INTRODUCTION TO GOOGLE APP ENGINE

Sam Guinea

guinea@elet.polimi.it

http://servicetechnologies.wordpress.com/exercises/

A special thanks to my friends at GoCloud
Outline

- General Architecture
- Software as a Service
  - Services and APIs
- Platform as a Service
  - Google App Engine
  - Google Web Toolkit
  - Pricing
Google’s General Architecture
Software as a Service

- Google Apps
  - Personal Productivity and Collaboration Environment
  - Email, Calendar, Drive, Sites, Video, ...
- Google Maps
- *Many* other well known services …

- Functional and administrative access by means of
  - Web Interface
  - rich set of APIs
    - based on HTTP + REST protocols
Platform as a Service

• Google App Engine (GAE)
  • runtime execution environment

• Google Web Toolkit
  • for the development of complex client components Javascript-based (optional)

• SDK and Google Plugin for Eclipse
  • for the development, full local debug and "automatic" application deployment
Google App Engine

- Execution environment for Google SaaS and custom applications
  - Web access through HTTP protocol

- Scales automatically, activating (Java) virtual machines on-the-fly on a very high number of physical servers
  - Load Balancing
  - Fine tuning options

- High data redundancy
  - there is no need to backup data (other than for logical recovery)

- Functionally rich Administration Console
What does it offer?

- dynamic web serving, with full support for common web technologies
- persistent storage with queries, sorting and transactions
- automatic scaling and load balancing
- APIs for authenticating users and sending email using Google Accounts
- a fully featured local development environment that simulates Google App Engine on your computer
- task queues for performing work outside of the scope of a web request
- scheduled tasks for triggering events at specified times and regular intervals
Google App Engine
Application Sandboxing

- Applications run inside a *Sandbox* with some limitations enforcing security and "good behavior"
  - No file system writes (use datastore and memcache)
  - No socket opens
  - Code only runs due to a
    - Specific web request
    - A queued task
    - A scheduled task
  - All code must terminate within 60 seconds
    - No thread control
- This allows Google App Engine to move apps across multiple servers depending on traffic demands
Main services

- Single Sign-on for authentication, integrated with *all* Google services
  - 2-step authentication
  - integration with corporate Single Sign-on Systems (SAML-based)
- Integration with Provisioning Systems (LDAP), like MS Active Directory
- URL Fetch
- Mail
- Memcache
- Image Manipulation
- Scheduled Tasks e Task Queues
- Database/Datastore
Data storage services

• **App Engine Datastore**,  
  • NoSQL schemaless object datastore  
    • High Replication  
    • Distributed Atomic Transactions  
    • Very high Scalability

• **Google Cloud SQL**  
  • relational database based on MySQL RDBMS  
    • Fully Managed  
    • High Availability: automatic data replicas "across multiple geographic regions"

• **Google Blobstore**

• **Google Cloud Storage**  
  • very large objects and files - up to terabytes
The Datastore

• A distributed datastore that grows with your data

• Data objects (entities) have a
  • Kind
  • Set of properties

• Queries retrieve entities of a given kind
  • Filtered and sorted by the values of the properties
Comparison with Traditional DBs

• Datastore uses a distributed architecture to automatically manage scaling to very large data sets.

• While the Datastore interface has many of the same features as traditional databases, it differs from them in the way it describes relationships between data objects.

• Entities of the same kind can have different properties, and different entities can have properties with the same name but different value types.
Main Characteristics

• The App Engine Datastore is designed to scale, allowing applications to maintain high performance as they receive more traffic:
  • Datastore writes scale by automatically distributing data as necessary.
  • Datastore reads scale because the only queries supported are those whose performance scales with the size of the result set (as opposed to the data set).
    • This means that a query whose result set contains 100 entities performs the same whether it searches over a hundred entities or a million. This property is the key reason some types of query are not supported.
Main Characteristics

- Because all queries on App Engine are served by pre-built indexes, the types of query that can be executed are more restrictive than those allowed on a relational database with SQL.
- In particular, the following are not supported:
  - Join operations
  - Inequality filtering on multiple properties
  - Filtering of data based on results of a subquery

https://developers.google.com/appengine/articles/datastore/overview
Acid Transactions

• ACID transactions
  • Optimistic concurrency control
  • A transaction manipulates entities within a single group.
  • Entities of the same group are stored together for efficient execution of transactions.
  • Your application can assign entities to groups when the entities are created.
**Optimistic Concurrency Control**

- Optimistic concurrency control (OCC) is a concurrency control method for relational database management systems that assumes that multiple transactions can complete without affecting each other, and that therefore transactions can proceed without locking the data resources that they affect.

