Google App Engine Java Experiments

Version 1.0

Romin K. Irani

Author: Romin K. Irani
Editorial Board: Romin K. Irani, Jérôme Bâton
Web Marketing : Aurélien Bernard (http://www.ab-comm.fr/)
Compositor: Devyani R. Irani

For any information/feedback regarding this publication, please contact the author directly at romin.k.irani@gmail.com.

The source code for this book is available to readers at: http://gaejexperiments.wordpress.com. If you like this book and would like to donate, please do via the *Donate* button on the above site. 100% of all donations will be sent to KIVA (www.kiva.org) charity of my choice.

About the Author

Romin K. Irani is a software developer with 15 years of industry experience, living in Mumbai, India. He is the author of the blog: http://gaejexperiments.wordpress.com.

He truly believes that software has the power to change lives. And it is that pursuit that keeps him passionate about his work.

He loves learning and writing about technology. Give him an API, he will conduct an experiment with it and release the results to the world.

He has contributed to several books at Wrox Publications, as writer and reviewer. He has also authored several key programming articles at various websites. The list of his articles/books is listed at http://iromin.wordpress.com/articles and http://iromin.wordpress.com/books.

His social application, *Save a Life* (http://iromin.wordpress.com/save-a-life) won the category award at the Ribbit Killer App Challenge in March 2009.

He is on the lookout for assignments that you may have. He would love to hear from you, if you are interested in consulting & training on various technical areas. Please reach him at romin.k.irani@gmail.com

You can follow him on Twitter: http://www.twitter.com/iromin

About Mumbai Health Tracker

I recently built a social portal called **Mumbai Health Tracker.**

I am pleased to let you know that a lot of material covered in this book was used to create this application and there is no reason why you should not be able to create a similar application using the material presented in this book.

http://mumbaihealth.appspot.com

Mumbai Health Tracker is a social web service that aggregates health issues reported by its residents.

These health issues are then categorized, aggregated to provide information that will be useful to Individuals, Government & Doctors.

All the above individuals/organizations can understand the typical health trends over a period of time. Some examples include:

- Which diseases are more prevalent in certain areas?
- Which months of the year see a spike in certain diseases?
- Co-relate certain health issues to civic infrastructure issues

The long term vision of this service is to do controlled pilots in small areas where individuals and medical practitioners provide data that may help to identify trends and address local issues.

A desktop version of the Health Tracker is also available. If you find this interesting or wish to adapt this application to your region, please contact me at romin.k.irani@gmail.com

Acknowledgements

We all stand on the shoulders of giants. And my case has been no different. In no particular order, I wish to thank the following:

- All the organizations that believe and promote open standards. By giving us their tools and environments to build on, it has created a win-win for all of us. “Nothing Else Matters…” sang Metallica and that is really true.
- To the team behind Google App Engine. This material would not exist without you.
- All individuals, who in the spirit of sharing, have penned great blogs/articles from which I have learned a lot.
- To my wife, Devyani, who managed the production of this eBook along with her patience and understanding, while I conducted these experiments.
- To a dear friend, Jérôme Bâton (aka Wadael – http://www.wadael.org), who lives in France, but has almost become a virtual next-to-my-cubicle co-worker. We have shared a lot of technical notes over the last few months and learnt much from each other. The world is indeed flat.
- To Aurélien Bernard (http://www.ab-comm.fr/) for his inputs on web marketing.
- To all the readers of http://gaejexperiments.wordpress.com. You folks have kept me motivated all along.
- To my parents. Dad & Mom – thank you for believing in me, encouraging and letting me pursue my interests.

Table of contents

Episode 1 : Google App Engine Java Development Setup

This blog post will be focused on getting your developer environment right for developing/deploying your Google App Engine for Java (**GAEJ** from now) application in your IDE. The version of Google App Engine that I shall be covering is 1.2.5 (**Released on 09/03/09**)– the very latest (at least at the time of this writing).

Before we begin, let me make a few assumptions:

- You are a Java developer who is familiar with developing basic Java Web applications. At most you need to know Servlets/JSP and the typical WAR file structure. Nothing more!
- You are comfortable with installing the JDK, Eclipse IDE and downloading / setting up plugins for Eclipse.

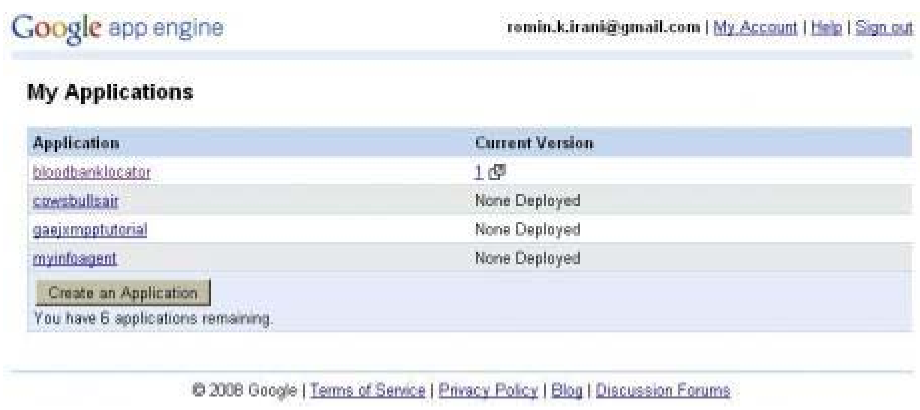
There are numerous resources on the web that address the book in addition to books, so please feel free to look them up before you proceed. As much as I would like to cover all details, I think I will need to keep this blog focused more on Google App Engine stuff for Java. So let's get going…

Sign up for Google App Engine

You need to first sign up for the Google App Engine Account. This is easy and is a breeze in case you already have a Google Account (Gmail, Apps, etc) in which case it is best that you use that straight away. Go to http://appengine.google.com and simply login with your account. Even if you do not have an account, you can sign up for a Google Account, a link for which is provided on the page.

Whether you already have a Google Account or have signed up for a new one, do login first. Once you are logged in, you will be presented with a list of applications that you are currently running on the Google App Engine under the account which we logged in with. Google provides a total of 10 applications that you can deploy under a single account. Please be aware that as of this date, there is no way to delete an application and use that slot for another application — so use with care.

Here is a screen shot of my account page at the Google App Engine after login:


Google app engine
romin.k.irani@gmail.com | My Account | Help | Sign out

My Applications

Application	Current Version
bloodbanklocator	1
cowsbullsair	None Deployed
gaejxmpptutorial	None Deployed
myinfoagent	None Deployed

You have 6 applications remaining.

© 2008 Google | Terms of Service | Privacy Policy | Blog | Discussion Forums

It shows that I have 4 applications registered on Google App Engine and have 6 more to go. You might have no applications listed if you are logging in for the first time and/or have not created any application.

Note: If you are **registering your 1st application**, the process is as follows:

1. Click on **Create an Application**.

2. Once you have selected a name, then you will be prompted for **your Mobile Number**. Enter your mobile number (with country code, etc).

3. Google will **send you a SMS** with an code. You will be prompted on the next screen to enter the code that you received on SMS. Enter that to confirm the process.

This is only required for registering your 1st application and once the SMS verification process is complete, you will not be prompted again when you create your 2nd application and so on.

Registering your Application

Cool, let’s create an application then but before that we must tell your application to show its ID. So let us talk about Identity first.

Identity, Identity, Identity….

Yes, everyone needs to show an ID. And the same applies to your application too that is hosted in the cloud.

Let me explain that a little. As you know Google App Engine is a PaaS (Platform as a Service), which means that you can develop and deploy your application (following some specification) into their infrastructure (in the cloud). Now what this means is that while it would be nice for you to get an application name of your choice, it is not feasible in a hosted environment where you are sharing all resources with other applications too.

By default, your application needs to be unique within this so called name space:

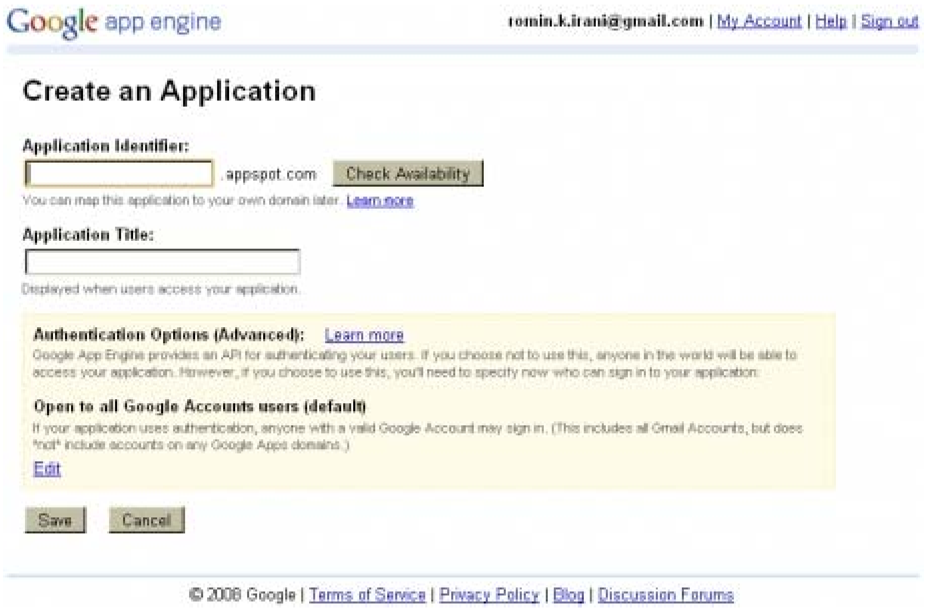
YOUR_APPLICATION_ID.appspot.com

where **YOUR_APPLICATION_ID** can be any name that you wish for your application, that has not already been taken by someone else. So before you start with an application in mind, it pays to plan it out a little first and that includes what name you would like for your application and then registering it as quickly as possible so that you have it.

Note: Please keep in mind that what I am discussing here is the basic facilities provided by Google. It is perfectly possible for you to purchase a domain name and then point it to the URL where your application is hosted in the Google App Engine cloud. But we will not discuss all that know and will keep the discussion to what is provided straight of the bat.

OK. So you have thought of the application name and wish to register it. So since we are going to write a **"Hello World"** sort of an application — we should go ahead and create an application identifier first.

Let us actually do that before moving on. Assuming that you are logged in as described earlier, click on the Create An Application button. This will bring up a page as shown below:



Create an Application

Application Identifier: .appspot.com Check Availability

You can map this application to your own domain later. Learn more

Application Title:

Displayed when users access your application.

Authentication Options (Advanced): Learn more

Google App Engine provides an API for authenticating your users. If you choose not to use this, anyone in the world will be able to access your application. However, if you choose to use this, you'll need to specify now who can sign in to your application:

Open to all Google Accounts users (default)

If your application uses authentication, anyone with a valid Google Account may sign in. (This includes all Gmail Accounts, but does *not* include accounts on any Google Apps domains.)

Edit
Save
Cancel
© 2008 Google | Terms of Service | Privacy Policy | Blog | Discussion Forums

This page will allow you to select an **Application Identifier** that we have been talking about. Go ahead. Try out a few names like myhelloworld, myfirstapp, etc and see which one is available. Give any string for the Application Title and go with the default options for the Account options (we can skip it for now). Click on the Save button. If all is well, the application identifier will be created and your first application is registered. Congratulations! But we have a lot to do including writing the application.

Please make sure that you note down your Application Identifier since this will be used later on to verify your entire setup. OK, let me not scare you — you can always go back to the Google AppEngine account and see the applications listed 😀

Do check out the excellent monitoring portal that Google App Engine provides to view/monitor a number of things about your application. Just click on the registered application in the application list and you will be lead to the portal where you can check the logs/quotas and configure several other stuff. As this blog develops, I will cover more of this but for now, just browse around and familiarize yourself.

Downloading, Installing and setting up the Eclipse IDE for GAEJ

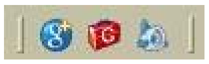
We will need the following software to be present on your development machine. I am assuming that you are comfortable with downloading and setting up of the software from public sites.

1. **Java Development Kit 1.6.x** : Please ensure that the JDK version is setup appropriately on your machine.
2. **Eclipse 3.3 / 3.4 / 3.5** : Choose any Eclipse IDE version that you want. I am using the latest version 3.5 Galileo Release.
3. **GAEJ Plug-in**: Go to the following page: http://code.google.com/eclipse/docs/getting_started.html This page contains links to the GAEJ plugin for Eclipse download/installation. I list below the update site URL for the GAEJ Eclipse plugin from that page:
    - **Eclipse 3.5 (Galileo)**
      http://dl.google.com/eclipse/plugin/3.5
    - **Eclipse 3.4 (Ganymede)**
      http://dl.google.com/eclipse/plugin/3.4
    - **Eclipse 3.3 (Europa)**
      http://dl.google.com/eclipse/plugin/3.3

   Depending on the version of Eclipse that you have, I reproduce from the Google site the links for installing the plugin depending on the version of Eclipse that you have. Please follow the instructions on the page, it is straightforward and I do not wish to replicate it here.

    - http://code.google.com/eclipse/docs/install-eclipse-3.5.html
    - http://code.google.com/eclipse/docs/install-eclipse-3.4.html
    - http://code.google.com/eclipse/docs/install-eclipse-3.3.html

Once you have finished the setup and restarted your Eclipse IDE, you should see the following 3 icons in the toolbar of the Eclipse IDE.



If not, please refer to the troubleshooting options that are described in each of the above links.

Writing our first GAEJ application

So far we have signed up for Google App Engine, registered our Application Id and setup the GAEJ plug-in for Eclipse. We are ready to write our application now. The Gods always tell us to start any new thing on an auspicious note by crying out “Hello World” and we shall stick to the tradition.

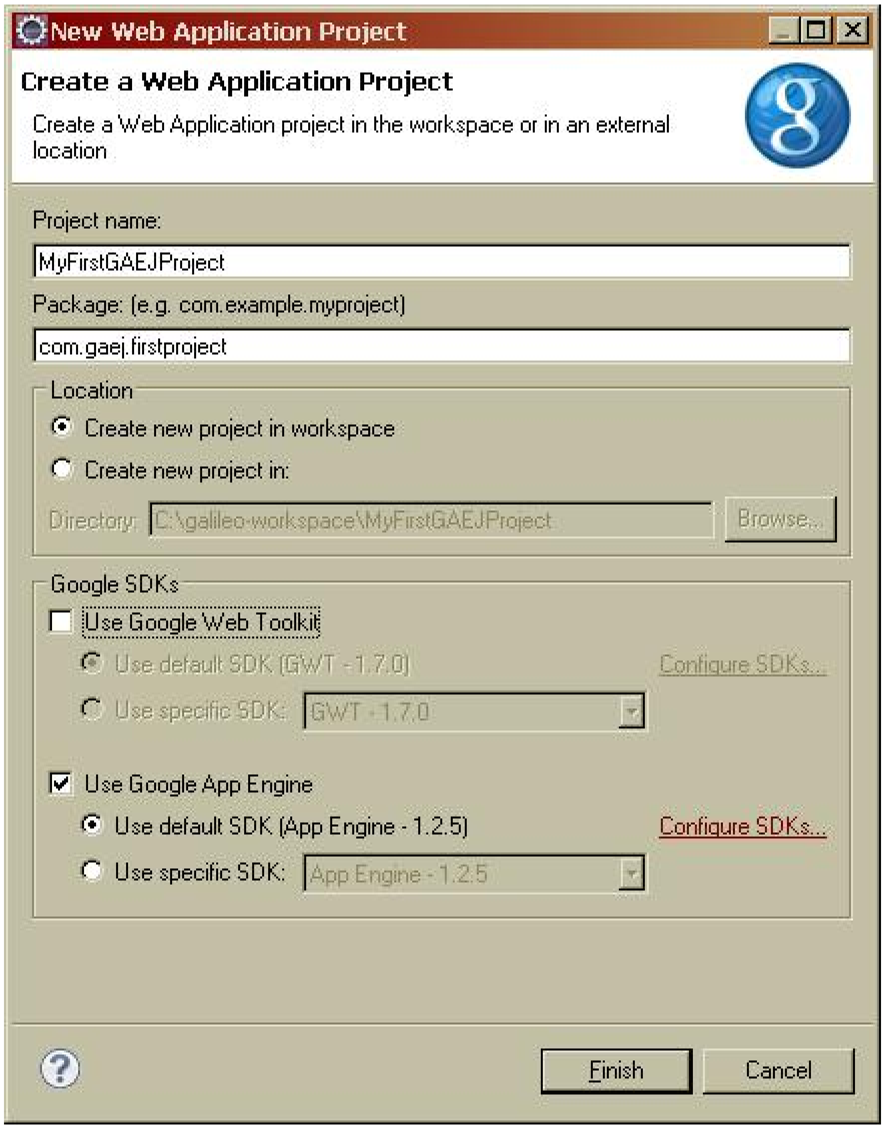
As we shall see things unfold, GAEJ plug-in for Eclipse makes things very simple. It does all the hard work after we give it some inputs and then it generates an entire web application for us along with sample servlet that pretty much does just that i.e. says “Hello World”. What we will be writing no code, you say? Yes. All this code is generated for us and is pretty much a template that we can take and put our application logic in there.

So let us move forward and generate our first project now (steps along with explanation follow):

1. Launch your Eclipse (if not started).

2. Either click on **File –> New –> Other** or press **Ctrl-N** to create a new project. Select **Google** and then **Web Application project**. Alternately, you could also click on the New Web Application Project Icon [part of GAEJ Eclipse Plugin] in the Toolbar:

3. This will bring up a New Web Application Project page as shown below:



New Web Application Project

Create a Web Application Project

Create a Web Application project in the workspace or in an external location

Project name: MyFirstGAEJProject

Package: (e.g. com.example.myproject) com.gaej.firstproject

Location

- [✓ Create new project in workspace]
- Create new project in:
- Directory: C:\galileo-workspace\MyFirstGAEJProject Browse...

Google SDKs

- [ ] Use Google Web Toolkit
  - [✓ Use default SDK (GWT - 1.7.0)] Configure SDKs...
  - Use specific SDK: GWT - 1.7.0
- [✓] Use Google App Engine
  - [✓ Use default SDK (App Engine - 1.2.5)] Configure SDKs...
  - Use specific SDK: App Engine - 1.2.5

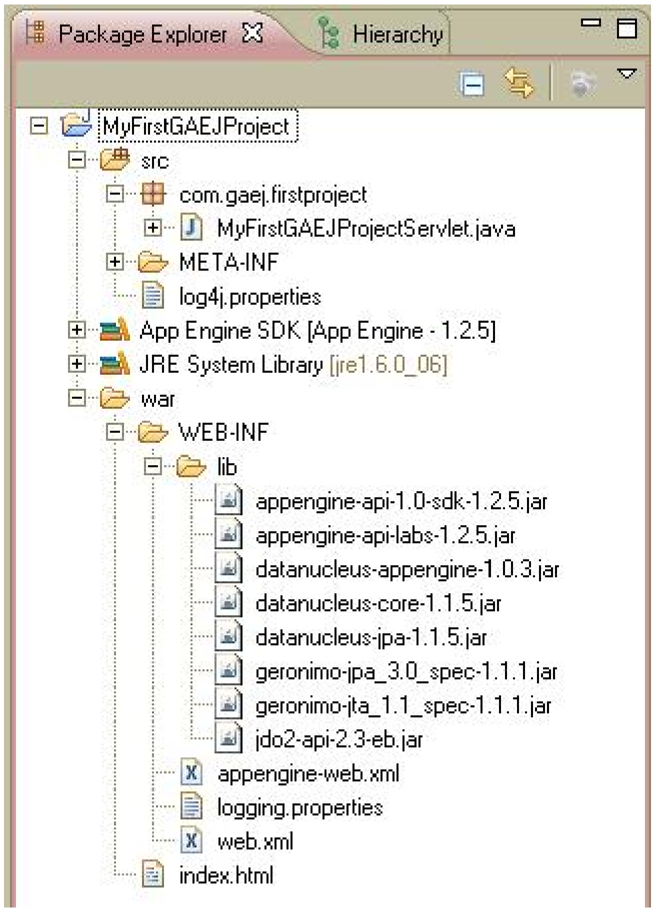
Finish Cancel

Enter the details as given below:

* **Project Name** : **MyFirstGAEJProject**. You can select any name that you like.
* **Package** : **com.gaej.firstproject**. You can select any name that you like.
* **Deselect** the **Use Google Web Toolkit** option.

4. Click on **Finish**. This will generate an entire project for you which is ready to be deployed.

5. But before that, let us look at the files that have been generated. I reproduce for you the screen from my Package Explorer as shown below:



```
MyFirstGAEJProject
  src
    com.gaej.firstproject
      MyFirstGAEJProjectServlet.java
    META-INF
    log4j.properties
  App Engine SDK [App Engine - 1.2.5]
  JRE System Library [jre1.6.0_06]
  war
    WEB-INF
      lib
        appengine-api-1.0-sdk-1.2.5.jar
        appengine-api-labs-1.2.5.jar
        datanucleus-appengine-1.0.3.jar
        datanucleus-core-1.1.5.jar
        datanucleus-jpa-1.1.5.jar
        geronimo-jpa_3.0_spec-1.1.1.jar
        geronimo-jta_1.1_spec-1.1.1.jar
        jdo2-api-2.3-eb.jar
      appengine-web.xml
      logging.properties
      web.xml
    index.html
```

6. If you are familiar with Java Web applications, this should be straightforward stuff with directories like WEB-INF, WEB-INF\classes, WEB-INF\lib, etc. But let us cover it in brief:

- **src** : This contains the source files. You will find that it has generated a sample file name **MyFirstGAEJProjectSerlvet.java** for you. It has been created as per the package hierarchy that you selected. You can keep all your source files over here as you develop more.

- **war** : This is the war file structure i.e. Web Application Archive. This if you recollect will be deployed into the Servlet Container of the App Engine that will eventually host your Java Web Application. In this directory, you have the following:
  - **WEB-INF\lib**: All appropriate JAR files that provide the Java SDK libraries along with Google Java APIs for the App Engine have been linked correctly into the project **build path** and the necessary JAR files are also present in the **WEB-INF\lib** folder
  - **index.html** : This is the first page served. You can actually customize it to any that you want. All your additional **html files/resources** could go in here.

- **logging.properties:** If you are familiar with Java logging, you should be able to make sense of this file and tweak your logging preferences over here.
- **web.xm**l : This is a standard web application configuration file. You will find the **Servlet** entries present here for the sample servlet that is generated for us.
- **appengine-web.xml** : This is a Google App Engine specific configuration file. And the main element here of interest to us is the application element that will contain the **Application ID** of the application that we registered for Google and which will be pointing to a hosted version of this WAR. We will set this up at the time of deploying the application, which we will cover later.

7. **MyFirstGAEJProjectServlet** : Let us cover this in brief, which can also act as a refresher for writing and setting up a servlet configuration in case you are little rusty (happens to the best of us!).

The servlet code is straightforward as shown below[Omitting package and import declarations]:

```
public void doGet(HttpServletRequest req, HttpServletResponse resp)
throws IOException {
resp.setContentType("text/plain");
 resp.getWriter().println("Hello, world");
 }
 }
```

It extends the **javax.servlet.http.HttpServlet** class and implements the **GET** method. You could parse out the request parameters if needed (in this case there are none of interest). All that the servlet does is to **get a handle to the response stream** and then mention its content type and send across the text **Hello World** to the client.

Now, we need to set the servlet url and its java class in the **web.xml** configuration file. Open up the **web.xml** file and you will find the appropriate entries as shown below:

```
<servlet>
<servlet-name>MyFirstGAEJProject</servlet-name>
 <servlet-
class>com.gaej.firstproject.MyFirstGAEJProjectServlet</servlet-class>
 </servlet>
 <url-pattern>/myfirstgaejproject</url-pattern>
 </servlet-mapping>
```

And finally, if you open **index.html**, you will find a link that references the servlet as shown below:

```
<td><a href="myfirstgaejproject"/>MyFirstGAEJProject</td>
```

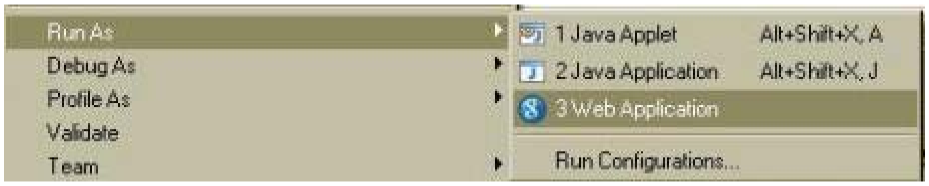
so that when the **index.html** page is served, you will see a **Hyperlink** named **MyFirstGAEJProject** and on **clicking** it will **invoke the MyFirsGAEProject servlet** implemented in **com.gaej.firstproject.MyFirstGAEJProjectServlet** which will then **print out Hello World**.

Testing it locally

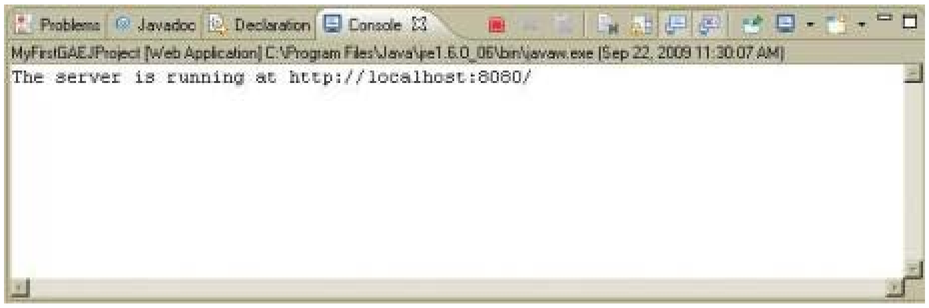
GAEJ Eclipse plug-in ships with a local server, where you can deploy your application and test it out. Ensure that port **8080** is free on your machine and that no other web server is running on it.

