

# Google AppEngine Tutorial

## Introduction to Google AppEngine with Java

In this tutorial, you will practice working with Google AppEngine, specifically in Java. The contents of this tutorial are based on the official [Google Tutorials](#). This tutorial assumes you already have **Java 8** installed, and Eclipse IDE for Java Developers, version 4.5 or higher.

## Setup

First, create a new Cloud Platform Console project or retrieve the project ID of an existing project from the Google Cloud Platform Console. To do this, visit the projects page and login. I used my gmail account since my utexas account didn't seem to have the right permissions.

Manage your AppEngine applications using Google Cloud SDK. Cloud SDK includes a local development server as well as the gcloud command-line tooling for deploying and managing your apps. So our next step is to install the [Google Cloud SDK](#) and initialize the gcloud tool. Download and install the Google Cloud SDK (be sure to follow steps one through six on the linked page.)

Run the following command to install the gcloud component that includes the App Engine extension for Java:

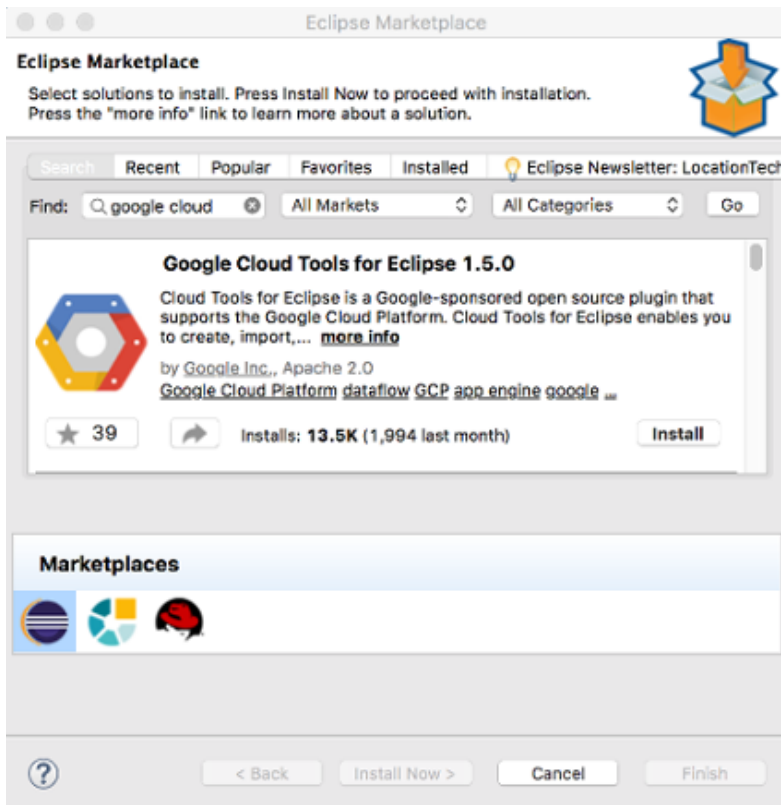
```
gcloud components install app-engine-java
```

And authorize your user account:

```
gcloud auth application-default login
```

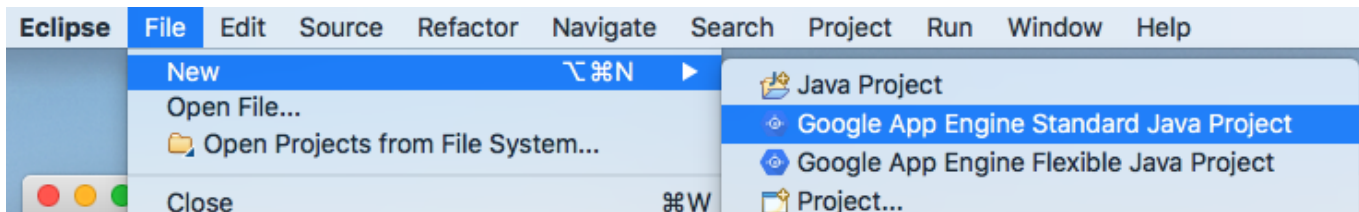
## Development Environment Configuration

Run eclipse. To download and install the Cloud Tools for Eclipse plugin, select **Help > Eclipse Marketplace...** and search for Google Cloud. After installation, restart Eclipse when prompted to do so.