- Before committing, each transaction verifies that no other transaction has modified its data. If the check reveals conflicting modifications, the committing transaction rolls back.
Entities

• An entity has one or more named properties, each of which can have one or more values.

• Property values can belong to a variety of data types, including integers, floating-point numbers, strings, dates, and binary data, among others.

• A query on a property with multiple values tests whether any of the values meets the query criteria. This makes such properties useful for membership testing.
Entity keys

- Each entity has its own **key**, which uniquely identifies it. The key consists of the following components:
  - The entity's **kind**
  - An **identifier**, which can be either
    - *key name* string
    - an integer *numeric ID*
  - An optional **ancestor path** locating the entity within the Datastore hierarchy

- The identifier is assigned when the entity is created. It can be assigned in either of two ways:
  - Your application can specify its own *key name* string for the entity.
  - You can have the Datastore automatically assign the entity an integer numeric ID.
Entity Ancestors

- Entities in the Datastore form a hierarchically structured space similar to the directory structure of a file system.
- When you create an entity, you can optionally designate another entity as its *parent*; the new entity is a *child* of the parent entity.
- An entity without a parent is a *root entity*.
- The association between an entity and its parent is permanent

```java
Entity employee = new Entity("Employee");
datastore.put(employee);

Entity address = new Entity("Address", employee.getKey());
datastore.put(address);
```
The Datastore

- Two possible setups
  - Master/slave (deprecated)
    - Single data center holds the master copy
    - Stored data is replicated asynchronously to slaves
    - Strong consistency for reads and queries
    - Central point of failure (downtime for maintenance)

- High Replication Datastore
  - Data is replicates across multiple data centers (Paxos algorithm)
## High replication vs. Master/slave

<table>
<thead>
<tr>
<th></th>
<th>High Replication</th>
<th>Master/Slave</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Put/delete latency</td>
<td>1/2x–1x</td>
<td>1x</td>
</tr>
<tr>
<td>Get latency</td>
<td>1x</td>
<td>1x</td>
</tr>
<tr>
<td>Query latency</td>
<td>1x</td>
<td>1x</td>
</tr>
<tr>
<td><strong>Consistency</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Put/get/delete</td>
<td>Strong</td>
<td>Strong</td>
</tr>
<tr>
<td>Most queries</td>
<td>Eventual</td>
<td>Strong</td>
</tr>
<tr>
<td>Ancestor queries</td>
<td>Strong</td>
<td>Strong</td>
</tr>
<tr>
<td>Occasional planned read-only period</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Unplanned downtime</td>
<td>Extremely rare; no data loss</td>
<td>Rare; possible to lose a small % of writes occurring near downtime</td>
</tr>
</tbody>
</table>
Consistency

• Strong Consistency
  • All accesses are seen by all parallel processes (or nodes, processors etc.) in the same order (sequentially)

• Weak Consistency
  • All accesses to synchronization variables are seen by all processes in the same order (sequentially) - these are synchronization operations. Accesses to critical sections are seen sequentially.
  • All other accesses may be seen in different order on different processes
  • The set of both read and write operations in between different synchronization operations is the same in each process.