To test the application, all you need to do is the following:

1. Select the project and then Run As –> Web Application



2. This will compile/deploy your application the local server and the server will be started on port 8080. You will see the server starting in the console window as shown below:

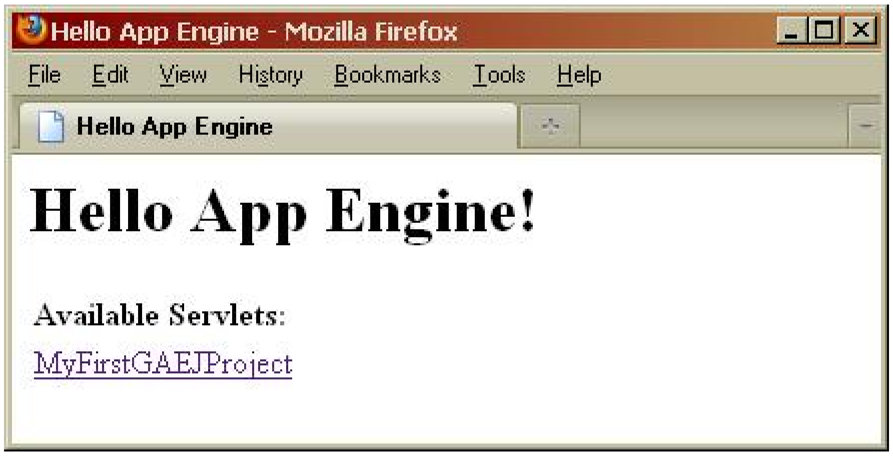


```
MyFirstGAEJProject [Web Application] C:\Program Files\Java\jre1.6.0_06\bin\javaw.exe (Sep 22, 2009 11:30:07 AM)
The server is running at http://localhost:8080/
```

3. If all is well, you will get the message that the server is running at http://localhost:8080 as shown above.

4. Once that notification is shown, launch your browser and navigate to the url : http://localhost:8080. This will bring up a default JSP page that is generated for you with

the servlet URL in it as shown below:

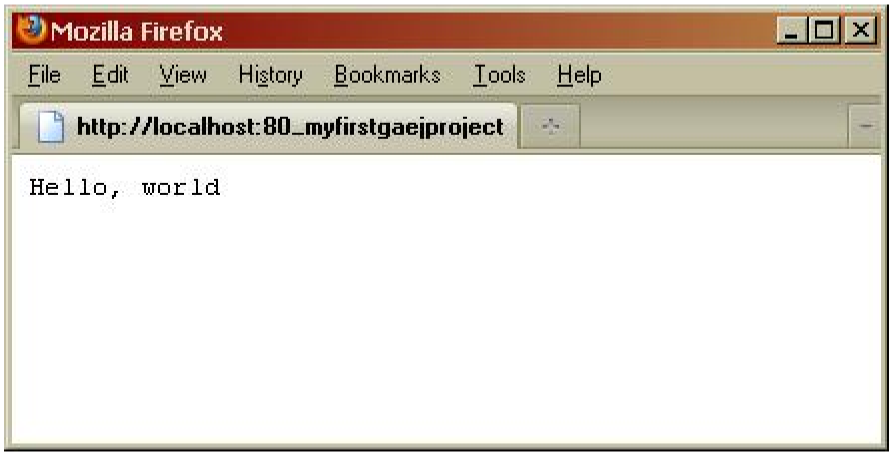

Hello App Engine - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Hello App Engine

Hello App Engine!

Available Servlets:

MyFirstGAEJProject

5. Click on the link and you will see your servlet getting invoked and telling you "Hello World".

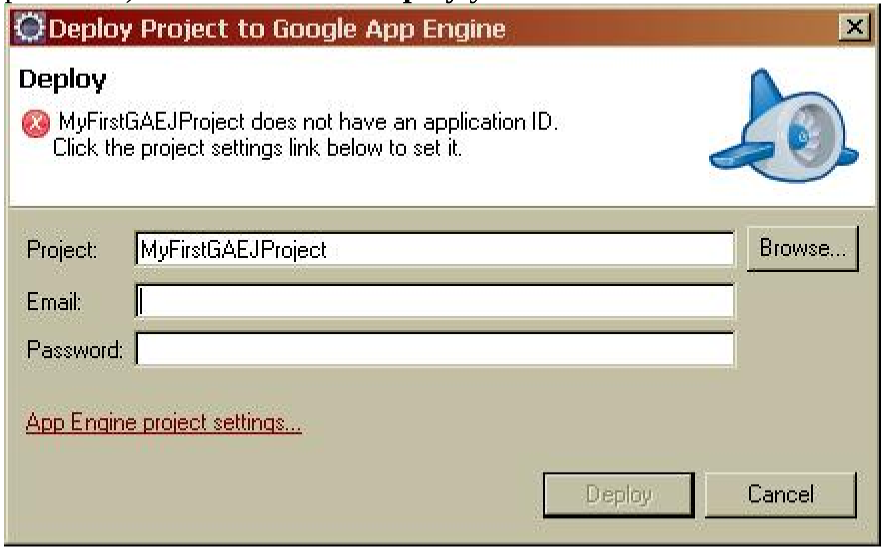

Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:80_myfirstgaejproject

```
Hello, world
```

Now that things work, we need to deploy this application into the Google App Engine infrastructure. So lets go to that.

Deploying it into the Cloud

To deploy the application in the cloud, simply click on the **Deploy App Engine Project Icon** available in Toolbar[part of GAEJ Eclipse Plugin]. This will bring up a dialog where you will need to provide your Google Account credentials (username and password). **Do not click on Deploy** yet.



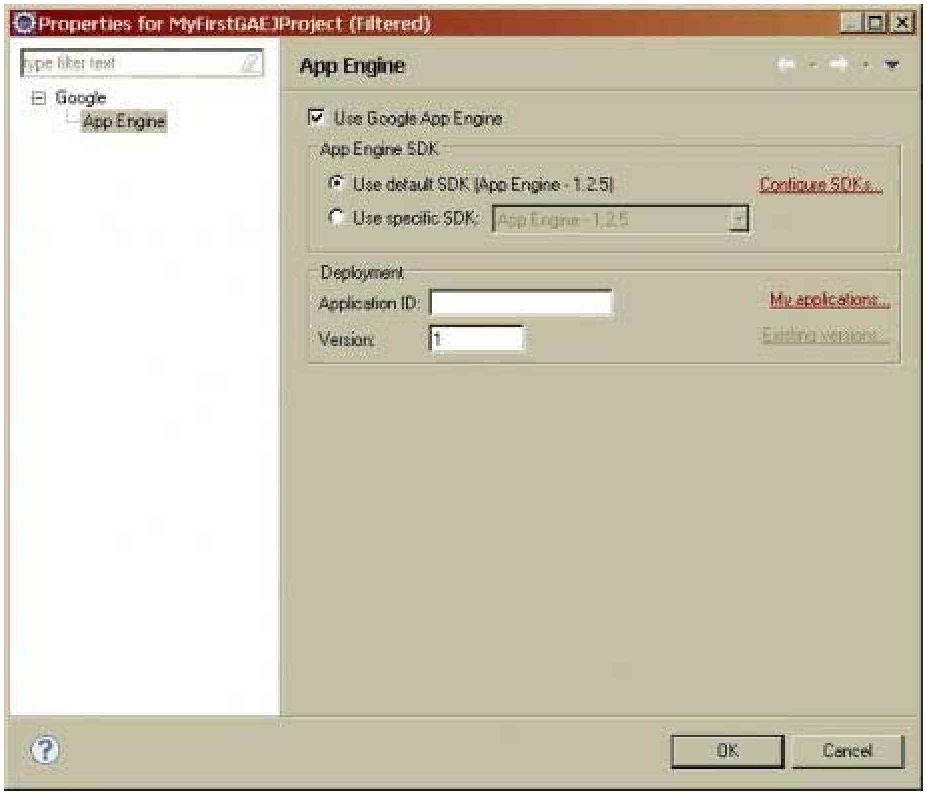
Deploy Project to Google App Engine

Deploy

MyFirstGAEJProject does not have an application ID.
Click the project settings link below to set it.

Project: MyFirstGAEJProject
Email:
Password:

Click on the **App Engine Project settings** link in the above dialog. This will bring up a properties page as shown where you need to entire the **Application ID**. Remember the Application ID that we discussed earlier and you registered one for you. Well that is the Application Id that you need enter over here. Enter that and click OK as shown below [Remember to enter your Application ID before clicking OK]:



Properties for MyFirstGAEJProject (Filtered)

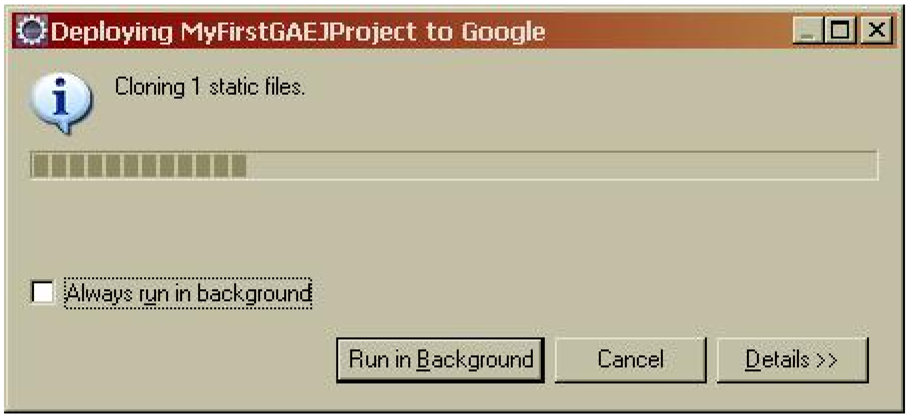
App Engine

Use Google App Engine: ✓

App Engine SDK
(● Use default SDK (App Engine - 1.2.5)) Use specific SDK: App Engine - 1.2.5

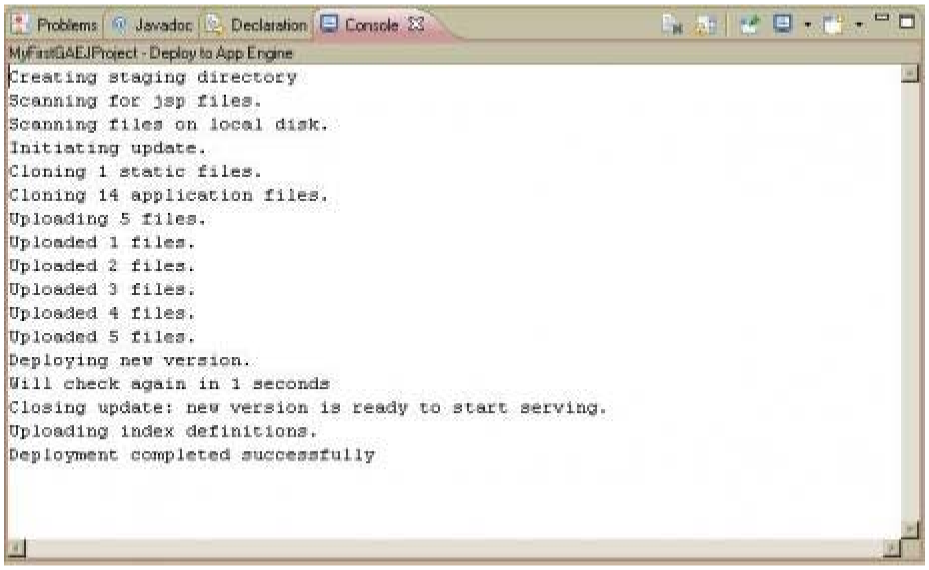
Deployment
Application ID:
Version: 1

This will lead you back to the Deploy Project to Google App Engine dialog box. Now simply click Deploy. This will connect to Google App Engine and deploy your application and register it with the Application Id that you used. A snapshot of the Progress dialog on my machine is shown below:



```
Cloning 1 static files.
```

You will a lot of activity in your Console bar (a snapshot is shown) and at the end of it you should see a message saying that the application is available.



```
Creating staging directory
Scanning for jsp files.
Scanning files on local disk.
Initiating update.
Cloning 1 static files.
Cloning 14 application files.
Uploading 5 files.
Uploaded 1 files.
Uploaded 2 files.
Uploaded 3 files.
Uploaded 4 files.
Uploaded 5 files.
Deploying new version.
Will check again in 1 seconds
Closing update: new version is ready to start serving.
Uploading index definitions.
Deployment completed successfully
```

You can deploy your application several times as you make changes. There is a certain quota limit to how many times you can deploy it in a day and you can read that up but for the moment, you can rest assured that the number is quite huge.

Congratulations! You now have your own application hosted on the Google App Engine. It is already available to the world and that is what we will verify in the next section.

Testing the hosted/online version of our application

Your application is available under the url :
http://YOUR_APPLICATION_ID.appspot.com. So if you **Application ID** is **myapp**, the url is **http://myapp.appspot.com**. Go ahead and access your URL.

Technically it is even possible to have different versions running, but for the purposes of this article we will limit it to the latest version that you deploy and which is available under the url that I have mentioned above.

Next Steps

So we have ended our mini journey for now in which we did the following:

- Registered for Google App Engine
- Registered our first application
- Setup the Developer Environment for developing and deploying GAEJ Java applications
- Used the Google Eclipse Plug-in to develop our first GAEJ application
- Tested and deployed our first GAEJ application

Thanks for joining in. I hope that this gives just about enough push to have your environment all set up correctly along with the complete end to end experience.

The next episode will contain details on a great feature that has got recently added to GAEJ and which is XMPP Support. I have used that to build the Mumbai Blood Bank Locator Agent. Go give it a try! And stay tuned for the next episode that will cover in detail on you can build your own **Agent 007**.

Episode 2 : GAEJ + XMPP and rolling your own Agent

I hope all of you have got started with development of GAEJ applications using the nifty Google Eclipse plugin. If not, I strongly recommend that you read it here. In this episode I will be discussing one significant enhancement that Google has brought about to their latest GAEJ release and that is **XMPP** support.

XMPP Protocol has for long been the backbone for a lot of systems till now and most likely in the future. It stands for Extensible Messaging and Presence Protocol. Whew! That was quite a complicated expansion if you ask me. But fear not. It emerged from the Jabber protocol and is in use today in a lot of Instant Messaging Systems. Google Talk is a prime example of an application that utilizes XMPP as its backbone. Several other IM clients utilize XMPP too. Several XMPP APIs in a variety of languages are available too for you to hack your XMPP client any time.

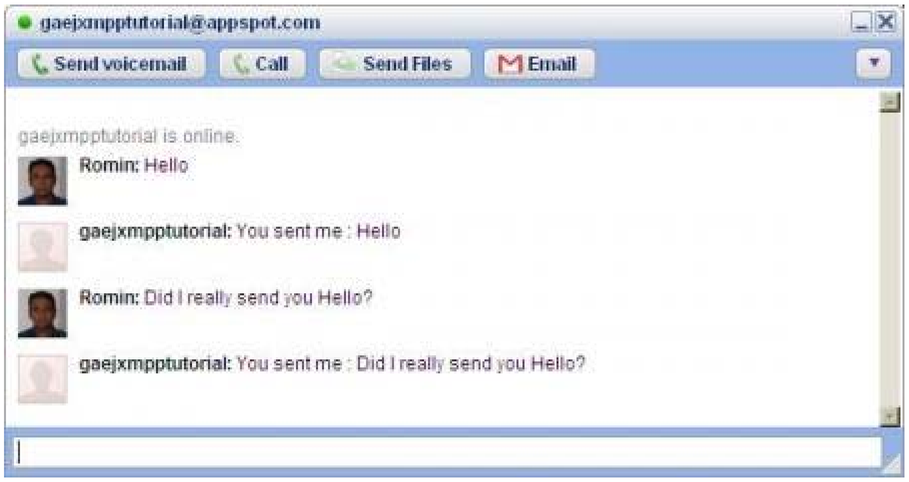
Agent in Action

What are we going to build today, you ask? We are going to a build a XMPP Agent that we can add in IM clients like Google and talk to it. We are going to cover here all that is needed for you as a Java developer to use GAEJ and get a basic XMPP Agent application working.

To see it in action, I suggest you do the following:

1. Start Google Talk on your machine.
2. Add **gaejxmpptutorial@appspot.com** as a friend. If it asks you to add itself as a friend, please do so.
3. Send it any message you like. For e.g. Hello.
4. It will reply back with whatever you typed.

Here is a running sample screenshot of what we will be achieving today:



gaejxmpptutorial is online.

Romin: Hello

gaejxmpptutorial: You sent me : Hello

Romin: Did I really send you Hello?

gaejxmpptutorial: You sent me : Did I really send you Hello?

We will cover how to write the same XMPP Agent (gaejxmpptutorial) using GAEJ. Once you get the mechanics in place, the world is your oyster when it comes to writing Agents that do what you want it to do. All you need to then write is your Agent logic and GAEJ platform does the rest for you.

First things first

Here are a few things that I shall assume:

1. You have an appropriate Eclipse setup along with the GAEJ for Eclipse plugin configured correctly
2. You have a Google Account and know what it means to register an application under your account

If you are unclear, I suggest to read up my first episode.

Our "Hello World" XMPP Agent

At a high level, we will be creating a XMPP Agent that will do the following:

- The Agent will be hosted in the GAEJ system. The application name will be any application id that you select for your account (More on that while deploying).
- This Agent can be added as your friend/buddy in Google Talk or any XMPP IM client
- You can type out a message to the Agent and it will simply send back the message that you typed.

Alright, let's go.

Create a New Project

The first thing to do is to create a New Google Web Application Project. Follow these steps:

1. Either click on **File –> New –> Other** or press **Ctrl-N** to create a new project. Select **Google** and then **Web Application project**. Alternately you could also click on the **New Web Application Project** Toolbar icon as part of the Google Eclipse plugin.
2. In the **New Web Application Project** dialog, **deselect** the **Use Google Web Toolkit** and give a name to your project. I have named mine **GAEJXMPPTutorial**. I suggest you go with the same name so that things are consistent with the rest of the article, but I leave that to you.
3. Click on **Finish**

This will generate the Google Web Application Project, which consists of a sample Servlet that prints out “Hello, World”.

Understand the GAEJ XMPP API

Before we write our Agent, we need to understand first the Google XMPP API. I will cover it in brief over here and the rest of the details you can understand from the link provided, once you need to dig deeper. Make no mistake, the API is detailed and you will need to refer to the finer points as the scope of your XMPP interaction increases but for our starter version, we do not need to know too much.

Think for a moment that you are the Agent that people are going to add to your Google Talk IM client. What do you need to do at a basic level. It all boils down to this:

1. Receiving a message
2. Interpreting it and composing an appropriate response message (This is where your Application logic will come in).
3. Sending a message.

Let us look at each of the 3 areas now and what the API looks like. Please note that all XMPP API classes are present in **com.google.appengine.api.xmpp.*** packages.

Receiving a message

Since XMPP support is now part of the GAEJ infrastructure, all GAEJ applications are capable of receiving an incoming XMPP Message. So once your application is hosted in the GAEJ cloud, the application can receive messages from other XMPP clients like Google Talk. To enable this receiving of XMPP messages from a code level, you need to do two things:

- Enable your application to receive XMPP messages by configuring the XMPP Service in your application. This is straightforward and all you need to do is add the following element to the **appengine-web.xml** file. The **appengine-web.xml** file as you know is specific to the Google Java Web Application project and is used for configuring certain services, XMPP being one of them. It is found in the **war\WEB-INF** folder of your Web Application Project. The XML fragment to add at the end but before the </**appengine-web-app**> element.

```
<inbound-services>
<service>xmpp_message</service>
</inbound-services>
```

We will cover this in detail again later, but this is sufficient for the moment.

- Configure and code a Java Servlet that will receive the incoming Message. All XMPP messages to your application are delivered via **POST** to following URL path in your application: **/_ah/xmpp/message/chat/.** So you will need to configure the servlet like the following snippet in the **web.xml** file, present in the **war\WEB-INF** folder of your Web Application Project.

```
<servlet>
<servlet-name>xmppreceiver</servlet-name>
<servlet-
class>com.gaejexperiments.xmpptutorial.XMPPAgentServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>xmppreceiver</servlet-name>
<url-pattern>/_ah/xmpp/message/chat/</url-pattern>
</servlet-mapping>
```

In the above snippet, you will find the fixed URL path **/_ah/xmpp/message/chat/** configured as the **<url-pattern/>**. And then I have a Java Servlet class **com.gaejexperiments.xmpptutorial.XMPPAgentServlet** as the **<servlet-class>**.

Now, all we have to do is write our Servlet. As mentioned, the incoming XMPP Message will be **POST**ed to our Servlet, so we need a simple doPost(…) implemented in our Servlet. The skeleton is shown below:

```
 package com.gaejexperiments.xmpptutorial;
 import com.google.appengine.api.xmpp.*;
 //Other imports
 public class XMPPAgentServlet extends HttpServlet {

 public void doPost(HttpServletRequest req, HttpServletResponse resp)
throws IOException
 {
 //get the incoming message from the Request object i.e. req
 // interpret it and compose a response
 //send the response message back
 }
 }
```

Interpreting the incoming message and composing a response

To interpret the incoming message, we need to extract out the message that has been passed to us in the **HTTP Request**. It is done in the following manner:

```
XMPPService xmpp = XMPPServiceFactory.getXMPPService();
Message msg = xmpp.parseMessage(req);
```

The **msg** object will contain the XMPP Message. There are several elements in the XMPP message that are of interest to us and chief among them being who has sent the message and what is the text of the message that has been sent to us. This is extracted out by the getter methods available on the **Message** class as shown below:

```
JID fromJid = msg.getFromJid();
String body = msg.getBody();
```

The JID class represents the Jabber ID of the sender whereas the body will contain text of the message sent to us.

Once you have the message, you can interpret it and then create a XMPP message that you need to send back i.e. respond with. To compose an XMPP Message, here is the API:

```
Message replyMessage = new MessageBuilder()
.withRecipientJids("JABBER_ID_OF_RECIPIENT")
.withBody("TEXT_TO_SEND_TO_RECIPIENT")
.build();
```

The above method is straightforward and we will see it more in the next section.

Sending a message

Finally, we have our message and all we need to do is send it. To do that, here is the API. We first check if the recipient is available i.e. Online and then send the message using the **sendMessage** method.

```
XMPPService xmpp = XMPPServiceFactory.getXMPPService();
//Compose the Message Object i.e. replyMessage
if (xmpp.getPresence("JABBER_ID_OF_RECIPIENT").isAvailable()) {
SendResponse status = xmpp.sendMessage(replyMessage);
//Take appropriate action based on status SUCCESS or FAIL i.e. log an
error, update database, etc
}
```

Write our Servlet and configure it

Create a new Java Servlet class named **XMPPAgentServlet** in the package **com.gaejexperiments.xmpptutorial**. You can use your own namespace and Java class name if you wish. The entire source code for the Agent is shown below:

```
package com.gaejexperiments.xmpptutorial;

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

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.google.appengine.api.xmpp.*;
import java.util.logging.Level;
import java.util.logging.Logger;

@SuppressWarnings("serial")

//STEP 1
```

```
public class XMPPAgentServlet extends HttpServlet {
public static final Logger _log =
Logger.getLogger(XMPPAgentServlet.class.getName());
public void doPost(HttpServletRequest req, HttpServletResponse resp)
throws IOException
{
try {
String strStatus = "";
XMPPService xmpp = XMPPServiceFactory.getXMPPService();
//STEP 2
Message msg = xmpp.parseMessage(req);
JID fromJid = msg.getFromJid();
String body = msg.getBody();
_log.info("Received a message from " + fromJid + " and body = " +
body);
//STEP 3
String msgBody = "You sent me : " + body;
Message replyMessage = new MessageBuilder()
.withRecipientJids(fromJid)
.withBody(msgBody)
.build();
//STEP 4
boolean messageSent = false;
if (xmpp.getPresence(fromJid).isAvailable()) {
SendResponse status = xmpp.sendMessage(replyMessage);
messageSent = (status.getStatusMap().get(fromJid) ==
SendResponse.Status.SUCCESS);
}
//STEP 5
if (messageSent) {
strStatus = "Message has been sent successfully";
}
else {
strStatus = "Message could not be sent";
}
_log.info(strStatus);
} catch (Exception e) {
_log.log(Level.SEVERE,e.getMessage());
}
}
}
```

The code should be familiar now but let us go through each of the steps above:

1. We have written a Java Servlet that extends the HttpServlet class and implements the doPost(…) method since the XMPP message will be delivered via POST to the application.
2. In this step, we extract out the XMPP Message object from the HTTP request. Then we invoke a couple of getter methods on the Message object to get the Jabber Id from where we got the message and also the body of the message. We log this into the log file.
3. We compose a string named **msgBody** that simply echoes back what was sent by the sender and then we compose a XMPP Message object named **replyMessage**

to be sent to the Jabber Id that sent us this message i.e. **fromJid** along with the text that we want to send in the message i.e. **msgBody**

4. We detect if the recipient that we are sending the response to i.e. **fromJid** is available i.e. online. If yes, we send the message using the **sendMessage** method. Then we determine if the message was sent successfully or not i.e. **messageSent**.
5. Finally, depending on the message status, we log the appropriate message in the log file.

