



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Getting Started with Memcached

Speed up and scale out your web applications with Memcached

Ahmed Soliman

[PACKT] open source*
PUBLISHING community experience distilled

www.allitebooks.com

Getting Started with Memcached

Speed up and scale out your web applications with Memcached

Ahmed Soliman

[PACKT] open source 
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

Getting Started with Memcached

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: November 2013

Production Reference: 1181113

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

ISBN 978-1-78216-322-0

www.packtpub.com

Cover Image by Aniket Sawant (aniket_sawant_photography@hotmail.com)

Credits

Author

Ahmed Soliman

Project Coordinator

Sageer Parkar

Reviewers

Jorge Arévalo

Nazimuddin Basha

David Hogue

Proofreader

Linda Morris

Indexer

Priya Subramani

Acquisition Editors

Vinay Argekar

Mary Jasmine Nadar

Production Coordinator

Manu Joseph

Commissioning Editor

Priyanka Shah

Cover Work

Manu Joseph

Technical Editor

Monica John

About the Author

Ahmed Soliman is an entrepreneur and software and systems engineer coming from a diverse background of highly scalable applications design, mission-critical systems, asynchronous data analytics, social networks design, reactive distributed systems, and systems administration and engineering. He has also published a technology patent in distributed computer-based virtual laboratories and designed numerous large-scale distributed systems for massive-scale enterprise customers.

A software engineer at heart, he is experienced in over 10 programming languages but most recently he was busy designing and writing applications in Python, Ruby, and Scala for several customers. He is also an opensource evangelist and activist. He contributed and maintained several open source projects on the Web.

Ahmed is a co-founder in Cloud Niners Ltd., a software and services company focusing on highly scalable cloud-based applications that have been delivering private and public cloud computing services to customers in the MEA region on different platforms and technologies.

Acknowledgments

I would like to thank some of the folks who changed my entire life for the better, upon meeting or working with them; those thanks do not come in a specific order but resembles a great appreciation for their support, help, and influence through my personal life and professional career:

- ▶ Al Sayed Al-Ghadban, for helping me switch to Linux back in late nineties
- ▶ Romain Sloodmaekers, for his priceless technical advice and continuous support
- ▶ Khaled El-Sersy, for helping me when I was a little kid to learn programming
- ▶ Ahmed Kamal, a great colleague and partner, who was always a believer and supporter
- ▶ Tarek El-Esseily, a great entrepreneur with endless energy
- ▶ Dr. Aser Farghal, for his endless care and support
- ▶ Dr. Kamal Shebl, who is like a father to me
- ▶ My Family, for everything!

A great thanks goes to my wife Sinar for her continuous support and help. She has been a great help and a deep source of inspiration. I have learned a lot from her patience and her support was the reason behind many of my/our successes together.

About the Reviewers

Jorge Arévalo is a computer engineer from the Universidad Autónoma de Madrid, UAM. He started developing Web applications with JS, PHP, and Python. In 2010, he began collaborating with PostGIS and GDAL projects, after participating in GSoC 2009, creating the PostGIS Raster GDAL driver. He currently works as a freelance Web/GIS developer and he collaborates with `geomati.co` group in projects such as gvSIG CE or QGIS. He also writes a blog about GIS: <http://www.libregis.org>.

"I worked with Jorge for a while. He acted as Scrum Master of our development team, where I was working as designer. He is a great professional, with proven ability as a developer and team manager. A good person and excellent colleague" (Carlos Azaustre, web developer).

"Jorge is a great project engineer, an excellent workmate. His commitment and devotion makes him a very valuable person." (Victor Serrano, GMS System Integration Engineer).

"Very organized, responsible and hard working colleague. Jorge is very autonomous and has a strong technical background. He is the kind of person you can trust in." (Diego Abia, Payload Data Ground Segment AIV Engineer).

Jorge Arévalo has co-written the book *Zurb Foundation 4 Starter*, for PacktPub Ltd. He has also worked as a reviewer for the books *PostGIS 2.0 Cookbook* and *OpenLayers beginner's guide (2nd edition)*, also for PacktPub Ltd.

I would like to thank my girlfriend, Elena Cedillo, for her continuous support and love.

Nazimuddin Basha has more than 15 years of experience in the information technology field in various positions. He started his career with Visual Basic and moved on to work in Java, later to .NET Framework. He is currently working as a .NET architect with a healthcare client providing services to the major insurance companies.

In his leisure time, he likes to watch movies and cricket, listen to music, and spend time with his kids (Aafreen and Asima).

His linkedin profile is <http://www.linkedin.com/in/nbasha/>.

David Hogue is a software developer currently working remotely out of the Portland, Oregon area. He's worked on a CMS called Pixelsilk that makes substantial use of Memcached for improving performance between clusters of Web servers. He has been programming in one language or another for over 15 years and is constantly looking out for new and useful tools or techniques.

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
Getting Started with Memcached	7
Basic installation of memcached on Ubuntu (Simple)	7
Basic installation of memcached on Mac (Simple)	9
Compiling memcached from a source on Ubuntu (Simple)	12
Talking with memcached (Advanced)	12
Setting up memcached to start on boot in Ubuntu (Simple)	16
Setting up distributed memcached (Intermediate)	18
Using memcached with PHP (Intermediate)	19
Using memcached with Python (Intermediate)	22
Using memcached with Ruby (Intermediate)	25
Using memcached with Java (Intermediate)	28
Setting up memcached support in Rails (Simple)	31
Setting up memcached support in Django (Intermediate)	34
Setting up memcached to support in Play (Intermediate)	37
Index	41

Preface

The Internet used to be a relatively smaller place compared to today's Internet. Applications were simpler, and most applications were getting a few hundred to thousands of requests per day. But that didn't last for long, the internet growth exceeded all expectations and with the boom of smart phones sales, servers started to choke, bottlenecks were congested and application scalability was inevitable. Memcached was written to reduce the number of database queries for the popular LiveJournal website, it was built as a general-purpose ultra-fast memory-based caching service. Today, memcached is used by many other sites, including Facebook, Youtube, Twitter, WordPress.com, Wikipedia, Digg, and the list keeps going. Many others use memcached to speed up their applications at very large scale infrastructures.

Caching is not only for big websites. In today's world the load characteristics changed dramatically, especially after the increasing number of mobile and tablet users accessing native and web applications, the average load on a medium-sized website in today's world is hundreds of thousands of requests per day. Serving pages as fast as possible is part of the user experience you are delivering to your users, you need to put in a lot of effort, because performance really matters!

Memcached is designed as a server-client service, this means that you connect to your memcached server, using a client written in your programming language of choice, or you will have to write a client yourself that speaks memcached protocol. Luckily, many clients are already robust and stable and waiting for your next big thing to be a part of.

Memcached is open source and publicly available under the New BSD License, a modern permissive license that allows you to do any kind of hacking and modifications if you want to. You can find many forks of memcached available with many interesting ideas already implemented in them, but even though, the original memcached code is still the most popular and you can grab it from `memcached.org` official website.

One of the interesting features of memcached is its ability to work in a highly scalable manner; you can run memcached in a consistent-hashing-based cluster and your cache will be partitioned across as many machines as you like, utilizing as much memory you already have available on your servers, to speed up the page loads.

This book is a hands-on guide on integrating memcached into your application in different programming languages, and if you are into web development, we have paid special attention to some of the most popular web development frameworks as well.

What this book covers

Basic installation of memcached on Ubuntu (Simple), provides easy installation steps for setting up memcached daemon on your Ubuntu machine.

Basic installation of memcached on Mac (Simple), provides easy installation steps for setting up memcached on MAC OS X.

