the current date in the name of the log when it is created. You may need to change the pattern for your log names so that the date is always included and the log is not renamed.



Do not use `crcSalt` if your logs change names!

If you enable `crcSalt` in an input where it was not already enabled, you will re-index all the data! You need to ensure that the old logs are moved aside or compressed and blacklisted before enabling this setting in an existing configuration.

Destructively indexing files

If you receive logfiles in `batch`, you can use the `batch` input to consume logs and then *delete* them. This should only be used against a copy of the logs.

See the following example:

```
[batch:///var/batch/logs/*/access.log*]
sourcetype=access
host_segment=4
move_policy = sinkhole
```

This stanza would index the files in the given directory and then delete the files. Be very sure this is what you want to do!

Network inputs

In addition to reading files, Splunk can listen to network ports. The stanzas take the following form:

```
[protocol://<remote host>:<local port>]
```

The remote host portion is rarely used, but the idea is that you can specify different input configurations for specific hosts. The usual stanzas look like this:

- `[tcp://1234]`: Specify that we will listen to port 1234 for TCP connections. Anything can connect to this port and send data in.
- `[tcp-ssl://importanthost:1234]`: Listen on TCP using SSL, and apply this stanza to the host `importanthost`. Splunk will generate self-signed certificates the first time it is launched.
- `[udp://514]`: This is generally used for receiving **syslog** events. While this does work, it is generally considered best practice to use a dedicated syslog receiver, such as `rsyslog` or `syslogng`. See *Chapter 11, Advanced Deployments*, for a discussion on this subject.
- `[splunktcp://9997]` or `[splunktcp-ssl://9997]`: In a distributed environment, your indexers will receive events on the specified port. It is a custom protocol used between Splunk instances. This stanza is created for you when you use the **Manager** page at **Manager | Forwarding and receiving | Receive data**.

For `tcp` and `udp` inputs, the following attributes apply:

- `source`: If not specified, `source` will default to `protocol:port`, for instance, `udp:514`.
- `sourcetype`: If not specified, `sourcetype` will also default to `protocol:port`, but this is generally not what you want. It is best to specify a source type and create a corresponding stanza in `props.conf`.
- `connection_host`: With network inputs, what value to capture for `host` is somewhat tricky. Your options essentially are:
 - `connection_host = dns`, which uses reverse DNS to determine the hostname from the incoming connection. When reverse DNS is configured properly, this is usually your best bet. This is the default.
 - `connection_host = ip`, which sets the `host` field to the IP address of the remote machine. This is your best choice when reverse DNS is unreliable.
 - `connection_host = none`, which uses the hostname of the Splunk instance receiving the data. This option can make sense when all traffic is going to an interim host.
 - `host = foo`, which sets the hostname statically.
 - It is also common to reset the value of `host` using a transform, for instance with `syslog` events. This happens after parsing, though, so is too late to change things such as time zone based on the host.
- `queueSize`: This value specifies how much memory Splunk is allowed to set aside for an input queue. A common use for a queue is to capture spikey data until the indexers can catch up.
- `persistentQueueSize`: This value specifies a persistent queue that can be used to capture data to disk if the in-memory queue fills up.

If you find yourself building a particularly complicated setup around network ports, I would encourage you to talk to Splunk support as there may be a better way to accomplish your goals.

Native Windows inputs

One nice thing about Windows is that system logs and many application logs go to the same place. Unfortunately, that place is not a file, so native hooks are required to access these events. Splunk makes those inputs available using stanzas of the form `[WinEventLog:LogName]`. For example, to index the `Security` log, the stanza simply looks like this:

```
[WinEventLog:Security]
```

There are a number of supported attributes, but the defaults are reasonable. The only attribute I have personally used is `current_only`, which is the equivalent of `followTail` for monitor stanzas. For instance, this stanza says to monitor the `Application` log, but to start reading from now:

```
[WinEventLog:Application]
current_only = 1
```

This is useful when there are many historical events on the server.

The other input available is **Windows Management Instrumentation (WMI)**. With WMI, you can:

- Monitor native performance metrics, like you would find in Windows Performance Monitor
- Monitor the Windows Event Log API
- Run custom queries against the database behind WMI
- Query remote machines



Though it is theoretically possible to monitor many Windows servers using WMI and a few Splunk forwarders, this is not advised. The configuration is complicated, does not scale well, introduces complicated security implications, and is not thoroughly tested. Also, reading Windows Event Logs via WMI produces different output than the native input, and most apps that expect Windows events will not function as expected.

The simplest way to generate the `inputs.conf` and `wmi.conf` configurations needed for Windows Event Logs and WMI is to install `Splunk for Windows` on a Windows host and then configure the desired inputs through the web interface. See the official Splunk documentation for more examples.

Scripts as inputs

Splunk will periodically execute processes and capture the output. For example, here is input from the `ImplementingSplunkDataGenerator` app:

```
[script://./bin/implSplunkGen.py 2]
interval=60
sourcetype=impl_splunk_gen_sourcetype2
source=impl_splunk_gen_src2
host=host2
index=implSplunk
```

Things to notice in this example are as follows:

- The present working directory is the root of the app that contains `inputs.conf`.
- Files that end with `.py` will be executed using the Python interpreter included with Splunk. This means the Splunk Python modules are available. To use a different Python module, specify the path to Python in the stanza.
- Any arguments specified in the stanza will be handed to the script as if executed at the command line.
- `interval` specifies how often this script should be run, in seconds.
 - If the script is still running, it will not be launched again.
 - Long-running scripts are fine. Since only one copy of a script will run at a time, the interval will instead indicate how often to check whether the script is still running.
 - This value can also be specified in `cron` format.

Any programming language can be used, as long as it can be executed at the command line. Splunk simply captures the standard output from whatever is executed.

Included with Splunk for Windows are scripts for querying WMI. One sample stanza looks like this:

```
[script://$SPLUNK_HOME\bin\scripts\splunk-wmi.path]
```

Things to note are:

- Windows paths require backslashes instead of slashes.
- `$SPLUNK_HOME` will expand properly.

transforms.conf

`transforms.conf` is where we specify transformations and lookups that can then be applied to any event. These transforms and lookups are referenced by name in `props.conf`.

For our examples in the later subsections, we will use this event:

```
2012-09-24T00:21:35.925+0000 DEBUG [MBX] Password reset called.
[old=1234, new=secret, req_time=5346]
```

We will use it with these metadata values:

```
sourcetype=myapp
source=/logs/myapp.session_foo-jA5MDkyMjEwMTIK.log
host=vlbmba.local
```

Creating indexed fields

One common task accomplished with `transforms.conf` is the creation of new indexed fields. Indexed fields are different from extracted fields in that they must be created at index time and can be searched for whether the value is in the raw text of the event or not. It is usually preferable to create extracted fields instead of indexed fields. See *Chapter 3, Indexed fields versus extracted fields*, for a deeper discussion about when indexed fields are beneficial.



Indexed fields are only applied to events that are indexed after the definition is created. There is no way to backfill a field without reindexing.

Creating a loglevel field

The format of a typical stanza in `transforms.conf` looks like this:

```
[myapp_loglevel]
REGEX = \s([A-Z])\s
FORMAT = loglevel::$1
WRITE_META = True
```

This will add to our events the field `loglevel=DEBUG`. This is a good idea if the values of `loglevel` are common words outside of this location, for instance `ERROR`.

Stepping through this stanza, we have:

- `[myapp_loglevel]`: The stanza can be any unique value, but it is in your best interest to make the name meaningful. This is the name referenced in `props.conf`.
- `REGEX = \s([A-Z]+)\s`: This is the pattern to test against each event that is handed to us. If this pattern does not match, this transform will not be applied.
- `FORMAT = loglevel::$1`: Create the field `loglevel`. Under the covers, all indexed fields are stored using a `::` delimiter, so we have to follow that form.
- `WRITE_META = True`: Without this attribute, the transform won't actually create an indexed field and store it with the event.

Creating a session field from source

Using our event, let's create another field, `session`, which appears to only be in the value of `source`.

```
[myapp_session]
SOURCE_KEY = MetaData:Source
REGEX = session_(.*?)\.log
FORMAT = session::$1
WRITE_META = True
```

Note the attribute `SOURCE_KEY`. The value of this field can be any existing metadata field or another indexed field that has already been created. See the *Attributes with class* subsection within the *props.conf* section for a discussion about transform execution order. We will discuss these fields in the *Modifying metadata fields* subsection.

Creating a "tag" field

It is also possible to create fields simply to tag events that would be difficult to search for otherwise. For example, if we wanted to find all events that were slow, we could search for:

```
sourcetype=myapp req_time>999
```

Without an indexed field, this query would require parsing every event that matches `sourcetype=myapp` over the time that we are interested in. The query would then discard all events whose `req_time` value was 999 or less.

If we know ahead of time that a value of `req_time>999` is bad, and we can come up with a regular expression to specify what bad is, we can tag these events for quicker retrieval. Say we have this `transforms.conf` stanza:

```
[myapp_slow]
REGEX = req_time=\d{4,}
FORMAT = slow_request::1
WRITE_META = True
```

This REGEX will match any event containing `req_time=` followed by four or more digits.

After adding `slow_request` to `fields.conf` (see the *fields.conf* section), we can search for `slow_request=1` and find all slow events very efficiently. This will not apply to events that were indexed before this transform existed. If the events that are slow are uncommon, this query will be *much* faster.

Creating host categorization fields

It is common, to have parts of a hostname mean something in particular. If this pattern is well known and predictable, it may be worthwhile to pull the value out into fields. Working from our fictitious host value, `v1bmba.local` (which happens to be my laptop), we might want to create fields for `owner` and `hosttype`. Our stanza might look like this:

```
[host_parts]
SOURCE_KEY = MetaData:Host
REGEX = (...)(...)\.
FORMAT = host_owner::$1 host_type::$2
WRITE_META = True
```

With our new fields, we can now easily categorize errors by whatever information is encoded into the hostname. Another approach would be to use a lookup, which has the advantage of being retroactive. This approach has the advantage of faster searches for the specific fields.

Modifying metadata fields

It is sometimes convenient to override the main metadata fields. We will look at one possible reason for overriding each base metadata value.



Remember that transforms are applied after parsing, so changing metadata fields via transforms cannot be used to affect which `props.conf` stanzas are applied for date parsing or line breaking. For instance, with syslog events that contain the hostname, you cannot change the time zone because the date has already been parsed before the transforms are applied.

The keys provided by Splunk include:

- `_raw` (this is the default value for `SOURCE_KEY`)
- `MetaData:Source`
- `MetaData:Sourcetype`
- `MetaData:Host`
- `_MetaData:Index`

Overriding host

If your hostnames are appearing differently from different sources, for instance, syslog versus Splunk Forwarders, you can use a transform to normalize these values. Given our hostname `v1bmba.local`, we may want to only keep the portion to the left of the first period. The stanza would look like this:

```
[normalize_host]
SOURCE_KEY = MetaData:Host
DEST_KEY = MetaData:Host
REGEX = (.*)\.
FORMAT = host::$1
```

This will replace our hostname with `v1bmba`. Note these two things:

- `WRITE_META` is not included because we are not adding to the metadata of this event; we are instead overwriting the value of a core metadata field
- `host::` must be included at the beginning of the format

Overriding source

Some applications will write a log for each session, conversation, or transaction. One problem this introduces is an explosion of `source` values. The values of `source` will end up in `$SPLUNK_HOME/var/lib/splunk/*/db/Sources.data`—one line per unique value of `source`. This file will eventually grow to a huge size, and Splunk will waste a lot of time updating it, causing unexplained pauses. A new setting in `indexes.conf` called `disableGlobalMetadata`, can also eliminate this problem.

To flatten this value, we could use a stanza like this:

```
[myapp_flatten_source]
SOURCE_KEY = MetaData:Source
DEST_KEY = MetaData:Source
REGEX = (*.session_)*.log
FORMAT = source::$1x.log
```

This would set the value of `source` to `/logs/myapp.session_x.log`, which would eliminate our growing source problem.

If the value of `session` is useful, the transform in the *Creating a session field from source* section could be run before this transform to capture the value. Likewise, a transform could capture the entire value of `source` and place it into a different metadata field.



A huge number of logfiles on a filesystem introduces a few problems, including running out of inodes and the memory used by the Splunk process tracking all of the files. As a general rule, a cleanup process should be designed to archive older logs.

Overriding sourcetype

It is not uncommon to change the `sourcetype` field of an event based on the contents of the event, particularly from `syslog`. In our fictitious example, we want a different source type for events that contain `[MBX]` after the log level so that we can apply different extracts to these events. The following examples will do this work:

```
[mbx_sourcetype]
DEST_KEY = MetaData:Sourcetype
REGEX = \d+\s[A-Z]+\s\([MBX\])
FORMAT = sourcetype::mbx
```

Use this functionality carefully as it easy to go conceptually wrong, and this is difficult to fix later.

Routing events to a different index

At times, you may want to send events to a different index, either because they need to live longer than other events or because they contain sensitive information that should not be seen by all users. This can be applied to any type of event from any source, be it a file, network, or script.

All that we have to do is match the event and reset the index.

```
[contains_password_1]
DEST_KEY = _MetaData:Index
REGEX = Password reset called
FORMAT = sensitive
```

Things to note are:

- In this scenario, you will probably make multiple transforms, so be sure to make the name unique
- `DEST_KEY` starts with an underscore
- `FORMAT` does not start with `index::`
- The index `sensitive` must exist on the machine indexing the data, or the event will be lost

Lookup definitions

A simple lookup simply needs to specify a filename in `transforms.conf`, thus:

```
[testlookup]
filename = test.csv
```

Assuming `test.csv` contains the columns `user` and `group`, and our events contain the field `user`, we can reference this lookup by using the `lookup` command in `search`, as follows:

```
* | lookup testlookup user
```

Or, we can wire this lookup to run automatically in `props.conf`, thus:

```
[mysourcetype]
LOOKUP-testlookup = testlookup user
```

That's all you need to get started, and this probably covers most cases. See the *Using lookups to enrich data* section in *Chapter 6, Extending Search*, for instructions on creating lookups.

Wildcard lookups

In *Chapter 9, Summary Indexes and CSV Files*, we edited `transforms.conf` but did not explain what was happening. Let's take another look. Our transform stanza looks like this:

```
[flatten_summary_lookup]
filename = flatten_summary_lookup.csv
match_type = WILDCARD(url)
max_matches = 1
```

Stepping through what we added, we have:

- `match_type = WILDCARD(url)`: This says that the value of the field `url` in the lookup file may contain wildcards. In our example, the URL might look like `/contact/*` in our CSV file.
- `max_matches = 1`: By default, up to 10 entries that match in the lookup file will be added to an event, with the values in each field being added to a multivalued field. In this case, we only want the first match to be applied.

CIDR wildcard lookups

CIDR wildcards look very similar to text-based wildcards but use Classless Inter-Domain Routing rules to match lookup rows against an IP address. Let's try an example.

Say we have this lookup file:

```
ip_range,network,datacenter
10.1.0.0/16,qa,east
10.2.0.0/16,prod,east
10.128.0.0/16,qa,west
10.129.0.0/16,prod,west
```

It has this corresponding definition in `transforms.conf`:

```
[ip_address_lookup]
filename = ip_address_lookup.csv
match_type = CIDR(ip_range)
max_matches = 1
```

And, there are a few events such as these:

```
src_ip=10.2.1.3 user=mary
src_ip=10.128.88.33 user=bob
src_ip=10.1.35.248 user=bob
```

We could use our lookup to enrich these events like so:

```
src_ip="*"
| lookup ip_address_lookup ip_range as src_ip
| table src_ip user datacenter network
```

This would match the appropriate IP address and give us a table like this one:

	src_ip ↕	user ↕	datacenter ↕	network ↕
1	10.2.1.3	mary	east	prod
2	10.128.88.33	bob	west	qa
3	10.1.35.248	bob	east	qa

The query also shows that you could use the same lookup for different fields by using the `as` keyword in the `lookup` call.

Using time in lookups

A temporal lookup is used to enrich events based on when the event happened. To accomplish this, we specify the beginning of a time range in the lookup source and then specify a format for this time in our lookup configuration. Using this mechanism, lookup values can change over time, even retroactively.

Here is a very simple example to attach a `version` field based on time. Say we have the following CSV file:

```
sourcetype,version,time
impl_splunk_gen,1.0,2012-09-19 02:56:30 UTC
impl_splunk_gen,1.1,2012-09-22 12:01:45 UTC
impl_splunk_gen,1.2,2012-09-23 18:12:12 UTC
```

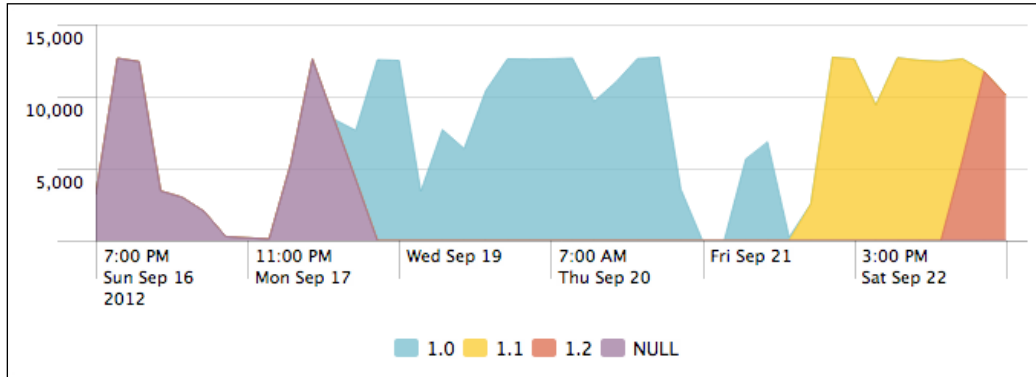
We then use the lookup configuration in `transforms.conf` to specify which field in our lookup will be tested against the time in each event and what the format of the time field will be:

```
[versions]
filename = versions.csv
time_field = time
time_format = %Y-%m-%d %H:%M:%S %Z
```

With this in place, we can now use our lookup in search, like so:

```
sourcetype=impl_splunk_gen error
| lookup versions sourcetype
| timechart count by version
```

This would give us a chart of errors (by version) over time, like so:



Other use cases include tracking deployments across environments and tracking activity from disabled accounts.

Using REPORT

Attributes of the format `REPORT-foo` in `props.conf` call stanzas in `transforms.conf` at search time, which means that they cannot affect metadata fields. `EXTRACT` definitions are more convenient to write as they live entirely in a single attribute in `props.conf`, but there are a couple of things that can only be done using a `REPORT` attribute paired with a transform defined in `transforms.conf`.

Creating multivalue fields

Assuming some value might happen multiple times in a given event, an `EXTRACT` definition can only match the first occurrence. For example, say we have the event:

```
2012-08-25T20:18:09 action=send a@b.com c@d.com e@f.com
```

We could pull the first e-mail address using the following extraction:

```
EXTRACT-email = (?i)(?P<email>[a-zA-Z0-9._]+@[a-zA-Z0-9._]+)
```

This would set the field `email` to `a@b.com`. Using a `REPORT` attribute and transform stanza, we can capture all of the e-mail addresses using the `MV_ADD` attribute.

The `props` stanza would look like this:

```
REPORT-mvemail = mvemail
```

The `transforms.conf` stanza would then look like this:

```
[mvemail]
REGEX = (?i) ([a-zA-Z0-9._]+@[a-zA-Z0-9._]+)
FORMAT = email::$1
MV_ADD = true
```

The `MV_ADD` attribute also has the effect that, if some other configuration has already created the `email` field, all values that match will be added to the event.

Creating dynamic fields

Sometimes, it can be useful to dynamically create fields from an event. For instance, say we have an event such as:

```
2012-08-25T20:18:09 action=send from_335353("a@b.com") to_223523("c@d.com") cc_39393("e@f.com") cc_39394("g@h.com")
```

It would be nice to pull `from`, `to`, and `cc` as fields, but we may not know all of the possible field names. This stanza in `transforms.conf` would create the fields we want, dynamically:

```
[dynamic_address_fields]
REGEX=\s(\S+)\_(\S+)\("(.*)"\)
FORMAT = $1::$2
MV_ADD=true
```

While we're at it, let's put the numeric value after the field name into a value:

```
[dynamic_address_ids]
REGEX=\s(\S+)\_(\S+)\("(.*)"\)
FORMAT = $1::$2
MV_ADD=true
```

This gives us multivalued fields like the ones in the following screenshot:

action ↕	cc ↕	from ↕	to ↕
send	e@f.com	a@b.com	c@d.com
	g@h.com	335353	223523
	39393		
	39394		

One thing that we cannot do is add extra text into the `FORMAT` attribute. For instance, in the second case, it would be nice to use a `FORMAT` attribute such as this one:

```
FORMAT = $1_id::$2
```

Unfortunately, this will not function as we hope and will instead create the field `id`.

Chaining transforms

As covered before in the *Attributes with class* section, transforms are executed in a particular order. In most cases, this order does not matter, but there are occasions when you might want to chain transforms together, with one transform relying on a field created by a previous transform.

A good example is the source flattening that we used previously, in the *Overriding source* section. If this transform happened before our transform in the *Creating a session field from source* section, our session field would always have the value `x`.

Let's reuse two transforms from previous sections and then create one more transform. We will chain them to pull the first part of `session` into yet another field. Say we have these transforms:

```
[myapp_session]
SOURCE_KEY = MetaData:Source
REGEX = session_(.*?)\.log
FORMAT = session::$1
WRITE_META = True
```

```
[myapp_flatten_source]
SOURCE_KEY = MetaData:Source
DEST_KEY = MetaData:Source
REGEX = (.?session_).*log
FORMAT = source::$1x.log
```

```
[session_type]
SOURCE_KEY = session
REGEX = (.?)-
FORMAT = session_type::$1
WRITE_META = True
```

To ensure that these transforms run in order, the simplest thing would be to place them in a single `TRANSFORMS` attribute in `props.conf`, like so:

```
[source:*session_*.log]
TRANSFORMS-s = myapp_session,myapp_flatten_source,session_type
```

We can use `source` from our sample event specified inside `transforms.conf` like this:

```
source=/logs/myapp.session_foo-jA5MDkyMjEwMTIK.log
```

Stepping through the transforms, we have:

- `myapp_session`: Reading from the metadata field, `source`, creates the indexed field `session` with the value `foo-jA5MDkyMjEwMTIK`
- `myapp_flatten_source`: Resets the metadata field, `source`, to `/logs/myapp.session_x.log`
- `session_type`: Reading from our newly indexed field, `session`, creates the field `session_type` with the value `foo`

This same ordering logic can be applied at search time using the `EXTRACT` and `REPORT` stanzas. This particular case needs to be calculated as indexed fields, if we want to search for these values, since the values are part of a metadata field.