• Eventual Consistency
  • It means that given a sufficiently long period of time over which no changes are sent, all updates can be expected to propagate eventually through the system and the replicas will be consistent
Persistency Example

```java
@PersistenceCapable
public class Employee {
    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;

    @Persistent
    private String firstName;

    @Persistent
    private String lastName;

    @Persistent
    private Date hireDate;

    public Employee(String firstName, String lastName, Date hireDate) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.hireDate = hireDate;
    } // ...
```
Queries

• A typical query includes the following:
  • An entity kind to which the query applies
  • Zero or more filters based on the entities' property values, keys, and ancestors
  • Zero or more sort orders to sequence the results

• A query can include an ancestor filter limiting the results to just the entity group descended from a specified ancestor
  • Ancestor queries return strongly consistent results. Non-ancestor queries can span the entire Datastore, but are only eventually consistent and may return stale results.
Datastore queries

// Get the Datastore Service
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();

Filter heightMinFilter = new FilterPredicate("height", FilterOperator.GREATER_THAN_OR_EQUAL, minHeight);

Filter heightMaxFilter = new FilterPredicate("height", FilterOperator.LESS_THAN_OR_EQUAL, maxHeight);

// Use CompositeFilter to combine multiple filters
Filter heightRangeFilter = CompositeFilterOperator.and(heightMinFilter, heightMaxFilter);

// Use class Query to assemble a query
Query q = new Query("Person").setFilter(heightRangeFilter);

// Use PreparedQuery interface to retrieve results
PreparedQuery pq = datastore.prepare(q);

for (Entity result : pq.asList()) {
    String firstName = (String) result.getProperty("firstName");
    String lastName = (String) result.getProperty("lastName");
    Long height = (Long) result.getProperty("height");

    System.out.println(firstName + " " + lastName + ", " + height + " inches tall");
}
Query variations

• Primary Key Queries
  Query q = new Query("Person").addFilter(Entity.KEY_RESERVED_PROPERTY,
  Query.FilterOperator.GREATER_THAN,
  lastSeenKey);

• Ancestor Filters
  Query q = new Query("Person").setAncestor(ancestorKey);

• Sort Orders
  // Order alphabetically by last name:
  Query q = new Query("Person").addSort("lastName", SortDirection.ASCENDING);

  // Order by height, tallest to shortest:
  Query q = new Query("Person").addSort("height", SortDirection.DESCENDING);
Projection queries

- Conceptually similar to
  - SELECT name, email, phone, FROM CUSTOMER

```java
//Get datastore and key needed for queries
DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
Key guestbookKey = KeyFactory.createKey("Guestbook", guestbookName);

// Create the projection.
Query proj = new Query("Greeting", guestbookKey);
proj.addProjection(new PropertyProjection("user", User.class));
proj.addProjection(new PropertyProjection("date", Date.class));

// Get the results and process them, casting to the required types
List<Entity> projTests = datastore.prepare(proj).asList(FetchOptions.Builder.withLimit(5));
for (Entity projtest : projTests) {
    User person = (User)projtest.getProperty("user");
    System.out.println(person.toString());
    Date stamp = (Date)projtest.getProperty("date");
    System.out.println(stamp.toString());
}
```
Query restrictions

• Entities lacking a property named in the query are ignored.

• Filtering on unindexed properties returns no results.

• Inequality filters are limited to at most one property.

• Ordering of query results is undefined when no sort order is specified.

• Sort orders are ignored on properties with equality filters.

• Properties used in inequality filters must be sorted first.
Write Operations

• Data is written to the Datastore in two phases:
  • In the Commit phase, the entity data is recorded in a log.
  • The Apply phase consists of two actions performed in parallel:
    • The entity data is written.
    • The index rows for the entity are written. (Note that this can take longer than writing the data itself.)

• The write operation returns immediately after the Commit phase and the Apply phase then takes place asynchronously.
Write Operations

• If a failure occurs during the Commit phase, there are automatic retries; but if failures continue, the Datastore returns an error message that your application receives as an exception.

• If the Commit phase succeeds but the Apply fails, the Apply is rolled forward to completion when one of the following occurs:
  • Periodic Datastore sweeps check for uncompleted Commit jobs and apply them.
  • Certain application operations (get, put, delete, and ancestor queries) that use the affected entity group cause any changes that have been committed but not yet applied to be completed before proceeding with the new operation.
Blobstore

- The Blobstore API allows your application to serve data objects, called *blobs*, that are much larger than the size allowed for objects in the Datastore service.

- Blobs are created by uploading a file through an HTTP request.

- Blobstore creates a blob from the file's contents and returns an opaque reference to the blob, called a *blob key*, which you can later use to serve the blob.
Blobstore and WebForms

• To prompt a user to upload a Blobstore value, your application presents a web form with a file upload field.

