O'REILLY®

Programming Google App Engine with Java

BUILD & RUN SCALABLE JAVA APPS ON GOOGLE'S INFRASTRUCTURE

Dan Sanderson

Programming Google App Engine with Java

Dan Sanderson

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Programming Google App Engine with Java
by Dan Sanderson

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Meghan Blanchette and Brian Anderson
Acquisition Editor: Mike Loukides
Production Editors: Colleen Lobner and Kara Ebrahim
Copyeditor: Jasmine Kwityn
Proofreader: Charles Roumeliotis
Indexer: Judy McConville
Interior Designer: David Futato
Cover Designer: Ellie Volckhausen
Illustrator: Rebecca Demarest

July 2015: First Edition

Revision History for the First Edition
2015-06-26: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491900208* for release details.

978-1-491-90020-8

[LSI]

Table of Contents

CHAPTER 1

Introducing Google App Engine

Google App Engine is a web application hosting service. By "web application," we mean an application or service accessed over the Web, usually with a web browser: storefronts with shopping carts, social networking sites, multiplayer games, mobile applications, survey applications, project management, collaboration, publishing, and all the other things we're discovering are good uses for the Web. App Engine can serve traditional website content too, such as documents and images, but the environment is especially designed for real-time dynamic applications. Of course, a web browser is merely one kind of client: web application infrastructure is well suited to mobile applications, as well.

In particular, Google App Engine is designed to host applications with many simultaneous users. When an application can serve many simultaneous users without degrading performance, we say it *scales*. Applications written for App Engine scale automatically. As more people use the application, App Engine allocates more resources for the application and manages the use of those resources. The application itself does not need to know anything about the resources it is using.

Unlike traditional web hosting or self-managed servers, with Google App Engine, you only pay for the resources you use. Billed resources include CPU usage, storage per month, incoming and outgoing bandwidth, and several resources specific to App Engine services. To help you get started, every developer gets a certain amount of resources for free, enough for small applications with low traffic.

App Engine is part of Google Cloud Platform, a suite of services for running scalable applications, performing large amounts of computational work, and storing, using, and analyzing large amounts of data. The features of the platform work together to host applications efficiently and effectively, at minimal cost. App Engine's specific role on the platform is to host web applications and scale them automatically. App Engine apps use the other services of the platform as needed, especially for data storage.

An App Engine web application can be described as having three major parts: application instances, scalable data storage, and scalable services. In this chapter, we look at each of these parts at a high level. We also discuss features of App Engine for deploying and managing web applications, and for building websites integrated with other parts of Google Cloud Platform.

The Runtime Environment

An App Engine application responds to web requests. A web request begins when a client, typically a user's web browser, contacts the application with an HTTP request, such as to fetch a web page at a URL. When App Engine receives the request, it identifies the application from the domain name of the address, either a custom domain name you have registered and configured for use with the app, or an *.appspot.com* subdomain provided for free with every app. App Engine selects a server from many possible servers to handle the request, making its selection based on which server is most likely to provide a fast response. It then calls the application with the content of the HTTP request, receives the response data from the application, and returns the response to the client.

From the application's perspective, the runtime environment springs into existence when the request handler begins, and disappears when it ends. App Engine provides several methods for storing data that persists between requests, but these mechanisms live outside of the runtime environment. By not retaining state in the runtime environment between requests—or at least, by not expecting that state will be retained between requests—App Engine can distribute traffic among as many servers as it needs to give every request the same treatment, regardless of how much traffic it is handling at one time.

In the complete picture, App Engine allows runtime environments to outlive request handlers, and will reuse environments as much as possible to avoid unnecessary initialization. Each instance of your application has local memory for caching imported code and initialized data structures. App Engine creates and destroys instances as needed to accommodate your app's traffic. If you enable the multithreading feature, a single instance can handle multiple requests concurrently, further utilizing its resources.