Dropping events

Some events are simply not worth indexing. The hard part is figuring out which ones these are and making very sure you're not wrong. Dropping too many events can make you blind to real problems at critical times and can introduce more problems than tuning Splunk to deal with the greater volume of data in the first place.

With that warning stated, if you know what events you do not need, the procedure for dropping events is pretty simple. Say we have an event such as this one:

```
2012-02-02 12:24:23 UTC TRACE Database call 1 of 1,000. [...]
```

I know absolutely that, in this case and for this particular source type, I do not want to index `TRACE` level events.

In `props.conf`, I create a stanza for my source type, thus:

```
[mysourcetype]
TRANSFORMS-droptrace=droptrace
```

Then, I create the following transform in `transforms.conf`:

```
[droptrace]
REGEX=^\d{4}-\d{2}-\d{2}\s+\d{1,2}:\d{2}:\d{1,2}\s+[A-Z]+\sTRACE
DEST_KEY=queue
FORMAT=nullQueue
```

This `REGEX` attribute is purposefully as strict as I can make it. It is vital that I do not accidentally drop other events, and it is better for this brittle pattern to start failing and to let through `TRACE` events rather than for it to do the opposite.

fields.conf

We need to add to `fields.conf` any indexed fields we create, or they will not be searched efficiently, or may even not function at all. For our examples in the `transforms.conf` section, `fields.conf` would look like this:

```
[session_type]
INDEXED = true

[session]
INDEXED = true

[host_owner]
INDEXED = true

[host_type]
INDEXED = true

[slow_request]
INDEXED = true

[loglevel]
INDEXED = true
```

These stanzas instruct Splunk to not look in the body of the events for the value being queried. Take, for instance, the following search:

```
host_owner=v1b
```

Without this entry, the actual query would essentially be:

```
v1b | search host_owner=v1b
```

With the expectation that the value `v1b` is in the body of the event, this query simply won't work. Adding the entry to `fields.conf` fixes this.

In the case of `loglevel`, since the value is in the body, the query will work, but it will not take advantage of the indexed field, instead only using it to filter events after finding all events that contain the bare word.

outputs.conf

This configuration controls how Splunk will forward events. In the vast majority of cases, this configuration exists on Splunk Forwarders, sending their events to Splunk indexers. An example would look like this:

```
[tcpout]
defaultGroup = nyc

[tcpout:nyc]
autoLB = true
server = 1.2.3.4:9997,1.2.3.6:9997
```

It is possible to use transforms to route events to different server groups, but it is not commonly used as it introduces a lot of complexity that is generally not needed.

indexes.conf

Put simply, `indexes.conf` determines where data is stored on disk, how much is kept, and for how long. An index is simply a named directory with a specific structure. Inside this directory structure, there are a few metadata files and subdirectories; the subdirectories are called **buckets** and actually contain the indexed data.

A simple stanza looks like this:

```
[implSplunk]
homePath = $SPLUNK_DB/implSplunk/db
coldPath = $SPLUNK_DB/implSplunk/colddb
thawedPath = $SPLUNK_DB/implSplunk/thaweddb
```

Let's step through these attributes:

- `homePath`: This is the location for recent data.
- `coldPath`: This is the location for older data.
- `thawedPath`: This is a directory where buckets can be restored. It is an unmanaged location. This attribute must be defined, but I for one, have never actually used it.

An aside about the terminology of buckets is probably in order. It is as follows:

- `hot`: This is a bucket that is currently open for writing. It lives in `homePath`.
- `warm`: This is a bucket that was created recently but is no longer open for writing. It also lives in `homePath`.

- `cold`: This is an older bucket that has been moved to `coldPath`. It is moved when `maxWarmDBCount` has been exceeded.
- `frozen`: For most installations, this simply means deleted. For customers who want to archive buckets, `coldToFrozenScript` or `coldToFrozenDir` can be specified to save buckets.
- `thawed`: A thawed bucket is a frozen bucket that has been brought back. It is special in that it is not managed, and it is not included in **All time** queries. When using `coldToFrozenDir`, only the raw data is typically kept, so `splunk rebuild` will need to be used to make the bucket searchable again.

How long data stays in an index is controlled by these attributes:

- `frozenTimePeriodInSecs`: This setting dictates the oldest data to keep in an index. A bucket will be removed when its newest event is older than this value. The default value is approximately 6 years.
- `maxTotalDataSizeMB`: This setting dictates how large an index can be. The total space used across all hot, warm, and cold buckets will not exceed this value. The oldest bucket is always frozen first. The default value is 500 gigabytes.

It is generally a good idea to set both of these attributes. `frozenTimePeriodInSecs` should match what users expect. `maxTotalDataSizeMB` should protect your system from running out of disk space.

Less commonly used attributes include:

- `coldToFrozenDir`: If specified, buckets will be moved to this directory instead of being deleted. This directory is not managed by Splunk, so it is up to the administrator to make sure that the disk does not fill up.
- `maxHotBuckets`: A bucket represents a slice of time and will ideally span as small a slice of time as is practical. I would never set this value to less than 3, but ideally, it should be set to 10.
- `maxDataSize`: This is the maximum size for an individual bucket. The default value is set by processor type and is generally acceptable. The larger a bucket, the fewer buckets have to be opened to complete a search, but the more disk space will be needed before a bucket can be frozen. The default is `auto`, which will never top 750 MB. The setting `auto_high_volume`, which equals 1 GB on 32-bit systems and 10 GB on 64-bit systems, should be used for indexes that receive more than 10 GB a day.

We will discuss sizing multiple indexes in *Chapter 11, Advanced Deployments*.

authorize.conf

This configuration stores definitions of capabilities and roles. These settings affect search and the web interface. They are generally managed through the interface at **Manager | Access controls**, but a quick look at the configuration itself may be useful.

A role stanza looks like this:

```
[role_power]
importRoles = user
schedule_search = enabled
rtsearch      = enabled
srchIndexesAllowed = *
srchIndexesDefault = main
srchDiskQuota  = 500
srchJobsQuota  = 10
rtSrchJobsQuota = 20
```

Let's step through these settings:

- `importRoles`: This is a list of roles to import capabilities from. The set of capabilities will be the merging of capabilities from imported roles and added capabilities.
- `schedule_search` and `rtsearch`: These are two capabilities enabled for the role `power` that were not necessarily enabled for the imported roles.
- `srchIndexesAllowed`: What indexes this role is allowed to search. In this case, all are allowed.
- `srchIndexesDefault`: What indexes to search by default. This setting also affects the data shown on **Search | Summary**. If you have installed the `ImplementingSplunkDataGenerator` app, you will see `impl_splunk_*` source types on this page even though this data is actually stored in the `implsplunk` index.
- `srchDiskQuota`: Whenever a search is run, the results are stored on disk until they expire. The expiration can be set explicitly when creating a saved search, but the expiration is automatically set for interactive searches. Users can delete old results from the **Jobs** view.
- `srchJobsQuota`: Each user is limited to a certain number of concurrently running searches. The default is 3. Users with the `power` role are allowed 10, while those with the `admin` role are allowed 50.
- `rtSrchJobsQuota`: Similarly, this is the maximum number of concurrently running real-time searches. The default is 6.

savedsearches.conf

This configuration contains saved searches and is rarely modified by hand.

times.conf

This configuration holds definitions for time ranges that appear in the time picker.

commands.conf

This configuration specifies commands provided by an app. We will use this in *Chapter 12, Extending Splunk*.

web.conf

The main settings changed in this file are the port for the web server, the SSL certificates, and whether to start the web server at all.

User interface resources

Most Splunk apps consist mainly of resources for the web application. The app layout for these resources is completely different from all other configurations

Views and navigation

Like `.conf` files, view and navigation documents take precedence in the following order:

1. `$(SPLUNK_HOME)/etc/users/$username/$appname/local`: When a new dashboard is created, it lands here. It will remain here until the permissions are changed to **App** or **Global**.
2. `$(SPLUNK_HOME)/etc/apps/$appname/local`: Once a document is shared, it will be moved to this directory.
3. `$(SPLUNK_HOME)/etc/apps/$appname/default`: Documents can only be placed here manually. You should do this if you are going to share an app.

Unlike `.conf` files, these documents *do not* merge.

Within each of these directories, views and navigation end up under the directories `data/ui/views` and `data/ui/nav`, respectively. So, given a view `foo`, for the user `bob`, in the app `app1`, the initial location for the document will be:

```
$SPLUNK_HOME/etc/users/bob/app1/local/data/ui/views/foo.xml
```

Once the document is shared, it will be moved to:

```
$SPLUNK_HOME/etc/apps/app1/local/data/ui/views/foo.xml
```

Navigation follows the same structure, but the only navigation document that is ever used is called `default.xml`, for instance:

```
$SPLUNK_HOME/etc/apps/app1/local/data/ui/nav/default.xml
```

You can edit these files directly on the disk instead of through the web interface, but Splunk will probably not see the changes without a restart—unless you use a little trick. To reload changes to views or navigation made directly on disk, load the URL `http://mysplunkserver:8000/debug/refresh`, replacing `mysplunkserver` appropriately. If all else fails, restart Splunk.

Appserver resources

Outside of views and navigation, there are a number of resources that the web application will use. For instance, applications and dashboards can reference CSS and images, as we did in *Chapter 7, Working with Apps*. These resources are stored under `$SPLUNK_HOME/etc/apps/$appname/appserver/`. There are a few directories that appear under this directory, as follows:

- `static`: Any static files that you would like to use in your application are stored here. There are a few magic documents that Splunk itself will use, for instance, `appIcon.png`, `screenshot.png`, `application.css`, and `application.js`. Other files can be referenced using `includes` or `templates`. See the *Using ServerSideInclude in a complex dashboard* section in *Chapter 7, Working with Apps* for an example of referencing `includes` and static images.
- `event_renderers`: Event renderers allow you to run special display code for specific event types. We will write an event renderer in *Chapter 12, Extending Splunk*.
- `templates`: It is possible to create special templates using the `mako` template language. It is not commonly done.

- `modules`: This is where new modules that are provided by apps are stored. Examples of this include the `Google Maps` and `Sideview Utils` modules. See <http://dev.splunk.com> for more information about building your own modules or use existing modules as an example.

Metadata

Object permissions are stored in files located at `$SPLUNK_HOME/etc/apps/$appname/metadata/`. The two possible files are `default.meta` and `local.meta`.

These files:

- Are only relevant to the resources in the app where they are contained
- Do merge, with entries in `local.meta` taking precedence
- Are generally controlled by the admin interface
- Can contain rules that affect all configurations of a particular type, but this entry must be made manually

In the absence of these files, resources are limited to the current app.

Let's look at `default.meta` for `is_app_one`, as created by Splunk:

```
# Application-level permissions
[]
access = read : [ * ], write : [ admin, power ]

### EVENT TYPES
[eventtypes]
export = system

### PROPS
[props]
export = system

### TRANSFORMS
[transforms]
export = system

### LOOKUPS
[lookups]
```

```

export = system

### VIEWSTATES: even normal users should be able to create shared
viewstates
[viewstates]
access = read : [ * ], write : [ * ]
export = system

```

Stepping through this snippet, we have:

- The `[]` stanza states that all users should be able to read everything in this app but that only users with the `admin` or `power` roles should be able to write to this app.
- The `[eventtypes]`, `[props]`, `[transforms]`, and `[lookups]` stanzas say that all configurations of each type in this app should be shared by all users in all apps, by default. `export=system` is equivalent to **Global** in the user interface.
- The `[viewstates]` stanza gives all users the right to share their viewstates globally. A **viewstate** contains information about dashboard settings made through the web application, for instance, chart settings. Without this, chart settings applied to a dashboard or saved search would not be available.

Looking at `local.meta`, we see settings created by the web application for the configurations we created through the web application.

```

[indexes/summary_impl_splunk]
access = read : [ * ], write : [ admin, power ]

[views/errors]
access = read : [ * ], write : [ admin, power ]
export = system
owner = admin
version = 4.3
modtime = 1339296668.151105000

[savedsearches/top%20user%20errors%20pie%20chart]
export = none
owner = admin
version = 4.3
modtime = 1338420710.720786000

[viewstates/flashtimeline%3Ah2v14xkb]
owner = nobody

```

```
version = 4.3
modtime = 1338420715.753642000

[props/impl_splunk_web/LOOKUP-web_section]
access = read : [ * ]
export = none
owner = admin
version = 4.3
modtime = 1346013505.279379000

...
```

You get the idea. The web application will make very specific entries for each object created. When distributing an application, it is generally easier to make blanket permissions in `metadata/default.meta`, as appropriate for the resources in your application.

For an application that simply provides dashboards, no metadata at all will be needed, as the default for all resources (apps) will be acceptable.

If your application provides resources to be used by other applications, for instance, lookups or extracts, your `default.meta` file might look like this:

```
### PROPS
[props]
export = system

### TRANSFORMS
[transforms]
export = system

### LOOKUPS
[lookups]
export = system
```

This states that everything in your `props.conf` and `transforms.conf` files, and all lookup definitions, are merged into the logical configuration of every search.

Summary

This chapter provided an overview of how configurations work and a commentary on the most common aspects of Splunk configuration. This is by no means a complete reference for these configurations, which I will leave to the official documentation. I find the easiest way to get to the official documentation for a particular file is to query your favorite search engine for `splunk configname.conf`.

In *Chapter 11, Advanced Deployments*, we will dig into distributed deployments, and look at how they are efficiently configured. What you have learned in this chapter will be vital to understanding what is considered best practice.

11

Advanced Deployments

When you first started Splunk, you probably installed it on one machine, imported some logs, and got to work searching. It is wonderful that you can try the product out so easily, but once you move into testing and production, things can get much more complicated, and a bit of planning will save you from trouble later.

In this chapter, we will discuss getting data in, the different parts of a distributed deployment, distributed configuration management, sizing your installation, security concerns, and backup strategies.

Planning your installation

There are a few questions that you need to answer to determine how many Splunk instances will be involved in your deployment:

- How much data will be indexed per day? How much data will be kept?
The rule of thumb is 100 gigabytes per day per Splunk indexer, assuming you have fast disks. See the *Sizing indexers* section for more information.
- How many searches will be running simultaneously?
This number is probably smaller than you think. This is not the number of users who may be using Splunk, but how many simultaneous queries are running. This varies by the type of queries your group runs.
- What are the sources of data?
Where your data comes from can definitely affect your deployment. Planning for all of the possible data that you might want to consume can save you from trouble later. See the *Common data sources* section for examples.

- How many data centers do you need to monitor?
Dealing with servers in multiple locations introduces another level of complexity, to which there is no single answer. See *Deploying the Splunk binary* section for a few example deployments.
- How will you deploy the Splunk binary?
- How will you distribute configurations?

We will touch on these topics and more.

Splunk instance types

In a distributed deployment, different Splunk processes will serve different purposes. There are four stages of processing that are generally spread across two to four layers. The stages of processing include:

- **input:** This stage consumes raw data, from log files, ports, or scripts
- **parsing:** This stage splits raw data into events, parses time, sets base metadata, runs transforms, and so on
- **indexing:** This stage stores the data and optimizes indexes
- **searching:** This stage runs queries and presents the results to the user

These different stages can all be accomplished in one process, but splitting them across servers can improve performance as log volumes and search load increase.

Splunk forwarders

Each machine that contains the log files generally runs a Splunk forwarder process. The job of this process is to read logs on that machine or to run scripted inputs. This installation is either:

- A full installation of Splunk, configured to forward data instead of indexing it
- **Splunk Universal Forwarder**, which is essentially Splunk with everything needed for indexing or searching removed

With a full installation of Splunk, the process can be configured as one of two kinds of forwarder:

- A **light forwarder** is configured to not parse events but instead to forward the raw stream of data to indexers. This installation has the advantages that it uses very few resources on the machine running the forwarder (unless the number of files being scanned is very large) and that the configuration is simple. It has the disadvantage that the indexers will do more work. If this is what you need, it is recommended that you use the Splunk Universal Forwarder.
- A **heavy forwarder** is configured to parse events, forwarding these parsed or "cooked" events to the indexers. This has the advantage that the indexer does less work but the disadvantage that more configurations need to be pushed to the forwarders. This configuration also uses approximately double the CPU and memory required for a light forwarder configuration.

For most customers, the Splunk Universal Forwarder is the right answer.

The most important configurations to a forwarder installation are:

- `inputs.conf`: This defines what files to read, network ports to listen to, or scripts to run.
- `outputs.conf`: This defines which indexer(s) should receive the data.
- `props.conf`: As discussed in *Chapter 10, Configuring Splunk*, very little of this configuration is relevant to the input stage, but much of it is relevant to the parse stage. The simplest way to deal with this complexity is to send `props.conf` everywhere so that whatever part of the configuration is needed is available. We will discuss this further in the *Using apps to organize configuration* section in this chapter.
- `default-mode.conf`: This configuration is used to disable processing modules. Most modules are disabled in the case of a light forwarder.
- `limits.conf`: The main setting here is `maxKBps`, which controls how much bandwidth each forwarder will use. The default setting for a light forwarder is very low to prevent flooding the network or overtaxing the forwarding machine. This value can usually be increased safely. It is often increased to the limits of the networking hardware.

We will discuss deploying the forwarder under the *Deploying the Splunk binary* section in this chapter.

Splunk indexer

In most deployments, indexers handle both parsing and indexing of events. If there is only one Splunk indexer, the search is typically handled on this server as well.

An **indexer**, as the name implies, indexes the data. It needs direct access to fast disks, whether they are local disks, SANs, or network volumes.



In my experience, **NFS** does not work reliably for storing Splunk indexes or files. Splunk expects its disks to act like a local disk, which, at times, NFS does not. It is fine to read logs from NFS. **iSCSI** works very well for indexers, as does **SAN**.

The configurations that typically matter to a Splunk indexer are:

- `inputs.conf`: This configuration typically has exactly one input, `[splunktcp://9997]`. This stanza instructs the indexer to listen for connections from Splunk forwarders on port 9997.
- `indexes.conf`: This configuration specifies where to place indexes and how long to keep data. By default:
 - all data will be written to `$SPLUNK_HOME/var/lib/splunk`
 - the index will grow to a maximum size of 500 gigabytes before dropping the oldest events
 - the index will retain events for a maximum of six years before dropping the oldest events

Events will be dropped when either limit is reached. We will discuss changing these values under the *Sizing indexers* section.

- `props.conf` and `transforms.conf`: If the indexer handles parsing, these configurations control how the data stream is broken into events, how the date is parsed, and what indexed fields are created, if any.
- `server.conf`: This contains the license server address.



See the *Sizing indexers* section for a discussion about how many indexers you might need.

Splunk search

When there is only one Splunk server, search happens along with indexing. Until log volumes increase beyond what one server can handle easily, this is fine. In fact, splitting off the search instance might actually hurt performance as there is more overhead involved in running a distributed search.

Most configurations pertaining to search are managed through the web interface. The configuration specifically concerning distributed search is maintained at **Manager | Distributed search**.

Common data sources

Your data may come from a number of sources; these can be files, network ports, or scripts. Let's walk through a few common scenarios.

Monitoring logs on servers

In this scenario, servers write their logs to a local drive, and a forwarder process monitors these logs. This is the typical Splunk installation.

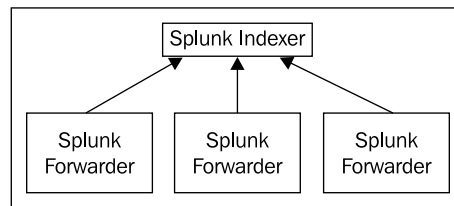
The advantages of this approach include:

- This process is highly optimized. If the indexers are not overworked, events are usually searchable within a few seconds.
- Slowdowns caused by network problems or indexer overload are handled gracefully. The forwarder process will pick up where it left off when the slowdown is resolved.
- The agent is light, typically using less than 100 megabytes of RAM and a few percent of one CPU. These values go up with the amount of new data written and the number of files being tracked. See `inputs.conf` in *Chapter 10, Configuring Splunk*, for details.
- Logs without a time zone specified will inherit the time zone of the machine running the forwarder. This is almost always what you want.
- The hostname will be picked up automatically from the host. This is almost always what you want.

The disadvantages of this approach include:

- The forwarder must be installed on each server. If you have a system for distributing software already, this is not a problem. We will discuss strategies under the *Deploying the Splunk binary* section.
- The forwarder process must have read rights to all logs to be indexed. This is usually not a problem but does require some planning.

This typical deployment looks like the following figure:



If your log volume exceeds 100 gigabytes of logs produced each day, you need to think about multiple indexers. We will talk about this further in the *Sizing indexers* section.

Monitoring logs on a shared drive

Some customers configure all servers to write their logs to a network share, NFS or otherwise. This setup can be made to work, but it is not ideal.

The advantages of this approach include:

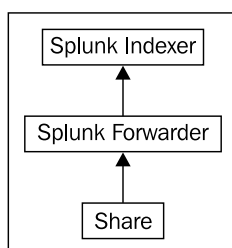
- A forwarder does not need to be installed on each server that is writing its logs to the share.
- Only the Splunk instance reading these logs needs rights to the logs.

The disadvantages of this approach include:

- The network share can become overloaded and can become a bottleneck.
- If a single file has more than a few megabytes of unindexed data, the Splunk process will only read this one log until all data is indexed. If there are multiple indexers in play, only one indexer will be receiving data from this forwarder. In a busy environment, the forwarder may fall behind.
- Multiple Splunk forwarder processes do not share information about what files have been read. This makes it very difficult to manage a failover for each forwarder process without a SAN.

- Splunk relies on the modification time to determine whether the new events have been written to a file. File metadata may not be updated as quickly on a share.
- A large directory structure will cause the Splunk process reading logs to use a lot of RAM and a large percentage of the CPU. A process to move old logs away would be advisable so as to minimize the number of files Splunk must track.

This setup often looks like the following figure:



This configuration may look simple, but unfortunately, it does not scale easily.

Consuming logs in batch

Another less common approach is to gather logs periodically from servers after the logs have rolled. This is very similar to monitoring logs on a shared drive, except that the problems of scale are possibly even worse.