• The application generates the form's action URL by calling the Blobstore API. The user's browser uploads the file directly to the Blobstore via the generated URL.

• Blobstore then stores the blob, rewrites the request to contain the blob key, and passes it to a path in your application. A request handler at that path in your application can perform additional form processing.
How it works

```html
<body>
  <!-- HTML form for uploading a file -->
  <form action="<%= blobstoreService.createUploadUrl("/upload") %>" method="post" enctype="multipart/form-data">
    <input type="file" name="myFile">
    <input type="submit" value="Submit">
  </form>
</body>
```

```java
Map<String, BlobKey> blobs = blobstoreService.getUploadedBlobs(req);
BlobKey blobKey = blobs.get("myFile");

if (blobKey == null) {
    res.sendRedirect("/");
} else {
    res.sendRedirect("/serve?blob-key=" + blobKey.getKeyString());
}
```
URL Fetch

• An app can use the URL Fetch service to issue HTTP and HTTPS requests and receive responses.
  • The URL Fetch service uses Google's network infrastructure for efficiency and scaling purposes.

• Use java.net.*

```java
URL url = new URL("http://www.example.com/atom.xml");
BufferedReader reader = new BufferedReader(new InputStreamReader(url.openStream()));
String line;
while ((line = reader.readLine()) != null) { … }
reader.close();
```
URL Fetch

- An app can fetch a URL using HTTP or HTTPS
  - Acceptable port ranges: 80-90, 440-450, 1024-65535
  - Can use GET, POST, PUT, HEAD, and DELETE
  - Deadline for requests (default to 5 seconds, maximum 60 seconds)
  - Support for synchronous and asynchronous requests
  - Max Request size 5M
  - Max Response size 32 M

- Contributes to incoming and outgoing bandwidth billing
Making a post connection

URL url = new URL("http://www.example.com/comment");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
connection.setDoOutput(true);
connection.setRequestMethod("POST");

Setting a Request header

connection.setRequestProperty("X-MyApp-Version", "2.7.3");

Managing redirects

connection.setInstanceFollowRedirects(false);
Managing the Response

OutputStreamWriter writer = new OutputStreamWriter(connection.getOutputStream());
writer.write("message=" + message);
writer.close();

if (connection.getResponseCode() == HttpURLConnection.HTTP_OK) {
    // OK
} else {
    // Server returned HTTP error code.
}
Secure Data Connector

• **Access your corporate data in the browser**
  • SDC lets you access your data from within Google Gadgets, Google App Engine, and Google Spreadsheets. SDC provides an agent to connect your Google Apps domain to your behind-the-firewall data sources.

• **Control the use of your data**
  • SDC lets you restrict which users and applications can make requests to your internal services. You can use your own internal authentication systems to validate and authorize those requests.

• **Build custom apps for your business**
  • SDC lets you extend your enterprise systems into Google Apps. You can easily build gadgets or
  • Google App Engine applications that make use of both private and public data.
Secure Data Connector

Google Apps generates a request for an intranet resource.

Google Apps checks the user and the application against the resource rules to ensure that the user has been granted access to the requested resource.

If Google Apps grants access, Google Apps wraps the request inside the SOCKS protocol, encrypts the request, and sends the request across the Internet inside the established tunnel to an Agent.

The SDC agent then issues the request to the relevant internal server within your corporate network, using the agent machine’s DNS and routing configuration.

The request arrives at the final destination, in this case, the web service application.

After authentication, the request is returned and data travels back to the SDC agent, which encrypts the data, and sends the data up the tunnel to Google Apps.
Mail

- Send email messages on behalf of the app's administrators, and on behalf of users with Google Accounts
- Apps can receive email at various addresses
  - in the form of HTTP requests initiated by App Engine and posted to the app
- Use javax.mail
  - No need to provide SMTP information
Sending Mail

Properties props = new Properties();
Session session = Session.getDefaultInstance(props, null);