Compiling memcached from a source on Ubuntu (Simple), will teach you how to build memcached from a source code on Ubuntu.

Talking with memcached (Advanced), will introduce you to the memcached ASCII protocol and help you to get acquainted with the different request types of memcached.

Setting up memcached to start on boot in Ubuntu (Simple), will guide you to set up memcached to start automatically after server reboots.

Setting up distributed memcached (Intermediate), how to scale up your memcached installation into a cluster and create a virtual shared caching memory pool on top of your memcached cluster.

Using memcached with PHP (Intermediate), will teach you to use memcached from your PHP program.

Using memcached with Python (Intermediate), will teach you to use memcached from your Python program.

Using memcached with Ruby (Intermediate), will teach you to use memcached from your Ruby program.

Using memcached with Java (Intermediate), will teach you to use memcached from your Java program.

Setting up memcached support in Rails (Simple), will teach you to integrate memcached into your Ruby On Rails setup to speed up your application.

Setting up memcached support in Django (Intermediate), will teach you to integrate memcached with your Django application.

Setting up memcached support in Play (Intermediate), will teach you to use memcached with the Play Framework, instead of the embedded Ehcache.

What you need for this book

You will need a computer with Ubuntu Linux (other distributions also work), but examples are explained based on Ubuntu. You can also use Mac OS X 10.7 or newer.

Who this book is for

This book targets software engineers and system administrators willing to configure and use memcached clusters in their future or current applications.

In this book, we are mainly using Ubuntu Linux 12.04LTS and Mac OS X 10.8 for our recipes, but you still can use any other operating system (Windows) if you like, however, you might find it harder to get memcached properly installed on Windows; you have been warned!

If you are a seasoned developer in any of the following frameworks such as Ruby on Rails, Django, and Play Framework 2.2.X, this book is definitely for you. You will learn how to configure those frameworks to rely on memcached for all of the caching tasks.

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, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
<?php
    $memcache = new Memcache;
    $memcache->connect('localhost', 11211) or die ("Could not connect");

    $version = $memcache->getVersion();
    echo "Server's version: ".$version."<br/>\n";
?>
```

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

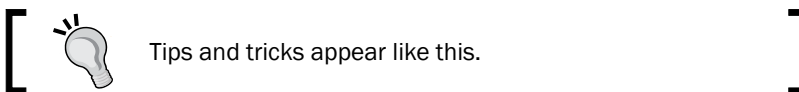
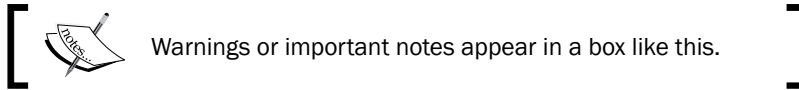
```
<?php
    $memcache = new Memcache;
    $memcache->connect('localhost', 11211) or die ("Could not connect");

    $version = $memcache->getVersion();
    echo "Server's version: ".$version."<br/>\n";
?>
```

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

```
# memcached -v
```

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: "clicking the **Next** button moves you to the next screen".



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.

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/submit-errata>, 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.

Getting Started with Memcached

Welcome to Getting Started with Memcached (A handy guide for memcached that helps you boost your application performance easily).

In this section, we will be covering the basic steps to get your memcached server up and running, either for testing or for a real production environment.

Basic installation of memcached on Ubuntu (Simple)

Let's get started with the basic installation of memcached on Ubuntu Linux 12.04LTS (long-time support) using apt-get. We have picked this particular version of Ubuntu because it's the latest LTS version that came out while writing this book, however, the steps of installation are the same for any other Ubuntu version. LTS is generally recommended for production servers because it gets a long period of maintenance and support from the folks at canonical.

Getting ready

You will need to have an administrator account on the Ubuntu box you are setting up. If you are performing some tests then most likely any machine would do the job but if you are setting this up as a production environment, you will need a machine with a decent amount of free memory for the caching job.

Update your apt local repository by using:

```
sudo apt-get update
```

When asked for the password, just enter your account password to give permission to the application to run as root.

How to do it...

1. Use `apt-get` to install the memcached service:

```
sudo apt-get install memcached
```

2. Now, let's verify that memcached service has been started.

```
ps aux | grep memcached
```

3. You are supposed to see something similar to the following:

```
memcache  830  0.0  0.1 323220 1188 ?          S1  17:33   0:00 /usr/bin/memcached -m 64 -p 11211 -u memcache -l 127.0.0.1
```

How it works...

First, we pulled the latest packages information from the apt repository online to make sure we are downloading the latest version of memcached to our local server. Then we simply used the `apt-get` command to download and auto-install the memcached package.

The installation script also starts the memcached daemon and marks this service to be auto-started on every boot of our Ubuntu box.

We validated that memcached was properly started by checking the running processes with the `ps` command and `grep-ing` to see only processes with the word memcached in them.

It's also important to note here that the default configuration of memcached limits the memcached daemon to listen only on the loopback device (localhost). This means that you can connect to your memcached daemons only from local processes running on the same computer.

There's more...

Let's take a look at the configuration of the memcached daemon installed on our Ubuntu server.

Open the configuration file located at `/etc/memcached.conf` and locate the line where you see something like the following:

```
-l 127.0.0.1
```

This tells the memcached daemon to listen only on the localhost, note that this is the only security measure that memcached can offer, so make sure it's listening on a firewalled interface.

Change it to the following if you want the daemon to listen on all interfaces.

```
-l 0.0.0.0
```

Also locate the following line:

```
-m 64
```

This configuration parameter configures the upper cap of how large the in-memory storage can grow to. The default here is 64 megabytes. This means that you can store up to 64 MB worth data on your memcached daemon, but this doesn't mean that the daemon will allocate this memory on its boot.

Installation on Windows

In most cases you will only need memcached on windows for development or testing, it's quite unlikely to see memcached installed on a production server.

Memcached is written in C so it's portable, but it is not officially supported or recommended to run on Windows. However, there have been a few ports to Windows, a popular one can be found at <http://code.jellycan.com/memcached/>, see memcached for win32. As advertised, it comes with no promises or support.

Basic installation of memcached on Mac (Simple)

Installation on Mac OS X is quite a straightforward process if you have the right tools installed.

Getting ready

We will be using a package manager for Mac OS X that is really a must-have tool for any Mac user and even more important if you are a developer or a system engineer.

The package manager we will be using is Homebrew, the missing package manager for OS X

You will need to install Homebrew first if you don't have it, installation is straightforward and all instructions are explained in different languages for your comfort at <http://brew.sh/>.

Or you can simply use this one liner to install Homebrew on your Mac:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/mxcl/homebrew/go)"
```

How to do it...

1. Update the Homebrew local repository:

```
brew update
```

2. Use Homebrew to install the memcached package:

```
brew install memcached
```

Memcached is not started by default after installation, if you manually want to start memcached use, `/usr/local/opt/memcached/bin/memcached`

3. If you are planning to start memcached on boot every time, you will need to create a link:

```
ln -sfv /usr/local/opt/memcached/*.plist ~/Library/LaunchAgents
```

Then, you may want to start memcached immediately using `launchctl`:

```
launchctl load ~/Library/LaunchAgents/homebrew.mxcl.memcached.plist
```

How it works...

We started by updating the local copy of the Homebrew repository from the Internet using the `brew update`, this ensures we are installing the latest version of any package we want to. Then we installed the latest version of memcached.

Homebrew does not start memcached automatically after installation nor during boot time, so we had to do this ourselves.

You can always start memcached manually in the foreground by using the memcached daemon executable. But if you want memcached to start on boot we created a symlink so that `launchctl` picks it up on boot.