The advantages of this approach include:

- A forwarder does not need to be installed on each server that is writing its logs to the share.

The disadvantages of this approach include:

- When new logs are dropped, if the files are large, the Splunk process will only read events from one file at a time. When this directory is on an indexer, this is fine, but when a forwarder is trying to distribute events across multiple indexers, only one indexer will receive events at a time.
- The oldest events in the rolled log will not be loaded until the log is rolled and copied.
- An active log cannot be copied, as events may be truncated during the copy or Splunk may be confused and believe the update file is a new log, indexing the entire file again.

Sometimes this is the only approach possible, and in those cases, you should follow a few rules:

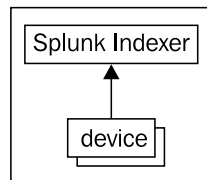
- Only copy complete logs to the watched directory.
- If possible, use batch stanzas in `inputs.conf`, instead of `monitor` stanzas, so that Splunk can delete files after indexing them.
- If possible, copy sets of logs to different Splunk servers, either to multiple forwarders that then spread the logs across multiple indexers, or possibly directly to watched directories on the indexers. Be sure to not copy the same log to multiple machines as Splunk has no mechanism for sharing file position information across instances.

Receiving syslog events

Another common source of data is **syslog**, usually from devices that have no filesystem or no support for installing software. These sources are usually devices or appliances, and usually send those events using UDP packets. Syslog management deserves a book of its own, so we will only discuss how to integrate syslog with Splunk at a high level.

Receiving events directly on the Splunk indexer

For very small installations, it may be acceptable to have your Splunk server listen directly for syslog events. This installation looks essentially like the following figure:



On the Splunk indexer, you would create an input for `syslog`, listening on `udp` or `tcp`. The `inputs.conf` configuration would look like:

```
[udp://514]
sourcetype = syslog
```

The advantage of this approach is its simplicity. The major caveat is that, if the Splunk process is down or busy for some reason, you will lose messages. Reasons for dropped events could include a heavy system load, large queries, a slow disk, network problems, or a system upgrade.

If your syslog events are important to you, it is worth the trouble to at least use a native syslog receiver on the same hardware, but you should ideally use separate hardware.

Using a native syslog receiver

The best practice is to use a standalone syslog receiver to write events to disk. Examples of syslog receivers include **syslog-ng** or **rsyslog**. Splunk is then configured to monitor the directories written by the syslog receiver.

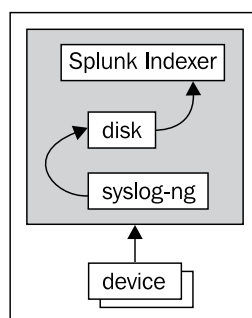


Ideally, the syslog receiver should be configured to write one file or directory per host. `inputs.conf` can then be configured to use `host_segment` or `host_regex` to set the value of `host`. This configuration has the advantage that `props.conf` stanzas can be applied by `host`, for instance, setting `TZ` by `hostname` pattern. This is not possible if `host` is parsed out of the log messages, as is commonly the case with `syslog`.

The advantages of a standalone process include:

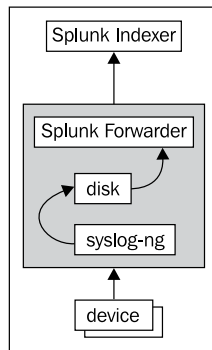
- A standalone process has no other tasks to accomplish and is more likely to have the processor time to retrieve events from the kernel buffers before data is pushed out of the buffer
- The interim files act as a buffer so that, in the case of a Splunk slowdown or outage, events are not lost
- The syslog data is on disk, so it can be archived independently or queried with other scripts, as appropriate
- If a file is written for each host, the hostname can be extracted from the path to the file, and different parsing rules (for instance time zone) can be applied at that time

A small installation would look like the following figure:



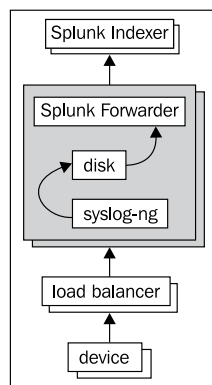
Since the configuration of the native syslog process is simple and unlikely to change, simply using another process on your single Splunk instance will add some level of protection from losing messages. A slow disk, high CPU load, or memory pressure can still cause problems, but you at least won't have to worry about restarting the Splunk process.

The next level of protection would be to use separate hardware to receive the syslog events and to use a Splunk forwarder to send the events to one or more Splunk indexers. That setup looks like the following figure:



This single machine is still a single point of failure, but it has the advantage that the Splunk server holding the indexes can be restarted at will and will not affect the instance receiving the syslog events.

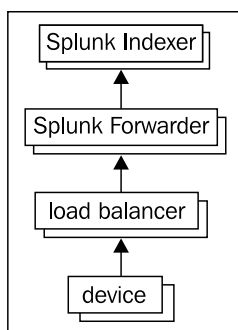
The next level of protection is to use a load balancer or a dynamic DNS scheme to spread the syslog data across multiple machines receiving the syslog events, which then forward the events to one or more Splunk indexers. That setup looks somewhat like the following figure:



This setup is complicated but very resilient as only a large network failure will cause loss of events.

Receiving syslog with a Splunk forwarder

It is also possible to use Splunk instances to receive the syslog events directly, which then forward the forwarders to Splunk indexers. This setup might look somewhat like the following figure:



These interim Splunk forwarder processes can be configured with a large input buffer using the `queueSize` and `persistentQueueSize` settings in `inputs.conf`. Note that these interim forwarders cannot be light forwarders. There are a few advantages to this approach that I can think of:

- If these Splunk forwarder processes are in the data center with the device producing the events, the forwarder process will set the time zone of the events. If you have devices in data centers in multiple time zones, this can be very helpful.
- The work of parsing the events will be handled at this stage, offloading some work from the indexers.

One disadvantage is that any parsing rules that are relevant to events parsed by these interim forwarders must be installed at this layer, which may require a restart when there are changes.

Consuming logs from a database

Some applications are built to store their logs in a database. This has the advantage that the logs are centralized, but the disadvantage that it is difficult to scale beyond the limits of the database server. If the logs are pulled into Splunk, it is possible to take advantage of the Splunk interface and correlate these events with other logs.

The process to consume database logs is essentially:

1. Build the query to retrieve the appropriate events; something as follows:

```
select date,id,log from log_table
```
2. Identify the field that you will use as your "pointer". This is usually either an ID field or a date field.
3. Modify the query to use this pointer field; use something such as the following code:

```
select date,id,log from log_table where id>4567
```
4. Use scripted input to run this query, capture the pointer field, and print the results.

There are a number of applications in a number of languages available at <http://splunkbase.com> to get you started, but you can use any language and any tool you like.

The app I know the best is `jdbc scripted input`, which uses Java and a user-provided jdbc driver. Just to quickly illustrate how it is used, perform the following steps:

1. Ensure Java 1.5 or greater is installed.
2. Download the app.
3. Copy your jdbc driver JAR to `bin/lib`.
4. Duplicate `bin/example` to `bin/myapp`.
5. Modify `bin/myapp/query.properties` to look something like the following code:

```
driverClass=com.mysql.jdbc.Driver
connectionString=jdbc:mysql://mydb:3306/myapp?user=u&password=p
iteratorField=id
query=select date,id,log from entries where id>${id} order by id
```

6. Add a matching stanza to `inputs.conf`.

```
[script://./bin/run.sh myapp]
interval = 60
sourcetype = myapp
source = jdbc
```

That should be it. `iteratorField` is not needed if your query handles not retrieving duplicate data some other way.

Using scripts to gather data

A scripted input in Splunk is simply a process that outputs text. Splunk will run the script periodically, as configured in `inputs.conf`. Let's make a simple example.

The configuration `inputs.conf` inside your app would contain an entry as follows:

```
[script://./bin/user_count.sh]
interval = 60
sourcetype = user_count
```

The script in `bin/user_count.sh` could contain something as follows:

```
#!/bin/sh
DATE=$(date "+%Y-%m-%d %H:%M:%S")
COUNT=$(wc -l /etc/passwd | awk '{print "users=$1}')
```

```
echo $DATE $COUNT
```

This would produce output such as this:

```
2012-10-15 19:57:02 users=84
```

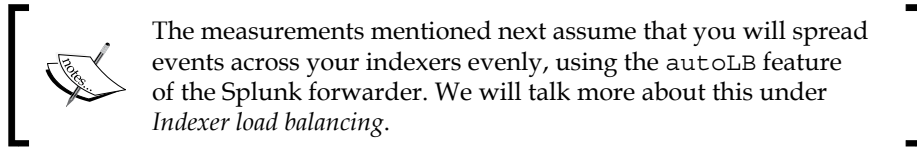
Good examples of this type of script are available in the Unix app available at splunkbase.com.

Please note that:

- New to Splunk 4.3: `interval` can be a cron schedule.
- If the name of the script ends in `.py`, Splunk will use its own copy of Python. Remember that there is no Python included with Universal Forwarder.
- Use `props.conf` to control event breaking as if this output was being read from a file.
- Set `DATETIME_CONFIG` to `CURRENT` if there is no date in the output.
- Set an appropriate `BREAK_ONLY_BEFORE` pattern if the events are multiline.
- Set `SHOULD_LINEMERGE` to `False` if the events are not multiline.
- Only one copy of each input stanza will run at a time. If a script should run continually, set `interval` to `-1`.

Sizing indexers

There are a number of factors that affect how many Splunk indexers you will need, but starting with a "model" system with typical usage levels, the short answer is 100 gigabytes of raw logs per day per indexer. In the vast majority of cases, the disk is the performance bottleneck, except in the case of very slow processors.



The model system looks like this:

- 8 gigabytes of RAM
If more memory is available, the operating system will use whatever Splunk does not use for the disk cache.
- Eight fast physical processors
On a busy indexer, two cores will probably be busy most of the time, handling indexing tasks. It is worth noting the following:
 - More processors won't hurt but will probably not make much of a difference to an indexer as the disks holding indexes will probably not keep up with the increased search load. More indexers, each with its own disks, will have more impact.
 - Virtualized slices of cores or oversubscribed virtual hosts do not work well, as the processor is actually used heavily during search, mostly decompressing raw data.
 - Slow cores designed for highly threaded applications do not work well. For instance, you should avoid older Sun SPARC processors or slices of cores on AIX boxes.
- Disks performing 800 random IOPS (input/output operations per second)
This is the value considered *fast* by Splunk engineering. Query your favorite search engine for `splunk bonnie++` for discussions about how to measure this value. The most important thing to remember when testing your disks is that you must test enough data to defeat disk cache. Remember, if you are using shared disks, that the indexers will share the available IOPS.
- No more than four concurrent searches
Please note that:
 - Most queries are finished very quickly
 - This count includes interactive queries and saved searches
 - Summary indexes and saved searches can be used to reduce the workload of common queries
 - Summary queries are simply saved searches



To test your concurrency on an existing installation, try this query:

```
index=_audit search_id action=search
| transaction maxpause=1h search_id
| concurrency duration=duration
| timechart span="1h" avg(concurrency)
max(concurrency)
```

A formula for a rough estimate (assuming eight fast processors and 8 gigabytes of RAM per indexer) might look like this:

```
indexers needed =
[your IOPs] / 800 *
[gigs of raw logs produced per day] / 100 *
[average concurrent queries] / 4
```



The behavior of your systems, network, and users make it impossible to reliably predict performance without testing. These numbers are a rough estimate at best.

Let's say you work for a mid-sized company producing about 80 gigabytes of logs per day. You have some very active users, so you might expect four concurrent queries on average. You have good disks, which `bonnie++` has shown to pull a sustained 950 IOPS. You are also running some fairly heavy summary indexing queries against your web logs, and you expect at least one to be running pretty much all the time. This gives us the following output:

```
950/800 IOPS *
80/100 gigs *
(1 concurrent summary query + 4 concurrent user queries) / 4
= 1.1875 indexers
```

You cannot really deploy 1.1875 indexers, so your choices are either to start with one indexer and see how it performs or to go ahead and start with two indexers. My advice would be to start with two indexers if at all possible. This gives you some fault tolerance, and installations tend to grow quickly as more data sources are discovered throughout the company. Ideally, when crossing the 100-gigabyte mark, it may make sense to start with three indexers and spread the disks across them. The extra capacity gives you the ability to take one indexer down and still have enough capacity to cover the normal load. See the discussion in the *Planning redundancy* section.

If we increase the number of average concurrent queries, increase the amount of data indexed per day, or decrease our IOPS, the number of indexers needed should scale more or less linearly.

If we scale up a bit more, say 120 gigabytes a day, 5 concurrent queries, and 2 summary queries running on average, we grow as follows:

```
950/800 IOPS *  
120/100 gigs *  
(2 concurrent summary query + 5 concurrent user queries) / 4  
= 2.5 indexers
```

Three indexers would cover this load, but if one indexer is down, we will struggle to keep up with data from forwarders. Ideally, in this case, we should have four or more indexers.

Planning redundancy

The term redundancy can mean different things, depending on your concern. Splunk has features to help with some of these concerns but not others. In a nutshell, up to and including Version 4.3, Splunk is excellent at making sure data is captured but provides essentially no mechanism for reliably replicating data across multiple indexers. Splunk 5, not covered in this book, adds data replication features that can eliminate most of these concerns.

Indexer load balancing

Splunk forwarders are responsible for load balancing across indexers. This is accomplished most simply by providing a list of indexers in `outputs.conf`, as shown in the following code:

```
[tcpout:nyc]  
server=nyc-splunk-index01:9997,nyc-splunk-index02:9997
```

If an indexer is unreachable, the forwarder will simply choose another indexer in the list. This scheme works very well and powers most Splunk deployments.

If the DNS entry returns multiple addresses, Splunk will balance between the addresses on the port specified.

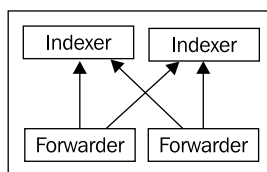
By default, the forwarder will use auto load balancing, specified by `autoLB=true`. Essentially, the forwarder will switch between indexers on a timer. This is the only option available for the Universal Forwarder and light forwarder.

On a heavy forwarder, the setting `autoLB=false` will load balance by event. This is less efficient and can cause results to be returned in a non-deterministic manner, since the original event order is not maintained across multiple indexers.

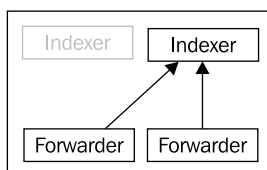
Understanding typical outages

With a single Splunk instance, an outage—perhaps for an operating system upgrade—will cause events to queue on the Splunk forwarder instances. If there are multiple indexers, forwarders will continue to send events to the remaining indexers.

Let's walk through a simplified scenario. Given these four machines, with the forwarders configured to load balance their output across two indexers as shown in the following figure:



While everything is running, half of the events from each forwarder data will be sent to each indexer. If one indexer is down, we are left with only one indexer as shown in the figure:

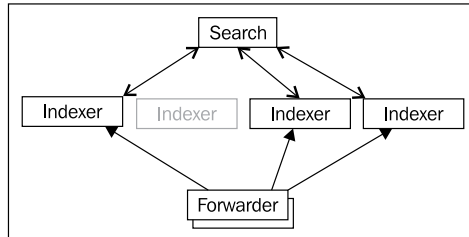


A few things happen in this case:

- All events will be sent to the remaining indexer.
- All events stored on our unavailable indexer will not be included in search results. Splunk 5 can help with this problem, at the cost of extra disks.
- Queries for recent events will work because these events will be stored on the remaining indexer, assuming the one indexer can handle the entire workload.

If our data throughput is more than a single indexer can handle, it will fall behind, which makes us essentially blind to new events until the other indexer comes back and we catch up.

As the size of our deployment increases, we can see that the impact of one indexer outage affects our results less, as shown in the following figure:



In this case, we have only lost 25 percent of our indexing capacity and have only lost access to 25 percent of our historical data. As long as three indexers can handle our indexing workload, our indexers will not fall behind and we will continue to have timely access to new events. As the number of indexers increases, the impact of one down indexer affects us less.

Working with multiple indexes

An **index** in Splunk is a storage pool for events, capped by size, time, or both. By default, all events will go to the index specified by `defaultDatabase`, which is called `main` but lives in a directory called `defaultdb`.

Directory structure of an index

Each index occupies a set of directories on disk. By default, these directories live in `$$SPLUNK_DB`, which, by default, is located in `$$SPLUNK_HOME/var/lib/splunk`. Looking at the following stanza for the `main` index:

```
[main]
homePath    = $$SPLUNK_DB/defaultdb/db
coldPath    = $$SPLUNK_DB/defaultdb/colddb
thawedPath  = $$SPLUNK_DB/defaultdb/thaweddb
maxHotIdleSecs = 86400
maxHotBuckets = 10
maxDataSize = auto_high_volume
```

If our Splunk installation lives at `/opt/splunk`, the index main is rooted at the path `/opt/splunk/var/lib/splunk/defaultdb`.

To change your storage location, either modify the value of `SPLUNK_DB` in `$(SPLUNK_HOME)/etc/splunk-launch.conf` or set absolute paths in `indexes.conf`.



`splunk-launch.conf` cannot be controlled from an app, which means it is easy to forget when adding indexers. For this reason, and for legibility, I would recommend using absolute paths in `indexes.conf`.

The `homePath` directories contain index-level metadata, hot buckets, and warm buckets. `coldPath` contains cold buckets, which are simply warm buckets that have aged out. See the upcoming sections *The lifecycle of a bucket* and *Sizing an index* for details.

When to create more indexes

There are several reasons for creating additional indexes. If your needs do not meet one of these requirements, there is no need to create more indexes. In fact, multiple indexes may actually hurt performance if a single query needs to open multiple indexes.

Testing data

If you do not have a test environment, you can use test indexes for staging new data. This then allows you to easily recover from mistakes by dropping the test index. Since Splunk will run on a desktop, it is probably best to test new configurations locally, if possible.

Differing longevity

It may be the case that you need more history for some source types than others. The classic example here is security logs, as compared to web access logs. You may need to keep security logs for a year or more but only need web access logs for a couple of weeks.

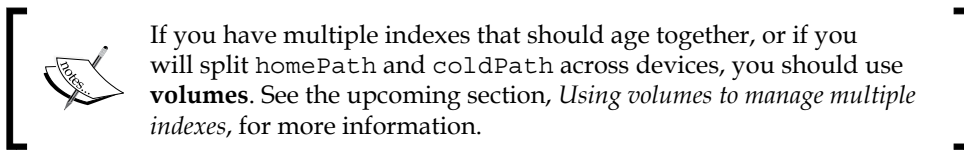
If these two source types are left in the same index, security events will be stored in the same buckets as web access logs and will age out together. To split these events up, you need to perform the following steps:

1. Create a new index called `security`, for instance.
2. Define different settings for the `security` index.
3. Update `inputs.conf` to use the new index for security source types.

For one year, you might make an `indexes.conf` setting such as this:

```
[security]
homePath   = $SPLUNK_DB/security/db
coldPath   = $SPLUNK_DB/security/colddb
thawedPath = $SPLUNK_DB/security/thaweddb
#one year in seconds
frozenTimePeriodInSecs = 31536000
```

For extra protection, you should also set `maxTotalDataSizeMB`, and possibly `coldToFrozenDir`.



Then, in `inputs.conf`, you simply need to add `index` to the appropriate stanza as follows:

```
[monitor:///path/to/security/logs/logins.log]
sourcetype=logins
index=security
```

Differing permissions

If some data should only be seen by a specific set of users, the most effective way to limit access is to place this data in a different index and then limit access to that index by using a role. The steps to accomplish this are essentially as follows:

1. Define the new index.
2. Configure `inputs.conf` or `transforms.conf` to send these events to the new index.
3. Ensure the `user` role does *not* have access to the new index.
4. Create a new role that has access to the new index.
5. Add specific users to this new role. If you are using LDAP authentication, you will need to map the role to an LDAP group and add users to that LDAP group.

To route very specific events to this new index, assuming you created an index called `sensitive`, you can create a transform as follows:

```
[contains_password]
REGEX = (?i)password[=:]
DEST_KEY = _MetaData:Index
FORMAT = sensitive
```

You would then wire this transform to a particular `sourcetype` or source index in `props.conf`. See *Chapter 10, Configuring Splunk*, for examples.

Using more indexes to increase performance

Placing different source types in different indexes can help increase performance, if those source types are not queried together. The disks will spend less time seeking when accessing the source type in question.

If you have access to multiple storage devices, placing indexes on different devices can help increase performance even more by taking advantage of different hardware for different queries. Likewise, placing `homePath` and `coldPath` on different devices can help performance.

However, if you regularly run queries that use multiple source types, splitting those source types across indexes may actually hurt performance. For example, let's imagine you have two source types called `web_access` and `web_error`.

We have the following line in `web_access`:

```
2012-10-19 12:53:20 code=500 session=abcdefg url=/path/to/app
```

And we have the following line in `web_error`:

```
2012-10-19 12:53:20 session=abcdefg class>LoginClass
```

If we want to combine these results, we could run a query like the following:

```
(sourcetype=web_access code=500) OR sourcetype=web_error
| transaction maxspan=2s session
| top url class
```

If `web_access` and `web_error` are stored in different indexes, this query will need to access twice as many buckets and will essentially take twice as long.

The lifecycle of a bucket

An index is made up of **buckets**, which go through a specific life cycle. Each bucket contains events from a particular period of time.

As touched on in *Chapter 10, Configuring Splunk*, the stages of this lifecycle are **hot**, **warm**, **cold**, **frozen**, and **thawed**. The only practical difference between hot and other buckets is that a hot bucket is being written to and has not necessarily been optimized. These stages live in different places on disk and are controlled by different settings in `indexes.conf`:

- `homePath` contains as many hot buckets as the integer value of `maxHotBuckets` and as many warm buckets as the integer value of `maxWarmDBCount`. When a hot bucket rolls, it becomes a warm bucket. When there are too many warm buckets, the oldest warm bucket becomes a cold bucket.



Do not set `maxHotBuckets` too low. If your data is not parsing perfectly, dates that parse incorrectly will produce buckets with very large time spans. As more buckets are created, these buckets will overlap, which means all buckets will have to be queried every time, and performance will suffer dramatically. A value of five or more is safe.