String msgBody = "...";

try {
    Message msg = new MimeMessage(session);
    msg.setFrom(new InternetAddress("admin@example.com", "Example.com Admin"));
    msg.addRecipient(Message.RecipientType.TO, new InternetAddress("user@example.com", "Mr. User"));
    msg.setSubject("Your Example.com account has been activated");
    msg.setText(msgBody);
    Transport.send(msg);
}
Receiving Mail

appengine-web.xml

<inbound-services>
  <service>mail</service>
</inbound-services>

Web.xml

<servlet>
  <servlet-name>mailhandler</servlet-name>
  <servlet-class>MailHandlerServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>mailhandler</servlet-name>
  <url-pattern>/_ah/mail/*</url-pattern>
</servlet-mapping>

<security-constraint>
  <web-resource-collection>
    <url-pattern>/_ah/mail/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>
public class MailHandlerServlet extends HttpServlet {

    public void doPost(HttpServletRequest req, HttpServletResponse resp) throws IOException {

        Properties props = new Properties();
        Session session = Session.getDefaultInstance(props, null);
        MimeMessage message = null;

        try {
            message = new MimeMessage(session, req.getInputStream());
            Address[] addresses = message.getFrom();
            log.info("Received a mail from " + addresses[0].toString);
            log.info("Message type was " + message.getContentType());
            log.info("Received " + message.getContent());
        } catch (MessagingException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
Mail

- Billing
  - Contributes to outgoing bandwidth

- Max size of outgoing mail (with attachments) 10M
- Max size of incoming mail (with attachments) 10M
Memcache

• Distributed in-memory data cache
  • in front of or in place of robust persistent storage
  • Supports both synchronous and asynchronous caching

• Use to speed up common datastore queries
  • Especially if showing slightly old content is not a problem
  • Collect new version after expiration

• Use for session data, user preferences, and temporary values

• Values can expire
  • Would we be fine without the session data
Data Expiration

• Values are retained as long as possible
• Values may be evicted from the cache when a new value is added to the cache if the cache is low on memory
  • Following a least recently used strategy

• Memcache values are not saved to disk, so a service failure may cause values to become unavailable
Synchronous Caching

String key =..
byte[] value;

// Using the synchronous cache
MemcacheService syncCache = MemcacheServiceFactory.getMemcacheService();
syncCache.setErrorHandler(ErrorHandlers.getConsistentLogAndContinue(Level.INFO));

value = (byte[]) syncCache.get(key); // read from cache

if (value == null) {
    // get value from other source
    // .......
    syncCache.put(key, value); // populate cache
}
Asynchronous Caching

// Using the asynchronous cache
AsyncMemcacheService asyncCache = MemcacheServiceFactory.getAsyncMemcacheService();
asyncCache.setErrorHandler(ErrorHandlers.getConsistentLogAndContinue(Level.INFO));

Future<Object> futureValue = asyncCache.get(key); // read from cache
// ... do other work in parallel to cache retrieval

value = (byte[]) futureValue.get();

if (value == null) {
    // get value from other source
    // .......
    // asynchronously populate the cache
    // Returns a Future<Void> which can be used to block until completion
    asyncCache.put(key, value);
}
Images

- Image manipulation
  - Resize, rotate, flip, and crop
  - Photo enhancements through predefined algorithms
  - Image composition
  - Format conversion

- The Image can come from the app or from storage
  - Final image must never be more than 32M

- Prepare an Image object and a Transform object

- Image formats
Images

byte[] oldImageData;
// ...

ImagesService imagesService = ImagesServiceFactory.getImagesService();

Image oldImage = ImagesServiceFactory.makeImage(oldImageData);
Transform resize = ImagesServiceFactory.makeResize(200, 300);

Image newImage = imagesService.applyTransform(resize, oldImage);
byte[] newImageData = newImage.getImageData();

From the BlobStore

BlobKey blobKey; // ...