Application code cannot access the server on which it is running in the traditional sense. An application can read its own files from the filesystem, but it cannot write to files, and it cannot read files that belong to other applications. An application can see environment variables set by App Engine, but manipulations of these variables do not necessarily persist between requests. An application cannot access the networking facilities of the server hardware, although it can perform networking operations by using services.

In short, each request lives in its own "sandbox." This allows App Engine to handle a request with the server that would, in its estimation, provide the fastest response. For web requests to the app, there is no way to guarantee that the same app instance will handle two requests, even if the requests come from the same client and arrive relatively quickly.

Sandboxing also allows App Engine to run multiple applications on the same server without the behavior of one application affecting another. In addition to limiting access to the operating system, the runtime environment also limits the amount of clock time and memory a single request can take. App Engine keeps these limits flexible, and applies stricter limits to applications that use up more resources to protect shared resources from "runaway" applications.

A request handler has up to 60 seconds to return a response to the client. While that may seem like a comfortably large amount for a web app, App Engine is optimized for applications that respond in less than a second. Also, if an application uses many CPU cycles, App Engine may slow it down so the app isn't hogging the processor on a machine serving multiple apps. A CPU-intensive request handler may take more clock time to complete than it would if it had exclusive use of the processor, and clock time may vary as App Engine detects patterns in CPU usage and allocates accordingly.

Google App Engine provides four possible runtime environments for applications, one for each of four programming languages: Java, Python, PHP, and Go. The environment you choose depends on the language and related technologies you want to use for developing the application.

The Java environment runs applications built for the Java 7 Virtual Machine (JVM). An app can be developed using the Java programming language, or most other languages that compile to or otherwise run in the JVM, such as PHP (using Quercus), Ruby (using JRuby), JavaScript (using the Rhino interpreter), Scala, Groovy, and Clojure. The app accesses the environment and services by using interfaces based on web industry standards, including Java servlets and the Any Java technology that functions within the sandbox restrictions can run on App Engine, making it suitable for many existing frameworks and libraries.

Similarly, the Python, PHP, and Go runtime environments offer standard execution environments for those languages, with support for standard libraries and third-party frameworks.

All four runtime environments use the same application server model: a request is routed to an app server, an application instance is initialized (if necessary), application code is invoked to handle the request and produce a response, and the response is returned to the client. Each environment runs application code within sandbox

restrictions, such that any attempt to use a feature of the language or a library that would require access outside of the sandbox returns an error.

You can configure many aspects of how instances are created, destroyed, and initialized. How you configure your app depends on your need to balance monetary cost against performance. If you prefer performance to cost, you can configure your app to run many instances and start new ones aggressively to handle demand. If you have a limited budget, you can adjust the limits that control how requests queue up to use a minimum number of instances.

I haven't said anything about which operating system or hardware configuration App Engine uses. There are ways to figure this out with a little experimentation, but in the end it doesn't matter: the runtime environment is an abstraction *above* the operating system that allows App Engine to manage resource allocation, computation, request handling, scaling, and load distribution without the application's involvement. Features that typically require knowledge of the operating system are either provided by services outside of the runtime environment, provided or emulated using standard library calls, or restricted in sensible ways within the definition of the sandbox.

Everything stated above describes how App Engine allocates application instances dynamically to scale with your application's traffic. In addition to a flexible bank of instances serving your primary traffic, you can organize your app into multiple "modules." Each module is addressable individually using domain names, and can be configured with its own code, performance characteristics, and scaling pattern—including the option of running a fixed number of always-on instances, similar to traditional servers. In practice, you usually use a bank of dynamically scaling instances to handle your "frontend" traffic, then establish modules as "backends" to be accessed by the frontends for various purposes.

The Static File Servers

Most websites have resources they deliver to browsers that do not change during the regular operation of the site. The images and CSS files that describe the appearance of the site, the JavaScript code that runs in the browser, and HTML files for pages without dynamic components are examples of these resources, collectively known as *static files*. Because the delivery of these files doesn't involve application code, it's unnecessary and inefficient to serve them from the application servers.