To complete our Servlet development, we will also need to add the **<servlet/>** and **<servlet-mapping/>** entry to the **web.xml** file. The necessary elements to be added to your **web.xml** file are shown below. Please note that you can use your own namespace and servlet class. Just modify it accordingly if you do so. But make sure that the fixed path URL i.e. **/_ah/xmpp/message/chat/** is correctly mentioned in the **<url-pattern/>**, since that is the path over which the Google XMPP Infrastructure will deliver the message to you via a **POST**.

```
<servlet>
<servlet-name>xmppreceiver</servlet-name>
<servlet-
class>com.gaejexperiments.xmpptutorial.XMPPAgentServlet</servlet-class>
</servlet><servlet-mapping>

<servlet-name>xmppreceiver</servlet-name>
<url-pattern>/_ah/xmpp/message/chat/</url-pattern>
</servlet-mapping>
```

Finally, we have used the INFO level to log if the message was sent out successfully or not, so we will have the change the logging level by modified the **logging.properties** file present in the **war\WEB-INF** folder. The necessary line after modification is shown below:

```
# Set the default logging level for all loggers to INFO
.level = INFO
```

Configure the XMPP Service for our Application

To enable your application to receive XMPP messages, you need to configure the XMPP service in your application. This is done by adding the XMPP service to the **appconfig-web.xml** file that is found in the **war\WEB-INF** folder of the Web Application project. Shown below is **appconfig-web.xml** and the relevant portion that you need to add to the file.

```
<?xml version="1.0" encoding="utf-8"?>
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
...
<inbound-services>
<service>xmpp_message</service>
</inbound-services>
```

```
...
</appengine-web-app>
```

Deploy the Application

To deploy the application, you will need to first create your **Application ID**. The Application Identifier can be created by logging in at http://appengine.google.com with your Google Account. You will see a list of application identifiers already registered under your account (or none if you are just getting started). To create a new Application, click on the **Create Application** button and provide the **Application Identifier** as requested. Please note this name down since you will be using it for deployment.

For e.g. I have registered an application identifier named ***gaejxmpptutorial.***

To deploy the application, click on the **Deploy** Icon in the Toolbar. This will present the dialog as shown below:

You will need to provide your Email and Password. **Do not click on Deploy button yet.** Click on the **App Engine Project settings link**. This will lead you to the screen as shown below, where you will need to enter your **Application ID [**For e.g. shown below is my Application Identifier ***gaejxmpptutorial*]**

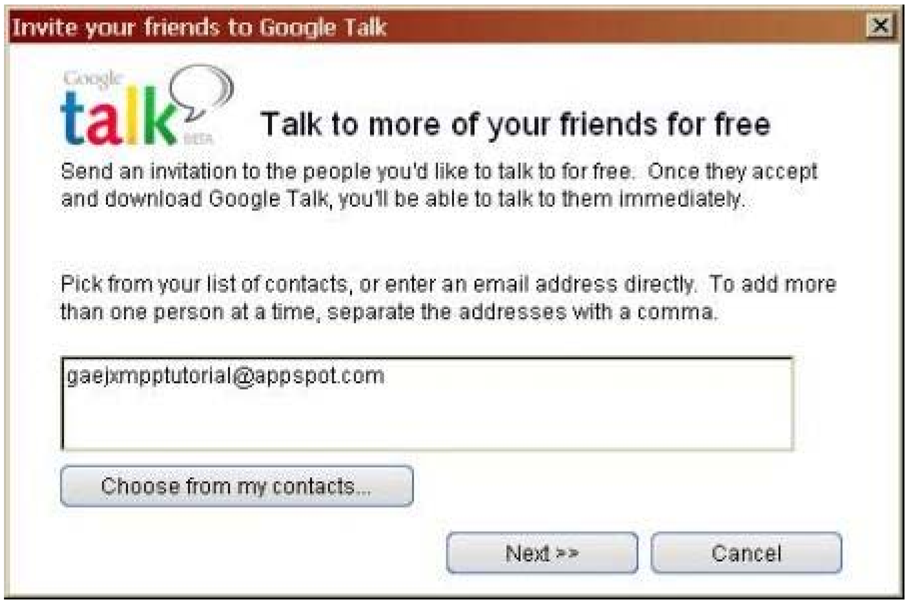
Click on **OK.** You will be lead back to the previous screen, where you can click on the **Deploy** button. This will start deploying your application to the GAEJ cloud. You should see several messages in the **Console** window as shown below as the application is being deployed:

You should see the message **"Deployment completed successfully"** as shown above. This means that you application is ready to serve.

See it in Action

Now that your Agent application has been deployed, it is time to communicate with it. To do that you will need to add your Agent as a friend/contact in your IM application. To do that you will need to know the ID of the Agent. This is made simple for you by GAEJ. If your Application ID is for e.g. **myagent** and your application is **XMPP** enabled, then the ID for your application will be **myagent@appspot.com**. In other words the ID is APPLICATION_ID@appspot.com where APPLICATION_ID is the application identifier for your application that you registered.

In my case, it was ***gaejxmpptutorial***, so all I did was start up Google Talk and clicked on **Add Contact** to add the Agent as shown below.


Invite your friends to Google Talk

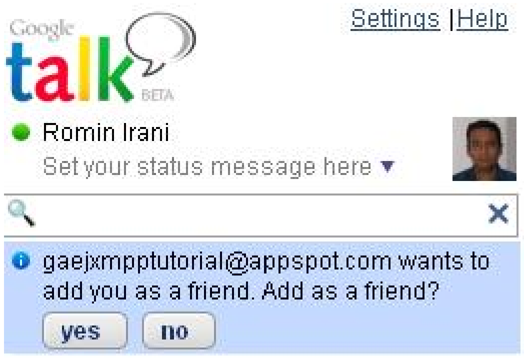
Talk to more of your friends for free

Send an invitation to the people you'd like to talk to for free. Once they accept and download Google Talk, you'll be able to talk to them immediately.

Pick from your list of contacts, or enter an email address directly. To add more than one person at a time, separate the addresses with a comma.

gaejxmpptutorial@appspot.com

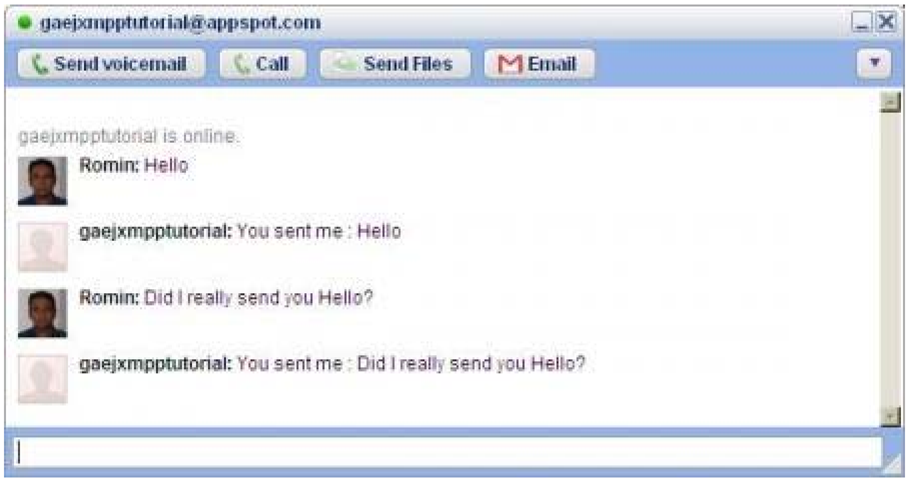
Once you click on Next and finish the invitation, you will receive a message in your Google Talk as shown below, where the Agent is requested to add you as friend. It is important that you accept that , so that it can send messages to you.



gaejxmpptutorial@appspot.com wants to add you as a friend. Add as a friend?

Once you do that, you will find that the Agent is available as shown below. The Agent is available **because your Application is online and running in the cloud. Cool, isn't it?**

Now you can start the conversation and see your Agent in action as shown below:



gaejxmpptutorial is online.

Romin: Hello

gaejxmpptutorial: You sent me : Hello

Romin: Did I really send you Hello?

gaejxmpptutorial: You sent me : Did I really send you Hello?

Moving forward

You should now be in a position to think, code and deploy XMPP Agents of your choice using GAEJ. I would love to see you write a few and sharing it out here with all of us. You could write Agents that could do the following:

- Play a word game
- Provide weather, sports, traffic,etc updates
- Examination Results
- Ticket Confirmation
- Train, Air Tickets routes… almost anything that you can think of.

If you would like to get an idea of a sample XMPP Agent written using GAEJ, look no further than the Mumbai Blood Bank Locator Agent that I wrote a while back. Take a look and inspire all of us with your next cool Agent 007.

Episode 2 : Update : Communicating to another XMPP Account via your Bot

I received a good comment on my second post, where we saw how to write your own XMPP Bot and deploying it on the Google App Engine for Java (GAEJ).

It asked whether one can communicate to another Jabber account from the XMPP Bot hosted on the GAEJ. The confusion arose because I covered the usage of Google Talk, which is an IM client itself. And it looked from my post that you can **only use Google Talk** as the client to talk the XMPP agent that you wrote.

The **short answer** is that **yes**, even if you are using another Instant Messaging (IM) client like Spark or Pidgin, it is possible to communicate to the XMPP bot that you have written and which is running inside the GAEJ cloud.

However, it turned out that **I made a mistake** in writing the code for the XMPP bot. The Google App Engine XMPP documentation clearly states that you can communicate to any other Jabber ID but **it is not possible for their infrastructure to check for the presence of a Jabber ID on another network** except their Google Talk network. This is fair enough. By presence, we are simply trying to see if the user is online or not.

So , we had some code in our XMPP bot that we saw earlier, which went something like this. This code fragment was at the very end when we are sending back the echo for our Agent.

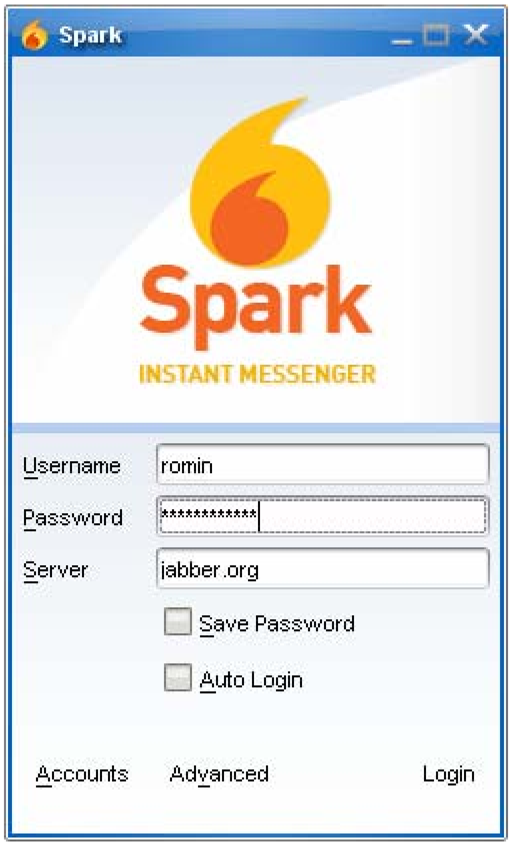
```
if (xmpp.getPresence(fromJid).isAvailable()) {
SendResponse status = xmpp.sendMessage(replyMessage);
messageSent = (status.getStatusMap().get(fromJid) ==
SendResponse.Status.SUCCESS);
 }
```

What we are doing here is that we are checking for the presence of another JabberID i.e. making sure that is online before sending the message. This works fine on the Google Network but not on other XMPP Networks like Jabber.org. So for e.g. our bot Jabber Id was **gaejxmpptutorial@appspot.com**. And if the Jabber Id which is talking to it is **someid@jabber.org**, then it will not be possible for your bot running inside of the Google network to determine if **someid@jabber.org** is online or not. As a result of this, the code will never enter the block and the message does not get sent.

So, all you have to do to send the message across to another network is to remove the check for the isAvailable() condition and simply call the sendMessage(…) method.

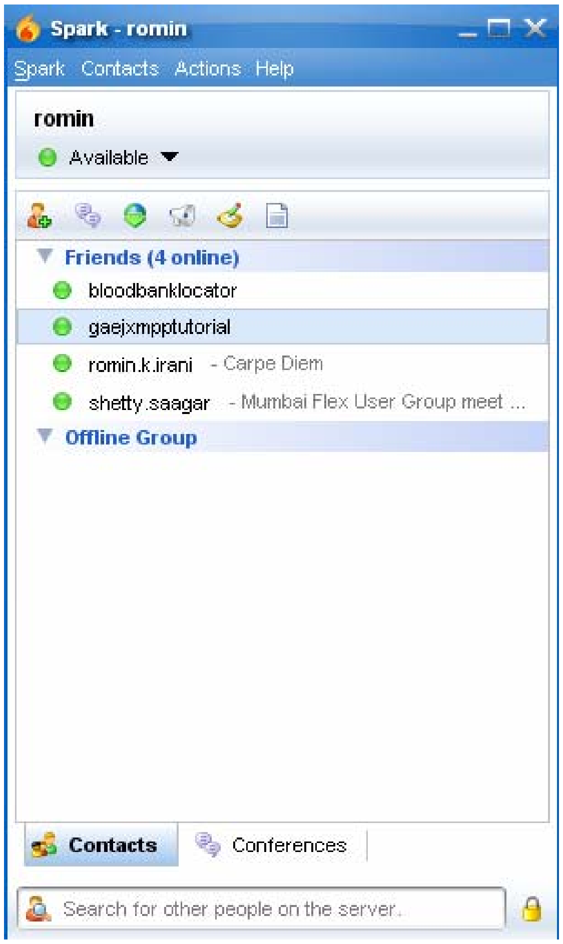
I have verified this by doing the following:

1. Used another IM client instead of Google Talk. I used the IM client called Spark
2. I have an account at **jabber.org** which I used to login as shown below in the Spark IM client:

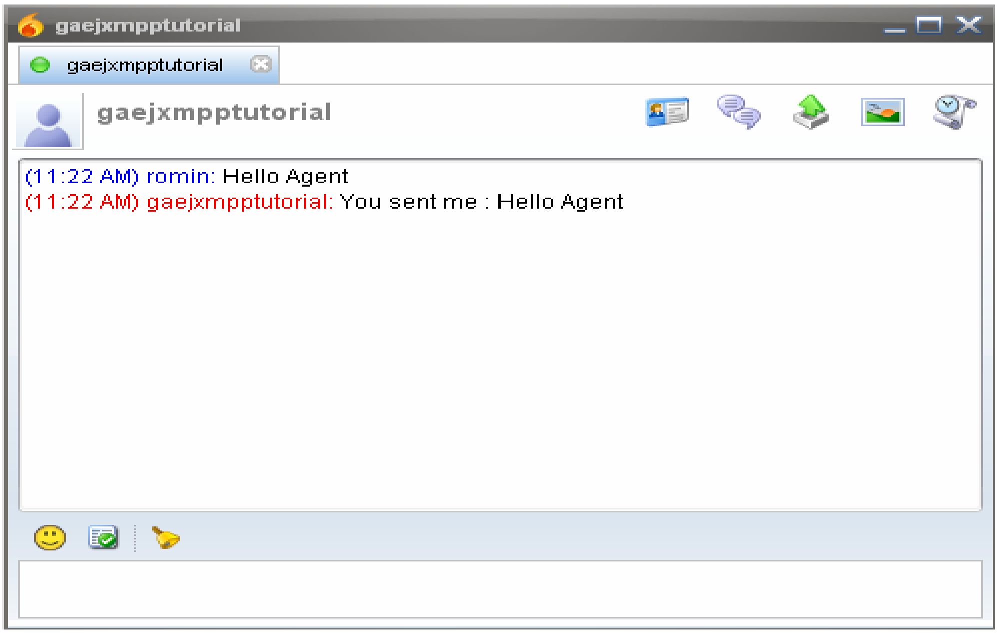


Username: romin
Password: *************
Server: jabber.org

3. I added **gaejxmpptutorial@appspot.com** as a contact in Spark IM client


Spark - romin
Spark Contacts Actions Help
romin
Available
Friends (4 online)
bloodbanklocator
gaejxmpptutorial
romin.k.irani - Carpe Diem
shetty.saagar - Mumbai Flex User Group meet ...
Offline Group
Contacts
Conferences
Search for other people on the server.

4. I sent a chat message across to the **gaejxmpptutorial** bot and got back the echo as shown below:



```
(11:22 AM) romin: Hello Agent
(11:22 AM) gaejxmpptutorial: You sent me : Hello Agent
```

Hope this clarifies that it is possible to communicate to your bot over XMPP from other XMPP Servers. Let me know if any of you are still having issues and I welcome your comments.

Episode 3: Using the GAEJ Email Service

Hope you have got going with the Google App Engine for Java (GAEJ) through the first two episodes where we got started with it in the first episode and then wrote our own XMPP bot in the second episode. In case you still need to get an introduction to the mechanics, please do not hesitate from reading the first episode.

In this episode, we are going to look at the Email Service in GAEJ. You can read the official documentation provided at the Google site over here. We will be keeping this episode simple but the key to take away from this episode is the ease with which you can integrate email sending into your GAEJ hosted applications. Please note that the service currently provides the ability to only send emails/attachments, etc. There is no provision at this point of time for your GAEJ hosted application to **receive** emails.

Our approach to this episode will be consistent with the earlier ones where we will follow these steps:

- See the application work first
- Create a new project and write our code that utilizes the Email Service
- Deploy and see it work

Email Service in Action

My approach in this and forthcoming episodes will be to focus on the Server side architecture and not much on the User Interface. What it means is that I could have demonstrated the Email Service by coding up a simple HTML form that accepts the To , Subject and Body of the Email to be sent and then it will hit a Server side script hosted in your GAEJ application and send off the email. But instead, I will assume that you are comfortable with whatever client (HTML, Flex, Java, etc) programming language and simply need a way to figure out what to do on the Server side. Hence we will focus on the Server side implementation.

To see the Email Service in Action, I have already the hosted the application that I wrote. The application id is **gaejexperiments** and hence available at **http://gaejexperiments.appspot.com.** In case you are still not clear on the Application Id, I suggest you read the first episode here.

To send an email, we will use a REST like style to invoke the Email Service. This is nothing much a servlet that has been hosted in my **gaejexperiments** application. To invoke the Email Service, all you need to do is punch in the following line in your favourite browser (IE, Firefox, Chrome, Safari,etc) and press enter:

```
http://gaejexperiments.appspot.com/gaejemail?email_to=[YourEmailId]&ema
il_subject=[EmailSubject]&email_body=[EmailBody]
```

You need to substitute the values [YourEmailId],[EmailSubject] and [EmailBody] with your email id, a sample subject and a sample body (message text). **Please make sure to encode your text as appropriate in case you have special characters, whitespaces, etc.**

An example is shown here:

```
http://gaejexperiments.appspot.com/gaejemail?email_to=romin.k.irani@gma
il.com&email_subject=Hi&email_body=Test
```

This should send out an email (on my behalf) and you will see a plain text message returned by our Service, which will indicate if it succeeded or failed in sending out the email. The sender of the message is coded in the service itself and we are not giving a provision here to specify that. It is just the way that I have designed the service and you can do it your way.

Remember that our application delivers the message onto the Google Infrastructure to deliver the message and then it will take over and try to deliver the message. That is pretty much standard stuff.

Develop our Project and utilize the Email Service

The first thing to do is to create a **New Google Web Application** Project. Follow these steps:

1. Either click on File –> New –> Other or press Ctrl-N to create a new project. Select Google and then Web Application project. Alternately you could also click on the New Web Application Project Toolbar icon as part of the Google Eclipse plugin.
2. In the New Web Application Project dialog, deselect the Use Google Web Toolkit and give a name to your project. I have named mine GAEJExperiments. I suggest you go with the same name so that things are consistent with the rest of the article, but I leave that to you.
3. Click on Finish

This will generate the project and also create a sample Hello World Servlet for you. But we will be writing our own Servlet.

Coding the GAEJEmailServlet.java

Create a new Servlet in your Project as shown below. I have created the GAEJEmailServlet.java in the package com.gaejexperiments.email. You can choose a package of your choice. The code is straightforward and is listed below:

```
package com.gaejexperiments.email;
```

```
import java.io.IOException;
import java.util.Properties;

import javax.mail.Message;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import javax.servlet.ServletException;
import javax.servlet.http.*;

@SuppressWarnings("serial")
public class GAEJEmailServlet extends HttpServlet {
 public void doGet(HttpServletRequest req, HttpServletResponse resp)
 throws IOException {

 String strCallResult = "";
 resp.setContentType("text/plain");
 try {
 //Extract out the To, Subject and Body of the Email to be sent
 String strTo = req.getParameter("email_to");
 String strSubject = req.getParameter("email_subject");
 String strBody = req.getParameter("email_body");

 //Do validations here. Only basic ones i.e. cannot be null/empty
 //Currently only checking the To Email field
 if (strTo == null) throw new Exception("To field cannot be empty.");

 //Trim the stuff
 strTo = strTo.trim();
 if (strTo.length() == 0) throw new Exception("To field cannot be
empty.");

 //Call the GAEJ Email Service
 Properties props = new Properties();
 Session session = Session.getDefaultInstance(props, null);
 Message msg = new MimeMessage(session);
 msg.setFrom(new InternetAddress("#YOUR EMAIL ADDRESS HERE#"));
 msg.addRecipient(Message.RecipientType.TO,
 new InternetAddress(strTo));
 msg.setSubject(strSubject);
 msg.setText(strBody);
 Transport.send(msg);
 strCallResult = "Success: " + "Email has been delivered.";
 resp.getWriter().println(strCallResult);
 }
 catch (Exception ex) {
 strCallResult = "Fail: " + ex.getMessage();
 resp.getWriter().println(strCallResult);
 }
 }

 @Override
 public void doPost(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException, IOException {
 doGet(req, resp);
 }

}
```

Let us go through the main pieces of the code:

1. I have provided both **GET** and **POST** handlers in the servlet and the **POST** handler simply invokes the **GET** handler here.
2. Then we parse the request parameters for the recipient email address (**email_to**) and the subject and body of the email i.e. **email_subject** and **email_body** respectively.
3. We do some basic validation on the **email_to** field to make sure that it is not empty. I am currently not doing at validation on the subject and body fields. Additionally, you can easily modify this to receive more than one email address and loop through them yourself.
4. Finally we have the block of code that sends email. This is pretty much standard **JavaMail** stuff that is used over here. We get a mail session object. Then we create the **MimeMessage** in which we can populate the sender (from Email id), the recipient and finally the subject and body. We use the Transport object to send the email out and return back a plain text string indicating success or failure in delivering the email to the transport.
5. Please note that the sender email id needs to be populated by you. I have intentionally left it as #YOUR EMAIL ADDRESS HERE# in the code above.

Configuring the Servlet in web.xml file

To complete our Servlet development, we will also need to add the **<servlet/>** and **<servlet-mapping/>** entry to the **web.xml** file. This file is present in the WEB-INF folder of the project. The necessary fragment to be added to your **web.xml** file are shown below. Please note that you can use your own namespace and servlet class. Just modify it accordingly if you do so.

```
<servlet>
 <servlet-name>GAEJEmailServlet</servlet-name>
 <servlet-class>com.gaejexperiments.email.GAEJEmailServlet</servlet-
class>
 </servlet>
 <servlet-mapping>
 <servlet-name>GAEJEmailServlet</servlet-name>
 <url-pattern>/gaejemail</url-pattern>
 </servlet-mapping>
```

Deploy the Application and utilize it

To deploy the application, you will need to first create your **Application ID**. The Application Identifier can be created by logging in at http://appengine.google.com with your Google Account. You will see a list of application identifiers already registered under your account (or none if you are just getting started). To create a new Application, click on the Create Application button and provide the Application Identifier as requested. Please note this name down since you will be using it for deployment.

For e.g. I have registered an application identifier named **gaejexperiments**.

To deploy the application, follow these steps (they should be familiar to you now):

1. Click on the Deploy Icon in the Toolbar.
2. In the Deploy dialog, provide your Email and Password. Do not click on Deploy button yet.
3. Click on the **App Engine Project settings** link. This will lead you to a dialog, where you need to enter your Application ID [For e.g. my Application Identifier **gaejexperiments**]
4. Click on **OK**. You will be lead back to the previous screen, where you can click on the **Deploy** button. This will start deploying your application to the GAEJ cloud. You should see several messages in the Console window as the application is being deployed.
5. Finally, you should see the message “Deployment completed successfully”.

This means that you application is ready to serve and you can verify the Email service by invoking the appropriate url which you can punch into your browser. The format will be the following:

```
http://[YourApplicationId].appspot.com/gaejemail?email_to=[YourEmailId]
&email_subject=[EmailSubject]&email_body=[EmailBody]
```

Moving forward

In this episode, we have seen how to utilize the Email Service provided by the Google App Engine. The code that we covered is straightforward to implement in your application. Do note once again that the Email Service currently does ***not allow*** for functionality that can receive emails. You can study the JavaMail API in more detail to understand how to do things like sending attachments in your email too. You can do that using the Google App Engine Email Service.

The other key thing to take away from this episode is also a focus on the server side api rather than building HTML front-ends, etc. I leave that to the user to design and implement since invoking a standards based HTTP GET or POST is now possible in almost any client programming platform.

In the next episode, we shall look at the Networking API available under GAEJ. The focus will be on invoking extrernal URLs and consuming external Web Services or even RSS feeds.