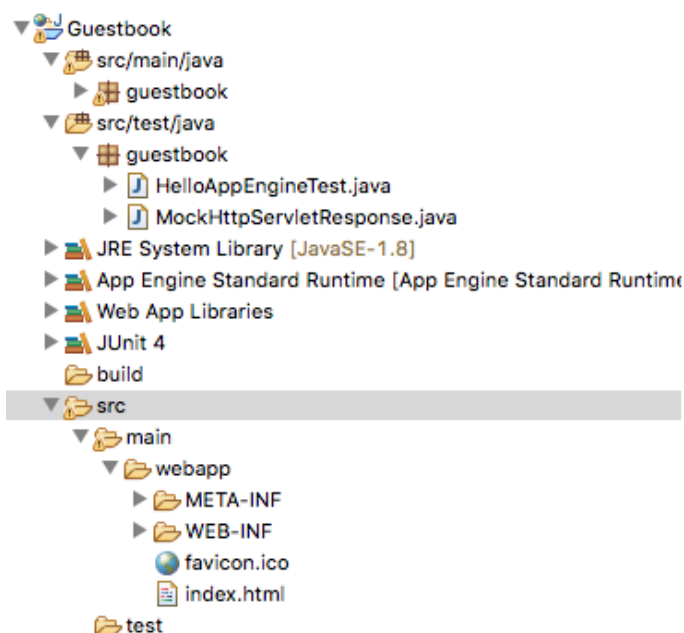
## Create A Project

In Eclipse, select the File menu > New > Google App Engine Standard Java Project. (If you don't see this menu option, select Window menu > Perspective > Reset Perspective... click OK, then try the File menu again.) Alternatively, click the Google Cloud Platform toolbar button , select Create New Project > Google App Engine Standard Java Project



The "New App Engine Standard Project" wizard opens. For Project name, enter a name for your project, such as Guestbook for the project described in the Getting Started Guide. For "Package", enter an appropriate package name, such as guestbook. Then click "Next", check the box that says "App Engine API", and click "Finish".

The wizard creates a directory structure for the project, including a src/ directory for Java source files, and a webapp/ directory for compiled classes and other files for the application, libraries, configuration files, static files such as images and CSS, and other data files. The wizard also creates a servlet source file and two configuration files. The directory structure looks like this:



## The Servlet Class

App Engine Java applications use the Java Servlet API to interact with the web server. An HTTP servlet is an application class that can process and respond to web requests. This class extends either the `javax.servlet.GenericServlet` class or the `javax.servlet.http.HttpServlet` class.

Our guestbook project begins with one servlet class, a simple servlet that displays a message.

In the directory `src/main/java`, make a file named `GuestbookServlet.java` with the following contents:

```

package guestbook;

import java.io.IOException;
import javax.servlet.http.*;

public class GuestbookServlet extends HttpServlet {

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

        resp.setContentType("text/plain");

        resp.getWriter().println("Hello, world");

    }

}

```

## The web.xml File

When the web server receives a request, it determines which servlet class to call using a configuration file known as the "web application deployment descriptor." This file is named web.xml, and resides in the webapp/WEB-INF/ directory. WEB-INF/ and web.xml are part of the servlet specification.

Modify the web.xml file so that it looks like this. You will need to add lines 6 through 20 that you see below.

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
5      version="3.1">
6      <servlet>
7
8          <servlet-name>guestbook</servlet-name>
9
10         <servlet-class>guestbook.GuestbookServlet</servlet-class>
11
12     </servlet>
13
14     <servlet-mapping>
15
16         <servlet-name>guestbook</servlet-name>
17
18         <url-pattern>/guestbook</url-pattern>
19
20     </servlet-mapping>
21
22
23     <welcome-file-list>
24         <welcome-file>index.html</welcome-file>
25         <welcome-file>index.jsp</welcome-file>
26     </welcome-file-list>
27
28 </web-app>

```

The web.xml file declares a servlet named guestbook, and maps it to the URL path /guestbook. It also indicates that whenever the user fetches a URL path that is not already mapped to a servlet, the server should check for a file named index.html in that directory and serve it if found.

In the file index.html, replace the line:

```
<td><a href='/hello'>The servlet</a></td>
```

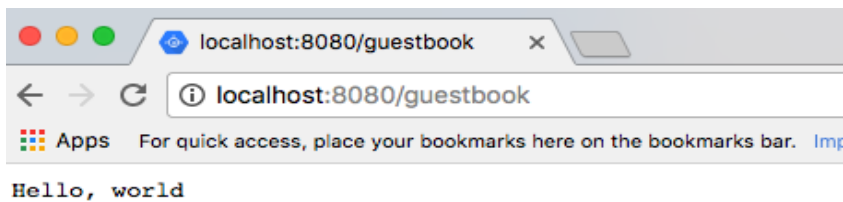
With the line:

```
<td><a href='/guestbook'>The servlet</a></td>
```

## Running the Project

Make sure the project "Guestbook" is selected, and then choose Run As > App Engine. Visit the server's URL in your browser. The server runs using port 8080 by default:

<http://localhost:8080/guestbook/>



The server calls the servlet, and displays the message in the browser.

## The appengine-web.xml Files

App Engine needs an additional configuration file to figure out how to deploy and run the application. This file is appengine-web.xml, and resides in WEB-INF/ also. It includes the registered ID of your application (Eclipse creates this with an empty ID for you to fill in later), the version number of your application, and lists of files that should be treated as static files (like images and CSS) and resource files (such as JSPs and other application data.)

appengine-web.xml is specific to App Engine, and is not part of the servlet standard. See [Configuring an App](#) for more information about this file.

## Using the Users Service

Google App Engine provides several useful services based on Google infrastructure, accessible by applications using libraries included with the SDK. One such service is the Users service, which lets your application integrate with Google user accounts. With the Users service, your users can use the Google accounts they already have to sign in to your application.

Let's use the Users service to personalize this application's greeting.

Edit src/guestbook/GuestbookServlet.java to resemble the following:

```
package guestbook;

import java.io.IOException;
import javax.servlet.http.*;
import com.google.appengine.api.users.User;
import com.google.appengine.api.users.UserService;
import com.google.appengine.api.users.UserServiceFactory;

public class GuestbookServlet extends HttpServlet {

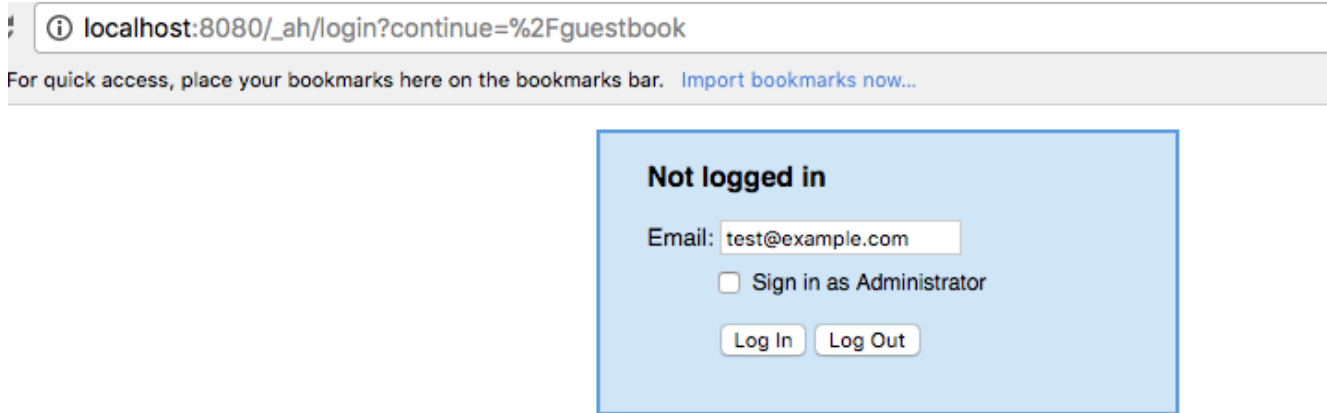
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {

        UserService userService = UserServiceFactory.getUserService();
        User user = userService.getCurrentUser();

        if (user != null) {
            resp.setContentType("text/plain");
            resp.getWriter().println("Hello, " + user.getNickname());
        } else {
            resp.sendRedirect(userService.createLoginURL(req.getRequestURI()));
        }
    }
}
```

If your development server is running, when you save your changes to this file, Eclipse compiles the new code automatically, then attempts to insert the new code into the already running server. Changes to classes, JSPs, static files and appengine-web.xml are reflected immediately in the running server without restarting. If you change web.xml or other configuration files, you must stop and restart the server to see the changes.

Test the application by visiting the servlet URL in your browser: <http://localhost:8080/guestbook>



Instead of displaying the message, the server now prompts you for an email address. Enter any email address (e.g., jane@example.com), then click "Log In". The app displays a message, this time containing the email address you entered.

The new code for the GuestbookServlet class uses the Users API to check if the user is signed in with a Google Account. If not, the user is redirected to the Google Accounts sign-in screen. `userService.createLoginURL(...)` returns the URL of the sign-in screen. The sign-in facility knows to redirect the user back to the app by the URL passed to `createLoginURL(...)`, which in this case is the URL of the current page.

The development server knows how to simulate the Google Accounts sign-in facility. When run on your local machine, the redirect goes to the page where you can enter any email address to simulate an account sign-in. When run on App Engine, the redirect goes to the actual Google Accounts screen.

You are now signed in to your test application. If you reload the page, the message will display again.

To allow the user to sign out, provide a link to the sign-out screen, generated by the method `createLogoutURL()`. Note that a sign-out link will sign the user out of all Google services. This is shown in the next section.

## Using JSPs

While we could output the HTML for our user interface directly from the Java servlet code, this would be difficult to maintain as the HTML gets complicated. It's better to use a template system, with the user interface designed and implemented in separate files with placeholders and logic to insert data provided by the application. There are many template systems available for Java, any of which would work with App Engine. For this tutorial, we'll use JSPs to implement the user interface for the guest book. JSPs are part of the servlet standard. App Engine compiles JSP files in the application's WAR automatically as one large JAR file, then maps the URL paths accordingly.

Our guestbook app writes strings to an output stream, but this could also be written as a JSP. Let's begin by porting the latest version of the example to a JSP.

In the directory `webapp/`, create a file named `guestbook.jsp` as follows:

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ page import="java.util.List" %>
<%@ page import="com.google.appengine.api.users.User" %>
<%@ page import="com.google.appengine.api.users.UserService" %>
<%@ page import="com.google.appengine.api.users.UserServiceFactory" %>
```

```
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

<html>

<body>

<%
    UserService userService = UserServiceFactory.getUserService();
    User user = userService.getCurrentUser();
    if (user != null) {
        pageContext.setAttribute("user", user);
    %>
    <p>Hello, ${fn:escapeXml(user.nickname)}! (You can
    <a href="<%= userService.createLogoutURL(request.getRequestURI()) %>">sign out</a>.)</p>
    <%
        } else {
    %>
    <p>Hello!
    <a href="<%= userService.createLoginURL(request.getRequestURI()) %>">Sign in</a>
    to include your name with greetings you post.</p>
    <%
        }
    %>

</body>
</html>
```