- `coldPath` contains cold buckets, which are warm buckets that have rolled out of `homePath` once there are more warm buckets than the value of `maxWarmDBCount`. If `coldPath` is on the same device, only a move is required; otherwise, a copy is required.
- Once the values of `frozenTimePeriodInSecs`, `maxTotalDataSizeMB`, or `maxVolumeDataSizeMB` are reached, the oldest bucket will be frozen. By default, frozen means *deleted*. You can change this behavior by specifying either:
 - `coldToFrozenDir`: This lets you specify a location to move buckets once they have aged out. The index files will be deleted, and only the compressed raw data will be kept. This essentially cuts disk usage in half. This location is unmanaged, so it is up to you to watch your disk usage.
 - `coldToFrozenScript`: This lets you specify a script to perform some action when the bucket is frozen. The script is handed the path to the bucket about to be frozen.

- `thawedPath` can contain buckets that have been restored. These buckets are not managed by Splunk and are not included in **All time** searches. To search these buckets, their time range must be included explicitly in your search. I have never actually used this directory. Search `http://splunk.com` for `restore archived`, for procedures.

Sizing an index

To determine how much disk space is needed for an index, use the following formula:

```
(gigabytes per day) * .5 * (days of retention desired)
```

Likewise, to determine how many days you can store an index, the formula is essentially:

```
(device size in gigabytes) / ( (gigabytes per day) * .5 )
```

The `.5` represents a conservative compression ratio. The log data itself is usually compressed to 10 percent of its original size. The index files necessary to speed up search brings the size of a bucket closer to 50 percent of the original size, though it is usually smaller than this.

If you plan to split your buckets across devices, the math gets more complicated unless you use volumes. Without using volumes, the math is essentially as follows:

- `homePath = (maxWarmDBCount + maxHotBuckets) * maxDataSize`
- `coldPath = maxTotalDataSizeMB - homePath`

For example, say we are given these settings:

```
[myindex]
homePath = /splunkdata_home/myindex/db
coldPath = /splunkdata_cold/myindex/colddb
thawedPath = /splunkdata_cold/myindex/thaweddb
maxWarmDBCount = 50
maxHotBuckets = 6
maxDataSize = auto_high_volume #10GB on 64-bit systems
maxTotalDataSizeMB = 2000000
```

Filling in the preceding formula, we get these values:

```
homePath = (50 warm + 6 hot) * 10240 MB = 573440 MB
coldPath = 2000000 MB - homePath = 1426560 MB
```

If we use volumes, this gets simpler and we can simply set the volume sizes to our available space and let Splunk do the math.

Using volumes to manage multiple indexes

Volumes combine pools of storage across different indexes so that they age out together. Let's make up a scenario where we have five indexes and three storage devices.

The indexes are as follows:

Name	Data per day	Retention required	Storage needed
web	50 GB	no requirement	?
security	1 GB	2 years	730 GB * 50 percent
app	10 GB	no requirement	?
chat	2 GB	2 years	1,460 GB * 50 percent
web_summary	1 GB	1 years	365 GB * 50 percent

Now let's say we have three storage devices to work with, mentioned in the following table:

Name	Size
small_fast	500 GB
big_fast	1,000 GB
big_slow	5,000 GB

We can create volumes based on the retention time needed. `security` and `chat` share the same retention requirements, so we can place them in the same volumes. We want our hot buckets on our fast devices, so let's start there with the following configuration:

```
[volume:two_year_home]
#security and chat home storage
path = /small_fast/two_year_home
maxVolumeDataSizeMB = 300000
```

```
[volume:one_year_home]
#web_summary home storage
path = /small_fast/one_year_home
maxVolumeDataSizeMB = 150000
```

For the rest of the space needed by these indexes, we will create companion volume definitions on `big_slow`, thus:

```
[volume:two_year_cold]
#security and chat cold storage
path = /big_slow/two_year_cold
maxVolumeDataSizeMB = 850000 #([security]+[chat])*1024 - 300000

[volume:one_year_cold]
#web_summary cold storage
path = /big_slow/one_year_cold
maxVolumeDataSizeMB = 230000 #[web_summary]*1024 - 150000
```

Now for our remaining indexes, whose timeframe is not important, we will use `big_fast` and the remainder of `big_slow`, thus:

```
[volume:large_home]
#web and app home storage
path = /big_fast/large_home
maxVolumeDataSizeMB = 900000 #leaving 10% for pad

[volume:large_cold]
#web and app cold storage
path = /big_slow/large_cold
maxVolumeDataSizeMB = 3700000
#(big_slow - two_year_cold - one_year_cold)*.9
```

Given the sum of `large_home` and `large_cold` is 4,600,000 MB, and a combined daily volume of approximately `web` and `app` is 60,000 MB, we should retain approximately 153 days of `web` and `app` logs with 50 percent compression. In reality, the number of days retained will probably be larger.

With our volumes defined, we now have to reference them in our index definitions:

```
[web]
homePath = volume:large_home/web
coldPath = volume:large_cold/web
thawedPath = /big_slow/thawed/web


[security]
homePath = volume:two_year_home/security
coldPath = volume:two_year_cold/security
thawedPath = /big_slow/thawed/security
```

```
coldToFrozenDir = /big_slow/frozen/security
```

```
[app]  
homePath = volume:large_home/app  
coldPath = volume:large_cold/app  
thawedPath = /big_slow/thawed/app
```

```
[chat]  
homePath = volume:two_year_home/chat  
coldPath = volume:two_year_cold/chat  
thawedPath = /big_slow/thawed/chat  
coldToFrozenDir = /big_slow/frozen/chat
```

```
[web_summary]  
homePath = volume:one_year_home/web_summary  
coldPath = volume:one_year_cold/web_summary  
thawedPath = /big_slow/thawed/web_summary
```

 thawedPath cannot be defined using a volume and must be specified for Splunk to start.

For extra protection, we specified `coldToFrozenDir` for the indexes `security` and `chat`. The buckets for these indexes will be copied to this directory before deletion, but it is up to us to make sure the disk does not fill up. If we allow the disk to fill up, Splunk will stop indexing until space is made available.

This is just one approach to using volumes. You could overlap in any way that makes sense to you as long as you understand that the oldest bucket in a volume will be frozen first, no matter what index put the bucket in that volume.

Deploying the Splunk binary

Splunk provides binary distributions for Windows and a variety of Unix operating systems. For all Unix operating systems, a compressed tar file is provided. For some platforms, packages are also provided.

If your organization uses packages, such as `deb` or `rpm`, you should be able to use the provided packages in your normal deployment process. Otherwise, installation starts by unpacking the provided tar to the location of your choice.

The process is the same whether you are installing the full version of Splunk or the Splunk Universal Forwarder.

The typical installation process involves the following process:

1. Installing the binary.
2. Adding a base configuration.
3. Configuring Splunk to launch at boot.
4. Restarting Splunk.



Having worked with many different companies over the years, I can honestly say that none of them used the same product or even methodology for deploying software. Splunk takes a hands-off approach to fit in as easily as possible into customer workflows.

Deploying from a tar file

To deploy from a tar file, the command depends on your version of tar. With a modern version of tar, you can run the following command:

```
tar xvzf splunk-4.3.x-xxx-Linux-xxx.tgz
```

Older versions may not handle gzip files directly, so you may have to run the following command:

```
gunzip -c splunk-4.3.x-xxx-Linux-xxx.tgz | tar xvf -
```

This will expand into the current directory. To expand into a specific directory, you can usually add `-C`, depending on the version of tar, as follows:

```
tar -C /opt/ -xvzf splunk-4.3.x-xxx-Linux-xxx.tgz
```

Deploying using msiexec

On Windows, it is possible to deploy Splunk using `msiexec`. This makes it much easier to automate deployment on a large number of machines.

To install silently, you can use the combination of `AGREETOLICENSE` and `/quiet`, as follows:

```
msiexec.exe /i splunk-xxx.msi AGREETOLICENSE=Yes /quiet
```

If you plan to use a deployment server, you can specify the following value:

```
msiexec.exe /i splunk-xxx.msi AGREETOLICENSE=Yes  
DEPLOYMENT_SERVER="deployment_server_name:8089" /quiet
```


Or, if you plan to overlay an app that contains `deploymentclient.conf`, you can forego starting Splunk until that app has been copied into place, as follows:

```
msiexec.exe /i splunk-xxx.msi AGREETOLICENSE=Yes LAUNCHSPLUNK=0 /quiet
```

There are options available to start reading data immediately, but I would advise deploying input configurations to your servers instead of enabling inputs via installation arguments.

Adding a base configuration

If you are using Splunk's deployment server, this is the time to set up `deploymentclient.conf`. This can be accomplished in several ways as follows:

- On the command line by running the following code:

```
$SPLUNK_HOME/bin/splunk set deploy-poll  
deployment_server_name:8089
```
- By placing a `deploymentclient.conf` in `$SPLUNK_HOME/etc/system/local/`
- By placing an app containing `deploymentclient.conf` in `$SPLUNK_HOME/etc/apps/`

The third option is what I would recommend because it allows overriding this configuration via a deployment server at a later time. We will work through an example later in the *Using Splunk deployment server* section.

If you are deploying configurations in some other way, for instance with puppet, be sure to restart the Splunk forwarder processes after deploying the new configuration.

Configuring Splunk to launch at boot

On Windows machines, Splunk is installed as a service that will start after installation and on reboot.

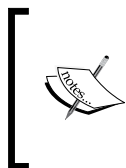
On Unix hosts, the `splunk` command line provides a way to create startup scripts appropriate for the operating system you are using. The command looks like this:

```
$SPLUNK_HOME/bin/splunk enable boot-start
```

To run Splunk as another user, provide the flag `-user`, as follows:

```
$SPLUNK_HOME/bin/splunk enable boot-start -user splunkuser
```

The startup command must still be run as root, but the startup script will be modified to run as the user provided.



If you do not run Splunk as root, and you shouldn't if you can avoid it, be sure that the Splunk installation and data directories are owned by the user specified in the `enable boot-start` command. You can ensure this by using `chmod`, such as in `chmod -R splunkuser $SPLUNK_HOME`

On Linux, you could then start the command using `service splunk start`.

Using apps to organize configuration

When working with a distributed configuration, there are a number of ways to organize these configurations. The most obvious approach might be to organize configurations by machine type. For instance, put all configurations needed by web servers into one app and all configurations needed by database servers in another app. The problem with this approach is that any changes that affect both types of machines must be made in both apps, and mistakes will most likely be made.

The less fragile but more complicated approach is to normalize your configurations, ensuring that there is only one copy of each configuration spread into multiple apps.

Separate configurations by purpose

Stepping through a typical installation, you would have configuration apps named like the following:

- **inputs-sometype**

For some logical set of inputs, you would create an app. You could use machine purpose, source type, location, operating system, or whatever makes sense in your situation. Normally, I would expect machine purpose or source type.

- **props-sometype**

This grouping should correspond to the grouping of the inputs, more or less. You may end up with props apps for more than one type, for instance machine type and location.

- **outputs-datacenter**

When deploying across data centers, it is common to place Splunk indexers in each data center. In this case, you would need an app per data center.

- **indexerbase**

Assuming your indexers are configured similarly, it is handy to put all indexer configuration into an app and deploy it like any other app.



All of these configurations are completely separate from search concerns, which should be stored in separate apps built and maintained through the Splunk web interface.

Let's imagine we have a distributed deployment across two data centers, `east` and `west`. Each data center has web servers, app servers, and database servers. In each data center we have two Splunk indexers. The apps for this setup could be as follows:

- `inputs-web`, `inputs-app`, and `inputs-db`
 - `inputs.conf` specifies the appropriate logs to monitor.
 - Each app should be distributed to each machine that is serving that purpose. If there are some machines that serve more than one purpose, they should receive all appropriate apps.
- `props-web`, `props-app`, and `props-db`
 - `props.conf` specifies how to parse the logs.
 - `transforms.conf` is included if there are relevant transforms.
 - Different portions of `props.conf` are needed at different stages of processing. Since it is difficult to know what stage is happening where, it is generally easiest to distribute these source type props apps everywhere.
- `props-west`, and `props-east`
 - Sometimes it is necessary to make configuration changes by location, for instance, configuring time zone on machines that are not set up properly. This can be accomplished by using the `TZ` setting in `props.conf` and sending this app to the appropriate data centers.
- `outputs-west`, and `outputs-east`
 - These would contain nothing but the `outputs.conf` configuration for the appropriate data center.

- `indexerbase`
 - Assuming all indexers are configured the same way, this app would contain a standard `indexes.conf` configuration, an `inputs.conf` configuration specifying the `splunktcp` port to listen to connections from Splunk forwarders, and `server.conf` specifying the address of the Splunk license server.

Let's look through an abbreviated listing of all of these files mentioned:

- For forwarders, we will need these apps:

```
inputs-web
  local/inputs.conf
    [monitor:///path/to/web/logs/access*.log]
      sourcetype = web_access
      index = web

    [monitor:///path/to/web/logs/error*.log]
      sourcetype = web_error
      index = web

inputs-app
  local/inputs.conf
    [monitor:///path/to/app1/logs/app*.log]
      sourcetype = app1
      index = app

    [monitor:///path/to/app2/logs/app*.log]
      sourcetype = app2
      index = app

inputs-db
  local/inputs.conf
    [monitor:///path/to/db/logs/error*.log]
      sourcetype = db_error

outputs-west
  local/outputs.conf
    [tcpout:west]
      server=spl-idx-west01.foo.com:9997,spl-idx-west02.foo.com:9997
      #autoLB=true is the default setting

outputs-east
  local/outputs.conf
    [tcpout:east]
      server=spl-idx-east01.foo.com:9997,spl-idx-east02.foo.com:9997
```

- All instances should receive these apps:

```
props-web
  local/props.conf
    [web_access]
      TIME_FORMAT = %Y-%m-%d %H:%M:%S.%3N %:z
      MAX_TIMESTAMP_LOOKAHEAD = 32
      SHOULD_LINEMERGE = False
      TRANSFORMS-squashpassword = squashpassword

    [web_error]
      TIME_FORMAT = %Y-%m-%d %H:%M:%S.%3N %:z
      MAX_TIMESTAMP_LOOKAHEAD = 32
      TRANSFORMS-squashpassword = squashpassword

  local/transforms.conf
    [squashpassword]
      REGEX = (?mi)^(.*)password[=:] [^,&]+$
      FORMAT = $1password=#####$2
      DEST_KEY = _raw

props-app
  local/props.conf
    [app1]
      TIME_FORMAT = %Y-%m-%d %H:%M:%S.%3N
      MAX_TIMESTAMP_LOOKAHEAD = 25
      BREAK_ONLY_BEFORE = ^\d{4}-\d{1,2}-\d{1,2}\s+\d{1,2}:\d{1,2}

    [app2]
      TIME_FORMAT = %Y-%m-%d %H:%M:%S.%3N
      MAX_TIMESTAMP_LOOKAHEAD = 25
      BREAK_ONLY_BEFORE = ^\d{4}-\d{1,2}-\d{1,2}\s+\d{1,2}:\d{1,2}

props-db
  local/props.conf
    [db_error]
      MAX_TIMESTAMP_LOOKAHEAD = 25

props-west
  local/props.conf
    [db_error]
      TZ = PST

    [web_error]
```

```
TZ = PST
```

```
props-east
  local/props.conf
    [db_error]
      TZ = EST

    [web_error]
      TZ = EST
```

- Finally, an app specifically for our indexers:

```
indexerbase
  local/indexes.conf
    [volume:two_year_home]
      path = /small_fast/two_year_home
      maxVolumeDataSizeMB = 300000

    [volume:one_year_home]
      path = /small_fast/one_year_home
      maxVolumeDataSizeMB = 150000

    [volume:two_year_cold]
      path = /big_slow/two_year_cold
      maxVolumeDataSizeMB = 1200000

    [volume:one_year_cold]
      path = /big_slow/one_year_cold
      maxVolumeDataSizeMB = 600000

    [volume:large_home]
      path = /big_fast/large_home
      maxVolumeDataSizeMB = 900000

    [volume:large_cold]
      path = /big_slow/large_cold
      maxVolumeDataSizeMB = 3000000

    [web]
      homePath = volume:large_home/web
      coldPath = volume:large_cold/web
      thawedPath = /big_slow/thawed/web

    [app]
```

```
homePath = volume:large_home/app
coldPath = volume:large_cold/app
thawedPath = /big_slow/thawed/app

[main]
homePath = volume:large_home/main
coldPath = volume:large_cold/main
thawedPath = /big_slow/thawed/main

local/inputs.conf
[splunktcp://9997]

local/server.conf
[license]
master_uri = https://spl-license.foo.com:8089
```

This is a minimal set of apps, but it should provide a decent overview of what is involved in configuring a distributed configuration. Next, we will illustrate where these apps should go.

Configuration distribution

As we have covered, in some depth, configurations in Splunk are simply directories of plain text files. Distribution essentially consists of copying these configurations to the appropriate machines and restarting the instances. You can either use your own system for distribution, such as puppet or simply a set of scripts, or use the deployment server included with Splunk.

Using your own deployment system

The advantage of using your own system is that you already know how to use it. Assuming that you have normalized your apps as described in the section *Using apps to organize configuration*, deploying apps to a forwarder or indexer consists of the following steps:

1. Set aside existing apps at `$SPLUNK_HOME/etc/apps/`.
2. Copy apps into `$SPLUNK_HOME/etc/apps/`.
3. Restart Splunk forwarder. Note that this needs to be done as the user that is running Splunk, either by calling the service script or calling `su`. On Windows, restart the `splunkd` service.

Assuming you already have a system for managing configurations, that's it.



If you are deploying configurations to indexers, be sure to only deploy configurations when downtime is acceptable, as you will need to restart the indexers to load the new configurations, ideally in a rolling manner. Do not deploy configurations until you are ready to restart as some (but not all) configurations will take effect immediately.

Using Splunk deployment server

If you do not have a system for managing configurations, you can use the deployment server included with Splunk.

Some advantages of the included deployment server are as follows:

- Everything you need is included in your Splunk installation
- It will restart forwarder instances properly when new app versions are deployed
- It is intelligent enough to not restart when unnecessary
- It will remove apps that should no longer be installed on a machine
- It will ignore apps that are not managed
- The logs for the deployment client and server are accessible in Splunk itself

Some disadvantages of the included deployment server are:


- As of Splunk 4.3, there are issues with scale beyond a few hundred deployment clients, at which point tuning is required
- The configuration is complicated and prone to typos

With these caveats out of the way, let's set up a deployment server for the apps we laid out before.

Step 1 – Deciding where your deployment server will run

For a small installation with less than a few dozen forwarders, your main Splunk instance can run the deployment server without issue. For more than a few dozen forwarders, a separate instance of Splunk makes sense.

Ideally, this instance would run on its own machine. The requirements for this machine are not large, perhaps 4 gigabytes of RAM and two processors, or possibly less. A VM would be fine.

 Define a DNS entry for your deployment server, if at all possible. This will make moving your deployment server later much simpler.

If you do not have access to another machine, you could run another copy of Splunk on the same machine running some other part of your Splunk deployment. To accomplish this, follow these steps:

1. Install Splunk in another directory, perhaps `/opt/splunk-deploy/splunk/`.
2. Start this instance of Splunk by using `/opt/splunk-deploy/splunk/bin/splunk start`. When prompted, choose different port numbers apart from the default and note what they are. I would suggest one number higher: 8090 and 8001.
3. Unfortunately, if you run `splunk enable boot-start` in this new instance, the existing startup script will be overwritten. To accommodate both instances, you will need to either edit the existing startup script, or rename the existing script so that it is not overwritten.

Step 2 – Defining your `deploymentclient.conf` configuration

Using the address of our new deployment server, ideally a DNS entry, we will build an app named `deploymentclient-yourcompanyname`. This app will have to be installed manually on forwarders but can then be managed by the deployment server.

This app should look somewhat like this:

```
deploymentclient-yourcompanyname
  local/deploymentclient.conf
  [deployment-client]

  [target-broker:deploymentServer]
  targetUri=deploymentserver.foo.com:8089
```

Step 3 – Defining our machine types and locations

Starting with what we defined under the *Separate configurations by purpose* section, we have, in the locations `west` and `east`, the following machine types:

- Splunk indexers
- db servers
- web servers
- app servers

Step 4 – Normalizing our configurations into apps appropriately

Let's use the apps we defined under the section *Separate configurations by purpose* plus the deployment client app we created in the section *Step 2 – Defining your deploymentclient.conf configuration*. These apps will live in `$SPLUNK_HOME/etc/deployment-apps/` on your deployment server.

Step 5 – Mapping these apps to deployment clients in serverclass.conf

To get started, I always start with Example 2 from `$SPLUNK_HOME/etc/system/README/serverclass.conf.example`:

```
[global]

[serverClass:AppsForOps]
whitelist.0=* .ops.yourcompany.com
[serverClass:AppsForOps:app:unix]
[serverClass:AppsForOps:app:SplunkLightForwarder]
```

Let's assume we have the machines mentioned next. It is very rare for an organization of any size to have consistently named hosts, so I threw in a couple of rogue hosts at the bottom, as follows:

```
spl-idx-west01
spl-idx-west02
spl-idx-east01
spl-idx-east02
app-east01
app-east02
app-west01
app-west02
web-east01
web-east02
web-west01
web-west02
db-east01
db-east02
db-west01
db-west02
qa01
homer-simpson
```

The structure of `serverclass.conf` is essentially as follows:

```
[serverClass:<className>]
#options that should be applied to all apps in this class

[serverClass:<className>:app:<appName>]
#options that should be applied only to this app in this serverclass
```

Please note that:

- `<className>` is an arbitrary name of your choosing.
- `<appName>` is the name of a directory in `$SPLUNK_HOME/etc/deployment-apps/`.
- The order of stanzas does not matter. Be sure to update `<className>` if you copy an `:app:` stanza. This is by far the easiest mistake to make.



It is important that configuration changes do not trigger a restart of indexers.