There's more...

If you have configured your memcached daemon to start on boot as previously described, you might be wondering, where is the default configuration? Is it in the same place as Ubuntu? The answer is No!

Because the memcached service will be started by `launchctl`, the configuration is controlled by it. You will find the configuration file at `/usr/local/opt/memcached/homebrew.mxcl.memcached.plist` and it's basically an XML file.

See the following section in the file:

```
<array>
  <string>/usr/local/opt/memcached/bin/memcached</string>
  <string>-l</string>
  <string>localhost</string>
</array>
```

As you may have discovered yourself, this represents a list of parameters passed to the memcached executable at runtime.

You can edit the `localhost` field, as previously described in the Ubuntu configuration section, but if you want to configure the amount of memory that memcached can use, you will need to insert a couple of directives for that directly after the `localhost` directive:

```
<string>-m</string>
<string>256</string>
```

This configures memcached to use up to 256 MB of memory for its on-memory storage.

Another option if you are not a Homebrew user, is to use MacPorts (<http://www.macports.org/>), it works almost the same way and you can use the `command port` instead of `brew`.

Another interesting feature of `brew`, is that you can specify options to control the way memcached is built (compiled), so most of the time you don't really need to compile memcached from source on Mac OS X, instead, you use `brew` options for that. An example, is to enable SASL support to disable the plain ASCII protocol or to add SASL with password option, as stated in the `brew info memcached`.

```
--enable-sasl
    Enable SASL support -- disables ASCII protocol!
--enable-sasl-pwdb
    Enable SASL with memcached's own plain text password db support
-- disables ASCII protocol!
```

So, for example, if you want to enable SASL support during installation, use the following:

```
brew install memcached --enable-sasl
```

Compiling memcached from a source on Ubuntu (Simple)

In some cases, you might want to enable some of the memcached features that have to be baked-in during the compile time of the program. In this recipe, we will learn how to compile memcached from a source on Ubuntu

Getting ready

We will install the requirements of the package by using `apt-get`:

```
sudo apt-get install g++ make libevent-dev
```

This installed the C++ compiler, make, and the libevent library headers needed to compile memcached.

How to do it...

1. Let's download the latest version of memcached in your home directory:

```
curl -O --location http://memcached.org/latest  
mv latest memcached-latest.tar.gz  
tar vxzf memcached-latest.tar.gz
```

2. Next, let's configure and compile

```
cd memcached-*  
./configure && make
```

3. If the compilation process went well, we install the binaries:

```
sudo make install
```

Talking with memcached (Advanced)

Now, since we have memcached installed, let's see what kind of commands memcached daemon supports and how simple the memcached protocol is.

We will be using a plain simple TELNET tool to connect to the memcached daemon.

Remember that memcached has no persistent storage whatsoever, so it's totally memory-based and once we terminate the daemon everything we have stored is simply gone!

Getting ready

You will need to have telnet client installed on your machine, in most cases you will find it already installed but in case you didn't find it you can install it on your Ubuntu box using

```
sudo apt-get install telnet
```

You also must make sure that the memcached daemon is actually running.

Connect to the running memcached daemon with telnet on port 11211.

```
telnet localhost 11211
```

You should see something like the following:

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^['.
```

How to do it...

So, let's play with some basic storage commands to understand the main concepts behind memcached. We believe that this way is the best way to understand the features of the service and to truly realize its design simplicity and power.

Memcached supports a plain ASCII (text) protocol, you can find the protocol definition and specifications in the document in this link <https://github.com/memcached/memcached/blob/master/doc/protocol.txt>

1. The first command is `stats` where you request some basic information about the running service:

```
stats
STAT pid 8141
STAT uptime 1926
STAT time 1380294691
STAT version 1.4.13
STAT libevent 2.0.16-stable
STAT pointer_size 64
STAT rusage_user 0.108006
-
```


2. So, let's now use it for settings:

```
stats settings
STAT maxbytes 67108864
STAT maxconns 1024
STAT tcpport 11211
STAT udpport 11211
STAT inter 127.0.0.1
STAT verbosity 0
STAT oldest 849
STAT evictions on
STAT domain_socket NULL
STAT umask 700
STAT growth_factor 1.25
STAT chunk_size 48
STAT num_threads 4
STAT num_threads_per_udp 4
STAT stat_key_prefix :
STAT detail_enabled no
STAT reqs_per_event 20
STAT cas_enabled yes
STAT tcp_backlog 1024
STAT binding_protocol auto-negotiate
STAT auth_enabled_sasl no
STAT item_size_max 1048576
STAT maxconns_fast no
STAT hashpower_init 0
STAT slab_reassign no
STAT slab_automove no
END
```

3. Now, let's store some value for a given key:

```
set mykey 0 300 5 16
I Love Memcached
```

4. After you hit the return key, you will see the `STORED` message. So the whole listing is as follows:

```
set mykey 0 300 16
I Love Memcached
STORED
```

5. Now, let's read this key by using the `get` command:

```
get mykey
VALUE mykey 0 16
I Love Memcached
END
```

How it works...

This recipe gives you a glimpse of the kind of commands you can send to your memcached daemon using a simple tool like telnet.

We started by connecting to the memcached daemon on the default port 11211 using telnet. Then we used the `stats` command which asks the daemon to send us some useful statistics from the service such as the uptime, how many `get` requests actually returned data `get_hits`, and how many `get` requests resulted in a miss hit `get_misses`.

Then, we used `stats settings` which prints out the settings and configuration of the current running daemon, you will be able to see things such as `tcpport` which points to the port it is listening to and something such as `maxbytes` which is the maximum number of bytes allowed in this cache server.

Then, we moved to the storage commands `set` and `get`. Storage commands have the following format:

```
<command name> <key> <flags> <exptime> <bytes>
```

The `<command name>` field can be `set`, `add`, `replace`, `append`, or `prepend`.

The `<key>` field is the name of the key you are storing, in our case that was `mykey`.

The `<flags>` field is an arbitrary 16-bit unsigned number that the server stores along with the key and is returned when the client requests to get the value. It's opaque to the server, so it doesn't give any special meaning to the server itself but the client can use this number to add some special meaning to this key if needed. In our case, we just passed 0 for this field.

The `<exptime>` field indicates the expiration time, if it's 0, the item never expires (although it might get deleted when the server needs to free up place for another key to be stored). If it's non-zero (either Unix time or offset in seconds from the current time), it is guaranteed that clients will not be able to retrieve this item after the expiration time arrives (measured by the server time).

The `<bytes>` field indicates the length of the value to be stored, in our case that was 16, which is the length of the words `I Love Memcached`.

After hitting your return (*Enter*) key, you are supposed to feed the server with the value to be stored along with the key. Then, after hitting another return, you receive a `STORED` message indicating that the key-value pair has been stored.

Then, we moved to `get`, it's very simple, you get a `<key>` field and the value returns along with the `<flags>` field and the length of the value, then the value is printed before the `END` sentinel.