Instead, App Engine provides a separate set of servers dedicated to delivering static files. These servers are optimized for both internal architecture and network topology to handle requests for static resources. To the client, static files look like any other resource served by your app.

You upload the static files of your application right alongside the application code. You can configure several aspects of how static files are served, including the URLs

for static files, content types, and instructions for browsers to keep copies of the files in a cache for a given amount of time to reduce traffic and speed up rendering of the page.

Frontend Caches

All App Engine traffic goes through a set of machines that know how to cache responses to requests. If a response generated by the app declares that another request with the same parameters should return the same response, the frontend cache stores the response for a period of time. If another matching request comes in, the cache returns the stored response without invoking the application. The resources conserved by exploiting frontend caches can be significant.

App Engine recognizes standard HTTP controls for proxy caches. Do a web search for "HTTP cache control" for more information (`Cache-Control`, `Expires`). By default, responses from an app have `Cache-Control` set to `no-cache`.

The static file servers can also be configured to serve specific cache controls. These are described by a configuration file. (More on that later.)

Cloud Datastore

Most useful web applications need to store information during the handling of a request for retrieval during a later request. A typical arrangement for a small website involves a single database server for the entire site, and one or more web servers that connect to the database to store or retrieve data. Using a single central database server makes it easy to have one canonical representation of the data, so multiple users accessing multiple web servers all see the same and most recent information. But a central server is difficult to scale once it reaches its capacity for simultaneous connections.

By far the most popular kind of data storage system for web applications in the past two decades has been the relational database, with tables of rows and columns arranged for space efficiency and concision, and with indexes and raw computing power for performing queries, especially "join" queries that can treat multiple related records as a queryable unit. Other kinds of data storage systems include hierarchical datastores (filesystems, XML databases) and object databases. Each kind of database has pros and cons, and which type is best suited for an application depends on the nature of the application's data and how it is accessed. And each kind of database has its own techniques for growing past the first server.

Google Cloud Platform offers several kinds of data storage you can use with an App Engine app, including a relational database (Google Cloud SQL). Most scalable apps use Google Cloud Datastore, or as it is known to App Engine veterans, simply "the

datastore."[1] The datastore most closely resembles an object database. It is not a join-query relational database, and if you come from the world of relational database–backed web applications (as I did), this will probably require changing the way you think about your application's data. As with the runtime environment, the design of the App Engine datastore is an abstraction that allows App Engine to handle the details of distributing and scaling the application, so your code can focus on other things.

If it turns out the scalable datastore does not meet your needs for complex queries, you can use Google Cloud SQL, a full-featured relational database service based on MySQL. Cloud SQL is a feature of Google Cloud Platform, and can be called directly from App Engine using standard database APIs. The trade-off comes from how you intend to scale your application. A Cloud SQL instance behaves like a single MySQL database server, and can get bogged down by traffic. Cloud Datastore scales automatically: with proper data design, it can handle as many simultaneous users as App Engine's server instances can.

Entities and Properties

With Cloud Datastore, an application stores its data as one or more datastore *entities*. An entity has one or more *properties*, each of which has a name, and a value that is of one of several primitive value types. Each entity is of a named *kind*, which categorizes the entity for the purpose of queries.

At first glance, this seems similar to a relational database: entities of a kind are like rows in a table, and properties are like columns (fields). However, there are two major differences between entities and rows. First, an entity of a given kind is not required to have the same properties as other entities of the same kind. Second, an entity can have a property of the same name as another entity, but with a different type of value. In this way, datastore entities are "schemaless." As you'll soon see, this design provides both powerful flexibility as well as some maintenance challenges.

Another difference between an entity and a table row is that an entity can have multiple values for a single property. This feature is a bit quirky, but can be quite useful once understood.