Image oldImage = ImagesServiceFactory.makeImageFromBlob(blobKey);
Image Serving URL

• A public URL for accessing an image
  • Can return the image resized or cropped

• http://your_app_id.appspot.com/randomStringImageId

  // Resize the image to 32 pixels (aspect-ratio preserved)
  http://your_app_id.appspot.com/randomStringImageId=s32

  // Crop the image to 32 pixels
  http://your_app_id.appspot.com/randomStringImageId=s32-c
Users

- Authenticate users that have
  - a Google account
  - An account on your Google apps domain
  - an OpenID identifier (experimental feature)

- The app can detect if the user is logged in
  - If not, it can redirect them to a login page

- It can also detect if the user is an administrator
User Authentication

public class MyServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {

        UserService userService = UserServiceFactory.getUserService();

        String thisURL = req.getRequestURI();

        resp.setContentType("text/html");

        if (req.getUserPrincipal() != null) {
            resp.getWriter().println("<p>Hello, " + req.getUserPrincipal().getName() + "! You can <a href="" + userService.createLogoutURL(thisURL) + "">sign out</a>.</p>");
        } else {
            resp.getWriter().println("<p>Please <a href="" + userService.createLoginURL(thisURL) + "">sign in</a>.</p>");
        }
    }
}
Task Queues

• For Background work
• Small tasks are added to queues
• Push Queues
  • Process tasks based on the processing rate in the queue’s definition
  • App Engine scales to make this possible
• Pull Queues
  • Allow a consumer to lease and process tasks at a specific time
  • More control
  • Integrate with code outside App Engine (REST interface)

• There is always a default push queue
  • 5 task invocations per second
## Quotas for Queues

<table>
<thead>
<tr>
<th>Limit</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>task object size</td>
<td>100KB</td>
</tr>
<tr>
<td>number of active queues (not including the default queue)</td>
<td>10 for free apps</td>
</tr>
<tr>
<td></td>
<td>100 for billed apps</td>
</tr>
<tr>
<td>queue execution rate</td>
<td>500 task invocations per second per queue</td>
</tr>
<tr>
<td>maximum countdown/ETA for a task</td>
<td>30 days from the current date and time</td>
</tr>
<tr>
<td>maximum number of tasks that can be added in a batch</td>
<td>100 tasks</td>
</tr>
<tr>
<td>maximum number of tasks that can be added in a transaction</td>
<td>5 tasks</td>
</tr>
</tbody>
</table>
Push Queues

Queue queue = QueueFactory.getDefaultQueue()
queue.add(withUrl("/worker").param("key", key));

• /worker determines the request handler that will be called
  • This is local to the applications root directory
  • This is also called a Web hook

• We also add a “key” parameter

• A task must finish within 10 minutes of the request
  • This is different from user requests which must complete within 60 seconds
Pull Queues

• The task consumer leases tasks, either via the Task Queue API or the Task Queue REST API.

• App Engine sends task data to the consumer.

• The consumer processes the tasks. If the task fails to execute before the lease expires, the consumer can lease it again. This counts as a retry attempt, and you can configure the maximum number of retry attempts before the system deletes the task.

• Once a task executes successfully, the task consumer must delete it.

• The task consumer is responsible for scaling instances based on processing volume.
Pull Queues

**Adding a task**

Queue q = QueueFactory.getQueue("pull-queue");
q.add(TaskOptions.Builder.withMethod(TaskOptions.Method.PULL).payload("hello world"));

**Leasing a task**

Queue q = QueueFactory.getQueue("pull-queue");
List<TaskHandle> tasks = q.leaseTasks(3600, TimeUnit.SECONDS, 100);

**Tagged tasks**

Queue q = QueueFactory.getQueue("pull-queue");

q.add(TaskOptions.Builder.withMethod(TaskOptions.Method.PULL).payload("parse1").tag("parse".getBytes()));


// Lease render tasks, but not parse
q.leaseTasksByTag(3600, TimeUnit.SECONDS, 100, "render");
Pull Queues

Delete a task

Queue q = QueueFactory.getQueue("pull-queue");
q.deleteTask("foo");

• Tasks can be pulled to an App Engine Backend
App Engine Backends

• Special App Engine instances that have
  • no request deadlines,
  • higher memory (up to 1 GB) and CPU limits (up to 4.8 GHz)
  • persistent state across requests
  • Resident or Dynamic

• They can run for long periods
  • They have unique URLs for requests (not anonymous)
  • Requests can be load-balanced across multiple instances

• They do not scale.
  • We must specify the configuration (number of instances)
GWT GOOGLE WEB TOOLKIT
GWT

• Google Web Toolkit is an SDK that allows the developer to write HTML pages with AJAX support using the Java programming language.