```
VALUE mykey 0 16
I Love Memcached
END
```

Setting up memcached to start on boot in Ubuntu (Simple)

How to make sure that memcached daemon is started by default on boot?

Getting ready

On server installations, we need to ensure that memcached is automatically started on boot if it's not already.

How to do it...

1. Check if memcached is already running or not:

```
/etc/init.d/memcached status
* memcached is running
```
2. If you want to disable starting memcached on boot:

```
sudo update-rc.d memcached disable
```
3. If you want to re-enable memcached to start on boot:

```
sudo update-rc.d memcached enable
```
4. To ensure it's running in the default run levels:

```
sudo update-rc.d memcached defaults
```

How it works...

We are using `update-rc.d` script to create and delete symbolic links at `/etc/rcX.d/` where `x` is the runlevel number.

Those symlinks are scanned on boot and they control whether the service is going to be started or not, based on the initial letter.

If you have seen the output of `update-rc.d memcached enable`

```
Enabling system startup links for /etc/init.d/memcached.  
Removing any system startup links for /etc/init.d/memcached:  
/etc/rc0.d/K20memcached  
/etc/rc1.d/K20memcached  
/etc/rc2.d/K80memcached  
/etc/rc3.d/K80memcached  
/etc/rc4.d/K80memcached  
/etc/rc5.d/K80memcached  
/etc/rc6.d/K20memcached  
Adding system startup for /etc/init.d/memcached:  
/etc/rc0.d/K20memcached -> ../init.d/memcached  
/etc/rc1.d/K20memcached -> ../init.d/memcached  
/etc/rc6.d/K20memcached -> ../init.d/memcached  
/etc/rc2.d/S20memcached -> ../init.d/memcached  
/etc/rc3.d/S20memcached -> ../init.d/memcached  
/etc/rc4.d/S20memcached -> ../init.d/memcached  
/etc/rc5.d/S20memcached -> ../init.d/memcached
```

You will see that the symlinks may start with `K` or `S`, which indicates that in a certain runlevel, the system should `Kill` or `Start` the service, respectively.

Setting up distributed memcached (Intermediate)

One of the most common use cases of using memcached is to build a distributed cache environment over multiple machines in a cluster. The setup allows you to scale up memcached horizontally by adding more machines to a cluster, you expand the total memory available for your application as a cache. The benefit of having a horizontally scalable caching, is that you are not limited by the amount of RAM you can install in a single server any more. It also means that you can utilize some of the free memory you have in your web server or so, and collectively you will have a distributed memcached environment with a large single virtual memory pool for your caching needs.

Building a distributed memcached environment is far simpler than you might have thought. The memcached daemon is blind about the cluster setup and has no special configuration on the server side to run the cluster, the client is actually doing the data distribution not the server.

Getting ready

So, it all starts when a single server cannot hold your entire cache and you need to split the cache pool across several servers.

If you are running multiple instances of the memcached daemon on the same server, make sure you are running them on different ports.

```
memcached -p 3030
memcached -p 3031
```

How to do it...

The server installation goes as previously described and the cluster configuration goes to your client by adding the list of servers to all your clients.

It's important to note that in order to ensure that the cluster is sane, is to have the same order of servers in all of your clients.

As an example, I'll be using python's `pylibmc` library to communicate with the memcached cluster:

```
import pylibmc
mc = pylibmc.Client(["127.0.0.1:3030", "127.0.0.1:3031"], binary=True,
behaviors={"tcp_nodelay": True, "ketama": True})
mc["ahmed"] = "Hello World"
mc["tek"] = "Hello World"
```

How it works...

What happens is that you specify a list of your servers to your client configuration and the client library uses consistent hashing to decide which server a certain key-value should go to.

The constructor of the client object here was fed with a couple of interesting parameters:

- ▶ `binary = True`: This is to configure `pylibmc` to use the memcached binary protocol not the ASCII protocol.
- ▶ `behaviors={"tcp_nodelay": True, "ketama": True}`: This configures the memcached connection socket to use the `tcp_nodelay` socket option which disables Nagle's algorithm (http://en.wikipedia.org/wiki/Nagle%27s_algorithm) on the socket level. Setting `"ketama" = True` means that `pylibmc` is using `md5` hashing and that it's using consistent hashing for key distribution.



The consistent hashing algorithm relies on the order of the list of the servers, so you need to have all your clients in-sync with the same configuration list in exact order.

After we have created the client object, we have set two keys `ahmed` and `tek` with the value `Hello World` and what actually happens behind the scenes is that each key-value pair is actually stored on a different daemon, according to the consistent hashing of the key.

Caching with persistence

Sometimes you want your caching server to be persistent; there are several very good alternatives to memcached that can help you achieve that.

You can checkout Redis at

<http://redis.io> and Kyoto Tycoon at <http://fallabs.com/kyototycoon/>.

Using memcached with PHP (Intermediate)

There are basically two memcached clients for PHP right now (**memcache**, and **memcached**), note the *d* at the latter. The **memcache** extension is older, lightweight, and most commonly used, and easier to install. The **memcached** module is feature-rich but still not widely adopted.

For the sake of simplicity, we will be using the **memcache** PHP extension in this recipe.

PHP is one of the most popular languages used for Web development today, it's very likely that you are actually using many pieces of software written in PHP on a daily basis without knowing.

Getting ready

I'm assuming you are using Ubuntu; you will need to have the simple setup of apache2 and php5. It's simple to get this stack working using this command:

```
sudo apt-get install apache2 php5
```

First, we need to install the PHP memcache extension using apt-get:

```
sudo apt-get install php5-memcache
```

This automatically installs the extension and gets everything wired and configured for you, if you want to use the memcached extension instead, all you need to do is to replace php5-memcache with php5-memcached and voila, everything just works!

If you are using Mac OS X, it's a slightly different story, and you will need to install apache2 and php5.

One of the quickest ways to do so is to install a nice package called MAMP (<http://www.mamp.info/en/index.html>); it will make life a lot easier for you. But, if you are an advanced user and want to go with the more manual route, you get really detailed instructions to get your OS X setup ready with Apache, MySQL, and PHP ready at (<http://jason.pureconcepts.net/2012/10/install-apache-php-mysql-mac-os-x/>).

How to do it...

1. First, we are going to start with a connection test to the memcached daemon:

```
<?php
    $memcache = new Memcache;
    $memcache->connect('localhost', 11211) or die ("Could not
connect");

    $version = $memcache->getVersion();
    echo "Server's version: ".$version."<br/>\n";
?>
```

2. The output of this script actually depends on the current version of the memcached server running, in my case the output is:

```
Server's version: 1.4.13
```

3. Next, let's set and get some keys from the connected memcached server:

```
<?php
    $memcache = new Memcache;
```

```
$memcache->connect('localhost', 11211) or die ("Could not
connect");

$sample_obj = new stdClass;
$sample_obj->str_attr = 'Memcache in PHP is cool';
$sample_obj->int_attr = 2468;

$memcache->set('sample_user', $sample_obj, false, 15) or die
("Failed to store data in memcached");
echo "Data stored in Memcached (will expire in 15
seconds)<br/>\n";

$get_result = $memcache->get('sample_user');
echo "Object from the cache:<br/>\n";

var_dump($get_result);

?>
```

How it works...

First, we are creating the Memcache object, that's the object we will be using to communicate with our memcached server in an object-oriented manner.

Then, we initialize the connection to the memcached server using the `connect` method that has the following signature which takes the host, port, and connection timeout:

```
bool Memcache::connect ( string $host [, int $port [, int $timeout ]]
)
```

The `connect` method closes the connection to the memcached server automatically at the end of the execution of the script.

Then, we used the `getVersion` method to retrieve the version of the memcached server we are connected to. We are using this method only to test our connection to the memcached server.

We then moved to the real work, we created an instance of the `stdClass` of PHP and added two attributes to the object to serialize and store in memcached under the key "sample_user"; We set the timeout to 15 seconds, this means that the memcached server will delete the key after 15 seconds. We also used `false` for flags, since we don't need compression or any other setting at the moment.

Then, we retrieved the value back from memcached using the `get` method of the `Memcache` object, and then we printed it on the screen. The output of the script would be as follows:

```
Data stored in cached (will expire in 15 seconds)
Object from the cache:
object(stdClass)#3 (2) { ["str_attr"]=> string(23) "Memcache in PHP is cool" ["int_attr"]=> int(2468) }
```

There's more...

If you are planning to connect to a cluster of memcached servers you will need to add all the servers using the `addServer` method:

```
<?php
/* OO API */
$memcache = new Memcache;
$memcache->addServer('memcached_host1', 11211);
$memcache->addServer('memcached_host2', 11211);
?>
```

Then, start using your memcache instance as usual and the magic will happen.

Using memcached with Python (Intermediate)

If you are planning to connect to memcached server(s) from your Python application, there are several clients available for you. The most popular ones are:

- ▶ **python-memcached:** This is a pure-python implementation of the memcached client (implemented 100 percent in Python). It offers good performance and is extremely simple to install and use.
- ▶ **pylibmc:** This is a Python wrapper on the `libmemcached` C/C++ library, it offers excellent performance, thread safety, and light memory usage, yet it's not as simple as `python-memcached` to install, since you will need to have the `libmemcached` library compiled and installed on your system.
- ▶ **Twisted memcache:** This client is part of the Python twisted event-driven networking engine for Python. It offers a reactive code structure and excellent performance as well, but it is not as simple to use as `pylibmc` or `python-memcached` but it fits perfectly if your entire application is built on twisted.

In this recipe, we will be using `python-memcached` for the sake of simplicity and since other clients have almost the same API, it does not make much difference from a developer's perspective.

Getting ready

It's always a good idea to create `virtualenv` for your experiments to keep your experiments contained and not to pollute the global system with the packages you install.

You can create `virtualenv` easily:

```
virtualenv memcache_experiments
source memcache_experiments/bin/activate
```

We will need to install `python-memcached` first, using the `pip` package manager on our system:

```
sudo pip install python-memcached
```

How to do it...

1. Let's start with a simple `set` and `get` script:

```
import memcache
client = memcache.Client([('127.0.0.1', 11211)])
sample_obj = {"name": "Soliman",
              "lang": "Python"}
client.set("sample_user", sample_obj, time=15)
print "Stored to memcached, will auto-expire after 15 seconds"

print client.get("sample_user")
```

2. Save the script into a file called `memcache_test1.py` and run it using `python memcache_test1.py`.
3. On running the script you should see something like the following:

```
Stored to memcached, will auto-expire after 15 seconds
{'lang': 'Python', 'name': 'Soliman'}
```

4. Let's now try other `memcached` features:

```
import memcache

client = memcache.Client([('127.0.0.1', 11211)])
client.set("counter", "10")

client.incr("counter")
print "Counter was incremented on the server by 1, now it's %s" %
client.get("counter")
```

```
client.incr("counter", 9)
print "Counter was incremented on the server by 9, now it's %s" %
client.get("counter")

client.decr("counter")
print "Counter was decremented on the server by 1, now it's %s" %
client.get("counter")
```

The output of the script looks like the following:

```
Counter was incremented on the server by 1, now it's 11
Counter was incremented on the server by 9, now it's 20
Counter was decremented on the server by 1, now it's 19
```

The `incr` and `decr` methods allow you to specify a delta value or to by default increment/decrement by 1.

Alright, now let's sync a Python dict to memcached with a certain prefix:

```
import memcache

client = memcache.Client(['127.0.0.1', 11211])

data = {"some_key1": "value1",
        "some_key2": "value2"}

client.set_multi(data, time=15, key_prefix="pfx_")

print "saved the dict with prefix pfx_"

print "getting one key: %s" % client.get("pfx_some_key1")

print "Getting all values: %s" % client.get_multi(["some_key1", "some_
key2"], key_prefix="pfx_")
```

How it works...

In this script, we are connecting to the memcached server(s) using the `Client` constructor, and then we are using the `set` method to store a standard Python dict as the value of the "sample_user" key. After that we use the `get` method to retrieve the value.



The client automatically serialized the python dict to memcached and deserialized the object after getting it from memcached server.

In the second script, we are playing with some of the features we never tried in the memcached server. The `incr` and `decr` are methods that allow you to increment and decrement integer values directly on the server automatically.

Then, we are using an awesome feature that we also didn't play with before, that is `get/set_multi` that allows us to set or get multiple key/values at a single request. Also it allows us to add a certain prefix to all the keys during the set or get operations.

The output of the last script should look like the following:

```
saved the dict with prefix pfx_  
getting one key: value1  
Getting all values: {'some_key1': 'value1', 'some_key2': 'value2'}
```

There's more...

In the `Client` constructor, we specified the server hostname and port in a tuple (host, port) and passed that in a list of servers. This allows you to connect to a cluster of memcached servers by adding more servers to this list. For example:

```
client = memcache.Client([('host1', 1121), ('host2', 1121), ('host3',  
1122)])
```

Also, you can also specify custom picklers/unpicklers to tell the memcached client how to serialize or de-serialize the Python types using your custom algorithm.

Using memcached with Ruby (Intermediate)

Ruby is also one of the most popular language used today by Web developers to build brilliant applications. The rise of the Rails framework was actually one of the main reasons this language received such popularity, however, Ruby is also a Swiss Army Knife language that is often used by system administrators for orchestration and automation.

There are of course, several clients to be memcached in Ruby but here we will be focusing on one of the most recent and stable clients that delivers high performance pure Ruby implementation of the memcached protocol, **Dalli!**

Dalli was written by the maintainer of memcache-client and is currently stable and being actively maintained.

The good thing about Dalli is that it can be integrated with Rails 3.x but, unfortunately, it does not integrate with the more popular Rails 2.x.

Getting ready

We need to install the Dalli gem using the following command:

```
gem install dalli
```

If you don't have the `gem` tool, then most likely you don't have Ruby properly installed. In Ubuntu you can always use the following to get Ruby installed:

```
sudo apt-get install ruby
```

How to do it...

1. Let's start by doing a very basic `set/get` operation on the memcached server from Ruby:

```
require 'dalli'
dc = Dalli::Client.new('localhost:11211', :threadsafe => true,
  :compress => true)
dc.set('somekey', 123)
puts("the value from cache is: #{dc.get('somekey')}")
```

2. This looks great; now let's store more complex structures in the memcached server:

```
require 'dalli'
dc = Dalli::Client.new('localhost:11211', :threadsafe => true,
  :compress => true)
user = { :name => "Ahmed",
  :job => "Engineer" }
dc.set('user1', user, ttl=20)
puts("user from cache: #{dc.get('user1')}")
```

3. Now, let's use a new feature which we did not use before (`replace`)

```
require 'dalli'
dc = Dalli::Client.new('localhost:11211', :threadsafe => true,
  :compress => true)
user = { :name => "Ahmed",
  :job => "Engineer" }
dc.set('user1', user, ttl=20)
puts("user from cache: #{dc.get('user1')}")

user[:age] = 31
dc.replace("user1", user, ttl=5)
puts("user from cache: #{dc.get('user1')}")
```

How it works...


First, we are importing Dalli into our namespace by using the `require` statement. Then, we are creating a client that connects to the memcached server and we are also setting some options. The following are some of the interesting options:

- ▶ `:compress => true`: This will ask Dalli to compress values larger than 1024 bytes.

- ▶ `:threadsafe => true`: This ensures that only a single thread is actively using the connection socket at a time; this is actually enabled by default, we added this to the snippet for clarity only.
- ▶ `:namespace => "app"`: This adds a prefix to all keys set in this connection.
- ▶ `:expires_in => 100`: This sets the default TTL(timeout) for all the keys where you are not specifying a TTL.

Then, we used the simple `set` method to set a basic integer value, after we retrieved that value and printed to the console using `puts`.

In the second snippet, we created a standard Ruby Hash and we used the built-in serializer to store this hash as a value for the `"user1"` key.

 This time we added a TTL parameter to specify that this key will expire after 20 seconds.

In the third snippet, we introduced a new feature of memcached, that is `replace`. This was used to replace the entire hash stored for the `"user1"` key with a modified version (we added age to it).

While we were replacing the value we also respecified the TTL value and changed the TTL of the value to be 5 seconds only. The `replace` feature fails if the key is not already stored in the memcached server and that's the main difference between it and the method that you are already familiar with `set`.

There's more...

The constructor of the `Client` class can also accept a list of servers to specify, in case you have a memcached cluster as you can see, it's a list of servers.

