



Community Experience Distilled

Mastering Google App Engine

Build robust and highly scalable web applications with Google App Engine

Mohsin Shafique Hijazee

[PACKT] open source*
PUBLISHING community experience distilled

Mastering Google App Engine

Build robust and highly scalable web applications
with Google App Engine

Mohsin Shafique Hijazee



BIRMINGHAM - MUMBAI

Mastering Google App Engine

Copyright © 2015 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: September 2015

Production reference: 1011015

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

ISBN 978-1-78439-667-1

www.packtpub.com

Credits

Author

Mohsin Shafique Hijazee

Project Coordinator

Harshal Ved

Reviewers

Aristides Villarreal Bravo

Johann du Toit

Proofreader

Safis Editing

Indexer

Priya Sane

Acquisition Editor

Nikhil Karkal

Production Coordinator

Komal Ramchandani

Content Development Editor

Athira Laji

Cover Work

Komal Ramchandani

Technical Editor

Naveenkumar Jain

Copy Editor

Ting Baker

Vedangi Narvekar

About the Author

Mohsin Shafique Hijazee started his programming adventure by teaching himself C, and later C++, mostly with the Win 32 API and MFC. Later, he worked with Visual Basic to develop an invoicing application for local distributors. In the meantime, .NET came along and Mohsin happened to be working with C# and Windows Forms. All of this was around desktop applications, and all of this happened during his days at university.

Very few people have had a chance to work with fonts, and that's exactly what Mohsin happened to do as his first job — developing OpenType fonts for complex right to left calligraphic styles such as Nastaleeq. He developed two different fonts, one based on characters and joining rules, and the other one contained more than 18,000 ligatures both of which are in public domain.

His first serious interaction with web development started with Ruby on Rails. Shortly after that, he discovered Google App Engine and found it to be a very interesting platform despite its initial limitations back in 2008, with Python being the only available runtime environment. Mohsin kept experimenting with the platform and deployed many production applications and mobile backends that are hosted on Google App Engine to this day.

Currently, Mohsin is working as a backend software engineer with a large multinational Internet company that operates in the online classified space in dozens of countries across the globe.

Acknowledgments

This book and a lot more would not have been possible without my parent's constant support in my earlier years. My father, Mohammad Shafique, taught me how to read, and later write, in multiple languages by using novels, literature, and other means that are not a part of traditional education in schools. This book would not have been possible if my mother, Azra Khanam, hadn't trained me in counting, adding, and playing with numbers even before I joined school. It was of course my mother who, in later years, helped me apply for a course in computer science for further education. My younger sisters Sara and Rida have been constant support by probing status on the book and keeping me motivated. Thank you both of you!

This book would not have been possible if my dear wife, Dr. Farzana, did not help to work around the tough schedule and absorbed moments or absent mindedness this piece of writing that you hold in your hands, brought to my life. She would quickly place a cup of tea on my table whenever I'd feel exhausted. Thanks a lot, patient is doing well now after being done with the writing project.

I'm not sure about whether I'd like to thank or complain about the little, cute, and aggressive boy Alyan, in our house who was way too young and in the cradle when I started the book, and by the time I finished it, he had started plucking out my laptop keys. Alyan, I hope you'll not repeat our mistakes and you'll make your own.

Special thanks goes to my newly found friend and colleague Naveed ur Rehman, who basically turned out to be an inspiration for me to write such technical text. Thank you Naveed, I've utmost respect for you.

I would like to thank my editors, Nikhil, Ajinkya, and Naveenkumar, for going through the tons of mistakes that I made throughout the text in painstaking detail, being tolerant about it, and constantly providing suggestions on how to improve the content. I became aware of their hard work when I had the chance to read my own script carefully. Thank you, gentlemen!

About the Reviewers

Aristides Villarreal Bravo is a Java developer. He is the CEO of Javscz Software Developers. He is also a member of the NetBeans Dream Team and he is part of Java User Groups leaders and members. He lives in Panamá. He has organized and participated in various national and international conferences and seminars related to Java, JavaEE, NetBeans, the NetBeans platform, free software, and mobile devices. He also writes tutorials and blogs related to Java and NetBeans and for web developers.

He has reviewed several books for Packt Publishing. He develops plugins for NetBeans and is a specialist in JSE, JEE, JPA, Agile, and Continuous Integration.

He shares his knowledge via his blog, which can be viewed by visiting <http://avbravo.blogspot.com>.

I would like to thank my family for the support throughout the book process.

Johann du Toit is an entrepreneur and a technical lead for various startups, ranging from national microchip databases to website verification systems.

He was appointed as the first Google Developer Expert in Africa for Cloud by Google.

He has experience that ranges from building large distributed systems that scale for millions of requests every day to embedded devices that serve Wi-Fi and medical information across Africa.

Visit <http://johanndutoit.net> for his latest details and whereabouts.