Episode 4: Building a Dictionary App : Using the GAEJ URL Fetch Service

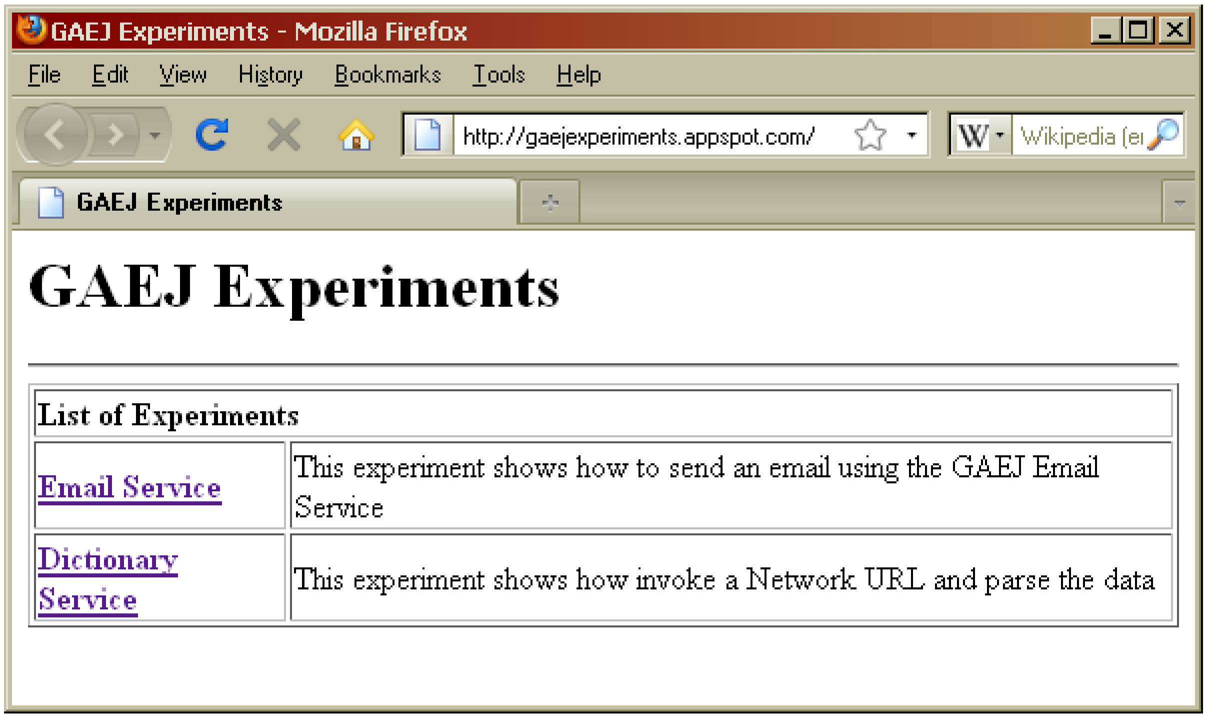
Welcome to Episode 4 of GAEJ Experiments. In this episode, we will learn how your GAEJ Application can invoke external URLs and get data back from them. In the process we will build a fully capable Dictionary Application. While it may not be a complete application, it will demonstrate just about enough how you can integrate external services into your application.

This episode is particularly important given the fact that there are hundreds of public APIs now available out there on the web, which your application can integrate. I strongly recommend visiting The Programmable Web that catalogs hundreds of Web APIs that are available today and which you can integrate into your application.

Dictionary Application in Action

In order to maintain a consistent style across the episodes so far, let us first watch the application in action to better understand what we will be building over here. Follow these easy steps:

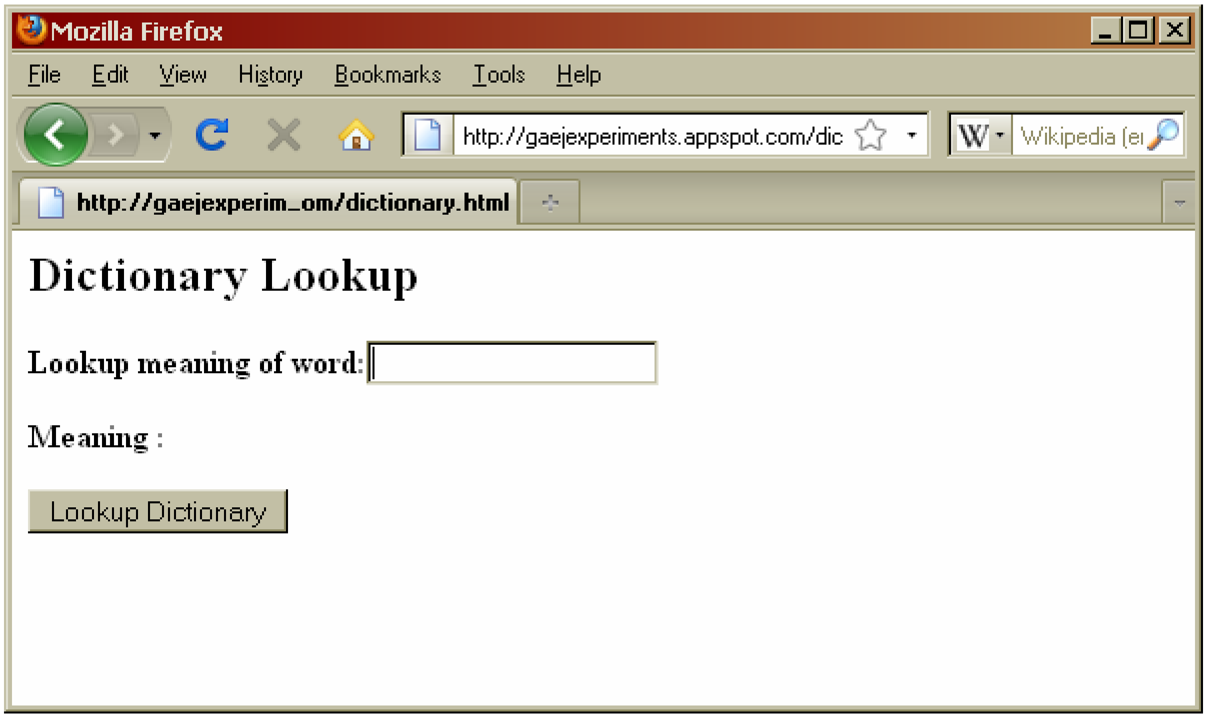
1. Navigate to **http://gaejexperiments.appspot.com**. This will result in a page as shown
   below:


GAEJ Experiments - Mozilla Firefox
File
Edit
View
History
Bookmarks
Tools
Help
http://gaejexperiments.appspot.com/
Wikipedia (e
GAEJ Experiments

GAEJ Experiments

List of Experiments	
Email Service	This experiment shows how to send an email using the GAEJ Email Service
Dictionary Service	This experiment shows how invoke a Network URL and parse the data

2. Click on **Dictionary Service** link. This will lead you to a page shown below:

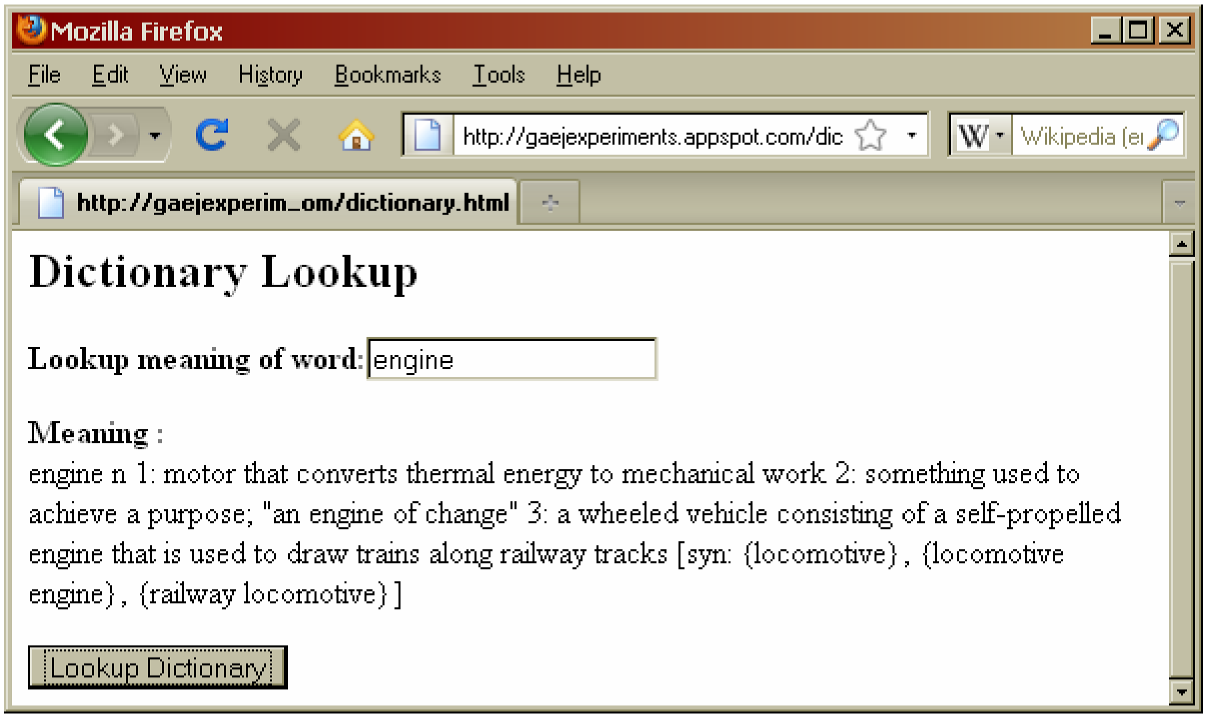

Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://gaejexperiments.appspot.com/dic
Wikipedia (e
http://gaejexperim...om/dictionary.html

Dictionary Lookup

Lookup meaning of word:

Meaning :

3. Enter any word that you wish to lookup in the dictionary and click on **Lookup Dictionary**. For e.g. *engine***.** This will display the meaning of the word as shown below:


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://gaejexperiments.appspot.com/dic
Wikipedia (e
http://gaejexperim...om/dictionary.html

Dictionary Lookup

Lookup meaning of word: engine

Meaning :

engine n 1: motor that converts thermal energy to mechanical work 2: something used to achieve a purpose; "an engine of change" 3: a wheeled vehicle consisting of a self-propelled engine that is used to draw trains along railway tracks [syn: {locomotive}, {locomotive engine}, {railway locomotive}]

Behind the Scenes

Let us understand how the above application is working. The key points to note in the flow are given below:

1. The dictionary page shown in **step 2** above is a simple HTML form page. The only user input needed in this step is the word that you wish to look up in the dictionary.
2. On click of the **Lookup Dictionary** button, a Servlet is invoked in our GAEJ Application. The Servlet will utilize the **GAEJ URL Fetch Service** to invoke an external Dictionary Service.
3. The Dictionary Service is hosted at a site called **http://services.aonaware.com/DictService/DictService.asmx** . One of the service operations is **Define** and you can try it out here. It supports a REST style interface (GET/POST) for invoking it, so that makes our task simple. Give it a try here to better understand what we are integrating or simply punch in the following in your browser:

```
http://services.aonaware.com/DictService/DictService.asmx/Define?
word=trial
```

4. This service takes in a word and returns us a XML Response which contains the meaning of the word in several dictionaries. We parse out the dictionary meaning and return that back to the client (HTML form).
5. The HTML form that renders the meaning as shown in the screen above.

URL Fetch Service

The core piece that we will be utilizing in our application here is an ability to invoke external URLs. This is provided by the **URL Fetch** service of the Google App Engine. It allows us to invoke any external URL and receive the data from the URL. At a high level, usage of the URL Fetch service is pretty simple and a boiler plate code is shown below: [Most details have been omitted]

```
try {
URL url = new URL(PUT_EXTERNAL_URL_HERE);
BufferedReader reader = new BufferedReader(new
InputStreamReader(url.openStream()));

String line;
StringBuffer responseData = new StringBuffer();

//Read the entire response in this loop
while ((line = reader.readLine()) != null) {
responseData.append(line);
}
reader.close();
//Process the data from the responseData StringBuffer
//Might involve Parsing the XML or JSON Response format, etc.

}
catch (Exception e) {
          //...
}
```

This is pretty much standard Java code and it involves opening an inputstream to the external URL and collecting the response in a local variable. You can then decide what to do with the response data received. For e.g. most external services provide a XML or JSON formatted response, which you may have to parse out and then proceed with your application logic.

In our Dictionary Application here, we will be receiving a XML response, which we will simply parse out and return back to the HTML client.

Developing our Application

The first thing to do is to create a **New Google Web Application** Project. Follow these steps:

1. Either click on File –> New –> Other or press Ctrl-N to create a new project. Select Google and then Web Application project. Alternately you could also click on the New Web Application Project Toolbar icon as part of the Google Eclipse plugin.
2. In the New Web Application Project dialog, deselect the Use Google Web Toolkit and give a name to your project. I have named mine GAEJExperiments. I suggest you go with the same name so that things are consistent with the rest of the article, but I leave that to you. **In case you are following the series, you could simply use the same project and skip all these steps altogether. You can go straight to the Servlet Development section.**
3. Click on Finish

This will generate the project and also create a sample Hello World Servlet for you. But we will be writing our own Servlet.

The Front end HTML form [dictionary.html]

Create a new HTML file in the war directory of your application. In that directory, you will find an **index.html** that is generated for you. So you can either use **index.html** or generate a new HTML file as I have done. Name this file as **dictionary.html** and code it as shown below:

The first thing that we will do is to code out a simple HTML form that takes a single parameter as an input. This parameter is the word that we need to lookup in the dictionary. We will use a bit of AJAX here to dynamically call our servlet and display the response that we get.

```
<html>
<head>
<script type="text/javascript">
var xmlhttp;
function lookupDictionary(word)
{
xmlhttp=null;
if (window.XMLHttpRequest)
 {// code for IE7, Firefox, Opera, etc.
 xmlhttp=new XMLHttpRequest();
 }
else if (window.ActiveXObject)
 {// code for IE6, IE5
 xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
 }
if (xmlhttp!=null)
 {
 xmlhttp.onreadystatechange=state_Change;
 var url = "/gaejdictionaryservice?word="+word;
 xmlhttp.open("GET",url,true);
 xmlhttp.send(null);
 }
else
 {
 alert("Your browser does not support XMLHTTP.");
 }
}

function state_Change()
{
if (xmlhttp.readyState==4)
 {// 4 = "loaded"
 if (xmlhttp.status==200)
 {
 // 200 = "OK"
 document.getElementById('DictionaryServiceResponse').innerHTML=xmlhttp
.responseText;
 }
 else
 {
 alert("Problem looking up Dictionary Service :" + xmlhttp.statusText);
 }
 }
}
</script>
</head>

<body>
<h2>Dictionary Lookup </h2>
<hr/>
<h3>(Powered by Aonaware <a
href="http://services.aonaware.com/DictService/">Dictionary
Service</a>)</h3>
<hr/>
<p>
<b>Lookup meaning of word:</b><input type="text" id="word"></input>
</p>
<p><b>Meaning :</b>
<br /><span id="DictionaryServiceResponse"></span>
</p>

<button onclick="lookupDictionary(word.value)">Lookup
Dictionary</button>

</body>
```

Here are some key points from the code above. For most web programmers it is pretty much standard stuff:

1. We have a single **input** field HTML element with an **id** named **word**. There is a button with a label **Lookup Dictionary**, on click on which, we invoke a Javascript function called **lookupDictionary**, passing in the word value.
2. The **lookupDictionary** method builds up the standard XMLHttpRequest object that we shall use to send our request across. The request url is **/gaejdictionaryservice** with a single parameter named **word**, which contains the value of the word that we need to lookup in the dictionary.
3. Note that **/gaejdictionaryservice** is our servlet endpoint that we will be seeing in the next section. This servlet will take it the word parameter, and use the URL Fetch Service to invoke the external Dictionary API and return us back the response.
4. The response when returned is inserted into the **span** element named **DictionaryServiceResponse** in the above HTML form to display the response received.

Next we shall code the heart of our application a **servlet** named **GAEJDictionaryService**.

Coding the GAEJDictionaryService Servlet [GAEJDictionaryService.java]

Create a new Servlet in your Project as shown below. I have created the GAEJDictionaryService.java in the package **com.gaejexperiments.networking**. You can choose a package of your choice. The code is straightforward and is listed below:

```
package com.gaejexperiments.networking;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.StringReader;
import java.net.URL;
import javax.servlet.ServletException;
import javax.servlet.http.*;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathFactory;

import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;

@SuppressWarnings("serial")
public class GAEJDictionaryService extends HttpServlet {
 public void doGet(HttpServletRequest req, HttpServletResponse resp)
 throws IOException {

 String strCallResult = "";
 resp.setContentType("text/plain");
 try {
 //Extract out the word that needs to be looked up in the Dictionary
Service
 String strWord = req.getParameter("word");

 //Do validations here. Only basic ones i.e. cannot be null/empty
 if (strWord == null) throw new Exception("Word field cannot be empty.");

 //Trim the stuff
 strWord = strWord.trim();
 if (strWord.length() == 0) throw new Exception("Word field cannot be
empty.");

 String strDictionaryServiceCall =
"http://services.aonaware.com/DictService/DictService.asmx/Define?word=
";
 strDictionaryServiceCall += strWord;
 URL url = new URL(strDictionaryServiceCall);
 BufferedReader reader = new BufferedReader(new
InputStreamReader(url.openStream()));
 StringBuffer response = new StringBuffer();
 String line;
 while ((line = reader.readLine()) != null) {
 response.append(line);
 }
 reader.close();

 strCallResult = response.toString();

 DocumentBuilderFactory builderFactory =
DocumentBuilderFactory.newInstance();
 DocumentBuilder builder = builderFactory.newDocumentBuilder();
 Document doc = builder.parse(new InputSource(new
StringReader(strCallResult.toString())));

 XPathFactory factory = XPathFactory.newInstance();
 XPath xpath = factory.newXPath();
 XPathExpression expr =
xpath.compile("//Definition[Dictionary[Id='wn']]/WordDefinition/text()"
);

 Object result = expr.evaluate(doc, XPathConstants.NODESET);
 NodeList nodes = (NodeList) result;
 for (int i = 0; i < nodes.getLength(); i++) {
 strCallResult = nodes.item(i).getNodeValue();
 }

 resp.getWriter().println(strCallResult);

 }
 catch (Exception ex) {
 strCallResult = "Fail: " + ex.getMessage();
 resp.getWriter().println(strCallResult);
 }
 }

 @Override
 public void doPost(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException, IOException {
 doGet(req, resp);
 }

}
```

Let us go over the key points in the code above:

1. I have provided both **GET** and **POST** handlers in the servlet and the **POST** handler simply invokes the **GET** handler here.
2. Then we parse the request parameters for the word parameter that we need to look up in the dictionary (**word**) and we do some basic validation to make sure that it is not empty.
3. The Dictionary Service that we plan to use is available at the following URL:

```
http://services.aonaware.com/DictService/DictService.asmx/Define?
word=[YOUR_WORD_HERE]
```

4. In the above URL, you need to provide the word. So what we do in the code is to simply append the word request parameter that was passed.
5. Next, we use the URL Fetch Service as discussed earlier to collect the entire response.
6. The response returned to use is in XML format and the service returns us the meaning of the word based on 6 dictionaries. We will be using just one of those dictionaries WordNet 2.0 which is the 3rd definition in the XML. I suggest that you punch in the following url to understand what we will be parsing out here. I have used the word ‘engine’ here.

```
http://services.aonaware.com/DictService/DictService.asmx/Define?
word=engine
```

7. Finally we use XPath. I intentionally used this to demonstrate how easy it is to use XPath to extract out the element text that we are interested in. You are free to choose an alternative way of extracting out the text value. You could use standard SAX/DOM parsing too if you wish. Whatever you are comfortable with will suffice for the example.
8. We first build the Document object by using the standard DocumentBuilderFactory and DocumentBuilder classes.
9. Then on the Document object **doc**, we evaluate the XPath expression. The XPath expression is **//Definition[Dictionary[Id='wn']]/WordDefinition/text()**.
10. The XPath expression given above can be read as following. First consider the //Definition[DictionaryId ='wn']] which means that find all definitions anywhere in the document which have a child element named DictionaryId whose value is ‘wn’. This is the Definition that we are interested in extracting.
11. Once that is found, comes the rest of the XPath expression, which says that for that Definition element found, get a child element named WordDefinition and extract out its text() value. This is returned as a collection of Text nodes.
12. Finally, we iterate through the Text Nodes and get the value which we then send back as a response.
13. I suggest that if you are still having a problem following the code, try out the URL as mentioned in step 6, study the XML and then the XPath expression will become clear. The rest of the code is standard XML/XPath code from the Java SDK.

Servlet Configuration

To complete our Servlet development, we will also need to add the **<servlet/>** and **<servlet-mapping/>** entry to the **web.xml** file. This file is present in the WEB-INF folder of the project. The necessary fragment to be added to your **web.xml** file are shown below. Please note that you can use your own namespace and servlet class. Just modify it accordingly if you do so.

```
<servlet>
 <servlet-name>GAEJDictionaryService</servlet-name>
 <servlet-
class>com.gaejexperiments.networking.GAEJDictionaryService</servlet-
class>
 </servlet>
<servlet-mapping>
 <servlet-name>GAEJDictionaryService</servlet-name>
 <url-pattern>/gaejdictionaryservice</url-pattern>
 </servlet-mapping>
```

Deploying and running your application

To deploy the application, you will need to first create your **Application ID**. The Application Identifier can be created by logging in at **http://appengine.google.com** with your Google Account. You will see a list of application identifiers already registered under your account (or none if you are just getting started). To create a new Application, click on the Create Application button and provide the Application Identifier as requested. Please note this name down since you will be using it for deployment.

For e.g. I have registered an application identifier named **gaejexperiments**.

To deploy the application, follow these steps (they should be familiar to you now):

1. Click on the Deploy Icon in the Toolbar.
2. In the Deploy dialog, provide your Email and Password. Do not click on Deploy button yet.
3. Click on the **App Engine Project settings** link. This will lead you to a dialog, where you need to enter your Application ID [For e.g. my Application Identifier **gaejexperiments**]
4. Click on **OK**. You will be lead back to the previous screen, where you can click on the **Deploy** button. This will start deploying your application to the GAEJ cloud. You should see several messages in the Console window as the application is being deployed.
5. Finally, you should see the message “Deployment completed successfully”.

This means that you application is ready to serve. Depending on whether you used index.html or dictionary.html, you should be able to access your application at the following url:

```
http://<YourAppId>.appspot.com/dictionary.html
```

or

```
http://<YourAppId>.appspot.com/index.html
```

Type in a word and check if the meaning is being returned.

Moving forward

In this episode, we have seen how to utilize the URL Fetch Service provided by the Google App Engine. This service provides us with a convenient way to access external services that can be integrated into our application. Creating Mashup applications would be straightforward with the URL Fetch Service.

You can integrate other external services, RSS feeds in this fashion. Just give it a try.

Episode 5: Upgrading to Google App Engine 1.2.6

If you have been following this blog, the version of the Google App Engine SDK was **1.2.5**. Recently Google announced a release **1.2.6** of its App Engine SDK.

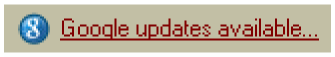
I think you should upgrade to the latest version since it a very cool feature that resonates with an earlier episode titled **Episode 3: Using the GAEJ Email Service**. That episode covered using the Email Service that allows you to send an email from your GAEJ application. The latest release 1.2.6 of the App Engine SDK for Java allows you to receive incoming emails too in your application. And that will be the topic in the next episode, so I thought it is best that I mention here that it is time to upgrade to version 1.2.6 of the App Engine SDK.

Here are a few steps and points to note to upgrade to version 1.2.6 of the SDK. I am assuming that all your development/deployment is being done via the Google App Engine plugin for Eclipse since that is what we have used throughout this series. Here are the steps:

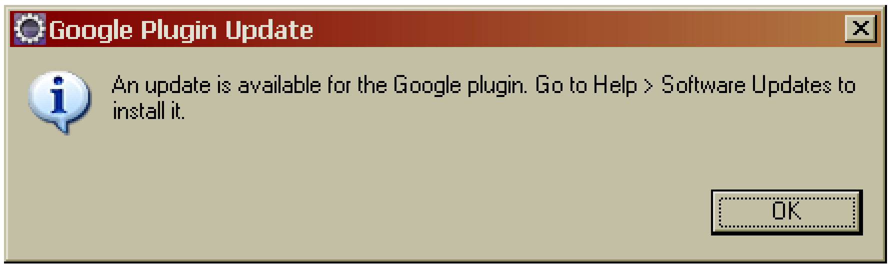
1. Launch the version of Eclipse that you have been using so far for the GAEJ development.