```
Dalli::Client.new(['localhost:11211:10', 'cache-2.example.com:11211:5', '192.168.0.1:22122:5'], :threadsafe => true, :failover => true, :expires_in => 300)
```

For every server, the format is `server:port:weight`, where `weight` allows you to distribute cache unevenly. Both `weight` and `port` are optional. If you pass in `nil`, Dalli will use the `MEMCACHE_SERVERS` environment variable or default to `localhost:11211` if it is not present.

Using memcached with Java (Intermediate)

In this recipe, we will be using Java to talk to our memcached server. Java is a very powerful programming language and is famous for enterprise-class applications.

There are, of course, a variety of memcached clients written for Java. We have chosen the most powerful client that is not hard to use, that is `spymemcached`.

Getting ready

The `spymemcached` java library has artifacts published on the maven central repository. If you are using maven you will need to add this to your `pom.xml` file (highlighted):

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.sample</groupId>
  <artifactId>spycache</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>spycache</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>net.spy</groupId>
      <artifactId>spymemcached</artifactId>
      <version>2.10.1</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

In my project here (named `spycache`), we will be showing snippets written inside the `src/main/java/com/sample/` directory and to run the application you will need to run the `mvn` package.

Then after seeing **BUILD SUCCESS** you will need to run the application by using the following:

```
java -cp target/spycache-1.0-SNAPSHOT.jar com.sample.App
```

How to do it...

1. Let's start by adding the following snippet into our main function:

```
try {
    MemcachedClient client = new MemcachedClient(new
InetSocketAddress("127.0.0.1", 11211));
    client.set("city", 20, "Istanbul");

    System.out.println((String)client.get("city"));

    client.shutdown();
} catch (IOException e) {
    e.printStackTrace();
}
```

2. We have just created a connection and did a simple set/get operation. Let's now store a more complex object:

```
class Employee implements Serializable{
    private static final long serialVersionUID =
2620538145665245947L;
    private String name;
    private int age;

    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```



```
@Override
public String toString() {
    return "Employee(\"" + name + "\", " + age + ")";
}
}
```

3. Then, let's store and retrieve an instance of this class:

```
Employee sample = new Employee("Ihab", 26);
client.set("engineer", 20, sample);

System.out.println((Employee)client.get("engineer"));
```

The output will be like the following:

```
2013-10-29 19:22:26.928 INFO net.spy.memcached.MemcachedConnection:
Added {QA sa=/127.0.0.1:11211, #Rops=0, #Wops=0, #iq=0, topRop=null,
topWop=null, toWrite=0, interested=0} to connect queue
2013-10-29 19:22:26.935 INFO net.spy.memcached.MemcachedConnection:
Connection state changed for sun.nio.ch.SelectionKeyImpl@680e62df
Employee("Ihab", 26)
2013-10-29 19:22:26.964 INFO net.spy.memcached.MemcachedConnection:
Shut down memcached client
```

How it works...

Basically, we are creating a `MemcachedClient` object that creates our memcached connection, note that this object is quite smart and does automatic reconnect on connection failure. By default, we are using the plain-text protocol but in the second example we are using the new binary protocol which is far more efficient.

The second snippet also shows how to configure `MemcachedClient` to connect to multiple memcached servers (cluster), automatic data distribution will be done for you.

The first method we used on the client object is `set` which is actually asynchronous, this means that it works like *fire and forget*, it does not block the current thread until the actual set operation happens. Then, we used `get` which is synchronous (the opposite), it blocks until data is retrieved and returns that data as `Object`, that's why we need casting to get a reference of the correct object type.

Java has an already very stable data serialization mechanism, and that's exactly what we have used. Standard types are serializable by default but what happens if you are trying to serialize your custom `Employee` object to memcached? You need to implement the `Serializable` interface for that, then magically everything works as presented in the last recipe.

There's more...

Spymemcached offers an asynchronous API for `get` as well, it's quite interesting actually and if you are familiar with `java.util.concurrent.Future` you will find it very easy to follow and understand.

The idea is that `client.asyncGet` returns a `Future<Object>` which you can use to retrieve the value asynchronously. You can use this future object to get the actual value later and set a timeout on your `get` request, as follows:

```
Future<Object> fobject = client.asyncGet("engineer");
try {
    fobject.get(10, TimeUnit.SECONDS);
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
} catch (TimeoutException e) {
    fobject.cancel(false);
}
```

What you can see here, is that we tried to `get` the future and we set 10 seconds for our trial, if the timeout expired, we get a `TimeoutException`, and then we can safely cancel the operation (by calling `cancel()` on the future). This means that we are not interested in the operation anymore.

You can also catch multiple exceptions to handle different types of potential failures, such as `InterruptedException`, which means that something interrupted the background thread, or `ExecutionException`, which means that an exception was thrown while trying to execute the background job.

Setting up memcached support in Rails (Simple)

Ruby on Rails Web development framework is extremely popular for rapid development of Web applications and, at some point in time it started an actual movement for a set of convention-over-configuration frameworks and most likely you have used one of them recently if you are a seasoned web developer.

We are assuming that you are familiar with Rails and you already have some experience using rails caching API, but even if you are not, that's a good introduction about caching in Rails anyway.

We will be using Dalli as the memcached client and we will be configuring a simple Rails application to use it as a backend for Rails Caching. If you want more information about Rails caching in general, you are advised to visit http://guides.rubyonrails.org/caching_with_rails.html.

Getting ready

You will need to have a working Rails installation on your system for that, you can find great tutorials on the Web for that, such as this one <https://www.digitalocean.com/community/articles/how-to-install-ruby-on-rails-on-ubuntu-12-04-lts-precise-pangolin-with-rvm>.

How to do it...

1. Let's now start by creating a really simple Rails application as a mock, to be the test base for our caching experiments: `rails new cachesample`.
2. In a few minutes, you will have your empty Rails application ready, you can run it using:

```
cd cachesample/  
rails server
```

3. You should see something like the following:

```
=> Booting WEBrick  
=> Rails 4.0.0 application starting in development on  
http://0.0.0.0:3000  
=> Run `rails server -h` for more startup options  
=> Ctrl-C to shutdown server  
[2013-10-29 16:40:56] INFO WEBrick 1.3.1  
[2013-10-29 16:40:56] INFO ruby 2.0.0 (2013-06-27) [x86_64-linux]  
[2013-10-29 16:40:56] INFO WEBrick::HTTPServer#start: pid=18450  
port=3000
```

4. Now, we need to configure our Rails application to use Dalli gem as a dependency.
5. Edit your `Gemfile` and add this line at the end of the file:

```
gem 'dalli'
```

6. Then, you will need to run:

```
bundle install
```

Now, we need to edit the application configuration to actually use memcached as the caching backend for the rails caching API.