I would like to thank my family and especially my sister, Philanie du Toit, for her support through out the book process.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a 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 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	ix
Chapter 1: Understanding the Runtime Environment	1
The overall architecture	2
The challenge of scale	2
How to scale with the scale?	2
Scaling in practice	4
Infrastructure as a Service	5
Platform as a Service	5
Containers	6
How does App Engine scales?	7
Available runtimes	9
Python	10
The Java runtime environment	10
Go	11
PHP	11
The structure of an application	11
The available services	12
Datastore	13
Google Cloud SQL	13
The Blobstore	13
Memcache	14
Scheduled Tasks	14
Queues Tasks	14
MapReduce	15
Mail	15
XMPP	15
Channels	16

Users	16
OAuth	16
Writing and deploying a simple application	16
Installing an SDK on Linux	17
Installing an SDK on Mac	18
Installing an SDK on Windows	18
Writing a simple app	18
Deploying	24
Summary	26
Chapter 2: Handling Web Requests	27
Request handling	27
The CGI program	28
Streams and environment variables	29
CGI and Google App Engine	30
WSGI	31
Problems with CGI	31
Solutions	32
What WSGI looks like	32
WSGI – Multithreading considerations	34
WSGI in Google App Engine	35
Request handling in App Engine	36
Rendering templates	39
Serving static resources	44
Cache, headers, and mime types	45
Serving files	47
Using web frameworks	48
Built-in frameworks	48
Using external frameworks	50
Using Bottle	52
Summary	54
Chapter 3: Understanding the Datastore	55
The BigTable	56
The data model	57
How is data stored?	59
The physical storage	60
Some limitations	61
Random writes and deletion	62
Operations on BigTable	64
Reading	64
Writing	65
Deleting	65
Updating	65
Scanning a range	66

Selecting a key	67
BigTable – a hands-on approach	69
Scaling BigTable to BigData	70
The datastore thyself	73
Supporting queries	77
Data as stored in BigTable	77
The implementation details	78
Summary	79
Chapter 4: Modeling Your Data	81
The data modeling language	81
Keys and internal storage	85
The application ID	86
Namespaces	87
The Kind	88
The ID	88
The key	92
Modeling your data	94
The first approach – storing a reference as a property	94
The second approach – a category within a key	97
Properties	102
The required option	103
The default option	103
The repeated option	104
The choices options	104
The indexed option	105
The validator option	105
The available properties	106
Structured Properties	108
The computed properties	109
The model	110
The constructor	110
Class methods	110
The allocate_ids() method	111
The get_by_id() method	111
The get_or_insert() method	111
The query() method	111
The instance methods	111
The populate() method	112
The put() method	112
The to_dict() method	112
Asynchronous versions	112
Model hooks	113
Summary	115

Chapter 5: Queries, Indexes, and Transactions	117
Querying your data	117
Queries under the hood	125
Single-property queries	126
Examples of single-property queries	128
Multiple property indexes	129
Working with indexes	132
The query API	134
The Query object	135
App	136
Namespace	136
Kind	136
The ancestor	136
The projection	137
Filters	137
The orders	137
Further query options	138
Filtering entities	141
Filtering repeated properties	142
Filtering structured properties	143
The AND and OR operations	144
Iterating over the results	146
Conclusions	149
Transactions	149
Summary	152
Chapter 6: Integrating Search	153
Background	153
The underlying principle	154
Indexing your data	155
Sample data	156
Indexing thyself	160
Documents	162
Fields	162
The text fields	163
Placing the document in an index	164
Getting a document	165
Updating documents	166
Deleting documents	166
Indexing the documents	167
Queries	169
Simple queries	169
Multiple value queries	170

Logical operations	170
Being specific with fields	171
Operators on NumberField	172
Operators on DateField	172
Operations on AtomField	173
Operations on TextField and HTMLField	173
Operations on GeoField	174
Putting it all together	175
Selecting fields and calculated fields	177
Sorting	181
Pagination	185
Offset-based pagination	186
Cursor-based pagination	187
Facets	190
Indexing facets	191
Fetching facets	194
Asking facets via automatic discovery	195
Asking specific facets	198
Asking facets with specific values	198
Asking facets in specific ranges	199
Filtering by facets	201
Summary	202
Chapter 7: Using Task Queues	203
The need to queue things	204
The queue	205
Defining queues	207
Adding to a queue	212
Processing tasks	216
Putting it all together	220
Using a deferred library	230
Pull queues	234
Summary	236
Chapter 8: Reaching out, Sending E-mails	237
About e-mails	237
Sending e-mails	239
The object-oriented API	242
E-mail on the development console	244
Headers	245
Receiving e-mails	246
Handling bounce notifications	250
Putting it all together	252
Summary	262

Chapter 9: Working with the Google App Engine Services	265
Memcache	266
The Memcache operations	267
Memcache in Google App Engine	269
The Memcache client	269
The object-oriented client	272
Multi-tenancy	273
Automatically setting the namespace	275
The API-specific notes	276
The Datastore	276
Memcache	276
Task queues	276
Search	277
Blobstore	277
Blobs	277
Uploads	278
Getting BlobInfo	281
More BlobInfo methods	283
Serving	284
Reading	287
Users	287
Storing users in datastore	291
Images	292
Putting it all together	294
Summary	303
Chapter 10: Application Deployment	305
Deployment configurations	305
Deployment revisited	306
Versions	307
The instance classes	308
Instance addressability	309
Scaling types	310
Manual scaling	310
Basic scaling	312
Automatic scaling	313
Modules	315
Accessing the modules	320
The dispatch.yaml file	321
Scheduled tasks	322
The Scheduled tasks format	324
Protecting cron handling URLs	327

Logs	328
The Remote API	332
AppStats	333
Summary	335
Index	337

Preface

Google App Engine is a Platform as a Service that builds and runs applications on Google's infrastructure. App Engine applications are easy to build, maintain, and scale.