2. Go to **Window –> Preferences**. And then click on **Google –> App Engine**. You can see the version of the App Engine SDK and most likely it is 1.2.5

3. Stay in **Preferences** and go one level higher in the tree on the left to **Google**. You will see a tiny checkbox that mentions **"Notify me about updates"**. If you have this selected, then you should have already got a notification status in the status bar in Eclipse, somewhere in the bottom of the screen. It looks something like this:



Clicking on this link would bring up a dialog that looks like the one shown below:

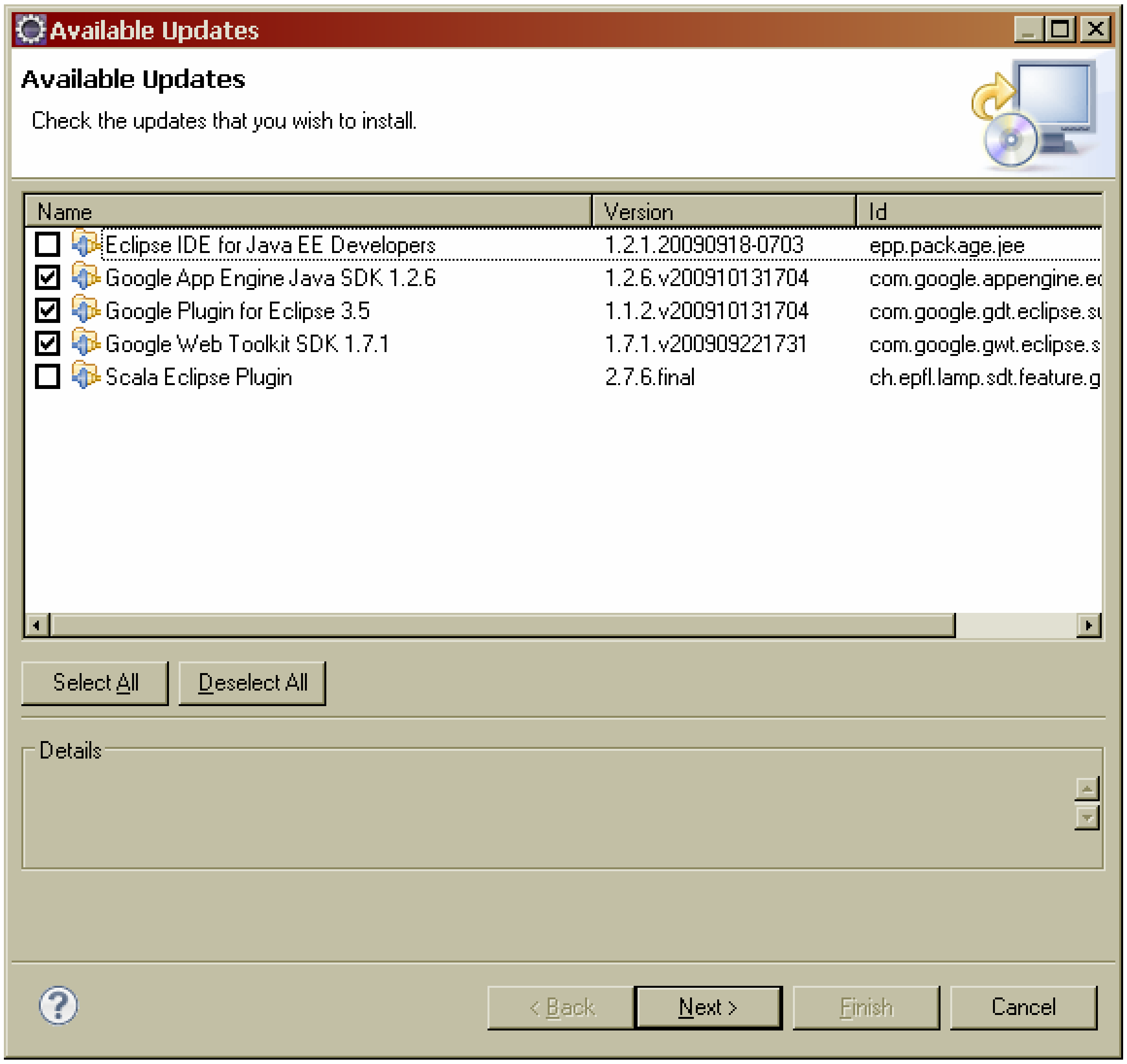

Google Plugin Update

An update is available for the Google plugin. Go to Help > Software Updates to install it.

Depending on your version of Eclipse, you might see a different message and maybe the menu options might be slightly different but the process is a standard one in Eclipse to update your plugins.

4. Even if you had deselected the option to Notify about updates, the process to upgrade the plugin/SDK to the latest version 1.2.6 is the same. Go to **Help –> Check for Updates** in the main menu. This is for version 3.5 of Eclipse that I use (Galileo version).

This will bring up a dialog and show all your installed components/plugins in Eclipse that have a newer version (update) for you to install. A sample screenshot from my Eclipse setup is shown below:

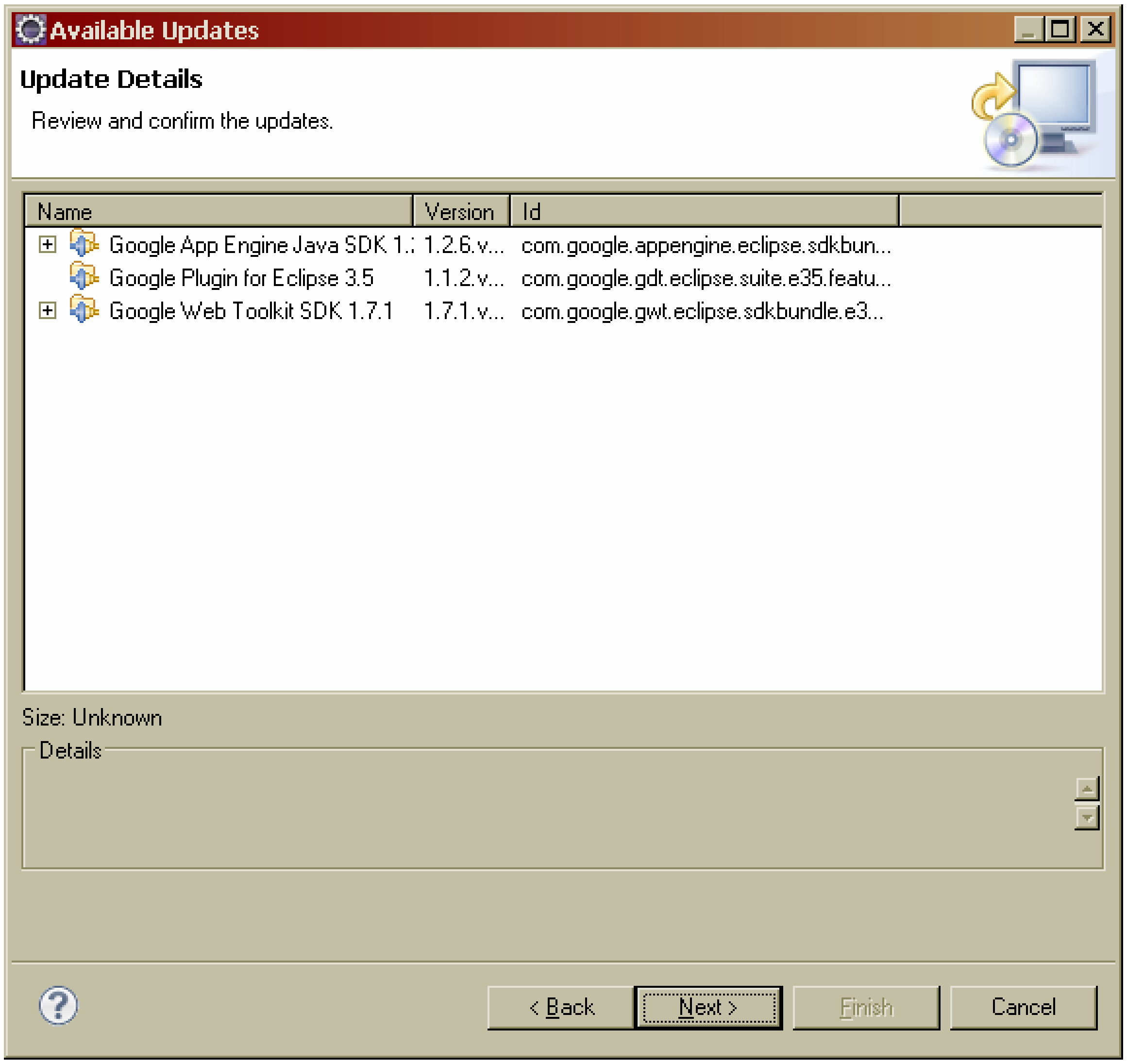


Available Updates

Check the updates that you wish to install.

	Name	Version	Id
☐	Eclipse IDE for Java EE Developers	1.2.1.20090918-0703	epp.package.jee
☑	Google App Engine Java SDK 1.2.6	1.2.6.v200910131704	com.google.appengine.e
☑	Google Plugin for Eclipse 3.5	1.1.2.v200910131704	com.google.gdt.eclipse.s
☑	Google Web Toolkit SDK 1.7.1	1.7.1.v200909221731	com.google.gwt.eclipse.s
☐	Scala Eclipse Plugin	2.7.6.final	ch.epfl.lamp.sdt.feature.g

5. Select only the Google specific upgrades. You should be able to see the version 1.2.6 of the SDK shown as above. Click on the **Next** button. This will confirm the updates as shown below:



Available Updates

Update Details

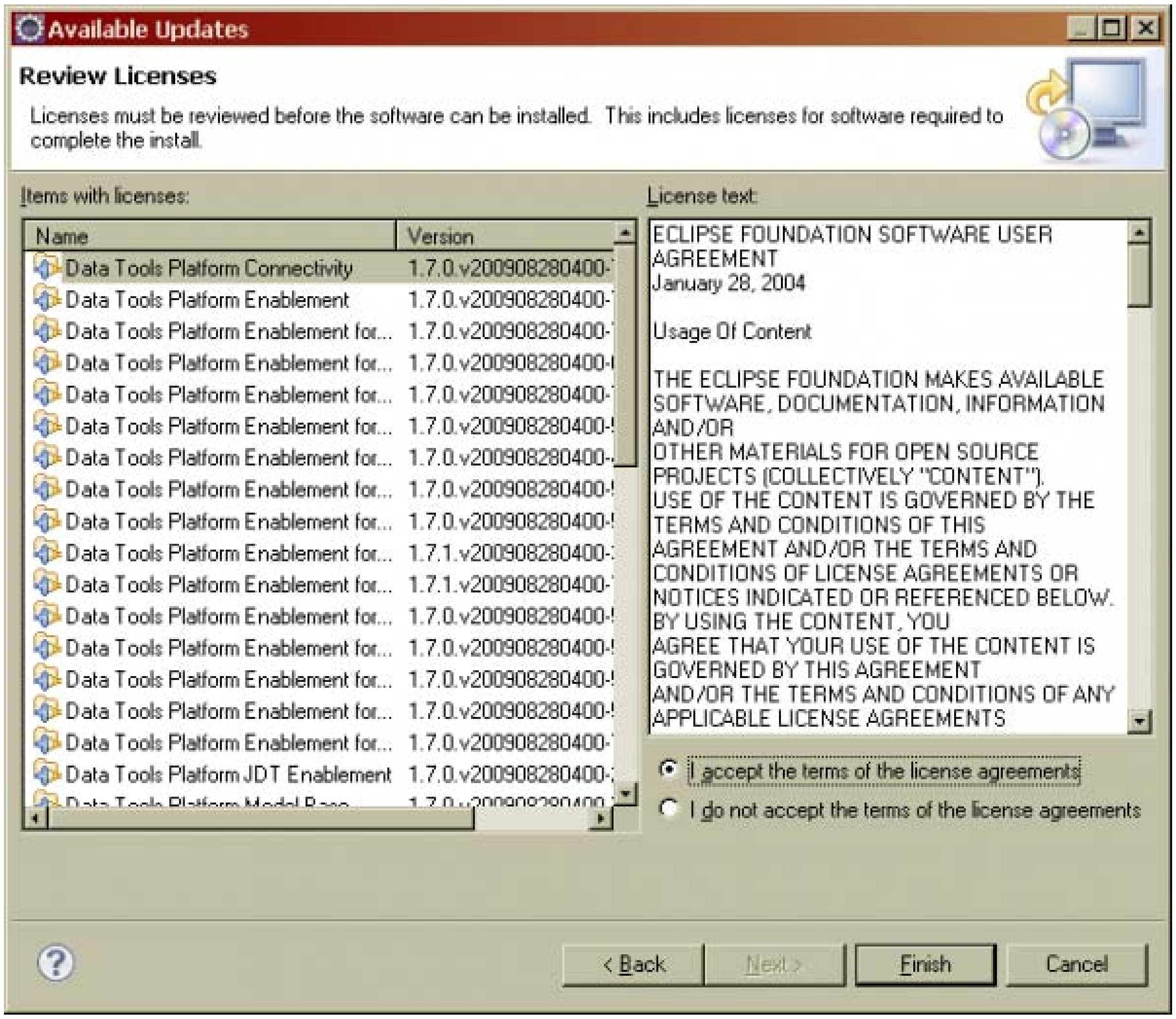
Review and confirm the updates.

Name	Version	Id
Google App Engine Java SDK 1.:	1.2.6.v...	com.google.appengine.eclipse.sdkbun...
Google Plugin for Eclipse 3.5	1.1.2.v...	com.google.gdt.eclipse.suite.e35.featu...
Google Web Toolkit SDK 1.7.1	1.7.1.v...	com.google.gwt.eclipse.sdkbundle.e3...

Size: Unknown

Details

Click on Next, accept all the terms as shown below and proceed with Finish appropriately.



Licenses must be reviewed before the software can be installed. This includes licenses for software required to complete the install.

Items with licenses:

Name	Version
Data Tools Platform Connectivity	1.7.0.v200908280400-
Data Tools Platform Enablement	1.7.0.v200908280400-
Data Tools Platform Enablement for...	1.7.0.v200908280400-
Data Tools Platform Enablement for...	1.7.0.v200908280400-
Data Tools Platform Enablement for...	1.7.0.v200908280400-
Data Tools Platform Enablement for...	1.7.0.v200908280400-
Data Tools Platform Enablement for...	1.7.0.v200908280400-
Data Tools Platform Enablement for...	1.7.0.v200908280400-
Data Tools Platform Enablement for...	1.7.0.v200908280400-
Data Tools Platform Enablement for...	1.7.1.v200908280400-
Data Tools Platform Enablement for...	1.7.1.v200908280400-
Data Tools Platform Enablement for...	1.7.0.v200908280400-
Data Tools Platform Enablement for...	1.7.0.v200908280400-
Data Tools Platform Enablement for...	1.7.0.v200908280400-
Data Tools Platform Enablement for...	1.7.0.v200908280400-
Data Tools Platform Enablement for...	1.7.0.v200908280400-
Data Tools Platform JDT Enablement	1.7.0.v200908280400-

License text:

ECLIPSE FOUNDATION SOFTWARE USER AGREEMENT
January 28, 2004

Usage Of Content

THE ECLIPSE FOUNDATION MAKES AVAILABLE SOFTWARE, DOCUMENTATION, INFORMATION AND/OR
OTHER MATERIALS FOR OPEN SOURCE PROJECTS (COLLECTIVELY "CONTENT").
USE OF THE CONTENT IS GOVERNED BY THE TERMS AND CONDITIONS OF THIS AGREEMENT AND/OR THE TERMS AND CONDITIONS OF LICENSE AGREEMENTS OR NOTICES INDICATED OR REFERENCED BELOW.
BY USING THE CONTENT, YOU
AGREE THAT YOUR USE OF THE CONTENT IS GOVERNED BY THIS AGREEMENT
AND/OR THE TERMS AND CONDITIONS OF ANY APPLICABLE LICENSE AGREEMENTS

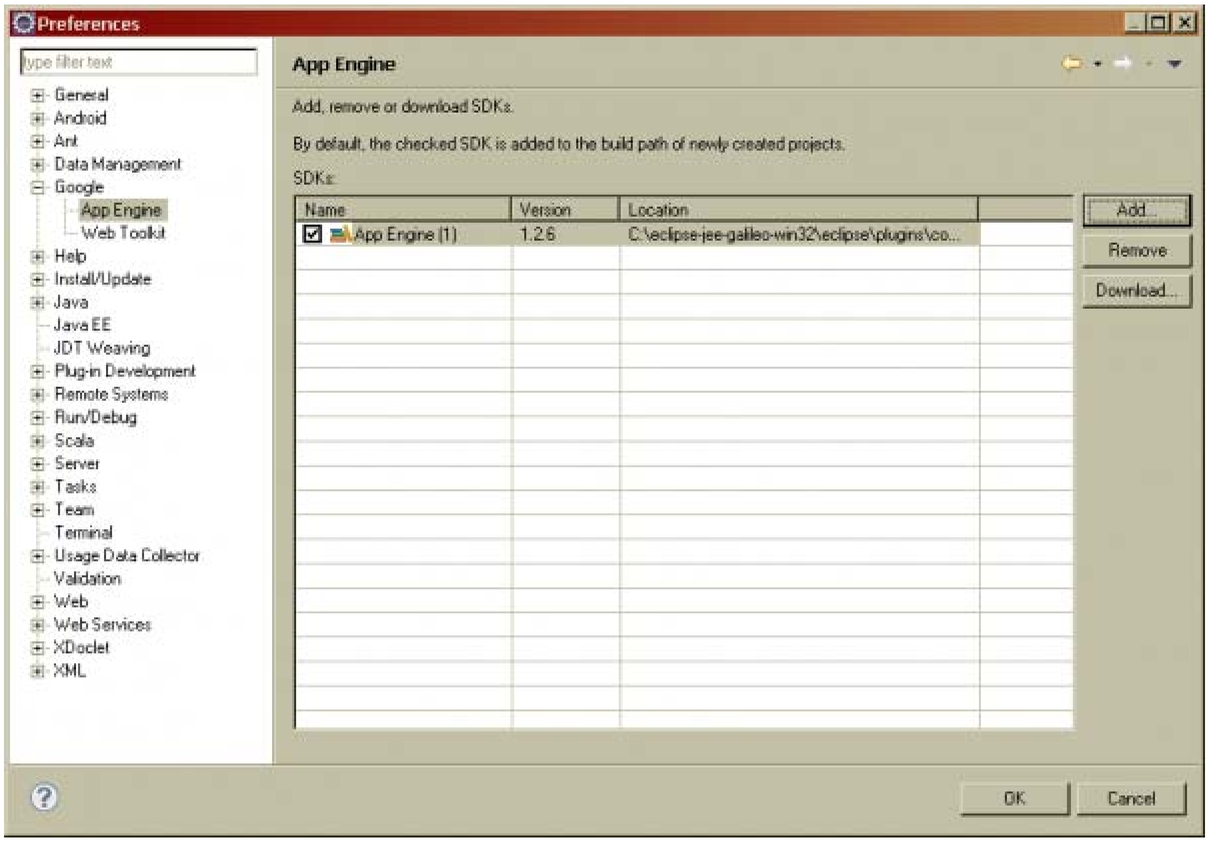
[✓ I accept the terms of the license agreements]
I do not accept the terms of the license agreements

This will download, install the update to the SDK. I recommend to restart Eclipse as prompted.

Post Installation Steps

The following two steps are very important in order to start using the latest version correctly. This is what worked for me and I am not sure if there are other things that are still required but this two steps should suffice for now:

1. Make sure that all your projects moving forward use version 1.2.6 of the SDK. This is not mandatory but required if you wish to use the new features available in version 1.2.6. To do this, go to **Window –> Preferences.** Click on **Google –> App Engine** as shown below:



App Engine

Add, remove or download SDKs.

By default, the checked SDK is added to the build path of newly created projects.

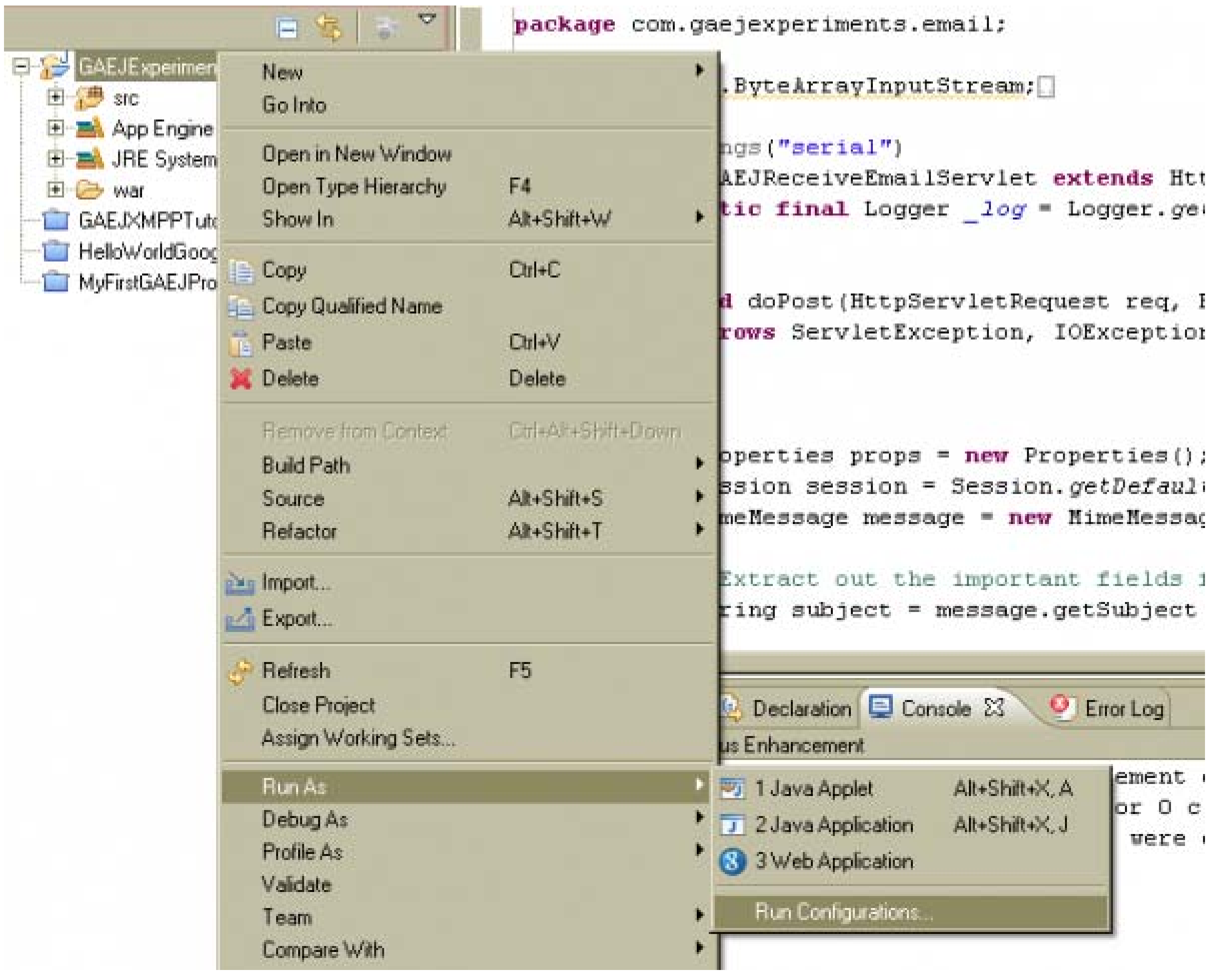
SDKs:

Name	Version	Location
☑ App Engine (1)	1.2.6	C:\eclipse-jee-galileo-win32\eclipse\plugins\co...

In this screen, you are seeing only only entry for the version 1.2.6 of the SDK. Normally you will see more than one (the older 1.2.5 SDK) will also be shown and it will be the default selected one. You are not seeing it over here in my setup, because I deleted it and am using only 1.2.6 of the SDK as shown. Select and you will find that as mentioned in the dialog above, the checked SDK will be used by default for newly created projects.

2. Google App Engine SDK also ships with a development server that you can use for local testing. We will be using it in the next episode. You can simply right click any project in the project hierarchy and select **Run As –> Web Application** to launch your GAEJ application in the development server. The development server is automatically started as you will notice in the console window and you can access your application at http://localhost:8080. **But an additional step** now needs to be performed to help the development server start. This has got added since version **1.2.6** – it looks like that. Follow the next steps to get this correctly setup.

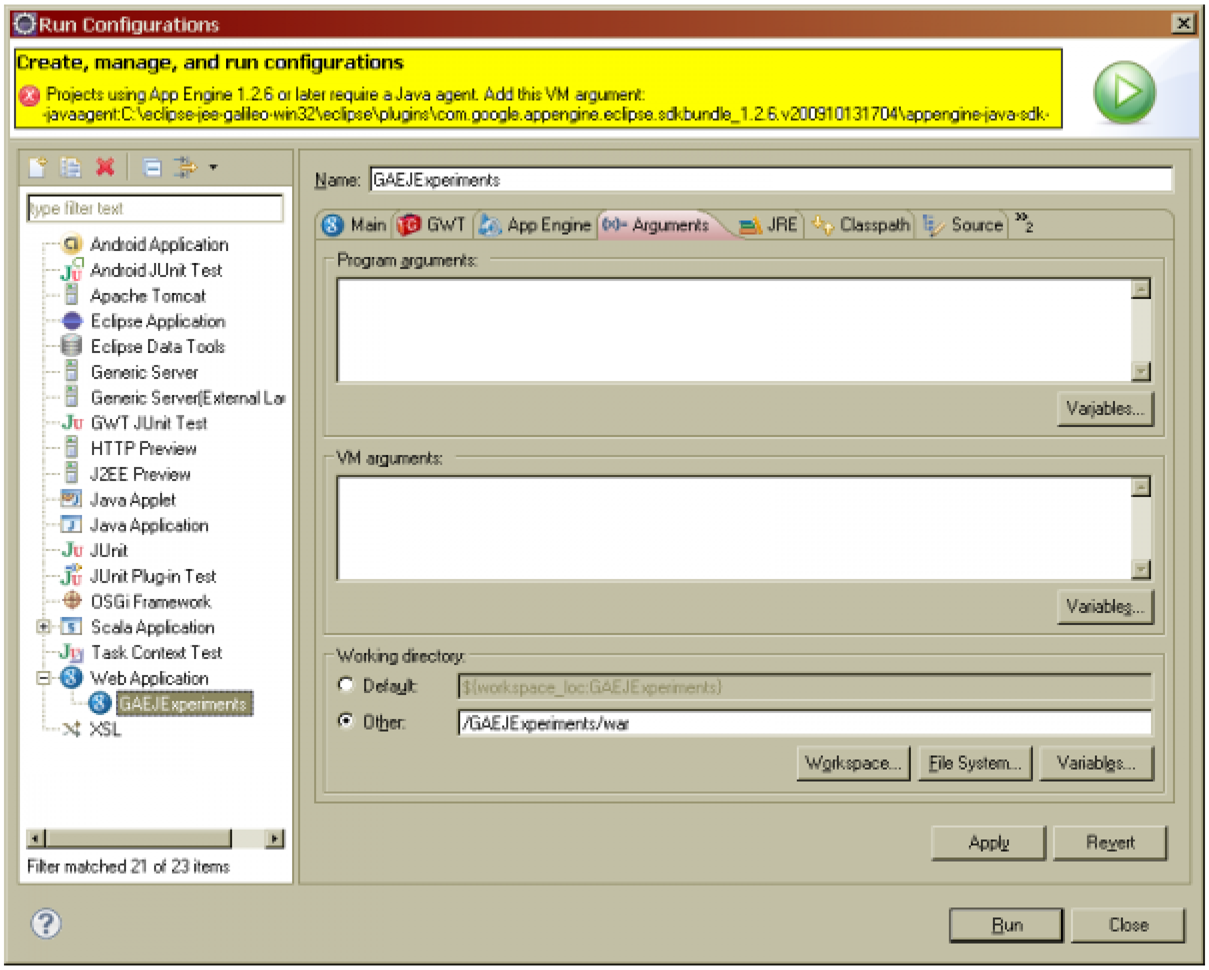
3. Select the project you want to run in the Project Hierarchy on left. Right click it and select **Run As –> Run Configurations…** as shown below:



```
package com.gaejexperiments.email;

.ByteArrayInputStream;

ngs("serial")
AEJReceiveEmailServlet extends Htt
tic final Logger _log = Logger.ge

d doPost(HttpServletRequest req, H
rows ServletException, IOExceptio

operties props = new Properties();
ssion session = Session.getDefaul
meMessage message = new MimeMessag

Extract out the important fields i
ring subject = message.getSubject
```

```
ement
or 0 c
were
```

4. This will bring up the Run Configuration as shown below. Notice the Error message displayed which I have highlighted in Yellow.