Let's apply this to our hosts, as follows:

```
[global]
restartSplunkd = True
#by default trigger a splunk restart on configuration change

####INDEXERS
##handle indexers specially, making sure they do not restart
[serverClass:indexers]
whitelist.0=spl-idx-*
restartSplunkd = False
[serverClass:indexers:app:indexerbase]
[serverClass:indexers:app:deploymentclient-yourcompanyname]
[serverClass:indexers:app:props-web]
[serverClass:indexers:app:props-app]
[serverClass:indexers:app:props-db]

#send props-west only to west indexers
[serverClass:indexers-west]
whitelist.0=spl-idx-west*
restartSplunkd = False
[serverClass:indexers-west:app:props-west]

#send props-east only to east indexers
```

```
[serverClass:indexers-east]
whitelist.0=spl-idx-east*
restartSplunkd = False
[serverClass:indexers-east:app:props-east]

####FORWARDERS
#send event parsing props apps everywhere
#blacklist indexers to prevent unintended restart
[serverClass:props]
whitelist.0=*
blacklist.0=spl-idx-*
[serverClass:props:app:props-web]
[serverClass:props:app:props-app]
[serverClass:props:app:props-db]

#send props-west only to west datacenter servers
#blacklist indexers to prevent unintended restart
[serverClass:west]
whitelist.0=*-west*
whitelist.1=qa01
blacklist.0=spl-idx-*
[serverClass:west:app:props-west]
[serverClass:west:app:deploymentclient-yourcompanyname]

#send props-east only to east datacenter servers
#blacklist indexers to prevent unintended restart
[serverClass:east]
whitelist.0=*-east*
whitelist.1=homer-simpson
blacklist.0=spl-idx-*
[serverClass:east:app:props-east]
[serverClass:east:app:deploymentclient-yourcompanyname]

#define our appserver inputs
[serverClass:appservers]
whitelist.0=app-*
whitelist.1=qa01
whitelist.2=homer-simpson
[serverClass:appservers:app:inputs-app]

#define our webserver inputs
[serverClass:webservers]
```

```
whitelist.0=web-*
whitelist.1=qa01
whitelist.2=homer-simpson
[serverClass:webserver:app:inputs-web]

#define our dbserver inputs
[serverClass:dbserver]
whitelist.0=db-*
whitelist.1=qa01
[serverClass:dbserver:app:inputs-db]

#define our west coast forwarders
[serverClass:fwd-west]
whitelist.0=app-west*
whitelist.1=web-west*
whitelist.2=db-west*
whitelist.3=qa01
[serverClass:fwd-west:app:outputs-west]

#define our east coast forwarders
[serverClass:fwd-east]
whitelist.0=app-east*
whitelist.1=web-east*
whitelist.2=db-east*
whitelist.3=homer-simpson
[serverClass:fwd-east:app:outputs-east]
```

You should organize the patterns and classes in a way that makes sense to your organization and data centers, but I would encourage you to keep it as simple as possible. I would strongly suggest opting for more lines than more complicated logic.

A few more things to note about the format of `serverclass.conf`:

- The number following `whitelist` and `blacklist` *must be sequential*, starting with zero. For instance, in the following example, `whitelist.3` will not be processed, since `whitelist.2` is commented:

```
[serverClass:foo]
whitelist.0=a*
whitelist.1=b*
# whitelist.2=c*
whitelist.3=d*
```

- `whitelist.x` and `blacklist.x` are tested against these values in the following order:
 - `clientName` as defined in `deploymentclient.conf`: This is not commonly used but is useful when running multiple Splunk instances on the same machine or when DNS is completely unreliable.
 - IP address: There is no CIDR matching, but you can use string patterns.
 - Reverse DNS: This is the value returned by DNS for an IP address. If your reverse DNS is not up to date, this can cause you problems, as this value is tested before the value of `hostname`, as provided by the host itself. If you suspect this, try `ping <ip of machine>` or something similar to see what the DNS is reporting.
 - Hostname as provided by forwarder: This is always tested after reverse DNS, so be sure your reverse DNS is up to date.
- When copying `:app:` lines, be very careful to update the `<className>` appropriately! This really is the most common mistake made in `serverclass.conf`.

Step 6 – Restarting the deployment server

If `serverclass.conf` did not exist, a restart of the Splunk instance running deployment server is required to activate the deployment server. After the deployment server is loaded, you can use the following command:

```
$SPLUNK_HOME/bin/splunk reload deploy-server
```

This command should be enough to pick up any changes to `serverclass.conf` and any changes in `etc/deployment-apps`.

Step 7 – Installing `deploymentclient.conf`

Now that we have a running deployment server, we need to set up the clients to call home. On each machine that will be running the deployment client, the procedure is essentially as follows:

1. Copy the `deploymentclient-yourcompanyname` app to `$SPLUNK_HOME/etc/apps/`.
2. Restart Splunk.

If everything is configured correctly you should see the appropriate apps appear in `$(SPLUNK_HOME)/etc/apps/`, within a few minutes. To see what is happening, look at the log `$(SPLUNK_HOME)/var/log/splunk/splunkd.log`.

If you have problems, enable debugging on either the client or the server by editing `$(SPLUNK_HOME)/etc/log.cfg`, followed by a restart. Look for the following lines:

```
category.DeploymentServer=WARN
category.DeploymentClient=WARN
```

Once found, change them to the following lines and restart Splunk:

```
category.DeploymentServer=DEBUG
category.DeploymentClient=DEBUG
```

After restarting Splunk, you will see the complete conversation in `$(SPLUNK_HOME)/var/log/splunk/splunkd.log`. Be sure to change the setting back once you no longer need the verbose logging!

Using LDAP for authentication

By default, Splunk authenticates using its own authentication system, which simply stores users and roles in flat files. The other two options available are LDAP and scripted authentication.

To enable LDAP authentication, perform the following steps:

1. Navigate to **Manager | Access controls | Authentication method**.
2. Check the **LDAP** checkbox.
3. Click on **Configure Splunk to use LDAP and map groups**.
4. Click on **New**.

You will then need the appropriate values to set up access to your LDAP server. Every organization sets up LDAP slightly differently, so I have never managed to configure this properly the first time. Your best bet is to copy the values from another application already configured in your organization.

Once LDAP is configured properly, you can map Splunk roles to LDAP groups through the admin interface. Whether to use an existing group or create Splunk-specific groups is of course up to your organization, but most companies I have worked with opted to create a specific group for each Splunk role. The common groups are often along the lines of: `splunkuser`, `splunkpoweruser`, `splunksecurity`, and `splunkadmin`. Rights are additive, so a user can be a member of as many groups as is appropriate.

New in Splunk 4.3 are the ability to use multiple LDAP servers at once, support for dynamic groups, support for nested groups, and more. The official documentation can be found at the following URL:

<http://docs.splunk.com/Documentation/Splunk/latest/Security/SetupUserAuthenticationWithLDAP>

Using Single Sign On

Single Sign On (SSO) lets you use some other web server to handle authentication for Splunk. For this to work, several assumptions are made, as follows:

- Your SSO system can act as an HTTP forwarding proxy, sending HTTP requests through to Splunk.
- Your SSO system can place the authenticated user's ID into an HTTP header.
- The IP of your server(s) forwarding requests is static.
- When given a particular username, Splunk will be able to determine what roles this user is a part of. This is usually accomplished using LDAP but could be accomplished by defining users directly through the Splunk UI or via a custom scripted authentication plugin.

Assuming all of these are true, the usual approach is to follow these steps:

1. Configure LDAP authentication in Splunk.
2. Configure your web server to send proxy requests through to Splunk: When this is configured properly, you should be able to use Splunk as if you were accessing the Splunk web application directly.
3. Configure your web server to authenticate: With this configured, your web server should ask for authentication, and you should still be asked for authentication by Splunk.
4. Look for the HTTP header containing the remote user: Proxying through your web server, change the URL to `http://yourproxyserver/debug/sso`. You should see your username under **Remote user HTTP header** or **Other HTTP headers**.
5. Configure SSO in `/$SPLUNK_HOME/etc/system/local/web.conf`: You need to add three attributes to the `[settings]` stanza, as shown in the following code:

```
[settings]
SSOMode = strict
remoteUser = REMOTE-USER
trustedIP = 192.168.1.1,192.168.1.2
```


That should be it. The hardest part is usually convincing the web server to both authenticate and proxy. Use the `/debug/sso` page to help diagnose what is happening.

There can also be issues with punctuation in the header fieldname. If it's possible, removing any punctuation in the header name may eliminate unexpected problems.

Load balancers and Splunk

Some organizations that have invested heavily in load balancers like to use them whenever possible to centralize network management. There are three services Splunk typically exposes, mentioned in the following sections:

web

Usually on port 8000, the Splunk web server can be load balanced when configured with **search head pooling**. The load balancer should be configured to be "sticky", as the web server will still rely on user sessions tied to the web server the user started on.

See the *Multiple search heads* section for more information.

splunktcp

Usually on port 9997, `splunktcp` is itself stateless. Splunk auto load balancing is very well tested and very efficient but does not support complicated logic. For instance, you could use a load balancer to prefer connections to indexers in the same data center, only using indexers in another data center as a last resort.

The problem is that when only one address is provided to a Splunk forwarder, the forwarder will open one connection and keep it open indefinitely. This means that when an indexer is restarted, it will never receive a connection until forwarders are restarted.

The easy solution is to expose two addresses on your load balancer and list both of these addresses in `outputs.conf`. The two addresses must be either two different ports or two different IP addresses. Two different CNAMEs on the same port will not work, as Splunk resolves the addresses and collapses the list of IP addresses.

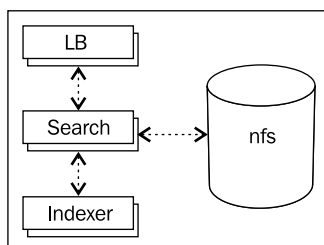
deployment server

Usually on port 8089, the deployment server listens using SSL, by default, with a self-signed certificate. There are a couple of problems with using a load balancer with the deployment server; they are as follows:

- The protocol is essentially REST over HTTP, but not quite. Use a TCP load balancer, not a load balancer that understands HTTP.
- While it is theoretically possible to load balance deployment servers, the issue is that, if the different deployment servers are out of sync, deployment clients may "flap", loading one set of apps and then the other. A better approach is probably running multiple deployment servers and using DNS or load balancers to ensure that certain sets of hosts always talk to a particular server.

Multiple search heads

Using the **search head pooling** feature, it is possible to run multiple **search head** instances. The feature requires a share of some sort behind the servers acting as search heads, which effectively means they must be in the same data center. The setup looks essentially like the following figure:



In short, the steps to configure the search are as follows:

1. Mount the NFS volume on each search head.
2. Enable the pooling feature on each instance.
3. Copy the existing configurations to the NFS volume.
4. Test the search heads.
5. Enable the load balancer.

The official documentation is available at <http://docs.splunk.com/Documentation/Splunk/latest/Deploy/Configuresearchheadpooling>.

Summary

We have touched upon a wide variety of subjects in this chapter, each of which possibly deserves a chapter of its own. Maybe that will be the next book.

We talked about the different purposes of Splunk instances, how to collect data from a variety of sources, how to install the Splunk binary, how to size your indexers, and how to manage the configuration of many instances, and finally, we touched upon a few advanced deployment topics.

In our final chapter, we will write some code to extend Splunk in a variety of ways.

12

Extending Splunk

While the core of Splunk is closed, there are a number of places where you can use scripts or external code to extend the default behaviors. In this chapter, we will write a number of examples, covering most of the places where external code can be added. Most code samples are written in Python, so if you are not familiar with Python, a reference may be useful.

We will cover:

- Writing scripts to create events
- Using Splunk from the command line
- Calling Splunk via REST
- Writing custom search commands
- Writing event type renderers
- Writing custom search action scripts

The examples used in this chapter are included in the app `ImplementingSplunkExtendingExamples`, which can be downloaded from the support page of the Packt Publishing website (www.packtpub.com/support).

Writing a scripted input to gather data

Scripted inputs allow you to run some piece of code on a scheduled basis, and capture the output as if it were simply being written to a file. It does not matter what language the script is written in, or where it lives, as long it is executable. We touched on this topic in the *Using scripts to gather data* section in *Chapter 11, Advanced Deployments*. Let's write a few more examples.

Capturing script output with no date

One common problem with script output is the lack of a predictable date or date format. In this situation, the easiest thing to do is to tell Splunk to not try to parse a date at all, and instead use the current date instead. Let's make a script that lists open network connections:

```
from subprocess import Popen
from subprocess import PIPE
from collections import defaultdict
import re

def add_to_key(fieldname, fields):
    return " " + fieldname + "+" + fields[fieldname]

output = Popen("netstat -n -p tcp", stdout=PIPE,
               shell=True).stdout.read()

counts = defaultdict(int)
for l in output.splitlines():
    if "ESTABLISHED" in l:
        pattern = r"(?P<protocol>\S+)\s+\d+\s+\d+\s+"
        pattern += r"(?P<local_addr>.*?)[^\d](?P<local_port>\d+)\s+"
        pattern += r"(?P<remote_addr>.*?)[^\d](?P<remote_port>\d+)"
        m = re.match(pattern, l)
        fields = m.groupdict()

        if "local_port" in fields and "remote_port" in fields:
            if fields["local_addr"] == fields["remote_addr"]:
                continue
            try:
                if int(fields["local_port"]) < 1024:
                    key = "type=incoming"
                    key += add_to_key("local_addr", fields)
                    key += add_to_key("local_port", fields)
                    key += add_to_key("remote_addr", fields)
                else:
                    key = "type=outgoing"
                    key += add_to_key("remote_addr", fields)
                    key += add_to_key("remote_port", fields)
                    key += add_to_key("local_addr", fields)
            except:
                print "Unexpected error:", sys.exc_info()[0]

        counts[key] += 1

for k, v in sorted(counts.items()):
    print k + " count=" + str(v)
```

Before we wire this up, we can test the command using the Python interpreter included with Splunk as follows:

```
$SPLUNK_HOME/bin/splunk cmd python connections.py
```



If you are using any Splunk Python modules, you must use Python included with Splunk, as other Python installations will not find these modules.

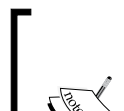
On my machine, this produces:

```
type=outgoing remote_addr=17.149.36.120 remote_port=5223
  local_addr=192.168.0.20 count=1
type=outgoing remote_addr=17.158.10.104 remote_port=443
  local_addr=192.168.0.20 count=2
type=outgoing remote_addr=17.158.10.42 remote_port=443
  local_addr=192.168.0.20 count=5
type=outgoing remote_addr=17.158.8.23 remote_port=993
  local_addr=192.168.0.20 count=4
type=outgoing remote_addr=173.194.64.109 remote_port=993
  local_addr=192.168.0.20 count=8
type=outgoing remote_addr=199.47.216.173 remote_port=443
  local_addr=192.168.0.20 count=1
type=outgoing remote_addr=199.47.217.178 remote_port=443
  local_addr=192.168.0.20 count=1
type=outgoing remote_addr=50.18.31.239 remote_port=443
  local_addr=192.168.0.20 count=1
```

Now that we have a working script, we need two pieces of configuration, namely `inputs.conf` and `props.conf`. As we covered in *Chapter 11, Advanced Deployments*, you will want to place these configurations in different apps if you are going to distribute this input across a distributed environment.

`inputs.conf` should contain something like the following code:

```
[script://./bin/connections.py]
interval=60
sourcetype=connections
```



If the script ends in `.py`, Splunk will automatically use the included Python interpreter. Otherwise, the script needs to be executable via the command line.

If you want to use a different Python executable, you will need to specify the full path to Python as the script, and the script itself as an argument.

props.conf should then contain something as follows:

```
[connections]
SHOULD_LINEMERGE = false
DATETIME_CONFIG = CURRENT
```

This configuration requires each line to be treated as an event and to not even try to find something that looks like a date in this event.

Let's build a query using the output of this scripted input. A useful query might be ports open by domain name. This query uses dnslookup and then flattens remote_host to either a domain name or subnet:

```
index=implsplunk sourcetype=connections
| fillnull value="-" remote_addr remote_port local_addr local_port
| dedup remote_addr remote_port local_addr local_port
| lookup dnslookup clientip as remote_addr
| rex field=clienthost ".*\.(?<domain>[^\.]+\.[^\.]*)"
| eval remote_host=coalesce(domain,remote_addr)
| eval remote_host=replace(remote_host,"(.*)\.\d+$","\\1.0")
| stats sum(count) as count values(remote_port) as remote_ports
  by remote_host local_addr local_port
| eval remote_ports=mvjoin(remote_ports, ", ")
```

On my laptop, I get the following results:

remote_host ↕	local_addr ↕	local_port ↕	count ↕	remote_ports ↕
138.108.7.0	172.16.14.25	-	2	80
172.16.14.0	172.16.14.25	-	8	55686, 55692, 55696, 61384, 61787, 61788, 61809, 62078
198.171.79.0	172.16.14.25	-	1	80
1e100.net	172.16.14.25	-	34	443, 80
206.33.35.0	172.16.14.25	-	1	1935
208.85.243.0	172.16.14.25	-	2	443, 80
209.170.117.0	172.16.14.25	-	3	80
akamaitechnologies.com	172.16.14.25	-	5	80
amazonaws.com	172.16.14.25	-	6	443
apple.com	172.16.14.25	-	1	5223

Capturing script output as a single event

When you want to capture the entire output of a script as a single event, the trick is to specify an impossible value for LINE_BREAKER. Let's write a shell script to output the different parts of uname with nice field names.

You can find the following script at

ImplementingSplunkExtendingExamples/bin/uname.sh:

```
#!/bin/sh

date "+%Y-%m-%d %H:%M:%S"
echo hardware="\$(uname -m)\ "
echo node="\$(uname -n)\ "
echo proc="\$(uname -p)\ "
echo os_release="\$(uname -r)\ "
echo os_name="\$(uname -s)\ "
echo os_version="\$(uname -v)\ "
```

This script produces output like the following code:

```
2012-10-30 19:28:05
hardware="x86_64"
node="mymachine.local"
proc="i386"
os_release="12.2.0"
os_name="Darwin"
os_version="Darwin Kernel Version 12.2.0: Sat Aug 25 00:48:52 PDT
2012; root:xnu-2050.18.24-1/RELEASE_X86_64"
```

You may notice that the last line definitely contains a date. Unless we specifically tell Splunk that the entire output is an event in one way or another, it will turn that last line into an event.

inputs.conf should contain something as follows:

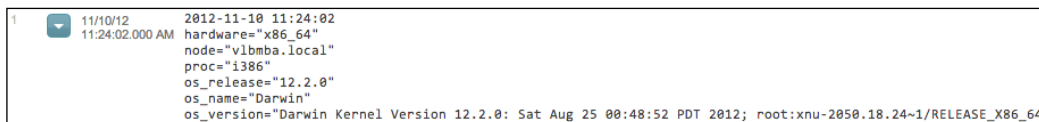
```
[script://./bin/uname.sh]
interval = 0 0 * * *
sourcetype=uname
```

Notice the cron syntax for interval. This will run the script each day at midnight. An alternative would be to set the value to 86400, which would run the script each time Splunk starts, and then every 24 hours thereafter.

props.conf should then contain something like the following:

```
[uname]
TIME_FORMAT = %Y-%m-%d %H:%M:%S
#treat each "line" as an event:
SHOULD_LINEMERGE = false
#redefine the beginning of a line to an impossible match,
#thus treating all data as one "line":
LINE_BREAKER = ((?!))
#chop the "line" at one megabyte, just in case:
TRUNCATE=1048576
```


Once installed, you can search for these events using `sourcetype=uname`, which produces output similar to the following screenshot:



```
1 11/10/12 2012-11-10 11:24:02
11:24:02.000 AM hardware="x86_64"
node="vibmba.local"
proc="i386"
os_release="12.2.0"
os_name="Darwin"
os_version="Darwin Kernel Version 12.2.0: Sat Aug 25 00:48:52 PDT 2012; root:xnu-2050.18.24~1/RELEASE_X86_64"
```

Because we used the `fieldname="fieldvalue"` syntax and we quoted values with spaces and strange characters, these field values will be automatically extracted. We can then use these fields immediately for reporting. A useful query might be:

```
earliest=-24h sourcetype=uname
| eventstats count by os_release os_name
| search count<10
```

This query would find the rare `os_release os_name` combinations.

Making a long-running scripted input

Sometimes a process needs to be long running, for instance, if it is polling some external source, like a database. A simple example might be:

```
import time
import random
import sys

for i in range(1, 1000):
    print "%s Hello." % time.strftime('%Y-%m-%dT%H:%M:%S')
    #make sure python actually sends the output
    sys.stdout.flush()
    time.sleep(random.randint(1, 5))
```

This script will run for somewhere between 1,000 and 5,000 seconds and then exit. Since this is a long-running script, our choices are either to treat each line as an event as we did in the *Capturing script output with no date* section, or, if we know there is a date to use, configure the input like a regular log file. In this case, we can see that there is always a date, so we will rely on that. The output is, unsurprisingly, as follows:

```
2012-10-30T20:13:29 Hello.
2012-10-30T20:13:33 Hello.
2012-10-30T20:13:36 Hello.
```

`inputs.conf` should contain something similar to the following:

```
[script://./bin/long_running.py]
interval = 1
sourcetype=long_running
```

With `interval = 1`, Splunk will try to launch the script every second, but will only run one copy of the script at a time.

`props.conf` should then contain something like:

```
[long_running]
TIME_FORMAT = %Y-%m-%dT%H:%M:%S
MAX_TIMESTAMP_LOOKAHEAD = 21
BREAK_ONLY_BEFORE = ^\d{4}-\d{1,2}-\d{1,2}T\d{1,2}:
```

This will create a long-running process that can do whatever is appropriate.



Though it is convenient to have Splunk execute scripts for you and capture the output, if the information you are capturing is vital, it may be safer to simply schedule the script with cron, direct its output to a file, and point Splunk at that file. This allows you to use the file in other ways; you can capture both standard output and errors, and the data will still be captured if Splunk is down. It, however, has the disadvantage that you have to clean up those logs yourself.

Using Splunk from the command line

Almost everything that can be done via the web interface can also be accomplished via the command line. For an overview, see the output of `/opt/splunk/bin/splunk help`. For help on a specific command, use `/opt/splunk/bin/splunk help [commandname]`.

The most common action to perform on the command line is search. For example, have a look at the following code:

```
$ /opt/splunk/bin/splunk search 'foo'
2012-08-25T20:17:54 user=user2 GET /foo?q=7148356 uid=MzA4MTc5OA
2012-08-25T20:17:54 user=user2 GET /foo?q=7148356 uid=MzA4MTc5OA
2012-08-25T20:17:54 user=user2 GET /foo?q=7148356 uid=MzA4MTc5OA
...
```

Things to note:

- By default, searches are performed over All time. Protect yourself by including `earliest=-1d` or an appropriate time range in your query.
- By default, Splunk will only output 100 lines of results. If you need more, use the `-maxout` flag.
- Search requires authentication, so the user will be asked to authenticate unless `-auth` is included as an argument.