Google App Engine allows you to develop highly scalable web applications or backends for mobile applications without worrying about the system administration's plumbing or hardware provisioning issues. You can just focus on writing your business logic, which is the meat of the application, and let Google's powerful infrastructure scale it to thousands of requests per second and millions of users without any effort on your part.

This book introduces you to cloud computing, managed Platform as a Service, the things that Google has to offer, and the advantages. It also introduces you to a sample app that will be built during the course of the book. It will be a small invoice management application where we have clients, products, categories, invoices, and payments as a sample SaaS application. The most complex part is that of reporting, as datastore has certain limitations on this.

What this book covers

Chapter 1, Understanding the Runtime Environment, explains the runtime environment, how requests are processed and handled, and how App Engine scales. This chapter also explores the limitations of runtime environments with respect to the request time and response size, among other factors.

Chapter 2, Handling Web Requests, introduces ways to handle web requests by using a built-in framework or Django and others. It also discusses how to serve static files and caching issues, render templates.

Chapter 3, Understanding the Datastore, covers the problem of storing huge amounts of data and processing it in bulk with the ability to randomly access it. This chapter explains the datastore in detail, which is built on top of Bigtable.

Chapter 4, Modeling Your Data, explains the new `ndb` Python library on top of Google datastore. It will also teach you how to model your data using its API.

Chapter 5, Queries, Indexes, and Transactions, focuses on how to query your data, the limitations, and ways to work around these limitations.

Chapter 6, Integrating Search, builds upon the datastore and shows how to make data searchable.

Chapter 7, Using Task Queues, introduces the reader to task queues, which enable the background repeated execution of tasks.

Chapter 8, Reaching out, Sending E-mails, talks about how the app can send and receive e-mails and how to handle bounce notifications.

Chapter 9, Working with the Google App Engine Services, introduces you to the other services that are provided by Google App Engine to make you aware of your available options.

Chapter 10, Application Deployment, talks in detail about deploying the GAE apps.

What you need for this book

In order to run the code demonstrated in this book, you need an interpreter that comes with the Python 2.7.x series and the latest Google App Engine SDK release of the 1.9.x series.

Additionally, to access the example application, once it runs on App Engine, you need a recent version of a web browser such as Google Chrome, Mozilla Firefox, Apple Safari, or Microsoft Internet Explorer.

Who this book is for

If you have been developing web applications in Python or any other dynamic language but have always been wondering how to write highly scalable web applications without getting into system administration and other areas that plumbing, this is the book for you. We will assume that you have no experience of writing scalable applications. We will help you build your skill set to a point where you can fully leverage the environment and services of Google App Engine, especially the highly distributed NoSQL datastore, to neatly knit and jot down a very robust and scalable solution for your users, be it a web application or a backend for your next killer mobile app.

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:


```
class Person(ndb.Model):
    name = ndb.StringProperty()
    age = ndb.IntegerProperty()
```

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

```
$ appcfg update /path/to/my/app/containing/app.yaml/
```

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: "Now, double-click on the **Launcher** icon that you just dragged to the Applications folder".

[ 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/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.

1

Understanding the Runtime Environment

In this chapter, we will look at the runtime environment that is offered by Google App Engine. Overall, a few details of the runtime environment pertaining to the infrastructure remain the same no matter which runtime environment—Java, Python, Go, or PHP—you opt for.

From all the available runtimes, Python is the most mature one. Therefore, in order to master Google App Engine, we will focus on Python alone. Many of the details vary a bit, but in general, runtimes have a commonality. Having said that, the other runtimes are catching up as well and all of them (including Java, PHP, and Go) are out of their respective beta stages.

Understanding the runtime environment will help you have a better grasp of the environment in which your code executes and you might be able to tweak code in accordance and understand why things behave the way they behave.

In this chapter, we will cover the following topics:

- The overall architecture
- Runtime environments
- Anatomy of a Google App Engine application
- A quick overview of the available services
- Setting up the development tools and writing a basic application

The overall architecture

The scaling of a web application is a hard thing to do. Serving a single page to a single user is a simple matter. Serving thousands of pages to a single or a handful of users is a simple matter, too. However, delivering just a single page to tens of thousands of users is a complex task. To better understand how Google App Engine deals with the problem of scale, we will revisit the whole problem of scaling in next chapter's, how it has been solved till date and the technologies/techniques that are at work behind the scenes. Once armed with this understanding, we will talk about how Google App Engine actually works.

The challenge of scale

The whole problem of complexity arises from the fact that to serve a simple page, a certain amount of time is taken by the machine that hosts the page. This time usually falls in milliseconds, and eventually, there's a limit to the number of pages that can be rendered and served in a second. For instance, if it takes 10 milliseconds to render a page on a 1 GHz machine, this means that in one second, we can serve 100 pages, which means that at a time, roughly 100 users can be served in a second.

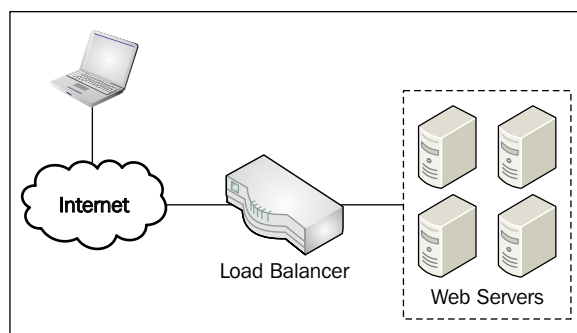
However, if there are 300 users per second, we're out of luck as we will only be able to serve the first 100 lucky users. The rest will get time-out errors, and they may perceive that our web page is not responding, as a rotating wait icon will appear on the browser, which will indicate that the page is loading.