• The Java classes are automatically compiled in JavaScript files and included in the associated web page.
GWT: Entry Point

• The main class that generates the JavaScript content of a page is called **Entry Point**, and it is always associated to a **Module**.

• A Module is an XML file containing dependencies and settings for the Java->JavaScript compiler.

• All the application Modules must be connected to each-other if there exists a common Java code used by all the classes referenced into the modules themselves.

• After the compilation process, the concepts of Entry Point and in general of Class decay in favor of a pure prototype-oriented JavaScript code.
Example – the Entry Point

```java
public class Test implements EntryPoint {
    public void onModuleLoad() {
        final Button sendButton = new Button("Send");
        final TextBox nameField = new TextBox();
        nameField.setText("GWT User");
        final Label errorLabel = new Label();

        RootLayoutPanel.get("nameFieldContainer").add(nameField);
        RootLayoutPanel.get("sendButtonContainer").add(sendButton);
        RootLayoutPanel.get("errorLabelContainer").add(errorLabel);

        // ...
    }
}
```
Example – the Module

```xml
<?xml version="1.0" encoding="UTF-8"?>
<module rename-to='test'>
  <inherits name='com.google.gwt.user.User'/> 
  <inherits name='com.google.gwt.user.theme.standard.Standard'/> 
  <entry-point class='it.gocloud.test.client.Test'/> 
  <source path='client'/> 
  <source path='shared'/> 
</module>
```
Communication with the backend

- GWT provides native APIs to communicate with the backend by serializing a Java object (which becomes a JavaScript object once compiled) and using the HTTP protocol.

- On the client-side for every service the developer must create two interfaces
  - an interface containing the methods
  - an interface containing the methods with **asynchronous callbacks**

- On the server-side for every service the developer creates a Servlet that implements the interface containing the methods.
Client-side code

**GreetingService.java**

```java
@RemoteServiceRelativePath("greet")
public interface GreetingService extends RemoteService {
    String greetServer(String name) throws IllegalArgumentException;
}
```

**GreetingServiceAsync.java**

```java
public interface GreetingServiceAsync {
    void greetServer(String input, AsyncCallback<String> callback)
        throws IllegalArgumentException;
}
```
Server-side code

GreetingServiceImpl.java

```java
@SuppressWarnings("serial")
public class GreetingServiceImpl extends RemoteServiceServlet
GreetingService {

    public String greetServer(String input) throws IllegalArgumentException {
        // ...
    }
}
```
In the EntryPoint

```java
private final GreetingServiceAsync greetingService = GWT.create(GreetingService.class);

// ...

greetingService.greetServer(textToServer, new AsyncCallback<String>() {
    public void onFailure(Throwable caught) {
        // ...
    }

    public void onSuccess(String result) {
        // ...
    }
});
```
General Information on Pricing

- **Quotas**
  - *safety and billable*
  - *per day and per minute*

- **Pricing types**
  - *Free Accounts*
    - Within the *billable quotas*
  - *Paid Applications and Premier Accounts*
    - 99,95% SLA
    - Monthly fixed fee (*Paid* $9 / app, *Premier* $500 / account) + variable fees
    - Weekly billing for *Paid*, and monthly for *Premier*

- Several items considered for the variable billing part, when the *billable quotas* are exceeded
  - adjustable *daily maximum budget*
Some quotas

- Instances
  - Max 28 CPU h / day
- Outgoing Bandwidth: 1 GB / day
- Stored Data:
  - 1 GB
  - 200 indexes
Main variable fees

• Instances
  • Dynamic
    • class 1 (128 MB, 600 MHz): $0.08 / h
    • class 2 (256 MB, 1.2 GHz): $0.16 / h
    • class 4, classe 8

• Outgoing Bandwidth: $0.12 / GB

• Stored Data
  • Blobstore: $0.13 / (GB x Month)
  • Datastore: $0.24 / (GB x Month)

• Low-level Datastore operations
  • Write: $0.10 / 100k_ops
  • Read: $0.07 / 100k_ops

• Recipients emailed: $0.0001 / email