Normally, you don't enable caching while development and you only turn it on in production, so we will be editing the `config/environments/production.rb` file (you may also want to add this to `config/environments/development.rb` if you want to see caching in development mode). Let's add:

```
config.cache_store = :dalli_store
```

How it works...

You will be able to use all of Rails automatic and manual caching features of Rails and the default cache store will be now `Dalli` (memcached client).

Rails has different features for caching, such as the following:

- ▶ **Action caching:** It caches an action response based on the input parameters and every request will go to all the *before filters*, this means that authentication is verified for example.
- ▶ **Fragment caching:** This means that specific pieces of template code can be cached, this is very useful, specially if you are building a huge page that contains pieces that are constantly changing and dynamic, you still can modularize parts of the page and cache those parts as fragments. For example, you want to cache a fragment that generates a list of products:

```
<% Order.find_recent.each do |o| %>
  <%= o.buyer.name %> bought <%= o.product.name %>
<% end %>

<% cache do %>
  All available products:
  <% Product.all.each do |p| %>    <%= link_to p.name, product_
url(p) %>
  <% end %>
<% end %>
```

- ▶ **SQL caching:** This is a feature that allows Rails to cache results of a SQL query so that if it encountered the same query again for that request, it will use the cached result set and will not be running the same query again on the database server.

There's more...

To use Dalli for Rails session storage that times out after 20 minutes, in `config/initializers/session_store.rb`:

```
config.cache_store = :dalli_store, 'cache-1.example.com', 'cache-2.
example.com',
  { :namespace => NAME_OF_RAILS_APP, :expires_in => 1.day, :compress
=> true }
```

Of course, you will need to write the correct names of the memcached hosts and ports for your caching cluster, instead of `cache-1.example.com` and `cache-2.example.com`.

Setting up memcached support in Django (Intermediate)

In the Python world, Django is the de facto standard choice as the most popular rich MVC/MVP framework around. It has a fantastic caching framework as well.

Django comes with a robust caching framework that lets you save dynamic pages so they don't have to be calculated for each request. Not only that, but also Django offers an abstract caching API that hides the specific implementation of the caching backend and offers a clean API to cache whatever you feel right, whenever you want to.

We are assuming you are a seasoned Django developer with some experience building Django applications in this recipe. Our goal here is to configure Django to use memcached as a caching backend and to introduce you to some of the features of Django's caching framework.

Getting ready

You will need to have a simple Django application to play with, if you don't have one you can create an empty project with an empty application by using the following:

```
django-admin.py startproject.djangocache
cd.djangocache/
python manage.py startapp.cachista
python manage.py runserver
```

This will create a project called `djangocache` and a simple app (module) inside your project that we called `cachista`.

If you don't have `python-memcached` installed already, you can simply use `pip` for that:

```
pip install python-memcached
```

How to do it...

1. Let's start by editing the `settings.py` file in your Django project (`djangocache/settings.py`, in our case), we will be using `python-memcached` for this recipe (you can use `pylibmc` too if you like).

2. The caching configuration parameter is controlled by the `CACHES` variable in the settings file. By default, you will not find this variable in your `settings.py` file, so we will need to add to it the `BACKEND` key in the 'default' dict which indicates the memcached client that you are planning to use.

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.
MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}
```

3. In this example, we used `MemcachedCache` which uses the `python-memcached` library.
4. If you are planning to use the faster `pylibmc`, you will need to replace this with `django.core.cache.backends.memcached.PyLibMCCache`.
5. The `LOCATION` key in the 'default' dict is where your memcached server is located, if you have a memcached cluster, you can change the value to be a list as follows:

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.memcached.
MemcachedCache',
        'LOCATION': [
            'cache-1.example.com:11211',
            'cache-2.example.com:11211',
        ]
    }
}
```

It's very important to understand that if you are planning to use multiple Django servers as a cluster, all the configurations of those servers need to have the same order as this caching list.

Now, let's tell Django to cache one of our views, it will automatically cache the view response for us. You will need to import that `cache_page` decorator first from `django.views.decorators.cache` import `cache_page`.

```
@cache_page(60 * 15, key_prefix="site1")
def my_view(request):
    """ my view code goes here """
```

Piece of cake! We told Django to cache this view for 15 minutes and the key prefix in the cache store will be "site1".

Now, do you remember the "default" we wrote in our `CACHES` setting? That was to setup multiple caching backends for Django! Yes, you can cache certain pages on certain caching backends. You can specify the caching backend in your `cache_page` view.

```
@cache_page(60 * 15, cache="memory_cache")
```

The "memory_cache" value must correspond to a key in your `CACHES` setting where you specify the caching backend settings. Fantastic!

As in Rails, you can specify fragments of your template to be cached.

```
{% load cache %}
{% cache 500 sidebar %}
  .. sidebar ..
{% endcache %}
```

Now, let's use the caching API to do manual caching of a value in our action/controller code from the following:

```
django.core.cache import get_cache
cache = get_cache('default')
cache.set('key', 'Hello Memcached!', 15)
print cache.get('key')
```

This looks very similar to the direct memcached API but it's not! It's an abstract API that can actually use multiple backends for you; memcached is one of them as configured in the `CACHES` setting.

How it works...

We started by defining the `CACHES` variable in the `settings.py` file and there we can define multiple cache regions with different backends. Django supports multiple cache backends, file-based, memory-based, and database-based. In our case, we used python-memcached backend and we specified that for the 'default' cache region.

Of course, it's very popular to use memcached as a cluster and to specify the list of servers to your configuration.

You can also specify some interesting options along with the `LOCATION` and `BACKEND` keys, some examples of the same are as follows:

- ▶ `TIMEOUT`: The default timeout, in seconds, to use for the cache. The default value is 300 seconds (5 minutes)
- ▶ `KEY_PREFIX`: A string that automatically will be prefixed to all cache keys.

Then we played with `cache_page` decorator which automatically caches a view for us, you can specify the prefix or the cache region you are planning to use for this particular page.

Then we have seen the template caching, you can cache pieces/fragments of your template code with the "cache" tag, you can specify in the identifier for this cached fragment and expiration.

In our case we used the `sidebar` identifier as stated in the following line:

```
{% cache 500 sidebar %}
```

The expiration is set to 500 seconds, but interestingly you can specify more keywords for your identifier for the same fragment.

```
{% cache 500 sidebar welcome %}
```

Also, you can use the low-level caching API if you want more granular control over your caching and that was described in the last code snippet.

Setting up memcached to support in Play (Intermediate)

Play Framework is a modern Java/Scala framework that promises a lightweight, stateless, Web-friendly architecture.

It's built on Akka and it's very reliable for building highly-scalable applications with predictable resource consumption.

Play 2 is the next generation of the framework; it has gone through almost a complete rewrite, and it's now fully written in Scala but offers a good Java API. We will be focusing on Scala examples right here.

Play 2 uses `Ehcache` by default as a backend, you can always replace the backend by writing plugins for Play 2, fortunately, someone already did that and it's using the `spymemcached` java client for memcached.

We are using Play 2.2.X for this recipe which uses sbt 0.13.X.

Getting ready

1. Let's start by creating a simple play project.

```
play new playcache
```

2. Select **create a simple Scala application**.

3. Then we need to configure our project's `build.sbt` to use `play2-memcached` as a dependency. Edit your `build.sbt` to look like the following:

```
name := "playcache"

version := "1.0-SNAPSHOT"
libraryDependencies += Seq(
  jdbc,
  anorm,
  cache,
  "com.github.mumoshu" %% "play2-memcached" % "0.3.0.2"
)

resolvers += "Spy Repository" at "http://files.couchbase.com/
maven2"

play.Project.playScalaSettings
```

How to do it...