Let's introduce a term here. Instead of pages per second, we will call it requests or queries per second, or simply **Queries Per Second (QPS)**, because users pointing the browser to our page is just a request for the page.

How to scale with the scale?

We have two options here. The first option is to bring the rendering time down from 10 milliseconds to 5 milliseconds, which will effectively help us serve double the number of users. This path is called **optimization**. It has many techniques, which involve minimizing disk reads, caching computations instead of doing on the fly, and all that varies from application to application. Once you've applied all possible optimizations and achieved a newer and better page rendering time, further reduction won't be possible, because there's always a limit to how much we can optimize things and there always will be some overhead. Nothing comes for free.

The other way of scaling things up will be to put more hardware. So, instead of a 1 GHz machine, we can put a 2 GHz machine. Thus, we effectively doubled the number of requests that are processed from 100 to 200 QPS. So now, we can serve 200 users in a second. This method of scaling is called **vertical scaling**. However, yet again, vertical scaling has its limits because you can put a 3 GHz processor, then a 3.5 GHz one, or maybe clock it to a 4.8 GHz one, but finally, the clock frequency has some physical limits that are imposed by how the universe is constructed, and we'll hit the wall sooner or later. The other way around is that instead of putting a single 1 GHz machine, we can put two such machines and a third one in front. Now, when a request comes to the third front-end machine, we can distribute it to either of the other two machines in an alternate fashion, or to the machine with the least load. This request distribution can have many strategies. It can be as simple as a random selection between the two machines, or **round-robin fashion** one after the other or delegating request to the least loaded machine or we may even factor in the past response times of the machines. The main idea and beauty of the whole scheme is that we are no more limited by the limitations of the hardware. If a 1 GHz machine serves 100 users, we can put 10 such machines to serve 1000 users. To serve an audience of 1 million users, we will need ten thousand machines. This is exactly how Google, Facebook, Twitter, and Amazon handle tens of millions of users. The image shows the process of load balancer:



Load balancer splitting the load among machines.

A critical and enabling component here is the machine at front called **load balancer**. This machine runs the software that receives requests and delegates them to the other machines. Many web servers such as Ngnix and Apache come with load-balancing capabilities and require configurations for activating load balancing. The HAProxy is another open source load balancer that has many algorithms at its disposal, which are used to distribute load among the available servers.

A very important aspect of this scaling magic is that each machine, when added to the network, must respond in a manner that is consistent with the responses of the other machines of the cluster. Otherwise, users will have an inconsistent experience, that is, they might see something different when routed to one machine and something else when routed to another machine. For this to happen, even if the operating system differs (consider an instance where the first machine runs on Ubuntu with Cpython and the second one runs on CentOS with Jython), the output produced by each node should be exactly the same. In order to keep things simple, each machine usually has an exactly identical OS, set of libraries, and configurations.

Scaling in practice

Now that you have a load balancer and two servers and you're able to ramp up about 200 QPS (200 users per second), what happens when your user base grows to about 500 people? Well, it's simple. You have to repeat the following process:

1. Go to a store and purchase three more machines.
2. Put them on racks and plug in the network and power cables.
3. Install an OS on them.
4. Install the required languages/runtimes such as Ruby or Python.
5. Install libraries and frameworks, such as Rails or Django.
6. Install components such as web servers and databases.
7. Configure all of software.
8. Finally, add the address of the new machines to the load balancer configuration so that it can start delegating requests from users to machines as well.

You have to repeat the same process for all the three machines that you purchased from the store.

So, in this way, we scaled up our application, but how much time did it take us to do that all? The setting up of the server cables took about 10 minutes, the OS installation another 15 minutes, and the installation of the software components consumed about 40 minutes. So approximately, it took about 1 hour and 5 minutes to add a single node to the machine. Add the three nodes yourself, this amounts to about 4 hours and 15 minutes, that too if you're efficient enough and don't make a mistake along the way, which may make you go back and trace what went wrong and redo the things. Moreover, the sudden spike of users may be long gone by then, as they may feel frustrated by a slow or an unresponsive website. This may leave your newly installed machines idle.

Infrastructure as a Service

This clunky game of scaling was disrupted by another technology called virtualization, which lets us emulate a virtual machine on top of an operating system. Now that you have a virtual machine, you can install another operating system on this virtual machine. You can have more than one virtual machine on a single physical machine if your hardware is powerful enough, which usually is the case with server-grade machines. So now, instead of wiring a physical machine and installing the required OS, libraries, and so on, you can simply spin a virtual machine from a binary image that contains an OS and all the required libraries, tools, software components, and even your application code, if you want. Spinning such a machine requires few minutes (usually about 40 to 150 seconds). So, this is a great time-saving technique, as it cuts down the time requirement from one and a half hour to a few minutes.

Virtualization has created a multibillion-dollar industry. It is a whole new cool term that is related to Cloud computing for consultants of all sorts, and it is used to furnish their resumes. The idea is to put hundreds of servers on racks with virtualization enabled, let the users spin the virtual machines of their desired specs and charge them based on the usage. This is called **Infrastructure as a Service (IaaS)**. Amazon, Rackspace, and Digital Ocean are the prime examples of such models.