By default, any file in webapp/ or a subdirectory other than WEB-INF/ and META-INF/ which has a name ending in .jsp is automatically mapped to a URL path. The URL path is the path to the .jsp file, including the filename. This JSP will be mapped automatically to the URL /guestbook.jsp.

For the guestbook app, we want this to be the application's homepage, displayed when someone accesses the URL /. An easy way to do this is to declare in web.xml that guestbook.jsp is the "welcome" servlet for that path.

Edit webapp/WEB-INF/web.xml and replace the current <welcome-file> element in the <welcome-file-list>. Be sure to remove index.html from the list, as static files take precedence over JSP and servlets.

Stop and then restart the development server. Visit <http://localhost:8080/>.

The app displays the contents of guestbook.jsp, including the user nickname if the user is signed in. We want to HTML-escape any text which users provide in case that text contains HTML. To do this for the user nickname, we use the JSP's pageContext so that java code can "see" the string, then call the escapeXML function we imported via the taglib element.

When you upload your application to App Engine, the SDK compiles all JSPs into one JAR file, and that is what gets

uploaded.

## The Guestbook Form

Our guest book application will need a web form so the user can post a new greeting, and a way to process that form. The HTML of the form will go into the JSP. The destination of the form will be a new URL, /sign, to be handled by a new servlet class, SignGuestbookServlet. SignGuestbookServlet will process the form, then redirect the user's browser back to /guestbook.jsp. For now, the new servlet will just write the posted message to the log.

Edit guestbook.jsp and put the following lines just above the closing </body> tag:

```
...

<form action="/sign" method="post">
  <div><textarea name="content" rows="3" cols="60"></textarea></div>
  <div><input type="submit" value="Post Greeting" ></div>
</form>

</body>
</html>
```

Create a new class named SignGuestbookServlet in the package guestbook. (Non-eclipse users, create the file SignGuestbookServlet.java in the directory src/guestbook/.) Give the source file the following contents:

```
package guestbook;

import java.io.IOException;
import java.util.logging.Logger;
import javax.servlet.http.*;
import com.google.appengine.api.users.User;
import com.google.appengine.api.users.UserService;
import com.google.appengine.api.users.UserServiceFactory;

public class SignGuestbookServlet extends HttpServlet {

    private static final Logger log = Logger.getLogger(SignGuestbookServlet.class.getName());

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        UserService userService = UserServiceFactory.getUserService();
        User user = userService.getCurrentUser();

        String content = req.getParameter("content");
        if (content == null) {
            content = "(No greeting)";
        }
    }
}
```



```
}  
if (user != null) {  
    log.info("Greeting posted by user " + user.getNickname() + ": " + content);  
} else {  
    log.info("Greeting posted anonymously: " + content);  
}  
resp.sendRedirect("/guestbook.jsp");  
}  
}
```

Edit WEB-INF/web.xml and add the following lines to declare the SignGuestbookServlet servlet and map it to the /sign URL:

```
<servlet>  
    <servlet-name>sign</servlet-name>  
    <servlet-class>guestbook.SignGuestbookServlet</servlet-class>  
</servlet>  
<servlet-mapping>  
    <servlet-name>sign</servlet-name>  
    <url-pattern>/sign</url-pattern>  
</servlet-mapping>  
  
...  
</web-app>
```

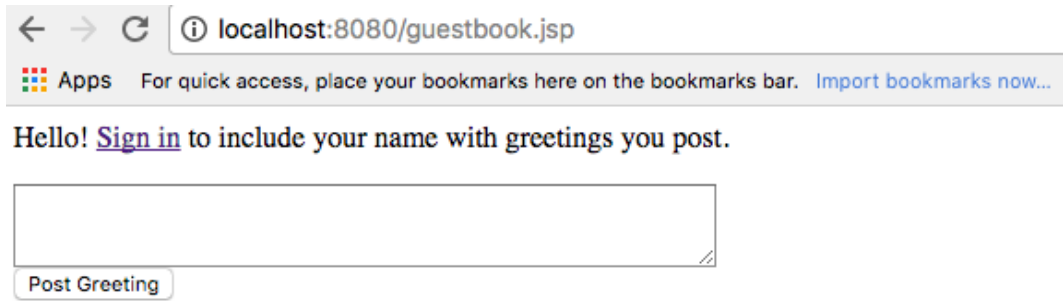
This new servlet uses the `java.util.logging.Logger` class to write messages to the log. You can control the behavior of this class using a `logging.properties` file in `Java\jdk<version>\jre\lib`, and a system property set in the app's `appengine-web.xml` file. In Eclipse, your app was created with a default version of this file in your app's `src/` and the appropriate system property.