Run Configurations

Create, manage, and run configurations

Projects using App Engine 1.2.6 or later require a Java agent. Add this VM argument:
-javaagent:C:\eclipse-jee-galileo-win32\eclipse\plugins\com.google.appengine.eclipse.sdkbundle_1.2.6.v200910131704\appengine-java-sdk-

Name: GAEJExperiments

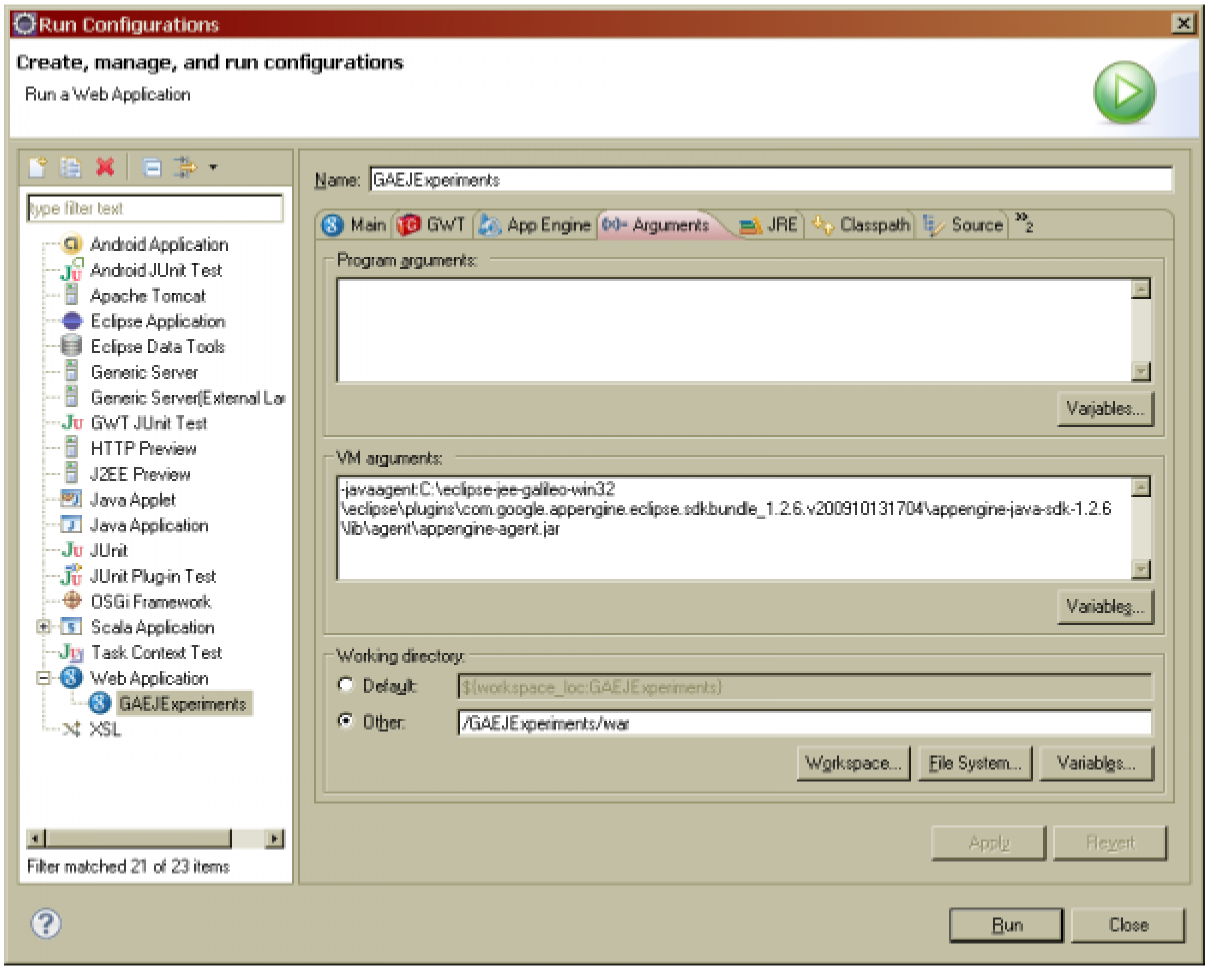
Program arguments:

VM arguments:

Working directory:
Default: ${workspace_loc:GAEJExperiments}
Other: /GAEJExperiments/war

Filter matched 21 of 23 items

5. So all you need to do is add the VM Argument that they are mentioning. Simply copy the string starting from -javaagent to the end. And paste it in the **VM Arguments** field in the same dialog as shown below:



Name: GAEJExperiments

Program arguments:

VM arguments:

```
-javaagent:C:\eclipse-jee-galileo-win32
\eclipse\plugins\com.google.appengine.eclipse.sdkbundle_1.2.6.v200910131704\appengine-java-sdk-1.2.6
\lib\agent\appengine-agent.jar
```

Working directory:

Default: ${workspace_loc:GAEJExperiments}

[✓ Other:] /GAEJExperiments/war

You will find that the error message will go away. You can click **Run** now to execute the Web Application. The server will be able to start successfully now.

This completes the upgrade of your Google plugin to version 1.2.6 of the SDK.

Upcoming Episode

Make sure that you have version 1.2.6 of the SDK setup as mentioned in this episode. The next episode coming in a day will show how your Google App Engine applications can receive incoming emails, which is one of the newly enabled features of version 1.2.6 of the SDK. Till then, happy upgrading!

Episode 6: Handling incoming Email in your application

Welcome to **Episode 6** of this series. In this episode, we will learn how your Google App Engine Java (GAEJ) Application can receive **incoming email**.

In an earlier **episode** of this series, we have covered how to use the Email Service of GAEJ to send out emails from your application. At that point in time, we were using version 1.2.5 of the AppEngine SDK. That version did not provide support for handling incoming email. Since then a newer version of the AppEngine SDK for Java 1.2.6 has been released. And one of the nice features of this new release is support for incoming email in your application. What it means is that anyone can send an email to your GAEJ hosted application, it can receive an email and then you can process and perform some business logic with that as required.

Prerequisites

Before we move on, it is important that you have a working setup of the environment and are comfortable with developing a new project using the Google Eclipse plugin. If not, please go through earlier episodes that contained detailed setup information along with a few development episodes like using Email Service, using the URL Fetch service, etc.

The most important prerequisite is to make sure that you have upgraded your Eclipse environment to the latest AppEngine SDK i.e. 1.2.6. Please go through the following :
Episode 5: Upgrading to Google App Engine 1.2.6

A quick check: To make sure that you have version 1.2.6 of the SDK installed, do the following:

1. In your Eclipse IDE, go to **Window –> Preferences.**
2. Navigate to **Google –> App Engine**
3. You should see version 1.2.6 of the SDK installed. And make sure that it is the default one by selecting it. By selecting it, it will be added to the Build Path of your Google Web Application projects. And we need the latest SDK since it will have support for the Incoming Email feature.

Receiving Email Feature

App Engine now supports incoming email in your applications. Read the official documentation here. Your applications can now receive email and you can parse out the email and determine any business logic that needs to be processed. This opens up a whole new range of applications where you can fulfill requests and send information from your application by allowing users to simply send an email from their favourite email client. Think of it as a Instant Message itself that your application can receive and react to it. We had seen in an earlier **Episode** how through XMPP Support, we can write our own Agent that can receive XMPP messages directly and respond to them. Now with version 1.2.6 of the SDK, the same functionality has got extended to email too. And the best part about it is the consistency with which Google has implemented it.

The steps to follow to receive an email is identical to the ones required to receive XMPP messages:

1. Configure your application to receive incoming email by configuring the Mail Service

2. Write and configure a servlet to receive email

3. Once the application is deployed, anyone can send an email to **SomeID@YourApplicationId**.appspotmail.com. **SomeID** is any id like test, admin, support,etc. It is just an id. And **YourApplicationId** is the application id of your hosted Google App Engine application.

Let us look at each of the above points in detail now. But before we begin, create a **New Google Web Application** Project (**If you wish you can continue to use an existing project to add the incoming Email support, which is what I have done personally, but the choice is yours)** . Follow these steps to create a new project:

1. Either click on **File –> New –> Other** or press **Ctrl-N** to create a new project. Select Google and then Web Application project. Alternately you could also click on the New Web Application Project Toolbar icon as part of the Google Eclipse plugin.
2. In the New Web Application Project dialog, **deselect** the **Use Google Web Toolkit** and give a name to your project. I have named mine **GAEJExperiments**. I suggest you go with the same name so that things are consistent with the rest of the article, but I leave that to you.
3. Click on Finish

This will generate the project and also create a sample Hello World Servlet for you. But we will be writing our own Servlet.

Configuring the incoming Email Service

This is straightforward and all you need to do is add the following element to the **appengine-web.xml** file. The **appengine-web.xml** file as you know is specific to the Google Java Web Application project and is used for configuring certain services. You need to configure the Incoming Email Service so that your application is enabled to receive it, being one of them. It is found in the **war\WEB-INF** folder of your Web Application Project. The XML fragment to add at the end but before the </**appengine-web-app**> element

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

```
</inbound-services>
```

Configure and code a Java Servlet that will receive the incoming Message

All Email messages to your application are delivered via **POST** to following URL path in your application: **/_ah/mail/** as per the Google AppEngine documentation**.** So you will need to configure the servlet like the following snippet in the **web.xml** file, present in the **war\WEB-INF** folder of your Web Application Project.

We need to add the **<servlet/>** and **<servlet-mapping/>** entry to the **web.xml** file. This file is present in the **WEB-INF** folder of the project. The necessary fragment to be added to your **web.xml** file are shown below. Please note that you can use your own namespace and servlet class. Just modify it accordingly if you do so.

```
<servlet>
 <servlet-name>emailhandler</servlet-name>
 <servlet-
class>com.gaejexperiments.email.GAEJReceiveEmailServlet</servlet-class>
 </servlet>
 <servlet-mapping>
 <servlet-name>emailhandler</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>
```

In the above snippet, you will find the fixed URL path **/_ah/mail/*** configured as the **<url-pattern/>**. And then I have a Java Servlet class **com.gaejexperiments.email.GAEJReceiveEmailServlet** as the **<servlet-class>**. The security constraint has been added so that in case anyone invokes your url directly, then only Google Account authenticated users will be able to do that.

Now, all we have to do is write our Servlet. As mentioned, the incoming Email messages will be **POST**ed to our Servlet, so we need a simple doPost(…) implemented in our Servlet. The code is shown below:

```
package com.gaejexperiments.email;

import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.mail.MessagingException;
import javax.mail.Multipart;
import javax.mail.Part;
import javax.mail.Session;
import javax.mail.internet.MimeMessage;
import javax.servlet.ServletException;
import javax.servlet.http.*;

@SuppressWarnings("serial")
public class GAEJReceiveEmailServlet extends HttpServlet {
 public static final Logger _log =
Logger.getLogger(GAEJReceiveEmailServlet.class.getName());

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

 try {

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

 //Extract out the important fields from the Mime Message
 String subject = message.getSubject();

 _log.info("Got an email. Subject = " + subject);

 String contentType = message.getContentType();
 _log.info("Email Content Type : " + contentType);

 printParts(message);
 //Parse out the Multiparts
 //Perform business logic based on the email
 }
 catch (Exception ex) {
 _log.log(Level.WARNING, "Failure in receiving email : " +
ex.getMessage());
 }
 }

 private static void printParts(Part p) throws IOException,
MessagingException {
 Object o = p.getContent();

 if (o instanceof String) {
 System.out.println("This is a String");
 System.out.println((String)o);
 }
 else if (o instanceof Multipart) {
```

```
 System.out.println("This is a Multipart");
 Multipart mp = (Multipart)o;

 int count = mp.getCount();
 for (int i = 0; i < count; i++) {
 printParts(mp.getBodyPart(i));
 }
 }
 else if (o instanceof InputStream) {
 System.out.println("This is just an input stream");
 InputStream is = (InputStream)o;
 int c;
 while ((c = is.read()) != -1)
 System.out.write(c);
 }
 }

}
```

Let us discuss the main parts of the code:

1. We have a **doPost()** method that gets invoked by the Google App Engine when an email is received.

2. In the **doPost()** method, we build out the email message (a instance of class MimeMessage) using the javax.mail.* classes as shown below:

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

3. We extract out key attributes from the Message like subject, content type, etc.

```
//Extract out the important fields from the Mime Message
 String subject = message.getSubject();
 _log.info("Got an email. Subject = " + subject);

 String contentType = message.getContentType();
 _log.info("Email Content Type : " + contentType);
```

4. We have a utility method **printParts()**, that helps simply print out the contents of the message. But you could explore the **Java Mail API** to parse out the multiparts as required and then incorporate your business logic.
5. To help debug our Servlet, we have put in some log statements along with System Out statements, which we shall look for to verify that the Application did receive email.

Finally, we have used the INFO level to log if the message was sent out successfully or not, so we will have the change the logging level by modified the **logging.properties** file present in the **war\WEB-INF** folder. The necessary line after modification is shown below:

```
# Set the default logging level for all loggers to INFO
.level = INFO
```

Deploying our application

To deploy the application, you will need to first create your **Application ID**. The Application Identifier can be created by logging in at **http://appengine.google.com** with your Google Account. You will see a list of application identifiers already registered under your account (or none if you are just getting started). To create a new Application, click on the Create Application button and provide the Application Identifier as requested. Please note this name down since you will be using it for deployment.

For e.g. I have registered an application identifier named **gaejexperiments**.

To deploy the application, follow these steps (they should be familiar to you now):

1. Click on the Deploy Icon in the Toolbar.
2. In the Deploy dialog, provide your Email and Password. Do not click on Deploy button yet.
3. Click on the **App Engine Project settings** link. This will lead you to a dialog, where you need to enter your Application ID [For e.g. my Application Identifier **gaejexperiments**]
4. Click on **OK**. You will be lead back to the previous screen, where you can click on the **Deploy** button. This will start deploying your application to the GAEJ cloud. You should see several messages in the Console window as the application is being deployed.
5. Finally, you should see the message “Deployment completed successfully”.

Testing our Application

To test our application, send an email from any mail client to an address within your application. As explained below, the email address to be used is shown below:

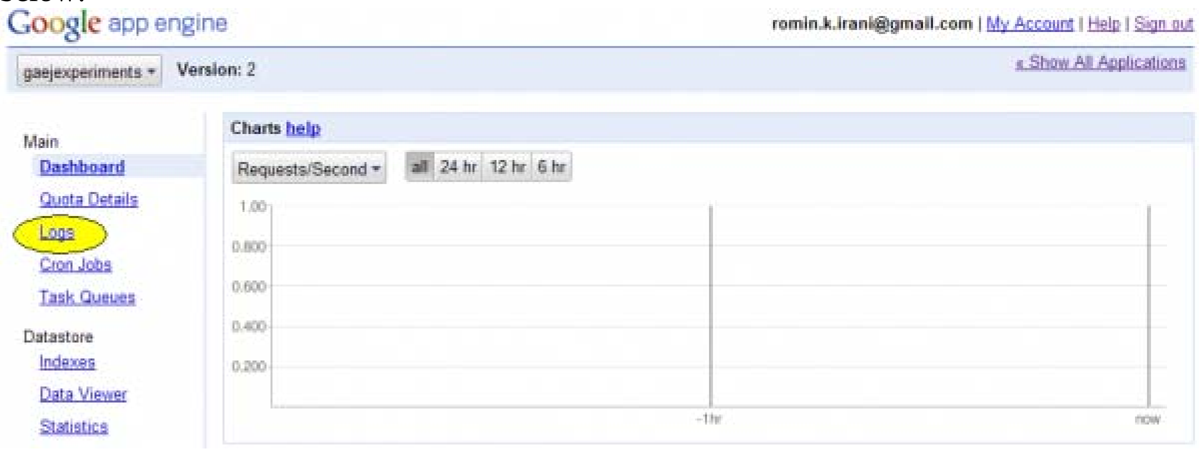
SomeID@YourApplicationId.appspotmail.com

For e.g. my Application Id is **gaejexperiments,** so I can send email to any of the following:

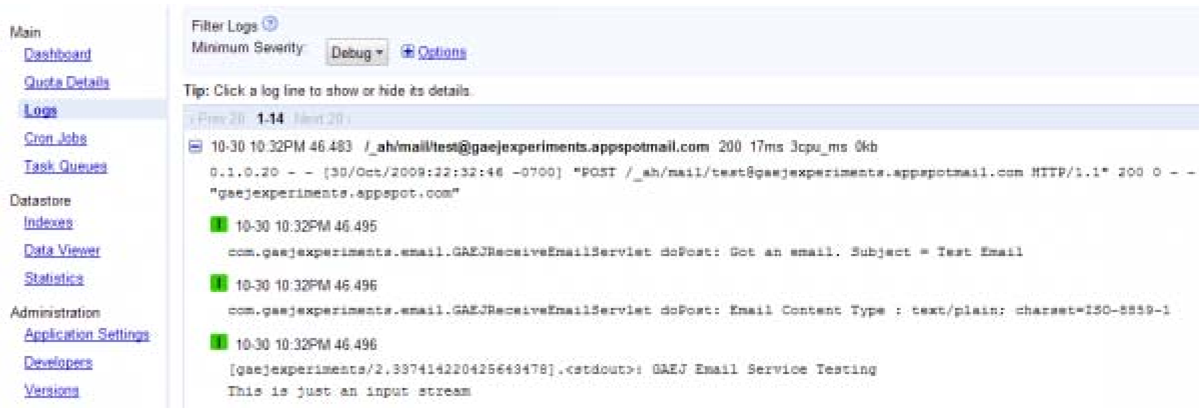
- test@gaejexperiments.appspotmail.com
- user1@gaejexperiments.appspotmail.com
- and so on…

Once the email has been sent successfully from a Google Mail Account or Yahoo Mail or Outlook/Thunderbird, etc – you can use the App Engine console to verify if your application received the email or not. To do that, perform the following steps:

1. Go to http://appengine.google.com and log in with your account.
2. You will see a list of applications registered. Click on the application that you just deployed. In my case, it is **gaejexperiments.**
3. When you click on a particular application, you will be taken to the **Dashboard** for that application, which contains a wealth of information around the requests, quotas, logs, versions, etc. This will be a subject of a future episode but for now, it is sufficient to say that you can come over here to monitor the health of your application and also to analyze what is going on.
4. Click on the **Logs** link as shown in the screenshot below:



5. This will display the application log. And all your application log statements that you code using the Logger class can be visible here.
6. By default, the severity level is set at ERROR and we can change that to DEBUG, the lowest level and you should be able your log statements that had the log level of INFO. This was the log level at which we had logged statements like Received an Email, etc in our Java Servlet, so that is what we need to check for.
7. If you see the statements, it means that the message has been received. Shown below is a screen shot of the log for a message that I sent.



Minimum Severity: Debug

```
10-30 10:32PM 46.483 /_ah/mail/test@gaejexperiments.appspotmail.com 200 17ms 3cpu_ms 0kb
  0.1.0.20 - - [30/Oct/2009:22:32:46 -0700] "POST /_ah/mail/test@gaejexperiments.appspotmail.com HTTP/1.1" 200 0 - -
  "gaejexperiments.appspot.com"
  10-30 10:32PM 46.495
    com.gaejexperiments.email.GAEJReceiveEmailServlet doPost: Got an email. Subject = Test Email
  10-30 10:32PM 46.496
    com.gaejexperiments.email.GAEJReceiveEmailServlet doPost: Email Content Type : text/plain; charset=ISO-8859-1
  10-30 10:32PM 46.496
    [gaejexperiments/2.337414220425643478].<stdout>: GAEJ Email Service Testing
    This is just an input stream
```

Once this mechanism is in place, it means that your application has been correctly setup and deployed for receiving email messages. You can now build in business logic on what you need to do when an email is received.

Hope you had a good time reading this episode. The next episode I plan to extend an existing episode in which we built an XMPP Bot and deployed it on the Google App Engine. I plan to extend the bot to support the collaboration tool Google Wave, so that your bot can participate in a Wave conversation too. Till then, happy GAEJ’ing!

Episode 7: Writing your First Google Wave Robot

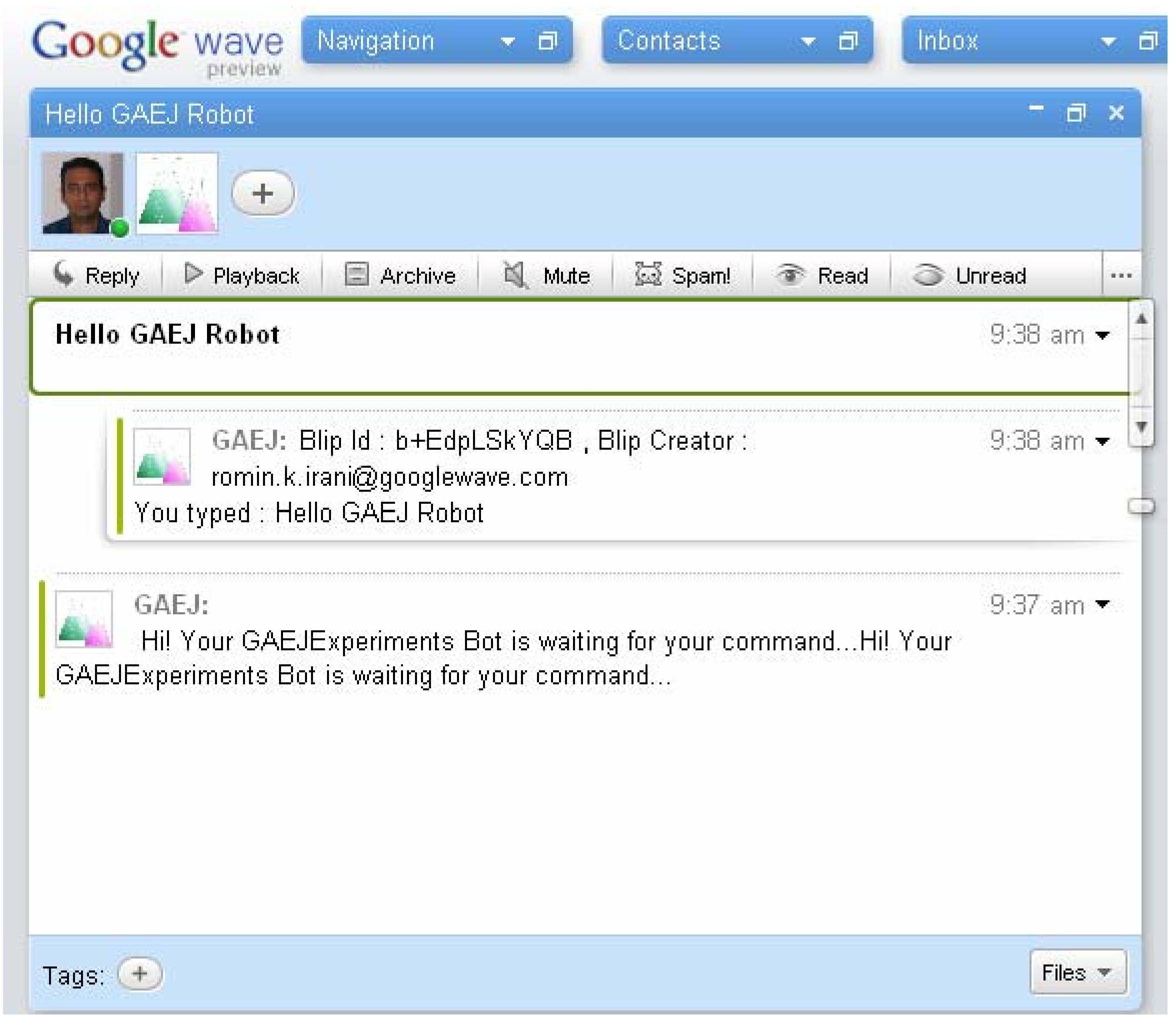
Welcome to **Episode 7** of this series. Hope that the series has been interesting for you so far. In this episode, we plan to jump onto the Google Wave bandwagon. Google Wave, as you must have heard is a collaborative messaging platform from Google. A Wave is a collaboration between several participants (humans and non-humans) using threaded conversations/documents/gadgets to achieve their task. The task could be as simple as chatting among themselves to figure out where to dine tonight to as complex as achieving a Return Order process between a purchaser and seller.