Platform as a Service

Although Infrastructure as a Service gives a huge boost in building scalable applications, it still leaves a lot of room for improvements because you have to take care of the OS, required libraries, tools, security updates, the load balancing and provisioning of new machine instances, and almost everything in between. This limitation or problem leads to another solution called **Platform as a Service (PaaS)**, where right from the operating system to the required runtime, libraries and tools are preinstalled and configured for you. All that you have to do is push your code, and it will start serving right away. Google App Engine is such a platform where everything else is taken care of and all that you have to worry about is your code and what your app is supposed to do.

However, there's another major difference between IaaS and PaaS. Let's see what the difference is.

Containers

We talked about scaling by adding new machines to our hosting fleet that was done by putting up new machines on the rack, plugging in the wires, and installing the required software, which was tedious and very time-consuming and took up hours. We then spoke about how virtualization changed the game. You can instantiate a whole new (virtual) machine in a few minutes, possibly from an existing disk image, so that you don't have to install anything. This is indeed a real game changer.

However, the machine is slow at the Internet scale. You may have a sudden increase in the traffic and you might not be able to afford waiting for a few minutes to boot new instances. There's a faster way that comes from a few special features in the Linux kernel, where each executing process can have its own allocated and dedicated resources. What this abstract term means is that each process gets its own partition of the file systems, CPU, and memory share. This process is completely isolated from the other processes. Hence, it is executed in an isolated container. Then, for all practical purposes, this containment actually works as a virtual machine. An overhead of creating such an environment merely requires spinning a new process, which is not a matter of minutes but of a few seconds.

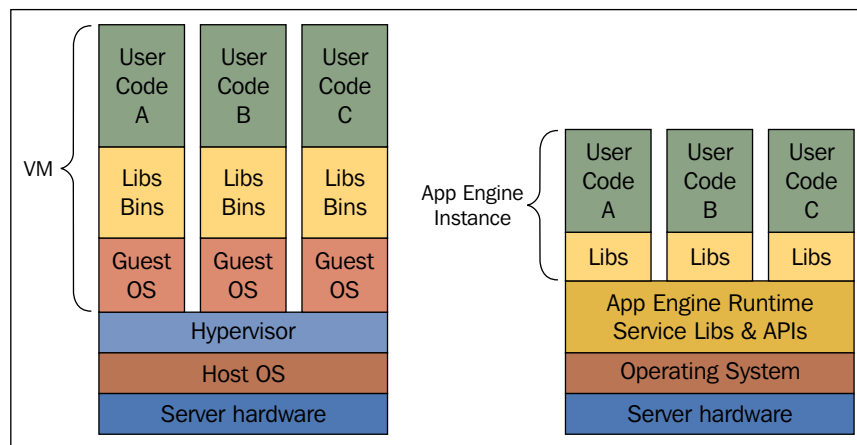
Google App Engine uses containment technology instead of virtualization to scale up the things. Hence, it is able to respond much faster than any IaaS solution, where they have to load a whole new virtual machine and then the whole separate operating system on top of an existing operating system along with the required libraries.

The containers use a totally different approach towards virtualization. Instead of emulating the whole hardware layer and then running an operating system on top of it, they actually are able to provide each running process a totally different view of the system in terms of file system, memory, network, and CPU. This is mainly enabled by **cgroups** (short for **control groups**). A kernel feature was developed by the engineers at Google in 2006 and later, it was merged into Linux kernel 2.6.24, which allows us to define an isolated environment and perform resource accounting for processes.

A container is just a separation of resources, such as file system, memory, and other resources. This is somewhat similar to chroot on Linux/Unix systems which changes the apparent root directory for the current running process and all of its parent-child. If you're familiar with it, you can change the system that you're working on, or simply put, you can replace the hard drive of your laptop with a hard drive from another laptop with identical hardware but a different operating system and set of programs. Hence, the mechanism helps to run totally different applications in each container. So, one container might be running **LAMP stack** and another might be running **node.js** on the same machine that runs at bare metal at native speed with no overhead.

This is called operating system virtualization and it's a vast subject in itself. Much more has been built on top of cgroups, such as **Linux Containers (LXC)** and Docker on top of LXC or using `libvirt`, but recently, docker has its own library called `libcontainer`, which sits directly on top of cgroups. However, the key idea is process containment, which results in a major reduction of time. Eventually, you will be able to spin a new *virtual machine* in a few seconds, as it is just about launching another ordinary Linux process, although contained in terms of what and how it sees the underlying system.

A comparison of virtual machines versus application containers (App Engine instances in our case) can be seen in the following diagram:

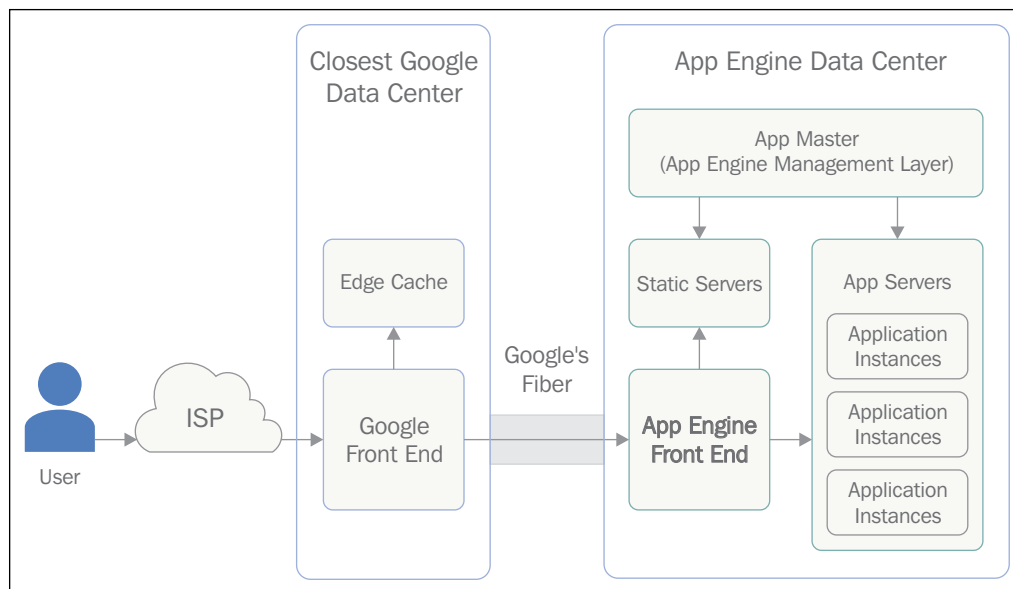


Virtualization vs container based App Engine machine instances.

How does App Engine scales?

Now that we understand many of the basic concepts behind how web applications can be scaled and the technologies that are at work, we can now examine how App Engine scales itself. When a user navigates to your app using their browser, the first thing that receives the users are the Google front end servers. These servers determine whether it is a request for App Engine (mainly by examining the HTTP Host header), and if it is, they are handed over to the **App Engine server**.

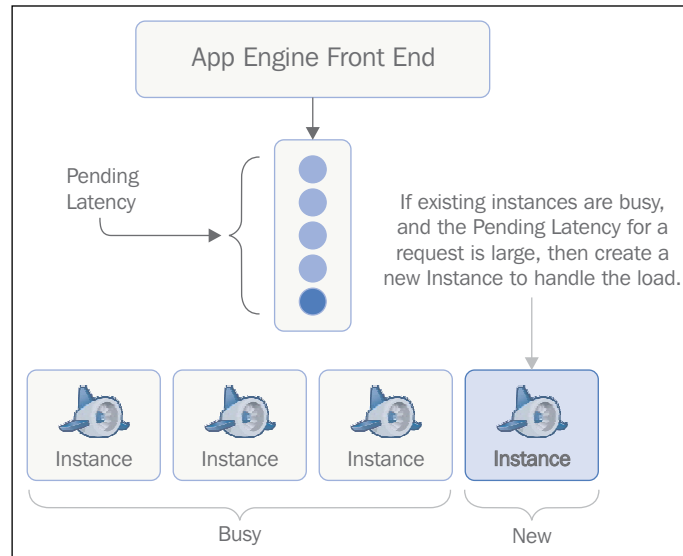
The App Engine server first determines whether this is a request for a static resource, and if that's the case, it is handed over to the static file servers, and the whole process ends here. Your application code never gets executed if a static resource is requested such as a JavaScript file or a CSS stylesheet. The following image shows the cycle of Google App Engine server request process:



Google App Engine Journey of a request.

However, in case the request is dynamic, the App Engine server assigns it a unique identifier based on the time of receiving it. It is entered into a request queue, where it shall wait till an instance is available to serve it, as waiting might be cheaper than spinning a new instance altogether. As we talked about in the section on containers, these instances are actually containers and just isolated processes. So eventually, it is not as costly as launching a new virtual machine altogether. There are a few parameters here that you can tweak, which are accessible from the application performance settings once you've deployed. One is the minimum latency. It is the minimum amount of time a request should wait in the queue. If you set this value to a higher number, you'll be able to serve more requests with fewer instances but at the cost of more latency, as perceived by the end user. App Engine will wait till the time that is specified as minimum latency and then, it will hand over the request to an existing instance. The other parameter is maximum latency, which specifies the maximum time for which a request can be held in the request queue, after which, App Engine will spin a new instance if none is available and pass the request to it. If this value is too low, App Engine will spin more instances, which will result in an increase in cost but much less latency, as experienced by the end user.

However by default, if you haven't tweaked the default settings. (we'll see how to do this in the *Chapter 10, Application Deployment*) Google App Engine will use heuristics to determine whether it should spin a new instance based on your past request history and patterns.



App Engine: Request, Request queues and Instances.

The last but a very important component in the whole scheme of things is the App Engine master. This is responsible for updates, deployments, and the versioning of the app. This is the component that pushes static resources to static servers and code to application instances when you deploy an application to App Engine.

Available runtimes

You can write web applications on top of Google App Engine in many programming languages, and your choices include Python, Java, Go, and PHP. For Python, two versions of runtimes are available, we will focus on the latest version.

Let's briefly look at each of the environments.

Python

The most basic and important principle of all runtime environments, including that of Python, is that you can talk to the outside world only by going through Google's own services. It is like a completely sealed and contained sandbox where you are not allowed to write to the disk or to connect to the network. However, no program will be very useful in that kind of isolation. Therefore, you can definitely talk to the outside world but only through the services provided by the App Engine. You can also ship your own code and libraries but they must all be in pure Python code and no C extensions are allowed. This is actually a limitation and tradeoff to ensure that the containers are always identical. Since no external libraries are allowed, it can be ensured that the minimal set of native required libraries is always present on the instance.