Every datastore entity has a unique key that is either provided by the application or generated by App Engine (your choice). Unlike a relational database, the key is not a

1 Historically, the datastore was a feature exclusive to App Engine. Today it is a full-fledged service of Google Cloud Platform, and can be accessed from Compute Engine VMs and from apps outside of the platform using a REST API. App Engine apps can access Cloud Datastore using the App Engine datastore APIs and libraries.

"field" or property, but an independent aspect of the entity. You can fetch an entity quickly if you know its key, and you can perform queries on key values.

An entity's key *cannot* be changed after the entity has been created. The entity's kind is considered part of its key, so the kind cannot be changed either. App Engine uses the entity's key to help determine where the entity is stored in a large collection of servers. (No part of the key guarantees that two entities are stored on the same server, but you won't need to worry about that anyway.)

Queries and Indexes

A datastore query returns zero or more entities of a single kind. It can also return just the keys of entities that would be returned for a query. A query can filter based on conditions that must be met by the values of an entity's properties, and can return entities ordered by property values. A query can also filter and sort using keys.

In a typical relational database, queries are planned and executed in real time against the data tables, which are stored just as they were designed by the developer. The developer can also tell the database to produce and maintain indexes on certain columns to speed up certain queries.

Cloud Datastore does something dramatically different. *Every* query has a corresponding index maintained by the datastore. When the application performs a query, the datastore finds the index for that query, locates the first row that matches the query, then returns the entity for each consecutive row in the index until the first row that doesn't match the query.

Of course, this requires that Cloud Datastore know ahead of time which queries the application is going to perform. It doesn't need to know the values of the filters in advance, but it does need to know the kind of entity to query, the properties being filtered or sorted, and the operators of the filters and the orders of the sorts.

Cloud Datastore provides a set of indexes for simple queries by default, based on which properties exist on entities of a kind. For more complex queries, an app must include index specifications in its configuration. The App Engine developer tools help produce this configuration file by watching which queries are performed as you test your application with the provided development web server on your computer. When you upload your app, the datastore knows to make indexes for every query the app performed during testing. You can also edit the index configuration manually.

When your application creates new entities and updates existing ones, the datastore updates every corresponding index. This makes queries very fast (each query is a simple table scan) at the expense of entity updates (possibly many tables may need updating for a single change). In fact, the performance of an index-backed query is not affected by the number of entities in the datastore, only the size of the result set.

It's worth paying attention to indexes, as they take up space and increase the time it takes to update entities. We discuss indexes in detail in Chapter 7.

Transactions

When an application has many clients attempting to read or write the same data simultaneously, it is imperative that the data always be in a consistent state. One user should never see half-written data or data that doesn't make sense because another user's action hasn't completed.

When an application updates the properties of a single entity, Cloud Datastore ensures that either every update to the entity succeeds all at once, or the entire update fails and the entity remains the way it was prior to the beginning of the update. Other users do not see any effects of the change until the change succeeds.

In other words, an update of a single entity occurs in a *transaction*. Each transaction is *atomic*: the transaction either succeeds completely or fails completely, and cannot succeed or fail in smaller pieces.

An application can read or update multiple entities in a single transaction, but it must tell Cloud Datastore which entities will be updated together when it creates the entities. The application does this by creating entities in *entity groups*. Cloud Datastore uses entity groups to control how entities are distributed across servers, so it can guarantee a transaction on a group succeeds or fails completely. In database terms, the datastore natively supports *local transactions*.

When an application calls the datastore API to update an entity, the call returns only after the transaction succeeds or fails, and it returns with knowledge of success or failure. For updates, this means the service waits for all entities to be updated before returning a result. The application can call the datastore asynchronously, such that the app code can continue executing while the datastore is preparing a result. But the update itself does not return until it has confirmed the change.