Most use cases for the command line involve counting events for outputting to other systems. Let's try a simple `stats` call to count instances of the word `error` over the last hour by host:

```
$ /opt/splunk/bin/splunk search 'earliest=-1h error | stats count by host'
```

This produces:

```
      host      count
-----  -
host2          3114
vlb.local      3063
```

Things to notice in this case are:

- `earliest=-1h` is included to limit the query to the last hour.
- By default, the output is in a `table` format. This is nicer to read, but much harder to parse in another scripting language. Use `-output` to control the output format.
- By default, Splunk will render a preview of the results as results are retrieved. This slows down the overall execution. Disable preview with `-preview false`. Previews are not calculated when the script is not being called from an interactive terminal, for instance, when run from cron.

To retrieve the output as CSV, try the following code:

```
$ /opt/splunk/bin/splunk search 'earliest=-1h error | stats count by host' -output csv -preview false
```

This gives us the following output:

```
count,host
3120,host2
3078,"vlb.local"
```

Note that if there are no results, the output will be empty.

Querying Splunk via REST

Splunk provides an extensive HTTP REST interface, which allows searching, adding data, adding inputs, managing users, and more. Documentation and SDKs are provided by Splunk at <http://dev.splunk.com/>.

To get an idea of how this REST interaction happens, let's step through a sample conversation to run a query and retrieve the results. The steps are essentially as follows:

1. Start the query (POST).
2. Poll for status (GET).
3. Retrieve results (GET).

We will use the command line program **cURL** to illustrate these steps. The SDKs make this interaction much simpler.

To start a query, the command is as follows:

```
curl -u user:pass -k https://yourserver:8089/services/search/jobs
-d"search=search query"
```

This essentially says to POST `search=search query`. If you are familiar with HTTP, you might notice that this is a standard POST from an HTML form.

To run the query `earliest=-1h index="_internal" warn | stats count by host`, we need to URL encode the query. The command then is as follows:

```
$ curl -u admin:changeme -k https://localhost:8089/services/search/
jobs -d"search=search%20earliest%3D-1h%20index%3D%22_internal%22%20
warn%20%7C%20stats%20count%20by%20host"
```

If the query is accepted, we will receive XML that contains our search ID:

```
<?xml version='1.0' encoding='UTF-8'?>
<response><sid>1352061658.136</sid></response>
```

The contents of `<sid>` are then used to reference this job. To check the status of our job, we run the following code:

```
curl -u admin:changeme -k https://localhost:8089/services/search/
jobs/1352061658.136
```

This returns a large document with copious amounts of information about our job as follows:

```
<entry ...>
  <title>search earliest=-1h index="_internal" warn | stats count by
host</title>
```

```
<id>https://localhost:8089/services/search/jobs/1352061658.136</id>
...
<link href="/services/search/jobs/1352061658.136/events"
rel="events"/>
<link href="/services/search/jobs/1352061658.136/results"
rel="results"/>
...
<content type="text/xml">
  <s:dict>
...
    <s:key name="doneProgress">1.00000</s:key>
...
    <s:key name="eventCount">67</s:key>
...
    <s:key name="isDone">1</s:key>
...
    <s:key name="resultCount">1</s:key>
```

Interesting fields include `doneProgress`, `eventCount`, `resultCount`, and the field we are most interested in at this point, `isDone`. If `isDone` is not 1, we should wait and poll again later. Once `isDone=1`, we can retrieve our results from the URL specified in `<link rel="results">`.

To retrieve our results, we call the following:

```
curl -u admin:changeme -k https://localhost:8089/services/search/
jobs/1352061658.136/results
```

This returns the following XML output:

```
<?xml version='1.0' encoding='UTF-8'?>
<results preview='0'>
  <meta>
    <fieldOrder>
      <field>host</field>
      <field>count</field>
    </fieldOrder>
  </meta>
  <result offset='0'>
    <field k='host'>
      <value><text>vlb.local</text></value>
    </field>
    <field k='count'>
      <value><text>67</text></value>
    </field>
  </result>
</results>
```

The list of fields is contained in `meta/fieldOrder`. Each result will then follow this field order.

Though not necessary (since jobs expire on their own) we can save disk space on our Splunk servers by cleaning up after ourselves. Simply calling the `DELETE` method on the job URL will delete the results and reclaim the used disk space.

```
curl -u admin:changeme -k -X DELETE https://localhost:8089/services/search/jobs/1352061658.136
```

Just to show the Python API action, here's a simple script:

```
import splunk.search as search
import splunk.auth as auth
import sys
import time

username = sys.argv[1]
password = sys.argv[2]
q = sys.argv[3]

sk = auth.getSessionKey(username, password)

job = search.dispatch("search " + q, sessionKey=sk)

while not job.isDone:
    print "Job is still running."
    time.sleep(.5)

for r in job.results:
    for f in r.keys():
        print "%s=%s" % (f, r[f])
    print "-----"

job.cancel()
```

This script uses the Python modules included with Splunk, so we must run it using Splunk's included Python as follows:

```
$ /opt/splunk/bin/splunk cmd python simplesearch.py admin changeme
'earliest=-7d index="_internal" warn | timechart count by source'
```

This produces output as follows:

```
_time=2012-10-31T00:00:00-0500
/opt/splunk/var/log/splunk/btool.log=0
/opt/splunk/var/log/splunk/searches.log=0
```

```
/opt/splunk/var/log/splunk/splunkd.log=31
/opt/splunk/var/log/splunk/web_service.log=0
_span=86400
_spandays=1
-----
_time=2012-11-01T00:00:00-0500
/opt/splunk/var/log/splunk/btool.log=56
/opt/splunk/var/log/splunk/searches.log=0
/opt/splunk/var/log/splunk/splunkd.log=87
/opt/splunk/var/log/splunk/web_service.log=2
_span=86400
_spandays=1
-----
...
```

For more examples and extensive documentation, check out <http://dev.splunk.com>.

Writing commands

To augment the built-in commands, Splunk provides the ability to write commands in Python and Perl. You can write the commands to modify events, replace events, or even dynamically produce events.

When not to write a command

While external commands can be very useful, if the number of events to be processed is large, or if performance is a concern, it should be considered a last resort. You should make every effort to accomplish the task at hand using the search language built into Splunk, or other built-in features. For instance, if you need:

- Regular expressions—learn to use `rex`, `regex`, and extracted fields
- To calculate a new field, or modify an existing field—look into `eval` (search for `splunk eval` functions with your favorite search engine)
- To augment your results with external data—learn to use **lookups**, which can also be a script, if need be
- To read external data that changes periodically—consider using `inputcsv`

The performance issues introduced by external commands come from the following two places:

- The work involved with launching a Python process, exporting events as CSV to the Python process, and then importing the results back into the Splunk process.
- The actual code of the command. A command that queries some external data source, for instance a database, will be affected by the speed of that external source.

In my testing, I could not make a command run faster than the speed that is 50 percent slower than native commands. To test this, let's try a couple of searches as follows:

```
* | head 100000 | eval t=_time+1 | stats dc(t)
```

On my laptop, this query takes roughly four seconds to execute, when run on the command line with preview disabled, as shown in the following code:

```
# time /opt/splunk/bin/splunk search '* | head 100000 | eval t=_time+1 | stats dc(t)' -preview false
```

Now let's throw in a command included in our sample app:

```
* | head 100000 | echo | eval t=_time+1 | stats dc(t)
```

This increases the search time to slightly over six seconds, an increase of 50 percent. Included in the sample app are three variations on the echo app of varying complexity:

- `echo`: This command simply echoes the standard input to standard output.
- `echo_csv`: This command uses `csvreader` and `csvwriter`.
- `echo_splunk`: This command uses the Python modules provided with Splunk to gather the incoming events and then output the results. We will use these Python modules for our example commands.

Using each of these commands, the times are nearly identical, which tells me most of the time is spent shuttling the events in and out of Splunk.



Adding `required_fields=_time` in `commands.conf` lowered times from 2.5x to 1.5x in this case. If you know the fields your command needs, this setting can dramatically increase performance.

When to write a command

Given the warning about performance, there are still times it will make sense to write a command. I can think of a few reasons:

- You need to perform a specific action that cannot be accomplished using internal commands
- You need to talk to an external system (though a lookup may be more efficient)
- You need to produce "events" out of thin air, perhaps from an external service or for testing

I'm sure you can think of your own reasons. Let's explore the nuts and bolts of different types of commands.

Configuring commands

Before we start writing commands, there is some setup that must be done for all commands. First, every command will need an entry in the `commands.conf` file of your app. Let's take a look at the following sample stanza:

```
[commandname]
filename = scriptname.py
streaming = false
enableheader = true
run_in_preview = true
local = false
retainsevents = false
```

Stepping through the following attributes:

- `[commandname]`: The command available to search will be the title of the stanza, in this case `commandname`.
- `filename = scriptname.py`: The script to run. It must live in the directory `bin` inside your app.
- `streaming = false`: By default, only one instance of each command will be run on the complete set of results. The assumption is that all events are needed for the script to do its work. If your script works on each event individually, set this value to `true`. This will eliminate the event limit, which by default is 50,000, as specified by `maxresultrows` in `limits.conf`.

- `enableheader = true`: By default, your script will receive a header that the Splunk Python modules know how to use. If this is set to `false`, your command will receive plain CSV.
- `run_in_preview = true`: By default, your command will be executed repeatedly while events are being retrieved, so as to update the preview in the GUI. This will have no effect on saved searches, but setting this to `false` can make a big difference in performance for interactive searches. This is particularly important if your command uses an external resource, as it will be called repeatedly.
- `local = false`: If you have a distributed environment, by default, your command will be copied to all indexers and executed there. If your command needs to be run on one machine, setting `local=true` will ensure the command only runs on the search head.
- `retainsevents = false`: By default, Splunk assumes that your command returns the transformed events, much like `stats` or `timechart`. Setting this to `true` will change the behavior to treat the results as regular events.

To make our commands available to other apps, for instance **Search**, we need to change the metadata in our app. Place the following two lines in the file `metadata/default.meta`:

```
[commands]
export = system
```

Finally, to use a newly configured command, we either need to restart Splunk or load the URL `http://yourserver/debug/refresh` in a browser. This may also be necessary after changing settings in `commands.conf`, but is not necessary after making changes to the script itself.

Adding fields

Let's start out with a simple command that does nothing more than add a field to each event. This example is stored in `ImplementingSplunkExtendingExamples/bin/addfield.py`:

```
#import the python module provided with Splunk
import splunk.Intersplunk as si

#read the results into a variable
results, dummyresults, settings = si.getOrganizedResults()
```

```
#loop over each result. results is a list of dict.
for r in results:
    #r is a dict. Access fields using the fieldname.
    r['foo'] = 'bar'

#return the results back to Splunk
si.outputResults(results)
```

Our corresponding stanza in `commands.conf` is as follows:

```
[addfield]
filename = addfield.py
streaming = true
retainsevents = true
```

We can use this command as follows:

```
* | head 10 | addfield | top foo
```

This gives us the result shown in the following screenshot:

	foo ↕	count ↕	percent ↕
1	bar	10	100.000000

This could be accomplished much more efficiently by simply using `eval foo="bar"`, but this illustrates the basic structure of a command.

Manipulating data

It is useful at times to modify the value of a field, particularly `_raw`. Just for fun, let's reverse the text of each event. We will also support a parameter that specifies whether to reverse the words or the entire value. You can find this example in `ImplementingSplunkExtendingExamples/bin/reverseraw.py`:

```
import splunk.Intersplunk as si
import re

#since we're not writing a proper class, functions need to be
#defined first
def reverse(s):
    return s[::-1]

#start the actual script
```

```

results, dummyresults, settings = si.getOrganizedResults()

#retrieve any options included with the command
keywords, options = si.getKeywordsAndOptions()

#get the value of words, defaulting to false
words = options.get('words', False)

#validate the value of words
if words and words.lower().strip() in ['t', 'true', '1', 'yes']:
    words = True
else:
    words = False

#loop over the results
for r in results:
    #if the words option is true, then reverse each word
    if words:
        newRaw = []
        parts = re.split('([a-zA-Z\']+)', r['_raw'])
        for n in range(0, len(parts) - 2, 2):
            newRaw.append(reverse(parts[n]))
            newRaw.append(parts[n + 1])
        newRaw.append(reverse(parts[-1]))
        r['_raw'] = ''.join(newRaw)
    #otherwise simply reverse the entire value of _raw
    else:
        r['_raw'] = reverse(r['_raw'])

si.outputResults(results)

```

The `commands.conf` stanza would look as follows:

```

[reverseraw]
filename = reverseraw.py
retainsevents = true
streaming = true

```

Let us assume the following event:

```

2012-10-27T22:10:21.616+0000 DEBUG Don't worry, be happy. [user=linda,
ip=1.2.3., req_time=843, user=extrauser]

```

Using our new command:

```

* | head 10 | reverseraw

```

Running the previous command on the preceding event, we see the entire event reversed, as shown in the following code:

```
]resuartxe=resu ,348=emit_qer ,.3.2.1=pi ,adnil=resu[ .yppah eb ,yrow  
t'noD GUBED 0000+616.12:01:22T72-01-2102
```

We can then add the words argument:

```
* | head 10 | reverseraw words=true
```

We maintain the order of the words, as shown in the following code:

```
2012-10-27T22:10:21.616+0000 GUBED t'noD yrow, eb yppah. [resu=adnil,  
pi=1.2.3., qer_emit=843, resu=resuartxe]
```

For fun, let's reverse the event again:

```
* | head 10 | reverseraw words=true | reverseraw
```

This gives us the following output:

```
]extrauser=user ,348=time_req ,.3.2.1=ip ,linda=user[ .happy be ,worry  
Don't DEBUG 0000+616.12:01:22T72-01-2102
```

happy be, worry Don't—Yoda could not have said it better.

Transforming data

So far, our commands have returned the original events with modifications to their fields. Commands can also transform data, much like the built-in functions `top` and `stats`. Let's write a function to count the words in our events. You can find this example in `ImplementingSplunkExtendingExamples/bin/countwords.py`:

```
import splunk.Intersplunk as si  
import re  
import operator  
from collections import defaultdict  
  
#create a class that does the actual work  
class WordCounter:  
    word_counts = defaultdict(int)  
    unique_word_counts = defaultdict(int)  
    rowcount = 0  
    casesensitive = False  
    mincount = 50  
    minwordlength = 3  
  
    def process_event(self, input):
```

```
self.rowcount += 1
words_in_event = re.findall('\W*([a-zA-Z]+)\W*', input)

unique_words_in_event = set()
for word in words_in_event:
    if len(word) < self.minwordlength:
        continue # skip this word, it's too short
    if not self.casesensitive:
        word = word.lower()
    self.word_counts[word] += 1
    unique_words_in_event.add(word)

for word in unique_words_in_event:
    self.unique_word_counts[word] += 1

def build_sorted_counts(self):
    #create an array of tuples,
    #ordered by the count for each word
    sorted_counts = sorted(self.word_counts.iteritems(),
                           key=operator.itemgetter(1))

    #reverse it
    sorted_counts.reverse()

    return sorted_counts

def build_rows(self):
    #build our results, which must be a list of dict
    count_rows = []
    for word, count in self.build_sorted_counts():
        if self.mincount < 1 or count >= self.mincount:
            unique = self.unique_word_counts.get(word, 0)
            percent = round(100.0 * unique / self.rowcount, 2)
            newrow = {'word': word,
                      'count': str(count),
                      'Events with word': str(unique),
                      'Event count': str(self.rowcount),
                      'Percent of events with word':
                        str(percent)}
            count_rows.append(newrow)
    return count_rows

#a helper method that doesn't really belong in the class
#return an integer from an option, or raise useful Exception
```

```
def getInt(options, field, default):
    try:
        return int(options.get(field, default))
    except Exception, e:
        #raise a user friendly exception
        raise Exception("%s must be an integer" % field)

#our main method, which reads the options, creates a WordCounter
#instance, and loops over the results
if __name__ == '__main__':
    try:
        #get our results
        results, dummyresults, settings = si.getOrganizedResults()
        keywords, options = si.getKeywordsAndOptions()

        word_counter = WordCounter()

        word_counter.mincount = getInt(options, 'mincount', 50)
        word_counter.minwordlength = getInt(options,
                                             'minwordlength', 3)

        #determine whether we should be case sensitive
        casesensitive = options.get('casesensitive', False)
        if casesensitive:
            casesensitive = (casesensitive.lower().strip() in
                             ['t', 'true', '1', 'y', 'yes'])
        word_counter.casesensitive = casesensitive

        #loop through the original results
        for r in results:
            word_counter.process_event(r['_raw'])

        output = word_counter.build_rows()
        si.outputResults(output)

    #catch the exception and show the error to the user
    except Exception, e:
        import traceback
        stack = traceback.format_exc()
        si.generateErrorResults("Error '%s'. %s" % (e, stack))
```

This is a larger script, but hopefully it is clear what is happening. Notice in this example a few new things:

- Most of the logic is in the class definition. This provides a better separation of Splunk-specific logic and business logic.
- Testing for `__main__`, as is the Python way.
- Exception handling.
- A nicer exception for failed parsing of integer arguments.
- Field names with spaces in them.

Our entry in `commands.conf` does not allow streaming, and does not retain events:

```
[countwords]
filename = countwords.py
retainsevents = false
streaming = false
```


We can then use our command as follows:

```
* | countwords
```

This will give us back a table, as shown in the following screenshot:

	count ↕	Events with word ↕	word ↕	Event count ↕	Percent of events with word ↕
1	49680	21822	error	50000	43.64
2	47860	41870	user	50000	83.74
3	40075	40070	time	50000	80.14
4	40065	40065	req	50000	80.13
5	39971	39971	logger	50000	79.94
6	25310	12655	this	50000	25.31
7	25243	25243	barclass	50000	50.49
8	25211	25211	don	50000	50.42
9	16131	16131	network	50000	32.26
10	16056	16056	session	50000	32.11
11	13442	13442	mary	50000	26.88
12	12655	12655	worthless	50000	25.31
13	12655	12655	nothing	50000	25.31
14	12655	12655	happened	50000	25.31
15	12655	12655	log	50000	25.31
16	12584	12584	debug	50000	25.17
17	12556	12556	worry	50000	25.11
18	12556	12556	happy	50000	25.11
19	12523	12523	warn	50000	25.05
20	12426	12426	info	50000	24.85
21	12378	12378	hello	50000	24.76
22	12378	12378	world	50000	24.76
23	8106	8106	red	50000	16.21

With my test data, this produced 132 rows, representing 132 unique words at least 3 characters long in my not-so-random data set. **count** represents how many times each word occurred overall, while **Events with word** represents how many events contained the word at all.

 Notice the value **50000** in the **Event count** column. Even though my query found more than 300,000 events, only 50,000 make their way to the command. You can increase this limit by increasing `maxresultrows` in `limits.conf`, but be careful! This limit is for your protection.

Trying out our options as follows:

```
* | head 1000
  | countwords casesensitive=true mincount=250 minwordlength=0
```

This query produces the following output:

	count ↕	Events with word ↕	word ↕	Event count ↕	Percent of events with word ↕
1	1000	1000	T	1000	100.0
2	968	837	user	1000	83.7
3	801	801	logger	1000	80.1
4	799	799	ip	1000	79.9
5	798	798	time	1000	79.8
6	798	798	req	1000	79.8
7	531	459	ERROR	1000	45.9
8	490	490	BarClass	1000	49.0
9	473	473	Don	1000	47.3
10	473	473	t	1000	47.3
11	330	330	network	1000	33.0
12	304	304	session	1000	30.4
13	282	282	mary	1000	28.2
14	271	271	INFO	1000	27.1
15	268	268	error	1000	26.8
16	268	268	Error	1000	26.8
17	259	259	Hello	1000	25.9
18	259	259	world	1000	25.9
19	251	251	WARN	1000	25.1

Notice that we now see one- and two-letter words, have entries for both T and t, and our results stop when **count** drops below our value for `mincount`.

Just for completeness, to accomplish this command using built-in commands, you could do something like the following code:

```
* | rex max_match=1000 "\W*(?<word>[a-zA-Z]+\W*"
  | eval id=1 | accum id | fields word id
  | eventstats count
  | mvexpand word
  | eval word=lower(word)
  | stats max(count) as event_count
    dc(id) as events_with_word
    count as word_count
    by word
  | sort -events_with_word
  | eval percent_events_containing =
    round(events_with_word/event_count*100.0,2)
  | rename word_count as count
    events_with_word as "Events with word"
    event_count as "Event count"
    percent_events_containing as "Percent of events with word"
  | table count "Events with word" word
    "Event count" "Percent of events with word"
```

There is probably a more efficient way to do this work using built-in commands, but this is what comes to mind initially.

Generating data

There are times when you want to create events out of thin air. These events could come from a database query, a web service, or simply some code that generates data useful in a query. Just to illustrate the plumbing, we will make a random number generator. You can find this example in `ImplementingSplunkExtendingExamples/bin/random_generator.py`:

```
import splunk.Intersplunk as si
from random import randint

keywords, options = si.getKeywordsAndOptions()

def getInt(options, field, default):
    try:
        return int(options.get(field, default))
    except Exception, e:
        #raise a user friendly exception
        raise Exception("%s must be an integer" % field)

try:
    min = getInt(options, 'min', 0)
```

```
max = getInt(options, 'max', 1000000)
eventcount = getInt(options, 'eventcount', 100)

results = []
for r in range(0, eventcount):
    results.append({'r': randint(min, max)})

si.outputResults(results)

except Exception, e:
    import traceback
    stack = traceback.format_exc()
    si.generateErrorResults("Error '%s'. %s" % (e, stack))
```

The entry in `commands.conf` then is as follows:

```
[randomgenerator]
filename = random_generator.py
generating = true
```

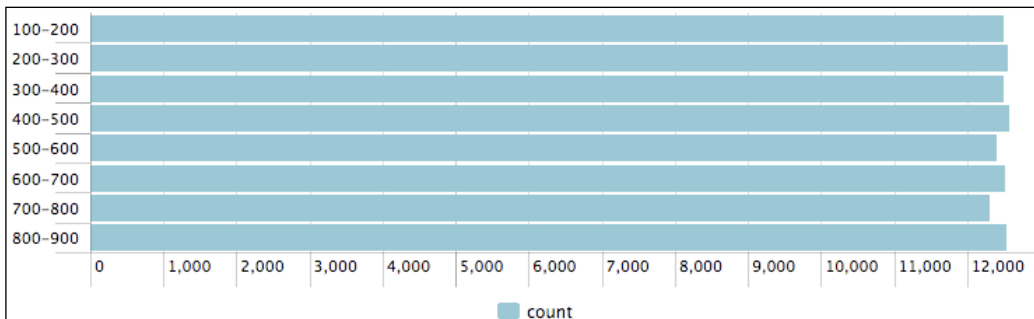
We can then use the command as follows:

```
|randomgenerator
```

Notice the leading pipe `|` symbol. This is the indication to run a command instead of running a search. Let's test the randomness of our Python:

```
|randomgenerator eventcount=100000 min=100 max=899
| bucket r
| chart count by r
```

This produces a graph, as shown in the following screenshot:



I guess that is not a bad distribution for 100,000 samples. Using Splunk's built-in commands, you could accomplish essentially the same thing using the following code:

```
index=_internal
| head 100000
| eval r=random()/2147483647*100000
| bucket r
| chart count by r
```

That is a very quick overview of commands, using fun demonstration commands to illustrate the plumbing required to execute your code. A number of samples ship with Splunk in `$SPLUNK_HOME/etc/apps/search/bin`.