At the very beginning, App Engine started with the Python runtime environment, and version 2.5 was the one that was available for you. It had a few external libraries too, and it provided a CGI environment for your web app to talk to the world. That is, when a web request comes in, the environment variables are set from the request, the body goes to `stdin` and the Python interpreter invoked with given program. It is up to your program to then handle and respond to the request. This runtime environment is now deprecated.

Later, the Python 2.7 runtime environment came along, with new language features and updated shipped libraries. A major departure from the Python 2.5 runtime environment was not only the language version, but also a switch from CGI to WSGI. Because of this switch, it became possible for web apps to process requests concurrently. This boosted the overall throughput per instance. We will examine CGI and WSGI in detail in the next chapter.

The Java runtime environment

Java runtime environment presents a standard Servlet version 2.5 environment, and there are two language versions available—Java 5 and Java 6. The Java 6 runtime environment is deprecated and will be soon removed. The Java 6 runtime environment will be replaced and new applications users can only be able to use Java 7. The `app.xml` is a file that defines your application, and you have various standard Java APIs available to talk to Google services, such as JPA for persistence, Java Mail for mail, and so on.

This runtime environment is also capable of handling concurrent requests.

Go

This runtime environment uses the new Go programming language from Google. It is a CGI environment too, and it's not possible to handle concurrent requests, the applications are written in Go version 1.4.

PHP

This is a preview platform, and the PHP interpreter is modified to fit in the scalable environment with the libraries patched, removed, or the individual functions disabled. You get to develop applications just as you would do for any normal PHP web application, but there are many limitations. Many of the standard library modules are either not available, or are partially functional, the applications are written in PHP version 5.5.

The structure of an application

When you are developing a web application that has to be hosted on Google App Engine, it has to have a certain structure so that the platform can deploy it. A minimal App Engine application is composed of an application manifest file called `app.yaml` and at least one script / code file that handles and responds to requests. The `app.yaml` file defines the application ID, version of the application, required runtime environment and libraries, static resources, if any, and the set of URLs along with their mappings to the actual code files that are responsible for their processing.

So eventually, if you look at the minimum application structure, it will comprise only the following two files:

- `app.yaml`
- `main.py`

Here, `app.yaml` describes the application and set of URLs to the actual code files mappings. We will examine `app.yaml` in greater detail in a later section. The `app.yaml` is not the only file that makes up your application. There are a few other optional configuration files as well. In case you are using datastore, there may be another file called `index.yaml`, which lists the kind of indexes that your app will require. Although you can edit this file, it is automatically generated for you, as your application runs queries locally.

You then might have a `crons.yaml` file as well, that describes various repeated tasks. The `queus.yaml` file describes your queue configurations so that you can queue in long running tasks for later processing. The `dos.yaml` is the file that your application might define to prevent DoS attacks.

However, most importantly, your application can have one or more logical modules, where each module will run on a separate instance and might have different scaling characteristics. So, you can have a module defined by `api.yaml` that handles your API calls, and its scaling type is set to **automatic** so that it responds to requests according to the number of consumers. Another named `backend.yaml` handles various long running tasks, and its scaling type is set to **manual** with 5 instances on standby, which will keep running all the time to handle whatever the long running tasks handled to them.

We will take a look at modules later in this book when discussing deployment options in *Chapter 10, Application Deployment*.

The available services

By now, you probably understand the overall architecture and atmosphere in which our app executes, but it won't be of much use without more services available at our disposal. Otherwise, with the limitation of pure Python code, we might have to bring everything that is required along with us to build the next killer web app.

To this end, Google App Engine provides many useful scalable services that you can utilize to build app. Some services address storage needs, others address the processing needs of an app, and yet, the other group caters to the communication needs. In a nutshell, the following services are at your disposal:

- **Storage:** Datastore, Blobstore, Cloud SQL, and Memcache
- **Processing:** Images, Crons, Tasks, and MapReduce
- **Communication:** Mail, XMPP, and Channels
- **Identity and security:** Users, OAuth, and App Identity
- **Others:** such as various capabilities, image processing and full text search

If the list seems short, Google constantly keeps adding new services all the time. Now, let's look at each of the previously listed services in detail.

Datastore

Datastore is a NoSQL, distributed, and highly scalable column based on a storage solution that can scale to petabytes of data so that you don't have to worry about scaling at all. App Engine provides a data modeling library that you can use to model your data, just as you would with any **Object Relational Mapping (ORM)**, such as the Django models or SQL Alchemy. The syntax is quite similar, but there are differences.

Each object that you save gets a unique key, which is a long string of bytes. Its generation is another topic that we will discuss later. Since it's a NoSQL solution, there are certain limitations on what you can query, which makes it unfit for everyday use, but we can work around those limitations, as we will explore in the coming chapters.

By default, apps get 1 GB of free space in datastore. So, you can start experimenting with it right away.

Google Cloud SQL

If you prefer using a relational database, you can have that too. It is a standard MySQL database, and you have to boot up instances and connect with it via whatever interface is available to your runtime environment, such as JDBC in case of Java and MySQLdb in case of Python. Datastore comes with a free quota of about 1 GB of data, but for Cloud SQL, you have to pay from the start.

Because dealing with MySQL is a topic that has been explored in much detail from blog posts to articles and entire books have been written on the subject, this book skips the details on this, it focuses more on Google Datastore.