The servlet logs messages using the INFO log level (using `log.info()`). The default log level is WARNING, which suppresses INFO messages from the output. To change the log level for all classes in the `guestbook` package, edit the `logging.properties` file and add an entry for `guestbook.level`, as follows:

```
.level = WARNING  
guestbook.level = INFO
```

Tip: When your app logs messages using the `java.util.logging.Logger` API while running on App Engine, App Engine records the messages and makes them available for browsing in the Admin Console, and available for downloading using the [AppCfgr](#) tool. The Admin Console lets you browse messages by log level.

Rebuild and restart, then test <http://localhost:8080/>. The form displays. Enter some text in the form, and submit. The browser sends the form to the app, then redirects back to the empty form. The greeting data you entered is logged to the console by the server.



← → ↻ ⓘ localhost:8080/guestbook.jsp

Apps For quick access, place your bookmarks here on the bookmarks bar. [Import bookmarks now...](#)

Hello! [Sign in](#) to include your name with greetings you post.

## Using the Datastore

Storing data in a scalable web application can be tricky. A user could be interacting with any of dozens of web servers at a given time, and the user's next request could go to a different web server than the previous request. All web servers need to be interacting with data that is also spread out across dozens of machines, possibly in different locations around the world.

With Google App Engine, you don't have to worry about any of that. App Engine's infrastructure takes care of all the distribution, replication and load balancing of data behind a simple API - and you get a powerful query engine and transactions as well.

The Datastore is one of several App Engine services offering a choice of standards-based or low-level APIs. The standards-based APIs decouple your application from the underlying App Engine services, making it easier to port your application to other hosting environments and other database technologies, if you ever need to. The low-level APIs expose the service's capabilities directly; you can use them as a base on which to implement new adapter interfaces, or just use them directly in your application.

App Engine includes support for two different API standards for the Datastore: Java Data Objects (JDO) and the Java Persistence API (JPA). These interfaces are provided by DataNucleus Access Platform, an open-source implementation of several Java persistence standards, with an adapter for the App Engine Datastore.

For clarity getting started, we'll use the low-level API to retrieve and post messages left by users.

## Updating Our Servlet to Store Data

Here is an updated version of `src/SignGuestbookServlet.java` that stores greetings in the Datastore. We will discuss the changes made here below.

```
package guestbook;

import com.google.appengine.api.datastore.DatastoreService;
import com.google.appengine.api.datastore.DatastoreServiceFactory;
import com.google.appengine.api.datastore.Entity;
import com.google.appengine.api.datastore.Key;
import com.google.appengine.api.datastore.KeyFactory;
```

```
import com.google.appengine.api.users.User;
import com.google.appengine.api.users.UserService;
import com.google.appengine.api.users.UserServiceFactory;

import java.io.IOException;
import java.util.Date;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class SignGuestbookServlet extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        UserService userService = UserServiceFactory.getUserService();
        User user = userService.getCurrentUser();

        // We have one entity group per Guestbook with all Greetings residing
        // in the same entity group as the Guestbook to which they belong.
        // This lets us run a transactional ancestor query to retrieve all
        // Greetings for a given Guestbook. However, the write rate to each
        // Guestbook should be limited to ~1/second.

        String guestbookName = req.getParameter("guestbookName");
        Key guestbookKey = KeyFactory.createKey("Guestbook", guestbookName);
        String content = req.getParameter("content");
        Date date = new Date();
        Entity greeting = new Entity("Greeting", guestbookKey);
        greeting.setProperty("user", user);
        greeting.setProperty("date", date);
        greeting.setProperty("content", content);

        DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
        datastore.put(greeting);

        resp.sendRedirect("/guestbook.jsp?guestbookName=" + guestbookName);
    }
}
```

```
}  
}
```

## Storing the Submitted Greetings

The low-level Datastore API for Java provides a schema-less interface for creating and storing entities. The low-level API does not require entities of the same kind to have the same properties, nor for a given property to have the same type for different entities. The following code snippet constructs the Greeting entity in the same entity group as the guestbook to which it belongs:

```
Entity greeting = new Entity("Greeting", guestbookKey);  
greeting.setProperty("user", user);  
greeting.setProperty("date", date);  
greeting.setProperty("content", content);
```

In our example, each Greeting is the posted content, and also stores the user information about who posted, and the date on which the post was submitted. When initializing the entity, we supply the entity name, Greeting, as well as a guestbookKey argument that sets the parent of the entity we are storing. Objects in the Datastore that share a common ancestor belong to the same entity group.