I will not delve into the specifics of Google Wave and will assume that you have a high level idea of what it is. In fact, a great online book has just been released that covers Google Wave in depth. I strongly recommend you to read it if you want to look into Google Wave in detail. The book is named **The Complete Guide to Google Wave** and you can read it here.

What does this episode cover?

In this episode, we are going to look at how you can write a **Google Wave Robot** using the Google plugin in Eclipse. We will then deploy it and host this robot in the Google App Engine cloud. This Robot is then live and can be added to any Wave and participate in the conversation.

It helps to see the final result before we start out. Even if you do not understand whats going on, just bear with me. So here is the screenshot:



Hello GAEJ Robot 9:38 am

GAEJ: Blip Id : b+EdpLSkYQB , Blip Creator : romin.k.irani@googlewave.com 9:38 am
You typed : Hello GAEJ Robot

GAEJ: 9:37 am
Hi! Your GAEJExperiments Bot is waiting for your command...Hi! Your GAEJExperiments Bot is waiting for your command...

Let us dissect this screen and in the process, I can hopefully explain to you some Google Wave terminology in a straightforward manner without too many details. The main points are:

1. What you see in the screen above is a Google Wave that me (the person with the photo created). A Wave is a collaboration between one or more participants (human or **Robots**).

2. Since we need participants to make it meaningful, you can add one or more of them. All you need to do is click the **+ sign** that you see on the top. And you can add one or more participants who are also signed up for Wave. In this case, my Robot GAEJ Application is hosted at http://gaejrobot.appspot.com and so I can add gaejrobot@appspot.com as a participant.

3. On adding the Robot participant, Google wave system pulls up its profile (image, etc) and adds it as a participant in the Wave that I created.

4. The Robot can receive events and it can respond to them. We shall see how to write them later (source code!!), but at this point it is enough to understand that the Robot can choose what events to listen to as the Wave collaboration happens and it can then react to them. So in the case of my GAEJ Robot, I have specified that I am interested in knowing when participants come and leave the Wave and when someone has submitted a message.

5. So you can see that the GAEJ Robot was notified when it was added to the Wave, so in response to that it printed out the message "Hi! Your GAEJExperiments Bot….." at the bottom. Similarly, when I finished typing the message (**Hello GAEJ Robot)**, the GAEJ Robot got notified. It then took the message and along with some metadata like ID and Creator Name, it simpy **echoed** back what I typed.

Hope this makes things clear. But a few more terms first : Wave, Wavelet and Blip. Let me keep it simple. In the above example, the **Wave** was the entire container. A **Wave** consists of one or more **Wavelets.** A Wavelet can be thought of as a threaded conversation that can go on within the wave (both public and private). And each Wavelet consists of one or more messages known as a **Blip.** A **Blip** is the actual message that was typed and submitted by the user. So when I typed "Hello GAEJ Robot" above, it was a **Blip.**

I have simplified things a bit here but even if things are not clear at this point, do not worry. Once you get your code working, things will fall in place.

A final note above Google Wave Extensions. Extensions are the mechanism by which you can extend Google Wave by adding your own creations. Extensions are of two types : Gadgets and Robots. Gadgets are like mini-applications that run within the Google Wave client. They can be typically thought of as a UI which several participants can share at the same time. A Robot is a full blown participant in the Wave. It can be aware of most things happening in the Wave by subscribing to Events. It also has a lot of potential to modify things in the Wave like messages, etc. This article focuses on writing a Robot. If you are interested in a good summary of Google Wave Extensions and some key differences between a Gadget and a Robot, read it here.

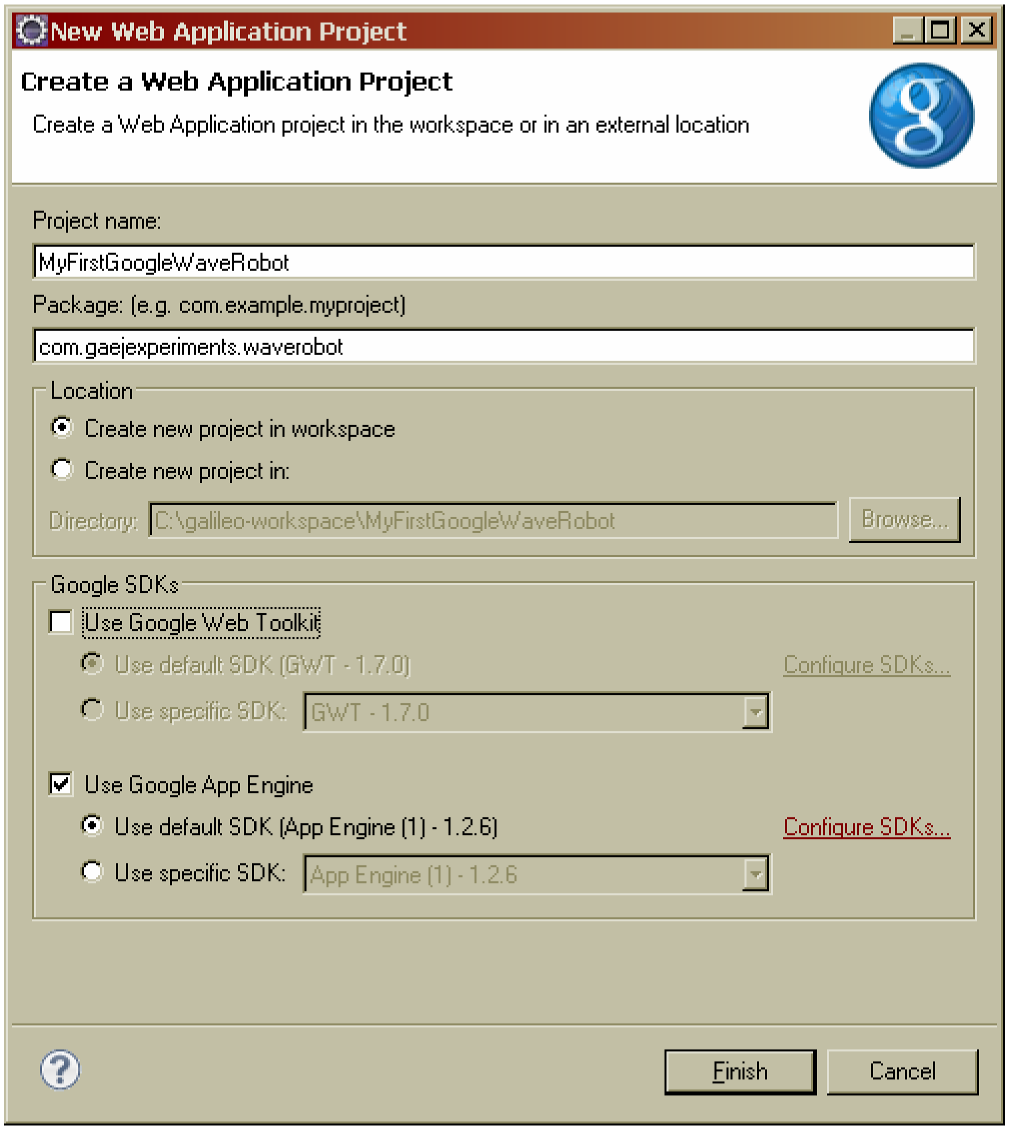
OK, so enough of theory. Let us get down to coding. *If you are new to developing with the Google Plugin for Eclipse, please read the earlier episodes to setup your environment (Episode 1 and Episode 5)*

Create a New Project

We need to create a New Project first. Follow the steps below:

1. Either click on **File –> New –> Other** or press **Ctrl-N** to create a new project. Select Google and then Web Application project. Alternately you could also click on the New Web Application Project Toolbar icon as part of the Google Eclipse plugin.
2. In the New Web Application Project dialog, **deselect** the **Use Google Web Toolkit**

and give a name to your project. I have named mine MyFirstGoogleWaveRobot and I suggest you go with the same name so that things are consistent with the rest of the article. The Screenshot is shown below:



New Web Application Project

Create a Web Application Project

Create a Web Application project in the workspace or in an external location

Project name: MyFirstGoogleWaveRobot

Package: (e.g. com.example.myproject) com.gaejexperiments.waverobot

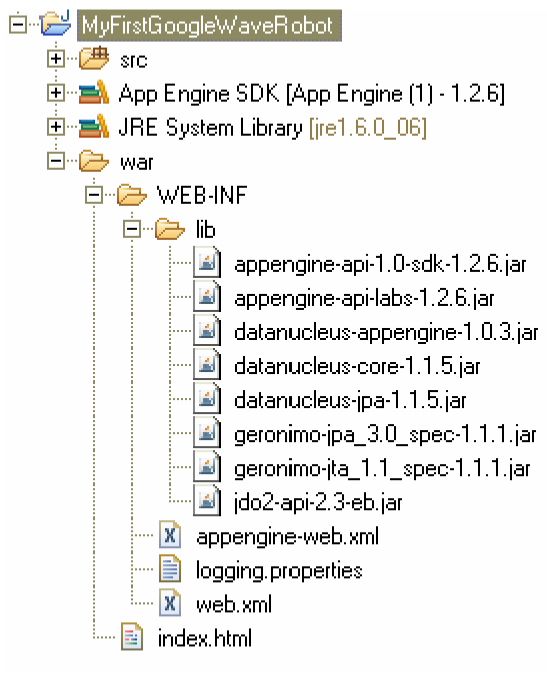
Location
- [✓ Create new project in workspace]
- Create new project in:

Directory: C:\galileo-workspace\MyFirstGoogleWaveRobot

Google SDKs
- ☐ Use Google Web Toolkit
  - Use default SDK (GWT - 1.7.0)
  - Use specific SDK: GWT - 1.7.0
- ☑ Use Google App Engine
  - [✓ Use default SDK (App Engine (1) - 1.2.6)]
  - Use specific SDK: App Engine (1) - 1.2.6

3. Click on Finish. This will generate the project and also create a sample Hello World Servlet for you. But we will be writing our own Servlet. **So I suggest that you can delete the Servlet Java class and the mappings made in the web.xml or you can leave it for now since we are going to write our own.**

The directory structure for your project should now look like this.



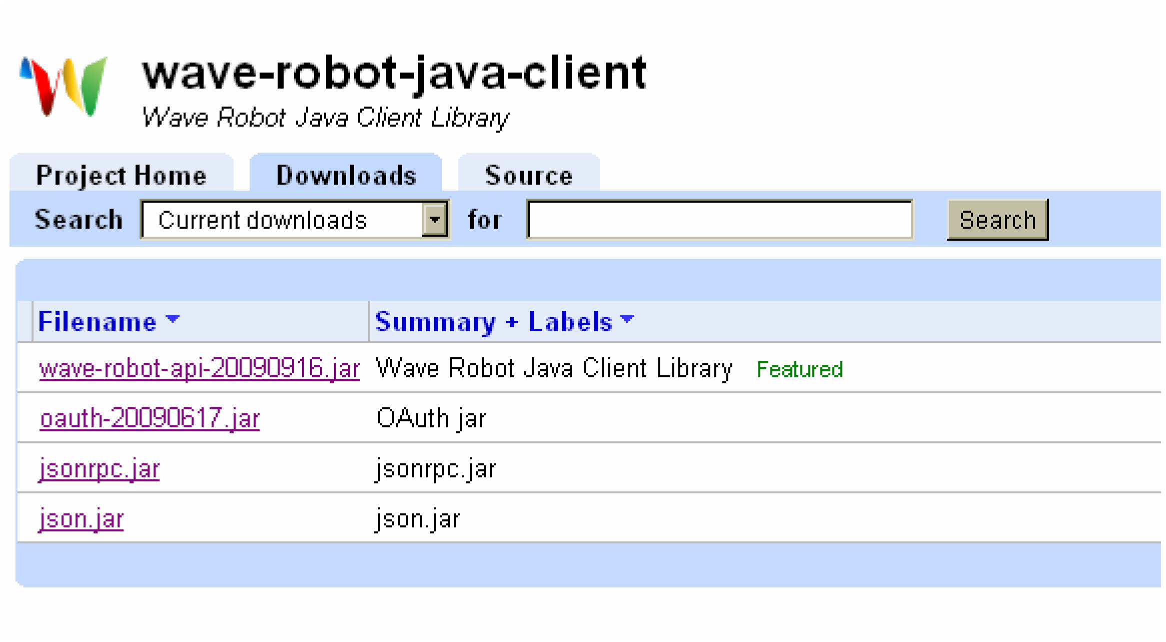
```
MyFirstGoogleWaveRobot
  src
  App Engine SDK [App Engine (1) - 1.2.6]
  JRE System Library [jre1.6.0_06]
  war
    WEB-INF
      lib
        appengine-api-1.0-sdk-1.2.6.jar
        appengine-api-labs-1.2.6.jar
        datanucleus-appengine-1.0.3.jar
        datanucleus-core-1.1.5.jar
        datanucleus-jpa-1.1.5.jar
        geronimo-jpa_3.0_spec-1.1.1.jar
        geronimo-jta_1.1_spec-1.1.1.jar
        jdo2-api-2.3-eb.jar
      appengine-web.xml
      logging.properties
      web.xml
    index.html
```

Adding Google Wave Robot JAR files to your Project Path

Since we are going to be writing a Wave Robot, we need some additional files on the client side. These additional files (JAR files) are required for the additional Wave API's and also for deployment in your **WEB-INF\lib** folder, so that they are correctly deployed and available to the run-time engine. These JAR files do not ship along with the Google Eclipse plugin, so you will need to download them for a website. The Google code website for the JAR files is:

http://code.google.com/p/wave-robot-java-client/downloads/list

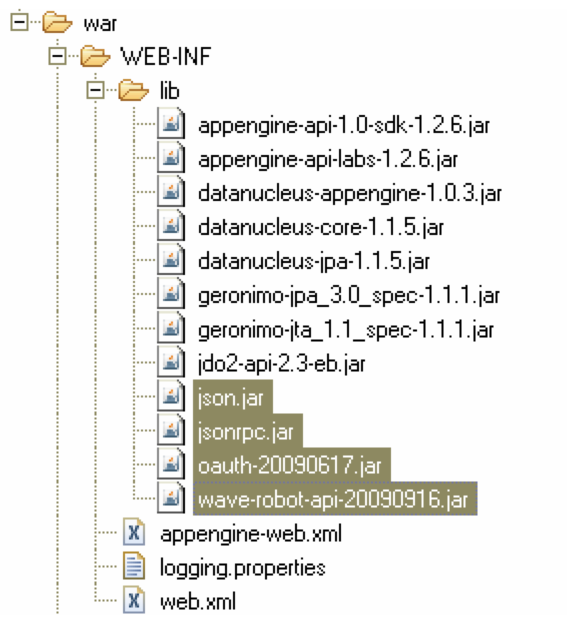
The web page when you navigate to the above URL is shown below:


wave-robot-java-client
Wave Robot Java Client Library

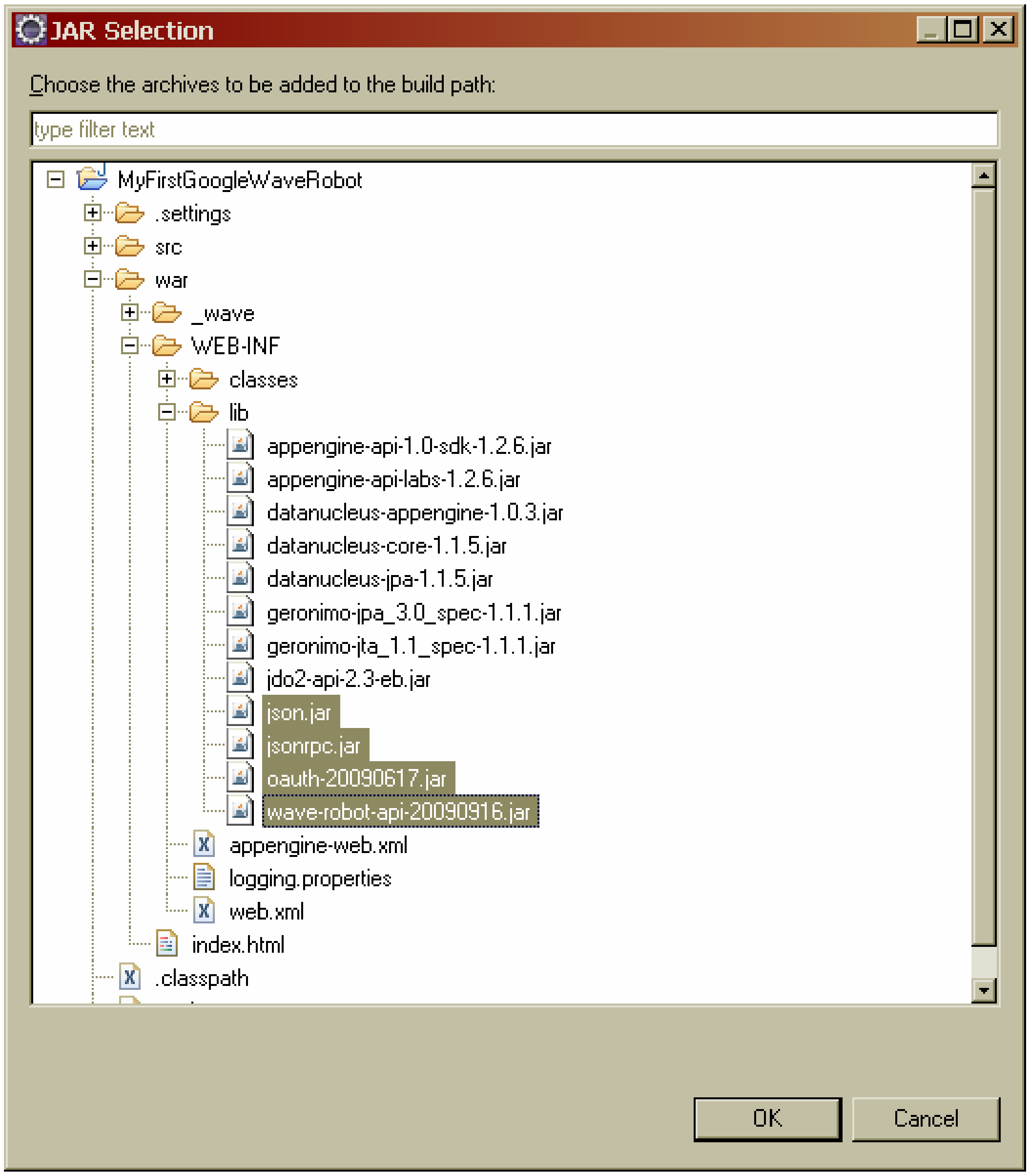
Filename	Summary + Labels
wave-robot-api-20090916.jar	Wave Robot Java Client Library Featured
oauth-20090617.jar	OAuth jar
jsonrpc.jar	jsonrpc.jar
json.jar	json.jar

Download all the above files to your machine. Once you have downloaded the files, follow these steps to setup your Project Build Path and runtime correctly.

1. Copy all the 4 JAR files to the WEB-INF\lib folder of your Eclipse Project. After copying you should see the files as shown in the project hierarchy below:

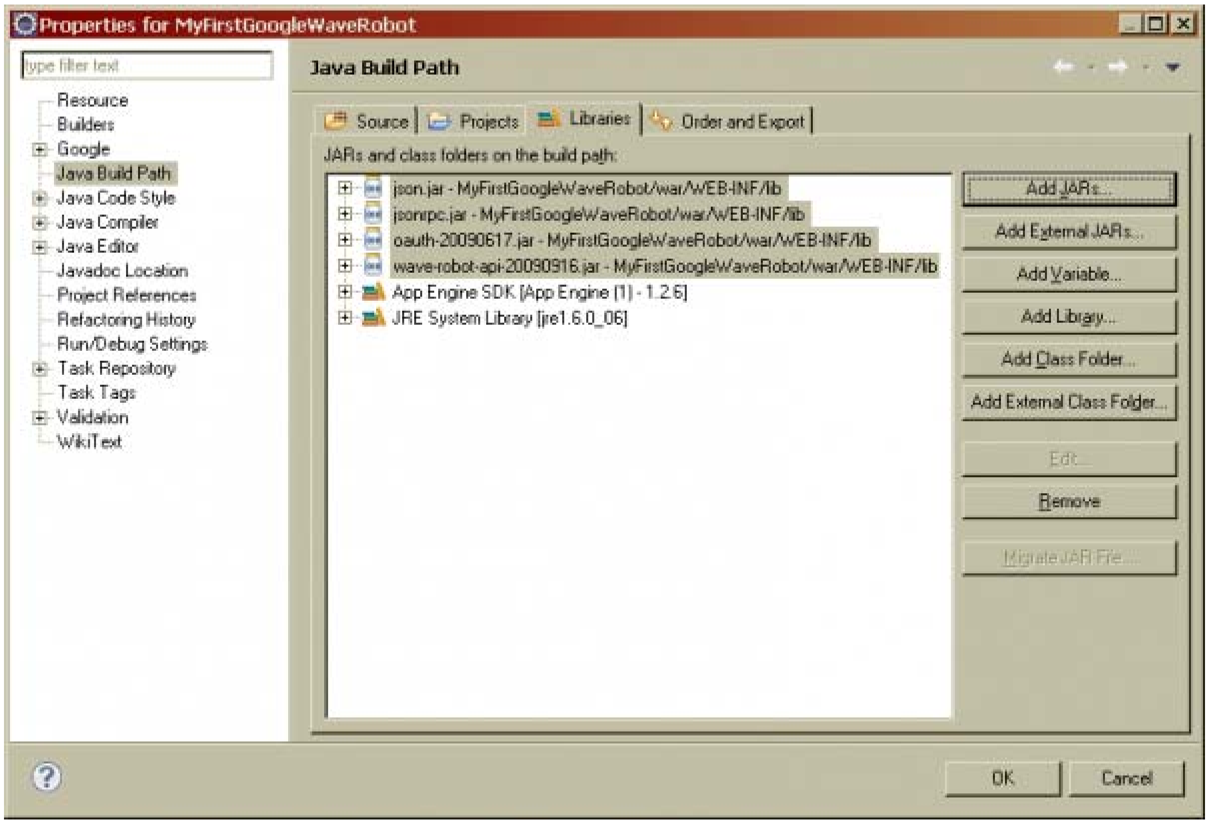

```
war
  WEB-INF
    lib
      appengine-api-1.0-sdk-1.2.6.jar
      appengine-api-labs-1.2.6.jar
      datanucleus-appengine-1.0.3.jar
      datanucleus-core-1.1.5.jar
      datanucleus-jpa-1.1.5.jar
      geronimo-jpa_3.0_spec-1.1.1.jar
      geronimo-jta_1.1_spec-1.1.1.jar
      jdo2-api-2.3-eb.jar
      json.jar
      jsonrpc.jar
      oauth-20090617.jar
      wave-robot-api-20090916.jar
    appengine-web.xml
    logging.properties
    web.xml
```

2. Right-click on the Project in the Project Hierarchy. Select **Properties** and then **Java Build Path.** Click on **Add JARs** and then select the 4 JAR files from your Project **WEB-INF\lib** folder as shown below and click on **OK**.



```
MyFirstGoogleWaveRobot
  .settings
  src
  war
    _wave
    WEB-INF
      classes
      lib
        appengine-api-1.0-sdk-1.2.6.jar
        appengine-api-labs-1.2.6.jar
        datanucleus-appengine-1.0.3.jar
        datanucleus-core-1.1.5.jar
        datanucleus-jpa-1.1.5.jar
        geronimo-jpa_3.0_spec-1.1.1.jar
        geronimo-jta_1.1_spec-1.1.1.jar
        jdo2-api-2.3-eb.jar
        json.jar
        jsonrpc.jar
        oauth-20090617.jar
        wave-robot-api-20090916.jar
      appengine-web.xml
      logging.properties
      web.xml
    index.html
  .classpath
```

3. Your Project Build Path should like the screenshot below.



JARs and class folders on the build path:

- json.jar - MyFirstGoogleWaveRobot/war/WEB-INF/lib
- jsonrpc.jar - MyFirstGoogleWaveRobot/war/WEB-INF/lib
- oauth-20090617.jar - MyFirstGoogleWaveRobot/war/WEB-INF/lib
- wave-robot-api-20090916.jar - MyFirstGoogleWaveRobot/war/WEB-INF/lib
- App Engine SDK [App Engine (1) - 1.2.6]
- JRE System Library [jre1.6.0_06]

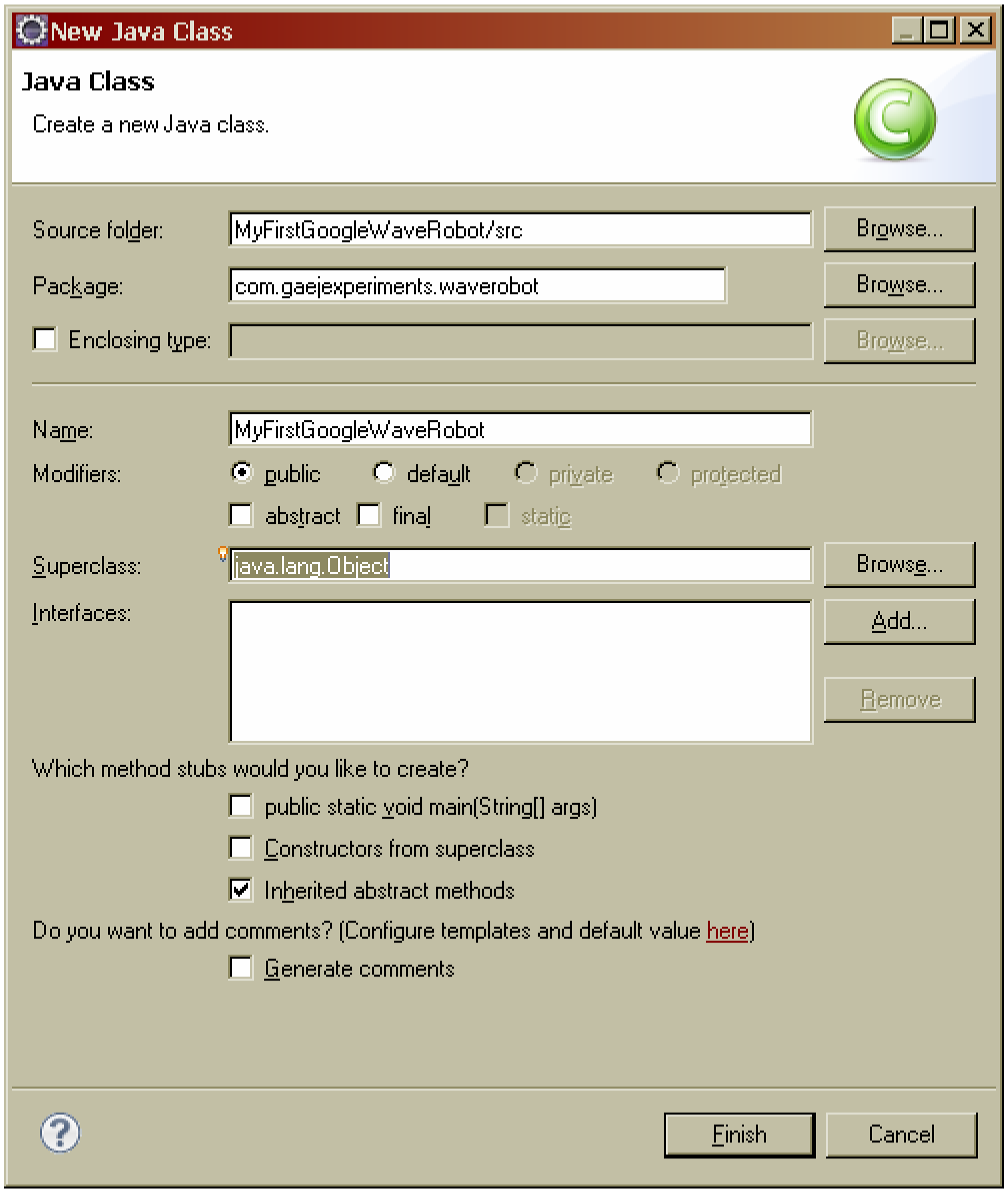
Click on OK to proceed. This completes your Build Path setup with the Google Wave Robot JAR files.