If a user tries to update an entity while another user's update of the entity is in progress, the datastore returns immediately with a contention failure exception. Imagine the two users "contending" for a single piece of data: the first user to commit an update wins. The other user must try her operation again, possibly rereading values and calculating the update from fresh data. Contention is expected, so retries are common. In database terms, Cloud Datastore uses *optimistic concurrency control*: each user is "optimistic" that her commit will succeed, so she does so without placing a lock on the data.

Reading the entity never fails due to contention. The application just sees the entity in its most recent stable state. You can also read multiple entities from the same entity group by using a transaction to ensure that all the data in the group is current and consistent with itself.

In most cases, retrying a transaction on a contested entity will succeed. But if an application is designed such that many users might update a single entity, the more popular the application gets, the more likely users will get contention failures. It is important to design entity groups to avoid a high rate of contention failures even with a large number of users.

It is often important to read and write data in the same transaction. For example, the application can start a transaction, read an entity, update a property value based on the last read value, save the entity, and then commit the transaction. In this case, the save action does not occur unless the entire transaction succeeds without conflict with another transaction. If there is a conflict and the app wants to try again, the app should retry the entire transaction: read the (possibly updated) entity again, use the new value for the calculation, and attempt the update again. By including the read operation in the transaction, the datastore can assume that related writes and reads from multiple simultaneous requests do not interleave and produce inconsistent results.

With indexes and optimistic concurrency control, Cloud Datastore is designed for applications that need to read data quickly, ensure that the data it sees is in a consistent form, and scale the number of users and the size of the data automatically. While these goals are somewhat different from those of a relational database, they are especially well suited to web applications.

The Services

The datastore's relationship with the runtime environment is that of a service: the application uses an API to access a separate system that manages all its own scaling needs separately from application instances. Google Cloud Platform and App Engine include several other self-scaling services useful for web applications.

The memory cache (or *memcache*) service is a short-term key-value storage service. Its main advantage over the datastore is that it is fast—much faster than the datastore for simple storage and retrieval. The memcache stores values in memory instead of on disk for faster access. It is distributed like the datastore, so every request sees the same set of keys and values. However, it is not persistent like the datastore: if a server goes down, such as during a power failure, memory is erased. It also has a more limited sense of atomicity and transactionality than the datastore. As the name implies, the memcache service is best used as a cache for the results of frequently performed queries or calculations. The application checks for a cached value, and if the value isn't there, it performs the query or calculation and stores the value in the cache for future use.

Google Cloud Platform provides another storage service specifically for very large values, called Google Cloud Storage.[2] Your app can use Cloud Storage to store, manage, and serve large files, such as images, videos, or file downloads. Cloud Storage can also accept large files uploaded by users and offline processes. This service is distinct from Cloud Datastore to work around infrastructure limits on request and response sizes between users, application servers, and services. Application code can read values from Cloud Storage in chunks that fit within these limits. Code can also query for metadata about Cloud Storage values.

For when you really need a relational database, Google Cloud SQL provides full-featured MySQL database hosting. Unlike Cloud Datastore or Cloud Storage, Cloud SQL does not scale automatically. Instead, you create *SQL instances*, virtual machines running managed MySQL software. Instances are large, and you only pay for the storage you use and the amount of time an instance is running. You can even configure instances to turn themselves off when idle, and reactivate when a client attempts to connect. Cloud SQL can be the basis for an always-on web app, or a part of a larger data processing solution.

Yet another storage service is dedicated to providing full-text search infrastructure, known simply as the Search service.[3] As Cloud Datastore stores entities with properties, the Search service stores *documents* with *fields*. Your app adds documents to *indexes*. Unlike the datastore, you can use the Search service to perform faceted text searches over the fields of the documents in an index, including partial string matches, range queries, and Boolean search expressions. The service also supports stemming and tokenization.

App Engine applications can access other web resources using the URL Fetch service. The service makes HTTP requests to other servers on the Internet, such as to retrieve pages or interact with web services. Because remote servers can be slow to respond, the URL Fetch API supports fetching URLs in the background while a request handler does other things, but in all cases the fetch must start and finish within the request handler's lifetime. The application can also set a deadline, after which the call is canceled if the remote host hasn't responded.