After we construct the entity, we instantiate the Datastore service, and put the entity in the Datastore:

```
DatastoreService datastore =  
  
DatastoreServiceFactory.getDatastoreService();  
datastore.put(greeting);
```

## Updating the JSP

We also need to modify the JSP we wrote earlier to display Greetings from the Datastore, and also include a form for submitting Greetings. Here is our updated guestbook.jsp:

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>  
<%@ page import="java.util.List" %>  
<%@ page import="com.google.appengine.api.users.User" %>  
<%@ page import="com.google.appengine.api.users.UserService" %>  
<%@ page import="com.google.appengine.api.users.UserServiceFactory" %>  
<%@ page import="com.google.appengine.api.datastore.DatastoreServiceFactory" %>
```

```
<%@ page import="com.google.appengine.api.datastore.DatastoreService" %>
<%@ page import="com.google.appengine.api.datastore.Query" %>
<%@ page import="com.google.appengine.api.datastore.Entity" %>
<%@ page import="com.google.appengine.api.datastore.FetchOptions" %>
<%@ page import="com.google.appengine.api.datastore.Key" %>
<%@ page import="com.google.appengine.api.datastore.KeyFactory" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

<html>
  <head>
  </head>

  <body>

<%
    String guestbookName = request.getParameter("guestbookName");
    if (guestbookName == null) {
        guestbookName = "default";
    }
    pageContext.setAttribute("guestbookName", guestbookName);
    UserService userService = UserServiceFactory.getUserService();
    User user = userService.getCurrentUser();
    if (user != null) {
        pageContext.setAttribute("user", user);
%>
<p>Hello, ${fn:escapeXml(user.nickname)}! (You can
<a href="<%= userService.createLogoutURL(request.getRequestURI()) %>">sign out</a>.)</p>
<%
    } else {
%>
<p>Hello!
<a href="<%= userService.createLoginURL(request.getRequestURI()) %>">Sign in</a>
to include your name with greetings you post.</p>
<%
    }
```

```
%>

<%
    DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
    Key guestbookKey = KeyFactory.createKey("Guestbook", guestbookName);
    // Run an ancestor query to ensure we see the most up-to-date
    // view of the Greetings belonging to the selected Guestbook.

    Query query = new Query("Greeting", guestbookKey).addSort("date", Query.SortDirection.DESENDING);
    List<Entity> greetings = datastore.prepare(query).asList(FetchOptions.Builder.withLimit(5));
    if (greetings.isEmpty()) {
        %>
        <p>Guestbook '${fn:escapeXml(guestbookName)}' has no messages.</p>
        <%
    } else {
        %>
        <p>Messages in Guestbook '${fn:escapeXml(guestbookName)}'.</p>
        <%
        for (Entity greeting : greetings) {
            pageContext.setAttribute("greeting_content",
                                    greeting.getProperty("content"));
            if (greeting.getProperty("user") == null) {
                %>
                <p>An anonymous person wrote:</p>
                <%
            } else {
                pageContext.setAttribute("greeting_user",
                                        greeting.getProperty("user"));

                %>
                <p><b>${fn:escapeXml(greeting_user.nickname)}</b> wrote:</p>
                <%
            }
        }
        %>
        <blockquote>${fn:escapeXml(greeting_content)}</blockquote>
        <%
    }
}
```

```
}  
%>  
  
<form action="/sign" method="post">  
  <div><textarea name="content" rows="3" cols="60"></textarea></div>  
  <div><input type="submit" value="Post Greeting" /></div>  
  <input type="hidden" name="guestbookName" value="{fn:escapeXml(guestbookName)}"/>  
</form>  
  
</body>  
</html>
```

Warning! Whenever you display user-supplied text in HTML, you must escape the string using the `fn:escapeXml` JSTL function, or a similar escaping mechanism. If you do not correctly and consistently escape user-supplied data, the user could supply a malicious script as text input, causing harm to later visitors.

## Retrieving the Stored Greetings

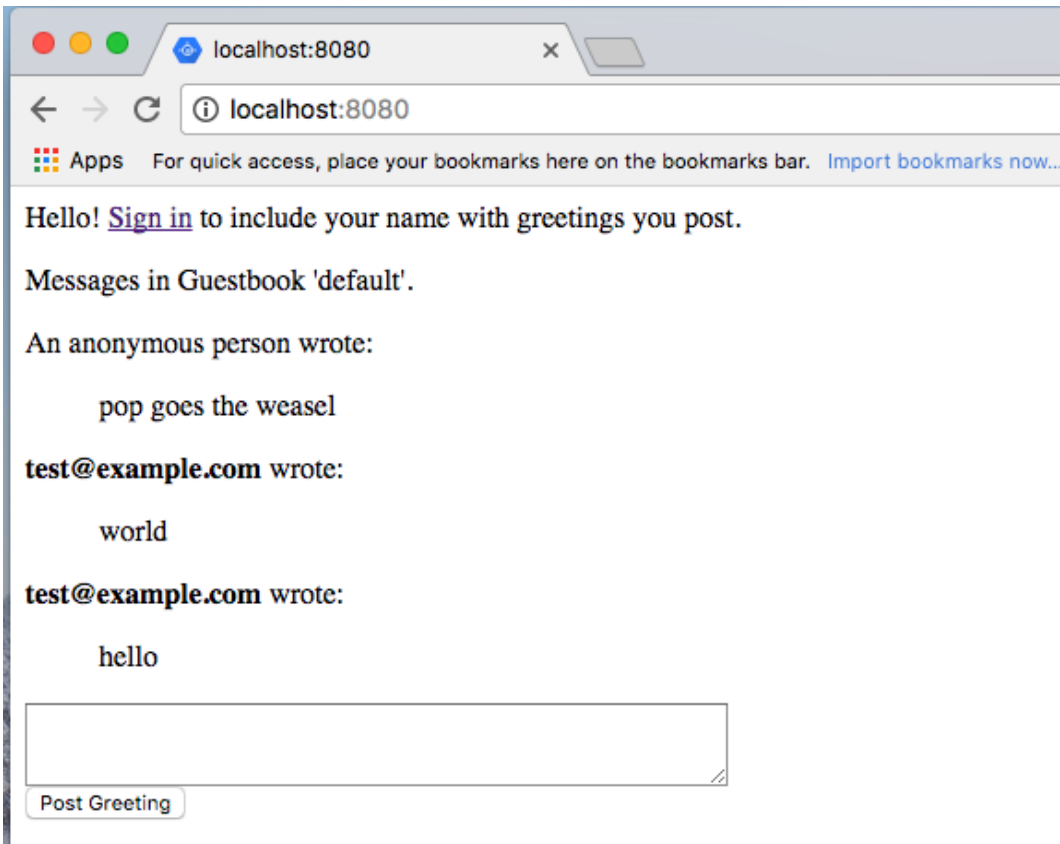
The low-level Java API provides a `Query` class for constructing queries and a `PreparedQuery` class for fetching and returning the entities that match the query from the Datastore. The code that fetches the data is here:

```
Query query = new Query("Greeting", guestbookKey).addSort("date", Query.SortDirection.DESENDING);  
List<Entity> greetings = datastore.prepare(query).asList(FetchOptions.Builder.withLimit(5));
```

This code creates a new query on the `Greeting` entity, and sets the `guestbookKey` as the required parent entity for all entities that will be returned. We also sort on the `date` property, returning the newest `Greeting` first.

After you construct the query, it is prepared and returned as a list of `Entity` objects. For a description of the `Query` and `PreparedQuery` interfaces, see [the Datastore reference](#).

Visit <http://localhost:8080/>.



## Using Static Files

There are many cases where you want to serve static files directly to the web browser. Images, CSS stylesheets, JavaScript code, movies and Flash animations are all typically served directly to the browser. For efficiency, App Engine serves static files from separate servers than those that invoke servlets.

By default, App Engine makes all files in the WAR available as static files except JSPs and files in WEB-INF/. Any request for a URL whose path matches a static file serves the file directly to the browser - even if the path also matches a servlet or filter mapping. You can configure which files App Engine treats as static files using the appengine-web.xml file.

Let's spruce up our guestbook's appearance with a CSS stylesheet. For this example, we will not change the configuration for static files. See App Configuration for more information on configuring static files and resource files.

In the directory webapp/, create a directory named stylesheets/. In this directory, create a file named main.css with the following contents:

```
body {  
  font-family: Verdana, Helvetica, sans-serif;  
  background-color: #FFFFCC;  
}
```

Edit webapp/guestbook.jsp and insert the following lines just after the <html> line at the top:

```
<html>  
<head>  
  <link type="text/css" rel="stylesheet"
```



```

href="/stylesheets/main.css" />
</head>

<body>
...
</body>
</html>

```

Visit <http://localhost:8080/>. The new version uses the stylesheet.

### Creating Index File

An index contains a list of properties of an entity type with corresponding order for each property, either ascending or descending. Cloud Datastore has already predefined an index for each property of an entity kind. Therefore, we do not need to define another index which only contains single property. For our guestbook, `guestbook.jsp` only requires Greetings to be sorted by a single property (i.e., date) which does not require us to add an index.

Indexes are defined in index configuration file `datastore-indexes.xml`.

Now we are going to sort the greetings by date and user. Create `datastore-indexes.xml` in `webapp/WEB-INF/` and paste this code inside the file:

```

<?xml version="1.0" encoding="utf-8"?>
<datastore-indexes autoGenerate="false">
  <datastore-index kind="Greeting" ancestor="true" source="manual">
    <property name="user" direction="desc" />
    <property name="date" direction="desc" />
  </datastore-index>
</datastore-indexes>

```

Update the query in `guestbook.jsp` with:

```

...
Query query = new Query("Greeting", guestbookKey).addSort("user",
Query.SortDirection.DESCENDING).addSort("date", Query.SortDirection.DESCENDING);
...

```

Now visit <http://localhost:8080/>. You will see that the greetings are sorted by date and user.

## Uploading Your Application

You create and manage App Engine web applications from the App Engine Administration Console, at the following URL:

<https://appengine.google.com>

Sign into App Engine using your Google account. If you do not have a Google account, create one.

To create a new application, click the "Create an Application" button. Follow the instructions to register an application ID, a name unique to this application.

For this tutorial, you should probably elect to use the free `appspot.com` domain name, and so the full URL for the application will be `http://your_app_id.appspot.com/`.

For Authentication Options (Advanced), the default option, "Open to all Google Accounts users", is the simplest choice for this tutorial. If you choose "Restricted to the following Google Apps domain", then your domain administrator must add your new app as a service on that domain. If you choose the Google Apps domain authentication option, then failure to add your app to your Google Apps domain will result in an HTTP 500 where the stack trace shows the error

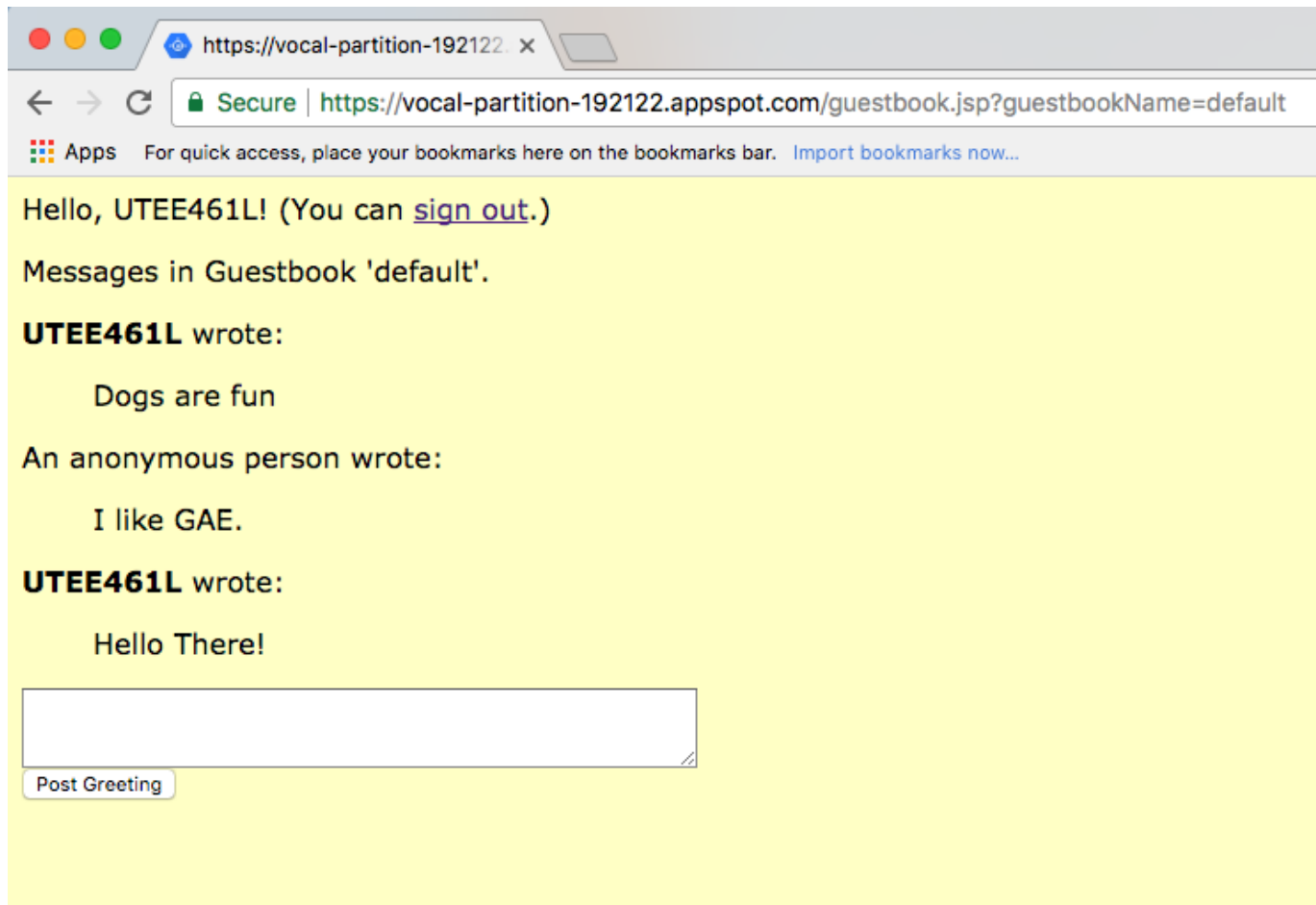
"Unexpected exception from servlet: java.lang.IllegalArgumentException: The requested URL was not allowed: /guestbook.jsp". If you see this error, add the app to your domain. See [Configuring Google Apps to Authenticate on Appspot](#) for instructions.

To upload your application from Eclipse, right click on Guestbook, and then select "Deploy to App Engine".

If prompted, follow the instructions to provide the Application ID from the [App Engine Console](#) that you would like to use for this app, your Google account username (your email address), and your password. Then click the Deploy button. Eclipse will automatically upload the contents of the webapp/ directory.

You can now see your application running on App Engine. If you set up a free appspot.com domain name, the URL for your website begins with your application ID:

`http://your_app_id.appspot.com/`



## What to Submit

All you need to submit is the URL for your final app. It will most likely be `http://***.appspot.com/`.