The Blobstore

Your application might want to store larger chunks of data such as images, audio, and video files. The Blobstore just does that for you. You are given a URL, which has to be used as the target of the upload form. Uploads are handled for you, while a key of the uploaded file is returned to a specified callback URL, which can be stored for later reference. For letting users download a file, you can simply set the key that you got from the upload as a specific header on your response, which is taken as an indication by the App Engine to send the file contents to the user.

Memcache

Hitting datastore for every request costs time and computational resources. The same goes for the rendering of templates with a given set of values. Time is money. Time really is money when it comes to cloud, as you pay in terms of the time your code spends in satisfying user requests. This can be reduced by caching certain content or queries that occur over and over for the same set of data. Google App Engine provides you with memcache to play with so that you can supercharge your app response.

When using App Engine's Python library to model data and query, the caching of the data that is fetched from datastore is automatically done for you, which was not the case in the previous versions of the library.

Scheduled Tasks

You might want to perform some certain tasks at certain intervals. That's where the scheduled tasks fit in. Conceptually, they are similar to the Linux/UNIX Cron jobs. However, instead of specifying commands or programs, you indicate URLs, which receive the HTTP GET requests from App Engine on the specified intervals. You're required to process your stuff in under 10 minutes. However, if you want to run longer tasks, you have that option too by tweaking the scaling options, which will be examined in the last chapter when we examine deployment.

Queues Tasks

Besides the scheduled tasks, you might be interested in the background processing of tasks. For this, Google App Engine allows you to create tasks queues and enqueue tasks in them specifying a target URL with payload, where they are dispatched on a specified and configurable rate. Hence, it is possible to asynchronously perform various computations and other pieces of work that otherwise cannot be accommodated in request handlers.

App Engine provides two types of queues—push queues and pull queues. In push queues, the tasks are delivered to your code via the URL dispatch mechanism, and the only limitation is that you must execute them within the App Engine environment. On the other hand, you can have pull requests where it's your responsibility to pull tasks and delete them once you are done. To that end, pull tasks can be accessed and processed from outside Google App Engine. Each task is retried with backoffs if it fails, and you can configure the rate at which the tasks get processed and configure this for each of the task queues or even at the individual task level itself. The task retries are only available for push queues and for pull queues, you will have to manage repeated attempts of failed tasks on your own.

Each app has a default task queue, and it lets you create additional queues, which are defined in the `queues.yaml` file. Just like the scheduled tasks, each task is supposed to finish its processing within 10 minutes. However, if it takes longer than this, we'll learn how to accommodate such a situation when we examine application deployment in the last chapter.

MapReduce

MapReduce is a distributed computing paradigm that is widely used at Google to crunch exotic amounts of data, and now, many open source implementations of such a model exist, such as Hadoop. App Engine provides the MapReduce functionality as well, but at the time of writing this book, Google has moved the development and support of MapReduce libraries for Python and Java to Open source community and they are hosted on Github. Eventually, these features are bound to change a lot. Therefore, we'll not cover MapReduce in this book but if you want to explore this topic further, check <https://github.com/GoogleCloudPlatform/appengine-mapreduce/wiki> for further details.

Mail

Google is in the mail business. So, your applications can send mails. You can not only send e-mails, but also receive them as well. If you plan to write your app in Java, you will use JavaMail as the API to send emails. You can of course use third-party solutions as well to send email, such as SendGrid, which integrates nicely with Google App Engine. If you're interested in this kind of solution, visit <https://cloud.google.com/appengine/docs/python/mail/sendgrid>.

XMPP

It's all about instant messaging. You may want to build chat features in your app or use in other innovative ways, such as notifying users about a purchase as an instant message or anything else whereas for that matter. XMPP services are at your disposal. You can send a message to a user, whereas your app will receive messages from users in the form of HTTP POST requests of a specific URL. You can respond to them in whatever way you see fit.

Channels

You might want to build something that does not work with the communication model of XMPP, and for this, you have channels at your disposal. This allows you to create a persistent connection from one client to the other clients via Google App Engine. You can supply a client ID to App Engine, and a channel is opened for you. Any client can listen on this channel, and when you send a message to this channel, it gets pushed to all the clients. This can be useful, for instance, if you wish to inform about the real-time activity of other users, which is similar to you notice on Google Docs when editing a spreadsheet or document together.

Users

Authentication is an important part of any web application. App Engine allows you to generate URLs that redirect users to enter their Google account credentials (yourname@gmail.com) and manage sessions for you. You also have the option of restricting the sign-in functionality for a specific domain (such as yourname@yourcompany.com) in case your company uses Google Apps for business and you intend to build some internal solutions. You can limit access to the users on your domain alone.

OAuth

Did you ever come across a button labeled **Sign in with Facebook**, **Twitter**, **Google**, and **LinkedIn** on various websites? Your app can have similar capabilities as well, where you let users not only use the credentials that they registered with on your website, but also sign in to others. In technical jargon, Google Engine can be an OAuth provider.

Writing and deploying a simple application

Now that you understand how App Engine works and the composition of an App Engine app, it's time to get our hands on some real code and play with it. We will use Python to develop applications, and we've got a few reasons to do so. For one, Python is a very simple and an easy-to-grasp language. No matter what your background is, you will be up and running it quickly. Further, Python is the most mature and accessible runtime environment because it is available since the introduction of App Engine, Further almost all new experimental and cutting-edge services are first introduced for Python runtime environment before they make their way to other runtimes.