App Engine applications can send email messages using the Mail service. The app can send email on behalf of the application itself or on behalf of the user who made the request that is sending the email (if the message is from the user). Many web applications use email to notify users, confirm user actions, and validate contact information.

2 An earlier version of this service was known as App Engine's "Blobstore" service. Both Blobstore and Cloud Storage are still available, with similar features. For new projects, prefer Cloud Storage. See the book's website for a free bonus chapter about the Blobstore service.

3 The Search service is currently exclusive to App Engine.

An application can also receive email messages. If an app is configured to receive email, a message sent to the app's address is routed to the Mail service, which delivers the message to the app in the form of an HTTP request to a request handler.

App Engine applications can send and receive instant messages to and from chat services that support the XMPP protocol. An app sends an XMPP chat message by calling the XMPP service. As with incoming email, when someone sends a message to the app's address, the XMPP service delivers it to the app by calling a request handler.

You can accomplish real-time two-way communication directly with a web browser using the Channel service, a clever implementation of the Comet model of browser app communication. Channels allow browsers to keep a network connection open with a remote host to receive real-time messages long after a web page has finished loading. App Engine fits this into its request-based processing model by using a service: browsers do not connect directly to application servers, but instead connect to "channels" via a service. When an application decides to send a message to a client (or set of clients) during its normal processing, it calls the Channel service with the message. The service handles broadcasting the message to clients, and manages open connections. Paired with web requests for messages from clients to apps, the Channel service provides real-time browser messaging without expensive polling. App Engine includes a JavaScript client so your code in the browser can connect to channels.

Google Accounts, OpenID, and OAuth

App Engine integrates with Google Accounts, the user account system used by Google applications such as Google Mail, Google Docs, and Google Calendar. You can use Google Accounts as your app's user authentication system, so you don't have to build your own. And if your users already have Google accounts, they can sign in to your app using their existing accounts, with no need to create new accounts just for your app.

Google Accounts is especially useful for developing applications for your company or organization using Google Apps for Work (or Google Apps for Education). With Google Apps, your organization's members can use the same account to access your custom applications as well as their email, calendar, and documents. You can add your App Engine application to a subdomain of your Apps domain from your Google Apps dashboard, just like any other Google Apps feature.

Of course, there is no obligation to use Google Accounts. You can always build your own account system, or use an OpenID provider. App Engine includes special support for using OpenID providers in some of the same ways you can use Google Accounts. This is useful when building applications for the Google Apps Marketplace, which uses OpenID to integrate with enterprise single sign-on services.

App Engine includes built-in support for OAuth, a protocol that makes it possible for users to grant permission to third-party applications to access personal data in another service, without having to share their account credentials with the third party. For instance, a user might grant a mobile phone application access to her Google Calendar account, to read appointment data and create new appointments on her behalf. App Engine's OAuth support makes it straightforward to implement an OAuth service for other apps to use. Note that the built-in OAuth feature only works when using Google Accounts, not OpenID or a proprietary identity mechanism.

There is no special support for implementing an OAuth client in an App Engine app, but most OAuth client libraries work fine with App Engine. For Google services and APIs, the easiest way is to use the Google APIs Client Libraries, which are known to run from App Engine and are available for many languages.

Google Cloud Endpoints

APIs are an essential part of the modern Web. It is increasingly common for browser-based web applications to be implemented as rich JavaScript clients: the user's first visit downloads the client code to the browser, and all subsequent interactions with the server are performed by structured web requests issued by the JavaScript code (as XMLHttpRequests, or XHRs). Nonbrowser clients for web apps, especially native mobile apps running on smartphones and tablets, are also increasingly important. Both kinds of clients tend to use REST (Representational State Transfer) APIs provided by the web app, and tend to need advanced features such as OAuth for authenticating calls.