Writing a scripted lookup to enrich data

We covered CSV lookups fairly extensively in *Chapter 6, Extending Search*, then touched on them again in *Chapter 9, Summary Indexes and CSV Files* and *Chapter 10, Configuring Splunk*. The capabilities built into Splunk are usually sufficient, but sometimes it is necessary to use an external data source or dynamic logic to calculate values. Scripted lookups have the following advantages over commands or CSV lookups:

- Scripted lookups are only run once per unique lookup value, as opposed to a command, which would run the command for every event
- The memory requirement of a CSV lookup increases with the size of the CSV file
- Rapidly changing values can be left in an external system and queried using the scripted lookup instead of being exported frequently

In the *Using a lookup with wildcards* section in *Chapter 9, Summary Indexes and CSV Files*, we essentially created a case statement through configuration. Let's implement that use case as a script, just to show how it would be done in Python. First, in `transforms.conf`, we need the following configuration:

```
[urllookup]
external_cmd = url_lookup.py
fields_list = url section call_count
```

The following are notes about this configuration:

- `fields_list` is the list of fields that will be sent to the script and the list of fields expected in the result
- `fields_list` must contain at least two fields or the script will fail silently

The script then looks as follows:

```
import sys
import re
from csv import DictReader
from csv import DictWriter

patterns = []

def add_pattern(pattern, section):
    patterns.append((re.compile(pattern), section))

add_pattern('^/about/.*', 'about')
add_pattern('^/contact/.*', 'contact')
add_pattern('^/.*/.*', 'unknown_non_root')
add_pattern('^/.*', 'root')
add_pattern('.*', 'nomatch')

# return a section for this url
def lookup(url):
    try:
        for (pattern, section) in patterns:
            if pattern.match(url):
                return section
        return ''
    except:
        return ''

#set up our reader
reader = DictReader(sys.stdin)
fields = reader.fieldnames

#set up our writer
writer = DictWriter(sys.stdout, fields)
writer.writeheader()
```

```

#start our output
call_count = 0
for row in reader:
    call_count = call_count + 1

    if len(row['url']):
        row['section'] = lookup(row['url'])
        row['call_count'] = call_count
        writer.writerow(row)

```

In a nutshell, this script takes the value of `url`, tries each regular expression in sequence, and then sets the value of `section` accordingly. A few points about the preceding script follow:

- The script receives the raw CSV with the fields listed in `transforms.conf`, but only the fields that are needed for lookup will have a value. In our case, that is `url`.
- The field `url` must be present in the data, or mapped in the `lookup` command using the `as` option.
- `call_count` is included to show that this scripted lookup is more efficient than an external command, as the lookup will only receive one line of input per unique value of `url`.

Let's try it out:

```

index=implsplunk sourcetype="impl_splunk_web"
| rex "\s[A-Z]+\s(?<url>.*?)\?"
| lookup urllookup url
| stats count values(call_count) by url section

```

This gives us the following results:

	url ↕	section ↕	count ↕	values(call_count) ↕
1	/about/	about	1443	1
2	/bar	root	1383	2
3	/contact/	contact	1389	3
4	/foo	root	1446	4
5	/products/	unknown_non_root	1364	5
6	/products/index.html	unknown_non_root	1389	6
7	/products/x/	unknown_non_root	2899	7
8	/products/y/	unknown_non_root	1430	8

The column **values(call_count)** tells us that our lookup script only received eight rows of input, one for each unique value of url. This is far better than 12,743 rows that an equivalent command would have received.

For more examples of scripted lookups, see `$(SPLUNK_HOME)/etc/system/bin/external_lookup.py` and the MAXMIND app available in Splunkbase.

Writing an event renderer

Event renderers give you the ability to make a specific template for a specific event type. To read more about creating event types, see *Chapter 6, Extending Search*.

Event renderers use **mako** templates (<http://www.makotemplates.org/>).

An event renderer is comprised of the following:

- A template stored at `$(SPLUNK_HOME)/etc/apps/[yourapp]/appserver/event_renderers/[template].html`
- A configuration entry in `event_renderers.conf`
- An optional event type definition in `eventtypes.conf`
- Optional CSS classes in `application.css`

Let's create a few small examples. All the files referenced are included in `$(SPLUNK_HOME)/etc/apps/ImplementingSplunkExtendingExamples`. These examples are not shared outside this app, so to see them in action, you will need to search from inside this app. Do this by pointing your browser at `http://[yourserver]/app/ImplementingSplunkExtendingExamples/flashtimeline`.

Using specific fields

If you know the names of the fields you want to display in your output, your template can be fairly simple. Let's look at the following event type `template_example`. The template is stored in `appserver/event_renderers/template_example.html`:

```
<%page args="job, event, request, options">
<ul class="template_example">
  <li>
    <b>time:</b>
    ${i18n.format_datetime_microseconds(event.get('_time', event.
time))}
  </li>
  <li>
    <b>ip:</b>
    ${event.get('ip', '')}
  </li>
</ul>
```

```

</li>
<li>
  <b>logger:</b>
  ${event.get('logger', '')}
</li>
<li>
  <b>message:</b>
  ${event.get('message', '')}
</li>
<li>
  <b>req_time:</b>
  ${event.get('req_time', '')}
</li>
<li>
  <b>session_id:</b>
  ${event.get('session_id', '')}
</li>
<li>
  <b>user:</b>
  ${event.get('user', '')}
</li>
<li>
  <b>_raw:</b>
  ${event.get('_raw', '')}
</li>
</ul>
</%page>

```

This template outputs a `` block for each event, with the specific fields we want displayed. To connect this template to a specific event type, we need the following entry in `default/event_renderers.conf`:

```

[template_example]
eventtype = template_example
template = template_example.html

```

Finally, if we want to format our output, we can use the following CSS in `appserver/static/application.css`:

```

ul.template_example {
  list-style-type: none;
}

ul.template_example > li {
  background-color: #dddddd;
  padding: 4px;
  margin: 1px;
}

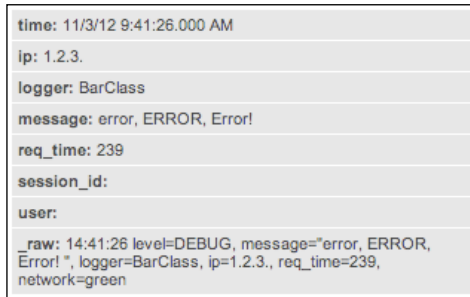
```


To test our event type renderer, we need the configuration to be loaded. You can accomplish this by restarting Splunk or by pointing your browser to `http://[yourserver]/debug/refresh`.

At this point, we can run a query and apply the event type manually:

```
index="implsplunk" sourcetype="template_example"
| eval eventtype="template_example"
```

This renders each event, as shown in the following screenshot:



To make this automatic, we can create an event type definition in `eventtypes.conf` as follows:

```
[template_example]
search = sourcetype=template_example
```

Now any query that finds events of `sourcetype=template_example` will be rendered using our template.

Table of fields based on field value

Since the template has access to everything in the event, you can use the fields in any way you like. The following example creates a horizontal table of fields, but lets the user specify a specific set of fields to display in a special field.

Our template, stored in `appserver/event_renderers/tabular.html`, looks as follows:

```
<%inherit file="//results/EventsViewer_default_renderer.html" />\
<%def name="event_raw(job, event, request, options, xslt)">\
  <%
  import sys
  _fields = str(event.fields.get('tabular', 'host,source,sourcetype,line
count')).split(',')
  </%>
```

```

head = ''
row = ''
for f in _fields:
    head += "<th>" + f + "</th>"
    row += "<td>" + str(event.fields.get(f, '-')) + "</td>"
%>
<table class="tabular_eventtype">
    <tr>
        ${head}
    </tr>
    <tr>
        ${row}
    </tr>
</table>
</%def>

```

Notice that we have extended the default event type renderer template, which means we will only change the rendering of the field `_raw`.

The entry in `event_renderers.conf` is as follows:

```

[tabular]
eventtype = tabular
template = tabular.html

```

Finally, our entries in `application.css` are as follows:

```

th.tabular_eventtype {
    background-color: #dddddd;
    border: 1px solid white;
    padding: 4px;
}

td.tabular_eventtype {
    background-color: #eeeeee;
    border: 1px solid white;
    padding: 4px;
}

```

We are not going to bother giving this event type a definition, but we can use it by setting the value of `eventtype` in the query. Let's try it out by running the following query:

```

index="implsplunk" | eval eventtype="tabular"

```

We see the following output, based on the default fields specified in the template:

1	11/3/12 9:41:26.000 AM	host	source	sourcetype	linecount
		vlbmba.local	/opt/splunk/etc/apps/ImplementingSplunkExtendingExamples/sample_data/template_example.log	template_example	1
host=vlbmba.local sourcetype=template_example					

2	11/3/12 9:41:26.000 AM	host	source	sourcetype	linecount
		vlbmba.local	/opt/splunk/etc/apps/ImplementingSplunkExtendingExamples/sample_data/template_example.log	template_example	1
host=vlbmba.local sourcetype=template_example					

Notice that we still see the event number, the workflow actions menu, local time as rendered by Splunk, and the selected fields underneath our template output. We have really only overridden the rendering of `_raw`.

If we specify the fields we want in our table in the field `tabular`, the template will honor what we specify in our table:

```
index="implsplunk" sourcetype="template_example"
| eval tabular="level,logger,message,foo,network"
| eval eventtype="tabular"
```

This gives us the output shown in the following screenshot:

1	11/3/12 9:41:26.000 AM	level	logger	message	foo	network
		DEBUG	BarClass	error, ERROR, Error!	-	green
host=vlbmba.local sourcetype=template_example						

2	11/3/12 9:41:26.000 AM	level	logger	message	foo	network
		ERROR	FooClass	Don't worry, be happy.	-	-
host=vlbmba.local sourcetype=template_example						

3	11/3/12 9:41:26.000 AM	level	logger	message	foo	network
		INFO	-	Nothing happened. This is worthless. Don't log this.	-	-
host=vlbmba.local sourcetype=template_example						

4	11/3/12 9:41:25.000 AM	level	logger	message	foo	network
		WARN	BarClass	error, ERROR, Error!	-	-
host=vlbmba.local sourcetype=template_example						

5	11/3/12 9:41:25.000 AM	level	logger	message	foo	network
		ERROR	AuthClass	Don't worry, be happy.	-	red
host=vlbmba.local sourcetype=template_example						

Any field that does not have a value is rendered as `-`, as per the following template code:

```
str(event.fields.get(f, '-'))
```

It would be much simpler to use the `table` command instead of writing an event renderer. This approach is only appropriate when you need a very specific rendering or still need access to `workflow` actions. For another approach, check out the `Table` and `Multiplexer` modules available in the app `Sideview Utils`.

Pretty print XML

In this example, we will use Python's `minidom` module to parse and "pretty print" XML, if possible. The template will look for a field called `xml`, or fallback to `_raw`. Let's look through the files included in `ImplementingSplunkExtendingExamples`.

The template file, located at `appserver/event_renderers/xml.html`, contains the following lines of code:

```
<%inherit file="//results/EventsViewer_default_renderer.html" />\
<%def name="event_raw(job, event, request, options, xslt)">\
<%
from xml.dom import minidom
import sys

def escape(i):
    return i.replace("<", "&lt;").replace(">", "&gt;")

_xml = str( event.fields.get('xml', event.fields['_raw']) )
try:
    pretty = minidom.parseString(_xml).toprettyxml(indent=' '*4)
    pretty = escape( pretty )
except Exception as inst:
    pretty = escape(_xml)
    pretty += "\n(couldn't format: " + str( inst ) + ")"
%>
<pre class="xml_eventtype">${pretty}</pre>
</%def>
```

Our entry in `event_renderers.conf` is as follows:

```
[xml]
eventtype = xml
template = xml.html
```

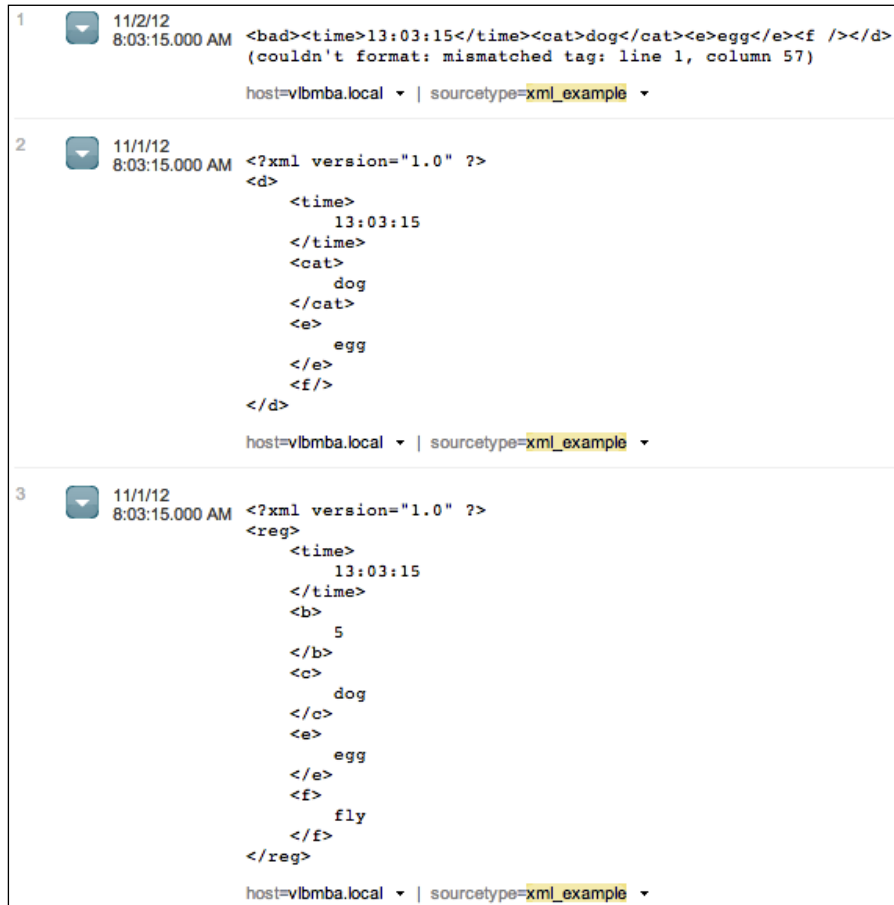
Our entry in `eventtypes.conf` is as follows:

```
[xml]
search = sourcetype="xml_example"
```

We can then simply search for our example source type as follows:

```
index="implsplunk" sourcetype="xml_example"
```

This renders the following output:



The XML in the first event is invalid, so an error message is appended to the original value.

Writing a scripted alert action to process results

Another option for interfacing with an external system is to run a custom **Alert action** using the results of a saved search. Splunk provides a simple example in `$$SPLUNK_HOME/bin/scripts/echo.sh`. Let's try it out and see what we get, using the following steps:

1. Create a saved search. For this test, do something cheap, such as the following:
`index=_internal | head 100 | stats count by sourcetype`
2. Schedule the search to run at some point in the future. I set it to run every five minutes, just for this test.
3. Enable **Run a script** and type in `echo.sh`.

Alert actions

Send email
 Enable

Add to RSS
 Enable
The RSS link is shown in Manager >> Searches and Reports.

Run a script
 Enable

File name of shell script to run

Splunk runs the script from \$\$SPLUNK_HOME/bin/scripts/

The script places the output into `$$SPLUNK_HOME/bin/scripts/echo_output.txt`. In my case, the output is as follows:

```
'/opt/splunk/bin/scripts/echo.sh' '4' 'index=_internal | head 100
| stats count by sourcetype' 'index=_internal | head 100 | stats
count by sourcetype' 'testingAction' 'Saved Search [testingAction]
always(4)' 'http://vlbmba.local:8000/app/search/@go?sid=scheduler__
admin__search__testingAction_at_1352667600_2efa1666cc496da4' '' '/
opt/splunk/var/run/splunk/dispatch/scheduler__admin__search__
testingAction_at_1352667600_2efa1666cc496da4/results.csv.gz' 'sessionK
ey=7701c0e6449bf5a5f271c0abdbae6f7c'
```

Let's look through each argument in the bullets that follow:

- \$0 - script path:
`'/opt/splunk/bin/scripts/echo.sh'`
- \$1 - number of events returned:
`'4'`
- \$2 - search terms:
`'index=_internal | head 100 | stats count by sourcetype'`
- \$3 - full search string:
`'index=_internal | head 100 | stats count by sourcetype'`
- \$4 - saved search name:
`'testingAction'`
- \$5 - the reason for the action:
`'Saved Search [testingAction] always(4)'`
- \$6 - a link to the search results. The host is controlled in `web.conf`:
`'http://vlbmba.local:8000/app/search/@go?sid=scheduler__admin__search__testingAction_at_1352667600_2efa1666cc496da4'`
- \$7 - deprecated:
`''`
- \$8 - the path to the raw results, which are always gzipped:
`'/opt/splunk/var/run/splunk/dispatch/scheduler__admin__search__testingAction_at_1352667600_2efa1666cc496da4/results.csv.gz'`
- STDIN - the session key when the search ran:
`'sessionKey=7701c0e6449bf5a5f271c0abdbae6f7c'`

The typical use for scripted alerts is to send an event to a monitoring system. You could also imagine archiving these results for some compliance reason or to import into another system.

Let's make a fun example that copies the results to a file, and then issues a cURL statement. That script might look like:

```
#!/bin/sh
DIRPATH='dirname "$8"'
```

```
DIRNAME='basename "$DIRPATH"'
DESTFILE="$DIRNAME.csv.gz"

cp "$8" /mnt/archive/alert_action_example_output/$DESTFILE

URL="http://mymonitoringsystem.mygreatcompany/open_ticket.cgi"
URL="$URL?name=$4&count=$1&filename=$DESTFILE"

echo Calling $URL
curl $URL
```

You would then place your script in `$(SPLUNK_HOME)/bin/scripts` on the server that will execute the script and refer to the script by name in **Alert actions**. If you have a distributed Splunk environment, the server that executes the scripts will be your search head.

If you need to perform an action for each row of results, then your script will need to open the results. The following is a Python script that loops over the contents of the gzip file and posts the results to a ticketing system, including a JSON representation of the event:

```
#!/usr/bin/env python

import sys
from csv import DictReader
import gzip
import urllib
import urllib2
import json

#our ticket system url
open_ticket_url = "http://ticketsystem.mygreatcompany/ticket"

#open the gzip as a file
f = gzip.open(sys.argv[8], 'rb')

#create our csv reader
reader = DictReader(f)
for event in reader:
    fields = {'json': json.dumps(event),
              'name': sys.argv[4],
              'count': sys.argv[1]}
```



```
#build the POST data
data = urllib.urlencode(fields)

#the request will be a post
resp = urllib2.urlopen(open_ticket_url, data)
print resp.read()

f.close()
```

Hopefully, these examples give you a starting point for your use case.

Summary

As we have seen in this chapter, there are a number of ways in which Splunk can be extended to input, manipulate, and output events. The search engine at the heart of Splunk is truly just the beginning. With a little creativity, Splunk can be used to extend existing systems, both as a data source and as a way to trigger actions.