1. We need to configure our play application to use memcached instead of the default Ehcache backend for the Caching API of Play Start by adding the `play2-memcached` plugin to `conf/play.plugins` (create the file if not created already)

```
5000:com.github.mumoshu.play2.memcached.MemcachedPlugin
```

2. Then, let's edit the configuration `conf/application.conf` file and add near the end of the file, the following line to disable the ehcache plugin:

```
ehcacheplugin=disabled
```

3. Now, let's configure the memcached plugin to the memcached server:

```
memcached.host="127.0.0.1:11211"
```

4. After that, you are ready to use memcached, start the application server by running:

```
play run
```

5. Then, let's edit the controller at `app/controllers/Application.scala` to look like the following snippet:

```
package controllers

import play.api._
import play.api.mvc._
import play.api.cache.Cache
import play.api.Play.current

object Application extends Controller {
```

```

def index = Action {
  Cache.getAs[String]("key") match {
    case Some(v) =>
      Ok(s"Got the value from cache: $v")
    case None =>
      Cache.set("key", "Fantastic Value", 50)
      Ok("Setting value in Cache")
  }
}

```

- From your browser, visit `http://localhost:9000/` and see what happens. On your first request you should see something like the following:

```
Setting value in Cache
```

- Then if you refreshed the page, you will see the following line:

```
Got the value from cache: Fantastic Value
```

- One more thing you can do is to cache the entire action response by using the `Cached` object for that.

```

def index = Cached("homePage") {
  Action {
    Ok("Hello world")
  }
}

```

Congratulations, Play 2 is now connected and uses memcached as the caching backend.

How it works...

First, we needed to edit the build script (`build.sbt`) to add the `play2-memcached` as a dependency, we did that by appending in the `libraryDependencies` setting key the value to `"com.github.mumoshu" %% "play2-memcached" % "0.3.0.2"`.

Then, we added a resolver to tell `sbt` where to get this plugin from. Next, we created/edited the `conf/play.plugins` file to add the plugin to play and we configured `conf/application.conf` to point to our memcached server.

In the controller code, we used the `play.api.cache.Cache` object to get and set values from the cache.

The last thing is that you can use the `Cached` object to cache the entire action response in a named cache key `"homePage"`.

There's more...

If you are planning to use Play2 with a memcached cluster, you will need to configure the list of your servers in `conf/application.conf`.

The configuration is really straightforward. Just replace `memcached.host="127.0.0.1:11211"` with the list of the servers as follows:

```
memcached.1.host="cache-1.example.com:11211"  
memcached.2.host="cache-1.example.com:11211"
```

As mentioned several times before, it's important to keep this list in-sync for all your Play servers.

Index

Symbols

<bytes> field 15
<command name> field 15
<exptime> field 15
<flags> field 15
<key> field 15

A

Action caching 33
addServer method 22
apt-get command 8

C

Cached object 39
caching server
 with persistence 19
connect method 21

D

distributed memcached
 setting up 18, 19
Django
 memcached Support, setting up 34-37

F

Fragment caching 33

G

get method 24
getVersion method 21

J

Java
 memcached, using with 28-31

K

Kyoto Tycoon
 URL 19

M

Mac
 memcached, installing on 9-11

MacPorts
 URL 11

MAMP
 URL 20

memcached
 compiling, from source on Ubuntu 12
 installing, on Mac 9-11
 installing, on Ubuntu 7-9
 installing, on Windows 9
 setting up, to start on boot in Ubuntu 16, 17
 talking with 12-15
 used, with Java 28-31
 used, with PHP 19-22
 used, with Python 22-25
 used, with Ruby 25-27

memcached Support
 setting up, in Django 34-37
 setting up, in Play Framework 37-40
 setting up, in Rails 31-34

Memcache object 21

P

PHP

memcached, using with 19-22

Play Framework

memcached Support, setting up 37-40

ps command 8

pylibmc 22

Python

memcached, using with 22-25

python-memcached 22

R

Rails

features, for caching 33

memcached Support, setting up 31-34

Redis

URL 19

Ruby

memcached, using with 25-27

S

SQL caching 33

stats command 15

T

Twisted memcache 22

U

Ubuntu

memcached, compiling from source 12

memcached, installing on 7-9

memcached, setting up to start

on boot 16, 17

W

Windows

memcached, installing on 9



Thank you for buying **Getting Started with Memcached**

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.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

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.

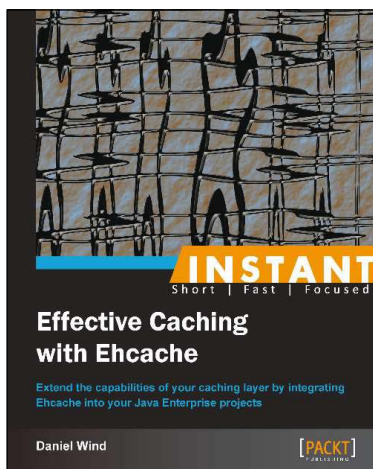


Lift Application Development Cookbook

ISBN: 978-1-84951-588-7 Paperback: 254 pages

Over 50 practical recipes to build web applications using Lift, the most secure web framework available

1. Lift made easy with step-by-steps recipes written by a developer for developers
2. Practical examples covering topics from basic to advanced levels
3. Learn to use Schemifier to automatically create tables and columns



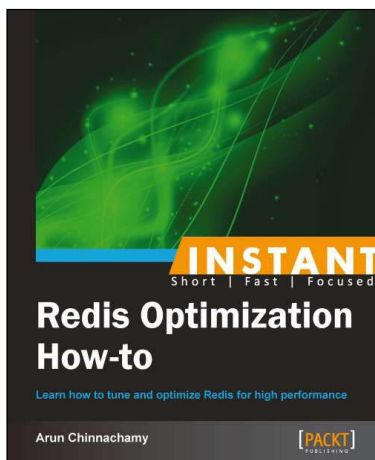
Instant Effective Caching with Ehcache

ISBN: 978-1-78216-038-0 Paperback: 86 pages

Extended the capabilities of your caching layer by intergrating Ehcache into your Java Enterprise projects

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results
2. Setup Ehcache and understand its configuration
3. Use Ehcache with popular Java frameworks
4. Monitor an Ehcache-based application using open source software

Please check www.PacktPub.com for information on our titles

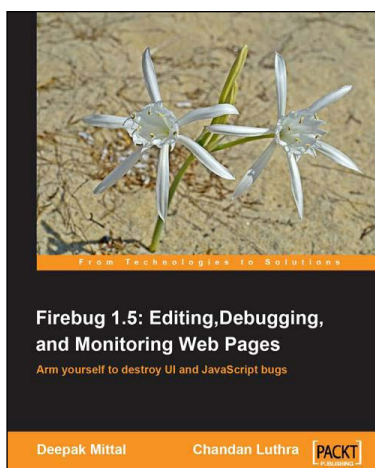


Instant Redis Optimization How-to

ISBN: 978-1-78216-480-7 Paperback: 56 pages

Learn how to tune and optimize redis for high performance

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results
2. Install, fine-tune, and add Redis to your application stack
3. Perform bulk writes into Redis efficiently
4. Debug and troubleshoot Redis



Firebug 1.5: Editing, Debugging, and Monitoring Web Pages

ISBN: 978-1-84719-496-1 Paperback: 436 pages

Arm yourself to destroy UI and JavaScript bugs

1. Expand your toolkit by learning to use Firebug to help you monitor, debug, develop and edit web pages on the fly
2. Create your own Firebug extensions and learn about popular third-party extensions
3. Covers JavaScript, AJAX, and CSS development

Please check www.PacktPub.com for information on our titles