To address this important need, Google Cloud Platform provides a service and a suite of tools called Google Cloud Endpoints. Endpoints make it especially easy for a mobile or rich web client to call methods on the server. Endpoints includes libraries and tools for generating server functionality from a set of methods in Python and Java, and generating client code for Android, iOS, and browser-based JavaScript. The tools can also generate a "discovery document" that works with the Google APIs Client Libraries for many client languages. And OAuth support is built in, so you don't have to worry about authentication and can just focus on the application logic.

Task Queues and Cron Jobs

A web application must respond to web requests very quickly, usually in less than a second and preferably in just a few dozen milliseconds, to provide a smooth experience to the user sitting in front of the browser. This doesn't give the application much time to do work. Sometimes there is more work to do than there is time to do it. In such cases, it's usually OK if the work gets done within a few seconds, minutes, or

hours, instead of right away, as the user is waiting for a response from the server. But the user needs a guarantee that the work will get done.

For this kind of work, an App Engine app uses *task queues*. Task queues let you describe work to be done at a later time, outside the scope of the web request. Queues ensure that every task gets done eventually. If a task fails, the queue retries the task until it succeeds.

There are two kinds of task queues: push queues and pull queues. With push queues, each task record represents an HTTP request to a request handler. App Engine issues these requests itself as it processes a push queue. You can configure the rate at which push queues are processed to spread the workload throughout the day. With pull queues, you provide the mechanism, such as a custom computational engine, that takes task records off the queue and does the work. App Engine manages the queuing aspect of pull queues.

A push queue performs a task by calling a request handler. It can include a data payload provided by the code that created the task, delivered to the task's handler as an HTTP request. The task's handler is subject to the same limits as other request handlers, with one important exception: a single task handler can take as long as 10 minutes to perform a task, instead of the 60-second limit applied to user requests. It's still useful to divide work into small tasks to take advantage of parallelization and queue throughput, but the higher time limit makes tasks easier to write in straightforward cases.

An especially powerful feature of task queues is the ability to enqueue a task within a Cloud Datastore transaction, when called via App Engine. This ensures that the task will be enqueued only if the rest of the datastore transaction succeeds. You can use transactional tasks to perform additional datastore operations that must be consistent with the transaction eventually, but that do not need the strong consistency guarantees of the datastore's local transactions. For example, when a user asks to delete a bunch of records, you can store a receipt of this request in the datastore and enqueue the corresponding task in a single transaction. If the transaction fails, you can report this to the user, and rest assured that neither the receipt nor the task are in the system.

App Engine has another service for executing tasks at specific times of the day: the scheduled tasks service. Scheduled tasks are also known as "cron jobs," a name borrowed from a similar feature of the Unix operating system. The scheduled tasks service can invoke a request handler at a specified time of the day, week, or month, based on a schedule you provide when you upload your application. Scheduled tasks are useful for doing regular maintenance or sending periodic notification messages.

We'll look at task queues, scheduled tasks, and some powerful uses for them in Chapter 16.

Namespaces

Cloud Datastore, Cloud Storage, memcache, Search, and task queues all store data for an app. It's often useful to partition an app's data across all services. For example, an app may be serving multiple companies, where each company sees its own isolated instance of the application, and no company should see any data that belongs to any other company. You could implement this partitioning in the application code, using a company ID as the prefix to every key. But this is prone to error: a bug in the code may expose or modify data from another partition.

To better serve this case, these storage services provide this partitioning feature at the infrastructure level. An app can declare it is acting in a *namespace* by calling an API. All subsequent uses of any of the data services will restrict themselves to the namespace automatically. The app does not need to keep track of which namespace it is in after the initial declaration.

The default namespace has a name equal to the empty string. This namespace is distinct from other namespaces. (There is no "global" namespace.) In the services that support it, all data belongs to a namespace.