Index

Symbols

- .conf files** 280
 - about 292
 - authorize.conf 325
 - commands.conf 326
 - fields.conf 322
 - indexes.conf 323, 324
 - inputs.conf 300
 - outputs.conf 323
 - props.conf 292
 - savedsearches.conf 326
 - times.conf 326
 - transforms.conf 310
 - web.conf 326
- <html> element** 187
- _indextime**
 - versus _time 42
- .ini files** 280
- |inputcsv command** 54
- |metadata command** 54
- () operator** 33
- [] operator** 33
- = operator** 33
- <row> element** 187
- <searchPostProcess> tag** 104, 106
- <searchString> tag** 98
- <searchTemplate> tag** 104
- .spl extension** 179
- .tgz extension** 179
- _time**
 - versus _indextime 42

A

- access.log file** 53
- actions** 51, 52
- actions icons** 17
- addterm** 218
- admin interface**
 - used, for building field 75, 76
- advanced XML**
 - reasons, for avoiding 202
 - reasons, for using 201
 - simple XML, converting to 205-210
- advanced XML structure**
 - about 203
 - example 204, 205
- aggregate of transaction statistics**
 - calculating 117
- alerts**
 - actions 51, 52
 - creating, from searches 48
 - Schedule step 49, 50
- AND operator** 32
- app, adding to Splunkbase**
 - about 196
 - directories, cleaning up 197, 198
 - packaging 198, 199
 - preparing 196
 - sharing settings, confirming 196
 - uploading 199, 200
- app directory structure** 194, 195
- appearance**
 - customizing, of app 184

apps

- about 10, 173
- adding, to Splunkbase 196
- appearance, customizing 184
- building 179-181
- customizing, custom CSS used 185, 186
- customizing, custom HTML used 187
- directory structure 194, 195
- installing 175
- installing, from files 178, 179
- installing, from Splunkbase 175, 176
- launcher icon, customizing 185
- purpose 173, 174
- used, for organizing configuration 361

appserver directory 194

appserver resources 327, 328

apps, Splunk

- gettingstarted 174
- search 174
- splunk_datapreview 174
- SplunkDeploymentMonitor 174
- SplunkForwarder 174
- SplunkLightForwarder 174

arguments

- used, for creating macro 159

arguments, lookup command

- as src_ip 177
- clientip 177
- geoip 176

arguments, timechart command

- bins 65
- limit 65
- usenull 65
- useother 65

attribute 281

authentication

- LDAP, using for 374

authorize.conf file 325

autoLB feature 346

automatic lookup

- defining 154-156
- fields 154, 155

average events per hour

- calculating 132-134

average events per minute

- calculating 132-134

average requests per minute

- calculating 131, 132

B

batch

- logs, consuming in 339, 340

bin directory 194

bins argument 65

blacklist

- using 302

boolean operators 32, 33

btool

- using 290, 291

bucket command 86, 130, 249, 251

buckets

- about 323, 354
- lifecycle 354, 355

buckets, lifecycle

- cold 354
- frozen 354
- hot 354
- thawed 354
- warm 354

by clause

- about 65
- concurrency, calculating with 124-129

C

cases, indexed fields 78-80

categorization 149

chart command

- about 63
- used, for turning data 61, 62

Chrome 7

CIDR wildcard lookups 316, 317

collect function

- about 258
- used, for producing custom summary indexes 258, 260

command line

- Splunk, using from 385, 386

commands

- configuring 392, 393
- data, generating 401, 402
- data, manipulating 394, 395

- data, transforming 396-401
- fields, adding 393, 394
- writing 390, 392
- writing, avoiding 390, 391
- commands.conf file** 326
- Comma Separated Values (CSV)** 150
- common attributes, props.conf**
 - about 292
 - index time 293
 - input time 296
 - parse time 293-295
 - search time 292, 293
- common field values**
 - displaying, top command used 54-56
- common input attributes, inputs.conf** 300
- complex dashboard**
 - ServerSideInclude, using in 188-191
- concurrency**
 - calculating, with by clause 124-129
 - determining 122
 - transaction, using with 122, 123
 - used, for estimating server load 123, 124
- configuration**
 - organizing, apps used 361
- configuration apps**
 - about 361, 362
 - indexerbase 362
 - inputs-sometype 361
 - outputs-datacenter 361
 - props-sometype 361
- configuration distribution**
 - about 366
 - deployment system, using 366
- configuration files, Splunk**
 - locating 279, 280
 - structure 280, 281
- configuration merging logic, Splunk**
 - about 281, 283
 - btool, using 290, 291
 - example 284-289
 - merging order 281
- configuration, Splunk Universal Forwarder**
 - default-mode.conf 335
 - inputs.conf 335
 - limits.conf 335
 - outputs.conf 335
 - props.conf 335
- configurations, Splunk indexer**
 - about 336
 - indexes.conf 336
 - inputs.conf 336
 - props.conf 336
 - server.conf 336
 - transforms.conf 336
- context macro**
 - building 167-169
- context workflow action**
 - building 165-167
- ConvertToDrilldownSearch module** 212, 219
- crcSalt**
 - using 305, 306
- CSV files**
 - used, for storing transient data 275
- cURL** 387
- custom CSS**
 - used, for customizing apps 185, 186
- custom HTML**
 - used, for customizing apps 187
 - using, in dashboard 187, 188
- custom query**
 - drilldown, building to 219-221

D

- dashboard panels**
 - placements 214, 215
- dashboards**
 - about 81
 - building, wizards used 82-90
 - converting, to forms 95-97
 - custom HTML, using 187, 188
 - development process 202
 - form, creating from 92, 94
 - generation, scheduling 91
 - need for 81
- data**
 - enriching, lookups used 150
 - gathering, scripts used 345
 - generating 401, 402
 - manipulating 394, 395
 - transforming 396-401
 - turning, chart command used 61, 62

database
logs, consuming from 343, 344

data gathering
scripted input, writing for 379

data generator 13

Data preview function 295

data sources 337

dedup command 156

deploymentclient.conf
installing 373

deploymentclient.conf configuration
defining 368

deployment server
about 377
advantages 367
apps, mapping to deployment clients in
serverclass.conf 369-372
configurations, normalizing into app 369
deploymentclient.conf configuration,
defining 368
deploymentclient.conf, installing 373
disadvantages 367
location, defining 368
location, for running 367
machine types, defining 368
restarting 373
using 367

deployment system
using 366

directory structure, index 350

divider tag 183

drilldown
about 219
building, to custom query 219-221
building, to multiple panels 224-228
building, to panel 222, 223

dropdown
prepopulating 276

dynamic fields
creating 319, 320

E

echo command 391

echo_csv command 391

echo_splunk command 391

EnablePreview module 212

epoch time 37

eval command
about 54, 68, 69
used, for building macro 160
used, for defining grouping fields 262, 263

eval function 169

event
script output, capturing as 382, 384

event renderer
about 406
pretty print XML 411, 412
specific fields, using 406-408
table of fields, based on field value 408,
409, 411
writing 406

events
dropping 321, 322
routing, to different index 314

event segmentation 34

events per slice of time
calculating 129

eventstats command 136

events viewer, search results 23, 24

event type 146

event types
used, for categorizing results 146-150
used, for grouping results 267, 268

ExtendedFieldSearch module 211

external commands
using 170

external site
workflow action, linking to 163, 164

extracted fields
versus indexed fields 77

Extract Fields interface
using 70-73

F

features, macro 169

features, tags 146

field
building, admin interface used 75, 76
prototyping, rex command used 73, 74

field context display
workflow action, building for 165

field picker
about 19
fields 19
using 26, 35, 36

fields
adding, to events 393, 394
using, for search 35
wildcards, supplementing in 37
working with 66

fields.conf file 322

field widgets 34

file
apps, installing from 178, 179

files
indexing, destructively 306
selecting, recursively 302

fillnull command 60

fill_summary_index.py script
about 256
used, for backfilling 256, 257

Firefox 7

Flash 7

FlashChart module 212

followTail attribute 304

form
panels, driving from 97-104

forms
about 92
building 92
creating, from dashboard 92, 94
dashboards, converting to 95-97
post-processing search results 104, 106

forwarders, Spunk 334

G

Geo Location Lookup Script
about 176
using 176, 177

gettingstarted app 174

Google
used, for generating results 172

Google Maps
about 176, 228, 229, 230
using 178

grep command 53

grouping fields
defining, eval command used 262, 263
defining, rex command used 262, 263

grouping operators 32, 33

H

head command 54

heavy forwarder 335

HiddenChartFormatter module 212

HiddenFieldPicker module 212

HiddenPostProcess
used for building drilldown, to multiple panels 224-228

HiddenSearch module 205, 211, 220

Home app 8-10

host 16

host categorization fields
creating 312

I

index
about 350
directory structure 350
events, routing to 314
sizing 355

indexed fields
advantages 77
cases 78-80
creating 310
disadvantages 77, 78
versus extracted fields 77

indexer 336

indexerbase app 362

indexer load balancing 348

indexers
sizing 345-347

indexes
about 243
reasons, for creating 351, 352
used, for increasing performance 353

indexes.conf file 323, 324

index time attributes, props.conf 293

inputcsv command 275

inputs.conf file
about 300
blacklist, using 302
common input attributes 300
crcSalt, using 305, 306
files, as input 301
files, indexing destructively 306
files, selecting recursively 302
native Windows inputs 308
network inputs 306, 307
old data, ignoring at installation 304, 305
patterns, used for selecting rolled logs 301
scripts, as inputs 309
symbolic link, following 303
value, setting of host from source 303
whitelist, using 302

inputs-sometype app 361

input time attributes, props.conf 296

installation, apps

about 175

from files 178, 179

from Splunkbase 175, 176

installation, deploymentclient.conf 373

instance types, Splunk 334

intentions

about 211, 217

addterm 218

stringreplace 217

using 217

J

JobProgressIndicator module 212

JSChart module 212, 220

L

latency

about 254

effect, on summary queries 254, 255

launcher icon

about 185

customizing 185

using 185

layoutPanel attribute

about 213

rules 213, 214

LDAP

about 8

using, for authentication 374

light forwarder 335

limit argument 65

load balancers

and Splunk 376

login screen, Splunk 8

loglevel

extracting 70

loglevel field

creating 310, 311

loglevel fields 82

logs

consuming, from database 343, 344

consuming, in batch 339, 340

monitoring, on server 337, 338

monitoring, on shared drive 338, 339

lookup command 151

lookup definition

about 152, 153, 315

fields 152, 153

wildcard lookups 315

lookups

about 390

troubleshooting 157

used, for enriching data 150

using, with wildcards 264-266

lookup table file

about 150

defining 150, 151

loosely related events

finding, subsearches used 111

M

macro

about 157

building, eval command used 160

creating 158

creating, with arguments 159

features 169

- mako templates**
 - about 406
 - URL 406
- Manager section**
 - about 27
 - using 27-29
- marker** 143
- merging order**
 - about 281
 - outside of search 281, 282
 - when searching 282, 283
- metadata** 328-330
- metadata fields**
 - events, routing to different index 314
 - hosts, overriding 313
 - modifying 312
 - source, overriding 313, 314
 - sourcetype, overriding 314
- minidom module** 411
- module logic flow** 210, 211
- modules**
 - functions 212
- msiexec**
 - used, for deploying Splunk binary 359
- multiple indexes**
 - managing, volumes used 356-358
 - working with 350
- multiple panels**
 - drilldown, building to 224-228
- multiple search heads** 377
 - configuring 377
- multivalued fields**
 - creating 318

N

- native syslog receiver**
 - using 341-343
- native Windows inputs** 308
- navigation**
 - about 182, 326
 - editing 182-184
 - object permissions, effects on 192
- nested subsearches** 113
- network inputs** 306, 307
- NOT operator** 33

O

- object permissions**
 - about 191
 - effects, on navigation 192
 - effects, on objects 192, 193
 - issues, correcting 193
 - options 191
- object permissions, options**
 - app 191
 - global 191
 - private 191
- OR operator** 33
- output**
 - controlling, for top command 56, 57
- outputs command** 275
- outputs.conf file** 323
- outputs-datacenter app** 361

P

- panel**
 - drilldown, building to 222, 223
- panels**
 - driving, from form 97-104
- parameter** 281
- parse time attributes, props.conf** 293-295
- patterns**
 - used, for selecting rolled logs 301
- Perl** 390
- Perl Compatible Regular Expressions (PCRE)** 68
- pipe symbol** 53, 54
- port 8000** 7
- post-processing search results**
 - about 104, 106
 - final XML 108
 - limitations 106
 - panel 1 106, 107
 - panel 2 107, 108
 - panel 3 108
- PostProcess module** 241
- processing stages, Splunk**
 - indexing 334
 - input 334
 - parsing 334
 - searching 334

props.conf file
about 292
attributes, with class 299, 300
common attributes 292
priorities, inside type 298
stanza types 296, 297

props-sometype app 361

Python 390

Q

query
reusing 215, 216
summary index events, using in 249-251

R

rare command 57

raw events
storing, in summary index 273, 275

Redirector module 232

redundancy
about 348
planning 348

redundancy, planning
indexer load balancing 348
typical outages 349, 350

REGEX attribute 322

regular expressions 66-68

REPORT
dynamic fields, creating 319, 320
multivalued fields, creating 318
using 318

REST
used, for querying Splunk 387-390

results
categorizing, event types used 146-150
generating, Google used 172
grouping, event types used 267, 268

rex command
about 54, 69
used, for defining grouping fields 262, 263
used, for prototyping field 73, 74

rolled logs
selecting, patterns used 301

rsyslog 341

running calculation
creating, for day 276, 278

S

Safari 7

savedsearches.conf file 326

saved tag 183

Schedule step 49, 50

scripted alert action
writing, for result processing 413-415

scripted input
about 379
creating 384, 385
writing, for data gathering 379

scripted lookup
advantages 403
writing, for data enrichment 403-406

script output
capturing, as single event 382, 384
capturing, with no date 380-382

scripts
used, for gathering data 345

search
about 149
clicking, for modification 34
fields, using for 35
performing, against time 39, 40, 41
running, values used 161-163
simplifying, tags used 143-146
time in-line, specifying in 41

search app
about 13, 174
actions icons 17
data generator 13
field picker 19
search results 16, 17, 21, 22
Summary view 14, 15
timeline 18

searches
alerts, creating from 48
making, faster 42
saving, for re-use 46-48
summary indexes, populating with 247, 248

search head pooling 376, 377

- search results**
 - about 16, 17, 21, 22
 - events viewer 23, 24
 - options 22, 23
 - sharing 43, 45
- search terms**
 - using, effectively 31, 32
- search time attributes, props.conf** 292, 293
- section** 280
- server load**
 - estimating, concurrency used 123, 124
- servers**
 - logs, monitoring on 337, 338
- ServerSideInclude**
 - using, in complex dashboard 188-191
- session field**
 - creating, from source 311
- session length**
 - determining, transaction command used 115, 116
- shared drive**
 - logs, monitoring on 338, 339
- Sideview**
 - views, linking with 232
- Sideview forms** 235, 238, 239, 241
- Sideview Search module** 231
- Sideview Utils**
 - about 230
 - Sideview forms 235, 238, 239, 241
 - Sideview Search module 231
 - URLLoader module 232-235
- simple XML**
 - converting, to advanced XML 205-210
- Single Sign On (SSO)**
 - about 375
 - using 375, 376
- sistats command** 251-254
- sitimechart command** 251-254
- sitop command** 251-254
- si* variants**
 - advantages 251
 - disadvantages 252
- size**
 - reducing, of summary index 261
- sort command** 58, 122
- source**
 - about 15
 - session field, creating from 311
- sourcetype** 15
- Splunk**
 - and load balancers 376
 - apps 174
 - configuration files, locating 279, 280
 - configuration files, structure 280, 281
 - configuration merging logic 281, 283
 - configuring, for boot launch 360
 - installation, planning 333, 334
 - instance types 334
 - logging into 7, 8
 - login screen 8
 - object permissions 191
 - processing stages 334
 - querying, via REST 387-390
 - regular expressions 66, 67, 68
 - summary indexes 243
 - time, displaying 38
 - time, parsing 37
 - time, storing 37
 - URL, for documentation 196
 - using, from command line 385, 386
- Splunk Answers**
 - URL 141
- Splunkbase**
 - about 10, 196
 - apps, adding to 196
 - apps, installing from 175, 176
 - URL 10, 196
- Splunk binary**
 - deploying 358
 - deploying, from tar file 359
 - deploying, msiexec used 359
- splunk_datapreview app** 174
- Splunk deployment**
 - base configuration, adding 360
- SplunkDeploymentMonitor app** 174
- Splunk deployment server**
 - using 367
- Splunk documentation** 11
- SplunkForwarder app** 174
- Splunk forwarders**
 - about 334
 - syslog, receiving with 343

- Splunk indexer**
 - about 336
 - configurations 336
 - sizing 345-347
 - syslog events, receiving on 340, 341
- Splunk interface**
 - about 7
 - field picker, using 26
 - Home app 8-10
 - Manager section, using 27-29
 - search app 13
 - time picker, using 25
 - top bar 11-13
- SplunkLightForwarder app** 174
- Splunk search** 337
- splunktcp** 376
- Splunk Universal Forwarder**
 - about 334
 - configuration, for installation 335
- Splunk Version 4.3** 7
- Splunk Versions 4.2** 7
- Splunk web server** 376
- stanza** 280
- stanza types, props.conf** 296, 297
- stats command** 172, 251
- stats function**
 - about 54, 130
 - structure 57
 - used, for aggregating values 57-61
- streamstats command** 125
- stringreplace** 217
- SubmitButton module** 211
- subnet field** 67
- subsearch**
 - about 111, 112
 - cautions 112
- subsearches**
 - combining, with transaction 118-121
 - used, for finding loosely related events 111
- summary data**
 - backfilling 256
- summary index**
 - about 243
 - avoiding 246
 - creating 244
 - events, using in query 249-251
 - populating, with saved searches 247, 248
 - producing, collect function used 258, 260
 - raw events, storing in 273, 275
 - size, reducing 261
 - using 245
- summary index events**
 - using, in query 249-251
- summary queries**
 - latency, effects 254, 255
- Summary view** 14, 15
- symbolic links**
 - following 303
- syslog**
 - about 340
 - receiving, with Splunk forwarder 343
- syslog events**
 - receiving 340
 - receiving, directly on Splunk indexer 340, 341
- syslog-ng** 341

T

- table command** 122
- tablespace** 243
- tag field**
 - creating 311, 312
- tagging** 149
- tags**
 - about 143
 - features 146
 - used, for simplifying search 143-146
- tar file**
 - Splunk binary, deploying from 359
- third-party add-ons**
 - about 228
 - Google Maps 228-230
 - Sideview Utils 230
- time**
 - about 35, 37
 - displaying 38
 - parsing 37
 - search, performing against 39-41
 - storing 37
 - using, in lookups 317, 318
- timechart command**
 - about 63, 249

- arguments 65
- used, for displaying values over time 63, 64
- using 129, 130

time in-line
specifying, in search 41

timeline 18

time picker
using 25

TimeRangePicker module 211

times.conf file 326

time zones
determining 38

top
calculating, for large time frame 269-272

top bar 11-13

top command
about 54, 134
output, controlling for 56, 57
recreating 134-140
used, for displaying common field values 54-56

transaction
subsearches, combining with 118-121
using, with concurrency 122, 123

transaction command
about 114
aggregate of transaction statistics,
calculating 117
properties 116
rules 114
used, for determining session length 115,
116

transforms
chaining 320, 321

transforms.conf file
about 310
events, dropping 321, 322
indexed fields, creating 310
lookup definitions 315
metadata fields, modifying 312
REPORT, using 318
transforms, chaining 320, 321

transient data
storing, CSV files used 275

typical outages 349, 350

U

UI Examples app 92

URLLoader module 232-235

URLs 262

useNULL argument 65

useOther argument 65

user interface resources
about 326
appserver resources 327, 328
metadata 328, 329, 330
navigation 326
views 326

V

values
aggregating, stats function used 57-61
extracting, from XML 170

ViewRedirectorLink module 212

ViewRedirector module 212, 220

views
about 326
linking, with Sideview 232

viewstate 211, 329

ViewstateAdapter module 211

view tag 183

volumes
about 356
used, for managing multiple
indexes 356-358

W

web.conf file 326

weblog. *See* **blog**

where command 54

whitelist 302

wildcard lookups
about 315
CIDR wildcard lookups 316, 317
time, using 317, 318

wildcards
lookups, using with 264-266
supplementing, in fields 37
using, efficiently 36

Windows Management Instrumentation (WMI) 308

wizards

used, for building dashboards 82-90

workflow actions

building, for field context display 165

creating 160-163

linking, to external site 163, 164

search, running with values 161-163

X

XML

values, extracting from 170

XML dashboards

editing 91

xmlkv command 170

XPath 171

Y

Your Apps section 10



Thank you for buying
**Implementing Splunk: Big Data
Reporting and Development for
Operational Intelligence**

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

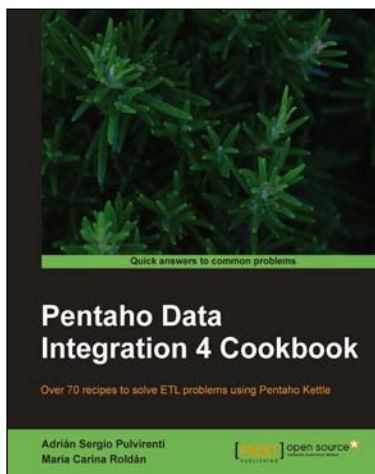
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



Pentaho Data Integration 4 Cookbook

ISBN: 978-1-84951-524-5 Paperback: 352 pages

Over 70 recipes to solve ETL problems using Pentaho Kettle

1. Manipulate your data by exploring, transforming, validating, integrating, and more
2. Work with all kinds of data sources such as databases, plain files, and XML structures among others
3. Use Kettle in integration with other components of the Pentaho Business Intelligence Suite



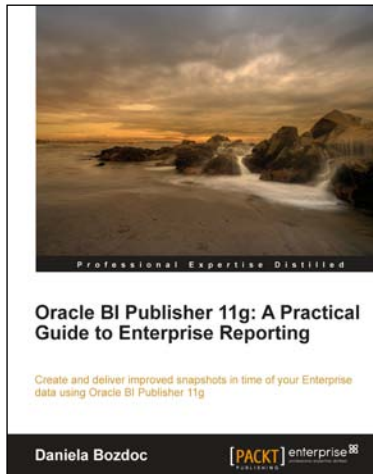
Learning Highcharts

ISBN: 978-1-84951-908-3 Paperback: 300 pages

Create rich, intuitive, and interactive JavaScript data visualization for your web and enterprise development needs using this powerful charting library — Highcharts

1. Step-by-step instructions with real-live data to create bar charts, column charts and pie charts, to easily create artistic and professional quality charts
2. Learn tips and tricks to create a variety of charts such as horizontal gauge charts, projection charts, and circular ratio charts
3. Use and integrate Highcharts with jQuery Mobile and ExtJS 4, and understand how to run Highcharts on the server-side

Please check www.PacktPub.com for information on our titles

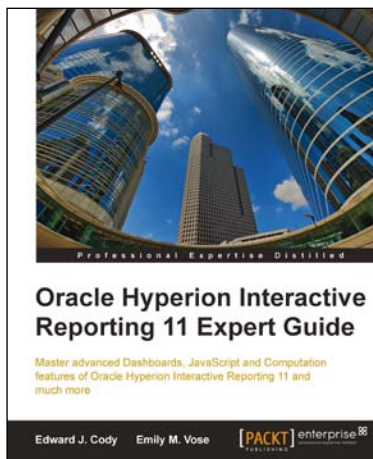


Oracle BI Publisher 11g: A Practical Guide to Enterprise Reporting

ISBN: 978-1-84968-318-0 Paperback: 254 pages

Create and deliver improved snapshots in time of your Enterprise data using Oracle BI Publisher 11g

1. A practical tutorial for improving your Enterprise reporting skills with Oracle BI Publisher 11g
2. Master report migration, template design, and E-Business Suite integration
3. A practical guide brimming with tips about all the new features of the 11g release



Oracle Hyperion Interactive Reporting 11 Expert Guide

ISBN: 978-1-84968-314-2 Paperback: 276 pages

Master advanced Dashboards, JavaScript and Computation features of Oracle Hyperion Interactive Reporting 11 and much more

1. Walk through a comprehensive example of a simple, intermediate, and advanced dashboard with a focus on Interactive Reporting best practices.
2. Explore the data analysis functionally with an in-depth explanation of built-in and JavaScript functions.
3. Build custom interfaces to create batch programs and exports for automated reporting.

Please check www.PacktPub.com for information on our titles