Developer Tools

Google provides a rich set of tools and libraries for developing for Cloud Platform and App Engine. The main tool suite is the Cloud SDK, which, among other things, includes a package installer and updater for the other tools. You will use this installer to acquire the App Engine SDK for Java, and other components you might need.

One of the most useful parts of the SDK is the development web server. This tool runs your application on your local computer and simulates the runtime environment and services. The development server automatically detects changes in your Java class files and reloads them as needed, so you can keep the server running while you develop the application. This is well-suited to Java IDEs that automatically compile as you write, such as Eclipse.

If you're using Eclipse, Google provides a plugin that adds App Engine development features directly to the IDE. You can run the development server in the interactive debugger, and set breakpoints in your application code.

The development server's simulated datastore can automatically generate configuration for query indexes as the application performs queries, which App Engine will use to prebuild indexes for those queries. You can turn this feature off to test that queries have appropriate indexes in the configuration.

The development web server includes a built-in browser-based developer console for inspecting and prodding your app. You use this console to inspect and modify the

contents of the simulated data storage services, manage task queue interactions, and simulate nonweb events such as incoming email messages.

You also use the toolkit to interact with App Engine directly, especially to deploy your application and run it on Cloud Platform. You can download log data from your live application, and manage the live application's datastore indexes and service configuration.

With a provided library, you can add a feature to your application that lets you access the running app's environment programmatically. This is useful for building administrative tools, uploading and downloading data, and even running a Python interactive prompt that can operate on live data.

But wait, there's more! The SDK also includes libraries for automated testing, and gathering reports on application performance. We'll cover one such tool, AppStats, in Chapter 17.

The Cloud Console

When your application is ready for its public debut, you create a *project*, then deploy your app's code to the project using a tool in the Cloud SDK. The project contains everything related to your app, including your App Engine code, data in all of Cloud Platform's data services, any Compute Engine VMs you might create with the app, and project-related settings and permissions. All of this is managed in a browser-based interface known as the Google Developers Console, or just "Cloud Console."

You sign in to the Cloud Console using your Google account. You can use your current Google account if you have one. You may also want to create a Google account just for your application, which you might use as the "from" address on email messages. Once you have created a project in Cloud Console, you can add additional Google accounts to the project. Each account has one of three possible roles: *owners* can change settings and manage permissions for other accounts, *editors* can change settings (but not permissions), and *viewers* can read (but not change) settings and project information.

The Console gives you access to real-time performance data about how your application is being used, as well as access to log data emitted by your application. You can query Cloud Datastore and other data services for the live application by using a web interface, and check on the status of datastore indexes and other features.

When you upload new code for your application, the uploaded version is assigned a version identifier, which you specify in the application's configuration file. The version used for the live application is whichever major version is selected as the "default." You control which version is the "default" by using the Cloud Console. You can access nondefault versions by using a special URL containing the version identi-

fier. This allows you to test a new version of an app running on App Engine before making it official.

You use the Console to set up and manage the billing account for your application. When you're ready for your application to consume more resources beyond the free amounts, you set up a billing account using a credit card and Google Accounts. The owner of the billing account sets a *budget*, a maximum amount of money that can be charged per calendar day. Your application can consume resources until your budget is exhausted, and you are only charged for what the application actually uses beyond the free amounts.

Getting Started

You can start developing applications for Google App Engine without creating an account. All you need to get started is the Cloud SDK, which is a free download from the Cloud Platform website:

https://cloud.google.com/sdk/

For a brief guided tour of creating an App Engine app with some sample code, see this quick-start guide (sign-in required):

https://console.developers.google.com/start/appengine

And while you're at it, be sure to bookmark the official App Engine documentation, which includes tutorials, articles, and reference guides for all of App Engine's features:

https://cloud.google.com/appengine/docs

In the next chapter, we'll describe how to create a new project from start to finish, including how to create an account, upload the application, and run it on App Engine.