Writing the Google Wave Robot Servlet : MyFirstGoogleWaveRobot.java

Let us first create our Robot Java class. The steps are straightforward and given below. All we need to do is write our class that extends the **com.google.wave.api.AbstractRobotServlet** class and provide an implementation for the **processEvents** method.

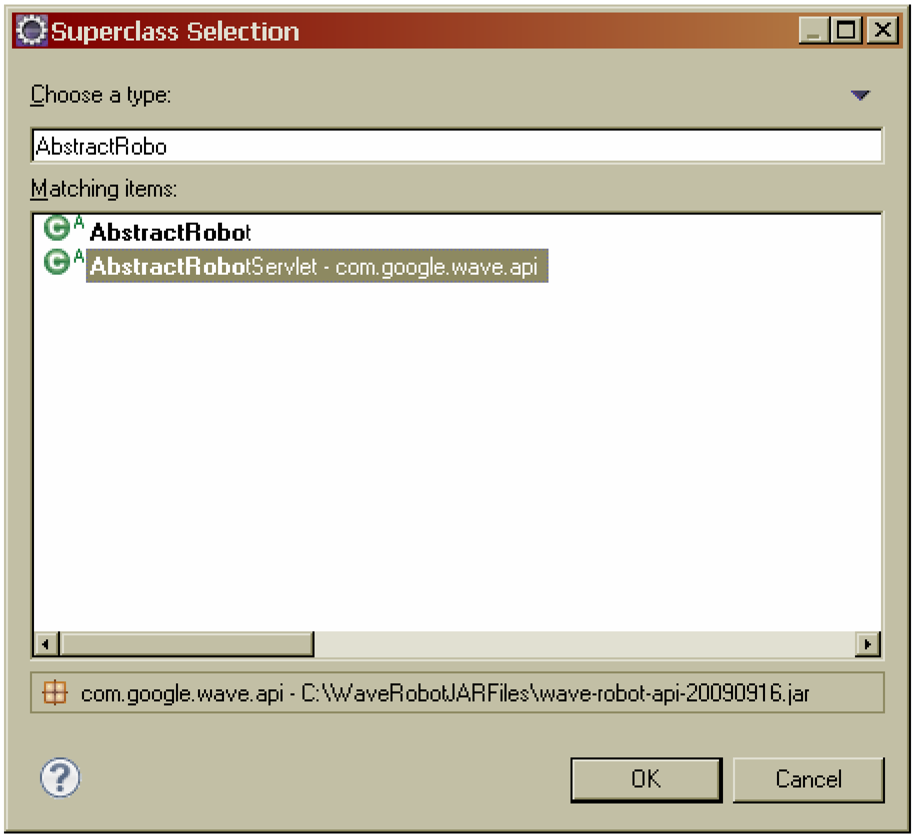
Follow these steps:

1. Create a new Java class within the same package. The New Java Class dialog is shown below. I have named the class **MyFirstGoogleWaveRobot** as shown below. Click on the **Browse** button to select a Super Class.

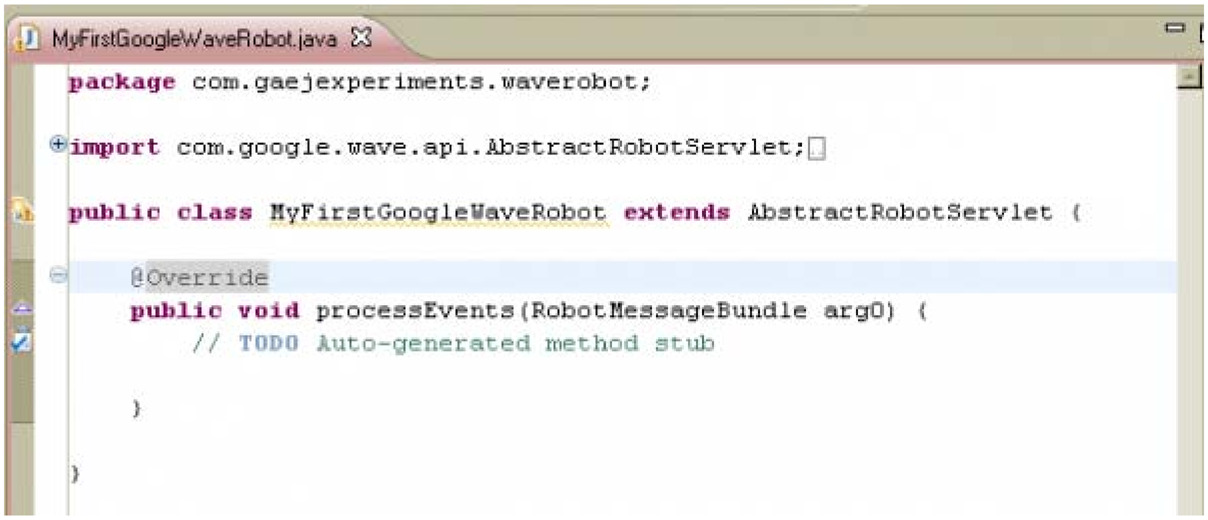


Source folder: MyFirstGoogleWaveRobot/src
Package: com.gaejexperiments.waverobot
☐ Enclosing type:
Name: MyFirstGoogleWaveRobot
Modifiers: [✓ public] default private protected
☐ abstract ☐ final ☐ static
Superclass: java.lang.Object
Interfaces:
Which method stubs would you like to create?
☐ public static void main(String[] args)
☐ Constructors from superclass
☑ Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
☐ Generate comments

2. In the **Superclass Selection** dialog shown below, type the word AbstractRobot (some part of it is also OK as the screenshot shows below) in the **Choose a type** field as shown. This will bring up the correct **Matching item**s i.e. **com.google.wave.api.AbstractRobotServlet**. Click on **OK**


Superclass Selection
Choose a type:
AbstractRobo
Matching items:
AbstractRobot
AbstractRobotServlet - com.google.wave.api
com.google.wave.api - C:\WaveRobotJARFiles\wave-robot-api-20090916.jar
OK
Cancel

This will generate the code as shown below in your Eclipse IDE.



```
package com.gaejexperiments.waverobot;

import com.google.wave.api.AbstractRobotServlet;

public class MyFirstGoogleWaveRobot extends AbstractRobotServlet {

    @Override
    public void processEvents(RobotMessageBundle arg0) {
        // TODO Auto-generated method stub

    }

}
```

Simply overwrite it with the code shown below:

```
package com.gaejexperiments.waverobot;

import com.google.wave.api.AbstractRobotServlet;
import com.google.wave.api.Blip;
import com.google.wave.api.Event;
import com.google.wave.api.EventType;
import com.google.wave.api.RobotMessageBundle;
import com.google.wave.api.TextView;
import com.google.wave.api.Wavelet;

public class MyFirstGoogleWaveRobot extends AbstractRobotServlet {

 @Override
 public void processEvents(RobotMessageBundle bundle) {
 Wavelet wavelet = bundle.getWavelet();
 if (bundle.wasSelfAdded()) {
 Blip blip = wavelet.appendBlip();
 TextView textView = blip.getDocument();
 textView.append("Hi! Your GAEJExperiments Bot is waiting for your
command...");
 }

 for (Event e: bundle.getEvents()) {
 if (e.getType() == EventType.BLIP_SUBMITTED) {
 //Get the Blip that was submitted
 Blip blip = e.getBlip();

 //Extract out MetaData information like ID and the creator of the Blip
 String strBlipMetaData = "Blip Id : " + blip.getBlipId() + " , " +
"Blip Creator : " + blip.getCreator();

 //Extract out the text that was entered in the blip
 String strBlipText = "You typed : " + blip.getDocument().getText();

 //Echo that out by creating a child within the submitted blip
 blip.createChild().getDocument().append(strBlipMetaData + "\r\n" +
strBlipText);
 }
 }
 }
}
```

Let us discuss the main points of the source code:

1. A Robot class needs to **extend** the **com.google.wave.api.AbstractRobotServlet** class

2. The only method that you need to implement to get going is the **processEvents()** method as shown above in the code. In this method, all we need to do is process the Events that we are interested in and take some action.

3. What we want our Robot to do is to announce itself to everyone when it is added to the Wave. This event is also pumped into the processEvents method but the API provides us a nice convenient method to detect if we have been added to the Wave via the bundle.wasSelfAdded() method. The code snippet is shown below:

```
public void processEvents(RobotMessageBundle bundle) {
Wavelet wavelet = bundle.getWavelet();
if (bundle.wasSelfAdded()) {
Blip blip = wavelet.appendBlip();
TextView textView = blip.getDocument();
textView.append("Hi! Your GAEJExperiments Bot is waiting for your
command...");
}

//...Rest of processEvents method
}
```

So the first thing we are doing is determining if we (i.e. the Robot has been added to the Wave. If yes, we need to add a message to the Wavelet (Threaded Conversation). Remember a message is known as a **Blip**. So we append a new Blip to the current wavelet. And now we need to set the message text for the Blip. To do that, we need to get the handle to a TextView for the Blip and then we append our Text to it. As simple as that.

4. Then we have the main Events loop in which we navigate through the events that are pumped into the loop by the Google Wave system. The code snippet is shown below:

```
for (Event e: bundle.getEvents()) {
if (e.getType() == EventType.BLIP_SUBMITTED) {
//Get the Blip that was submitted
Blip blip = e.getBlip();

//Extract out MetaData information like ID and the creator of the Blip
String strBlipMetaData = "Blip Id : " + blip.getBlipId() + " , " + "Blip
Creator : " + blip.getCreator();

//Extract out the text that was entered in the blip
String strBlipText = "You typed : " + blip.getDocument().getText();

//Echo that out by creating a child within the submitted blip
blip.createChild().getDocument().append(strBlipMetaData + "\r\n" +
strBlipText);
}
}
```

If the EventType is BLIP_SUBMITTED i.e. someone has submitted a message, then we first get the handle to the Blip. The Blip is not just the Text but a lot of other useful metadata. I simply demonstrate here two kinds of information about the blip, a BlipId and a Blip Creator. The blip creator is the name of the participant that created this blip. Then as we saw, we get to the Text of the Blip, by getting a handle on the getDocument() and then the getText() method. This gives us the text that was typed by the creator of the Blip. Finally, I insert a child Blip inside the Blip so that it looks like a response to that blip within the Google Wave. The response is nothing but a text appended with the metadata extracted and the echoing of the text that the creator typed.

Using this boiler plate of code, you can do your own thing. You could interpret commands given in a Blip by anyone and execute them and append the responses to their Blip. You could look up data on Amazon or any other web site if given the ISBN. The possibilities are limitless if you wish to modify the above code and make your Robot do something else.

That is all to writing a Robot but we still have some configuration to do to let Google Wave know that we have a Robot running in our GAEJ Application. And we will do that through the standard servlet entries in the **web.xml** file along with new XML configuration file that you need for the Robot called the **capabilities.xml** file.

Configuring the Robot Servlet

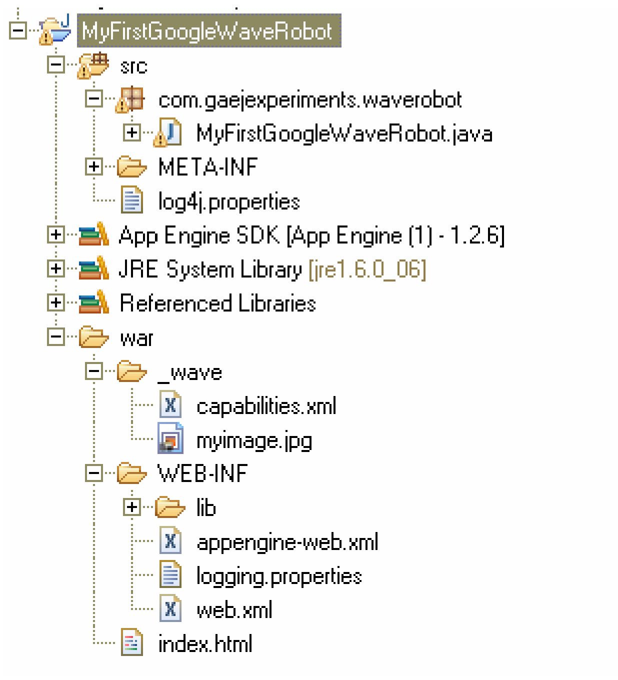
We need to add the **Robot Servlet <servlet/>** and **<servlet-mapping/>** entry to the **web.xml** file. This file is present in the **WEB-INF** folder of the project. The necessary fragment to be added to your **web.xml** file are shown below.

```
<servlet>
 <servlet-name>MyFirstGoogleWaveRobot</servlet-name>
 <servlet-
class>com.gaejexperiments.waverobot.MyFirstGoogleWaveRobot</servlet-
class>
 </servlet>
 <servlet-mapping>
 <servlet-name>MyFirstGoogleWaveRobot</servlet-name>
 <url-pattern>/_wave/robot/jsonrpc</url-pattern>
 </servlet-mapping>
```

In the above fragment, you will note that url-pattern **/_wave/robot/jsonrpc** has to be mapped to the Robot Servlet that you have written. This is because the Google Wave system will invoke this url to communicate with your Robot using its protocol.

Creating the Robot capabilities.xml files

We need an additional file to describe the capabilities of the Robot that we have written. This file is called the **capabilities.xml** and it needs to reside in a certain location. You need to create a **_wave** directory inside of the **war** directory of your project. The location of this file is shown below under the war/_wave directory.



```
MyFirstGoogleWaveRobot
  src
    com.gaejexperiments.waverobot
      MyFirstGoogleWaveRobot.java
    META-INF
    log4j.properties
  App Engine SDK [App Engine (1) - 1.2.6]
  JRE System Library [jre1.6.0_06]
  Referenced Libraries
  war
    _wave
      capabilities.xml
      myimage.jpg
    WEB-INF
      lib
      appengine-web.xml
      logging.properties
      web.xml
    index.html
```

You will need to create the _wave directory and create the **capabilities.xml** file over there. The capabilities file shown below is pretty straightforward. Two items are of interest. One is the a **<capability>** element for each EVENT that you wish your Robot get notified about. If you go back to the Source code of our Robot, you will notice that we were primarily interested in two events:

1. **BLIP_SUBMITTED** : Take a look at the source code. You will find that we checked for this event in our Events Loop and once a Blip (Message) was available, we extracted information and sent back a child Blip.

2. **WAVELET_PARTICIPANTS_CHANGED**: Take a look at the source code. This event was fired and we used a convenience method called bundle.wasSelfAdded() to find if we were added. In fact, you can put in an if else clause and catch this event in the Events loop too to get notified whenever anyone joins or leaves the Wave.

Now that it is clear that we need these two events, we **subscribe** to them by specifying the events in the **capabilities.xml** document. Any other events (Look at the **com.google.wave.api.EventType** class) that you are interested in should be mentioned here, otherwise your robot will not get notified about them. The other element is the **<version>** element. If you change any capabilities in your robot, then it is recommended that before you deploy, you change the version value over here, so that Google Wave can detect that there is a newer version and hence it can then query for your modified capabilities if any.

```
<?xml version="1.0" encoding="utf-8"?>
<w:robot xmlns:w="http://wave.google.com/extensions/robots/1.0">
 <w:capabilities>
 <w:capability name="WAVELET_PARTICIPANTS_CHANGED" content="true" />
 <w:capability name="BLIP_SUBMITTED" content="true" />
 </w:capabilities>
 <w:version>1</w:version>
</w:robot>
```

Writing our Robot Profile Servlet (not required but nice)

This is not a required step but it would be good practice to do so to make your Robot look more professional. A Profile Servlet is used to tell the following about your Robot:

1. A Name for your Robot

2. A custom image for your Robot

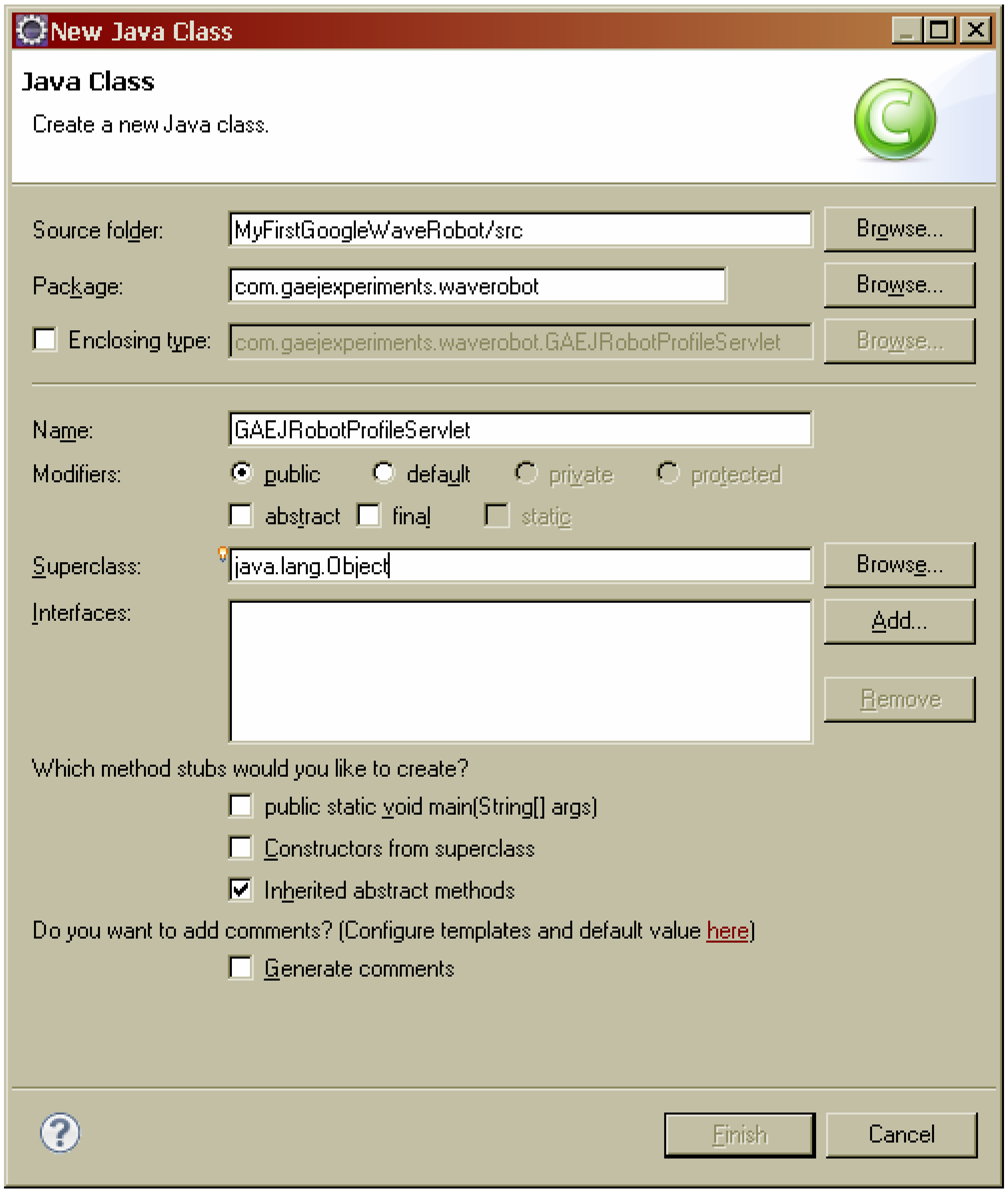
3. A profile page for your Robot (a URL)

If you provide these, then the Google Wave client is able to retrieve them and set it for your Robot when it is added as a participant. This makes the Robot look more professional.

This profile information needs to be provided by you by writing a Profile Servlet. The Profile Servlet is nothing but extending the com.google.wave.api.ProfileServlet class and providing simple implementations for the overwritten methods.

Follow these steps to write the Profile Servlet:

1. Create a new Java class within the same package. The New Java Class dialog is shown below. I have named the class **GAEJRobotProfileServlet** as shown below. Click on the **Browse** button to select a Super Class.


New Java Class

Java Class

Create a new Java class.

Source folder: MyFirstGoogleWaveRobot/src

Package: com.gaejexperiments.waverobot

Enclosing type: com.gaejexperiments.waverobot.GAEJRobotProfileServlet

Name: GAEJRobotProfileServlet

Modifiers: [✓ public] default private protected

abstract final static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

public static void main(String[] args)

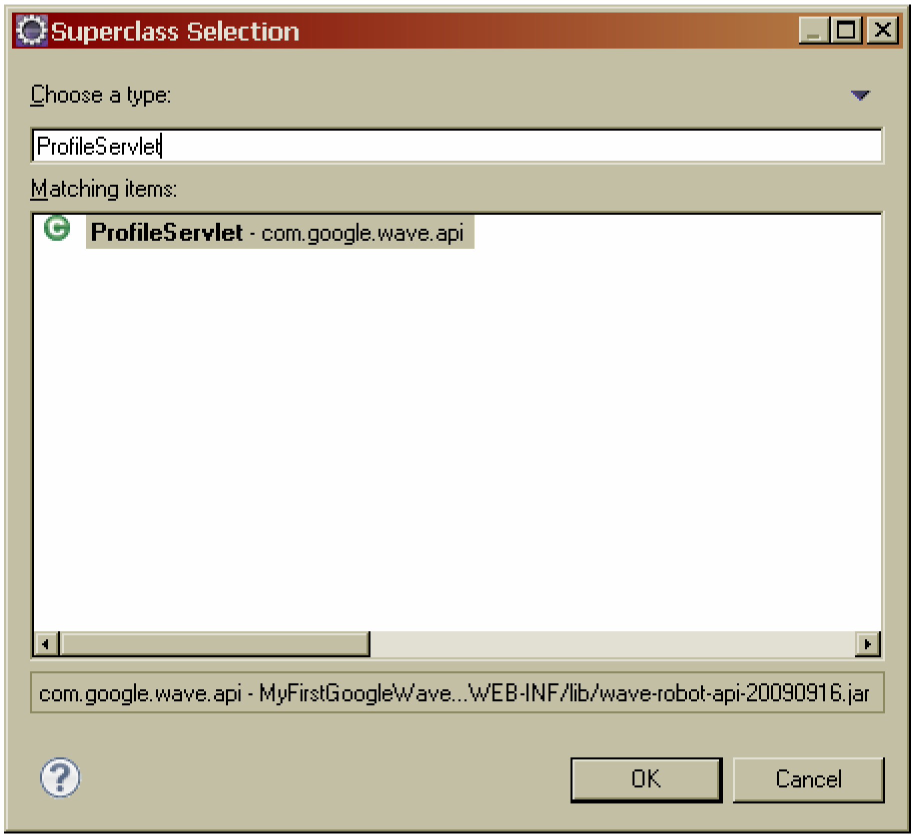
Constructors from superclass

[✓ Inherited abstract methods]

Do you want to add comments? (Configure templates and default value here)

Generate comments

2. In the **Superclass Selection** dialog shown below, type the word **ProfileServlet** in the **Choose a type** field as shown. This will bring up the correct **Matching items**s i.e. **com.google.wave.api.ProfileServlet** class. Click on **OK**.



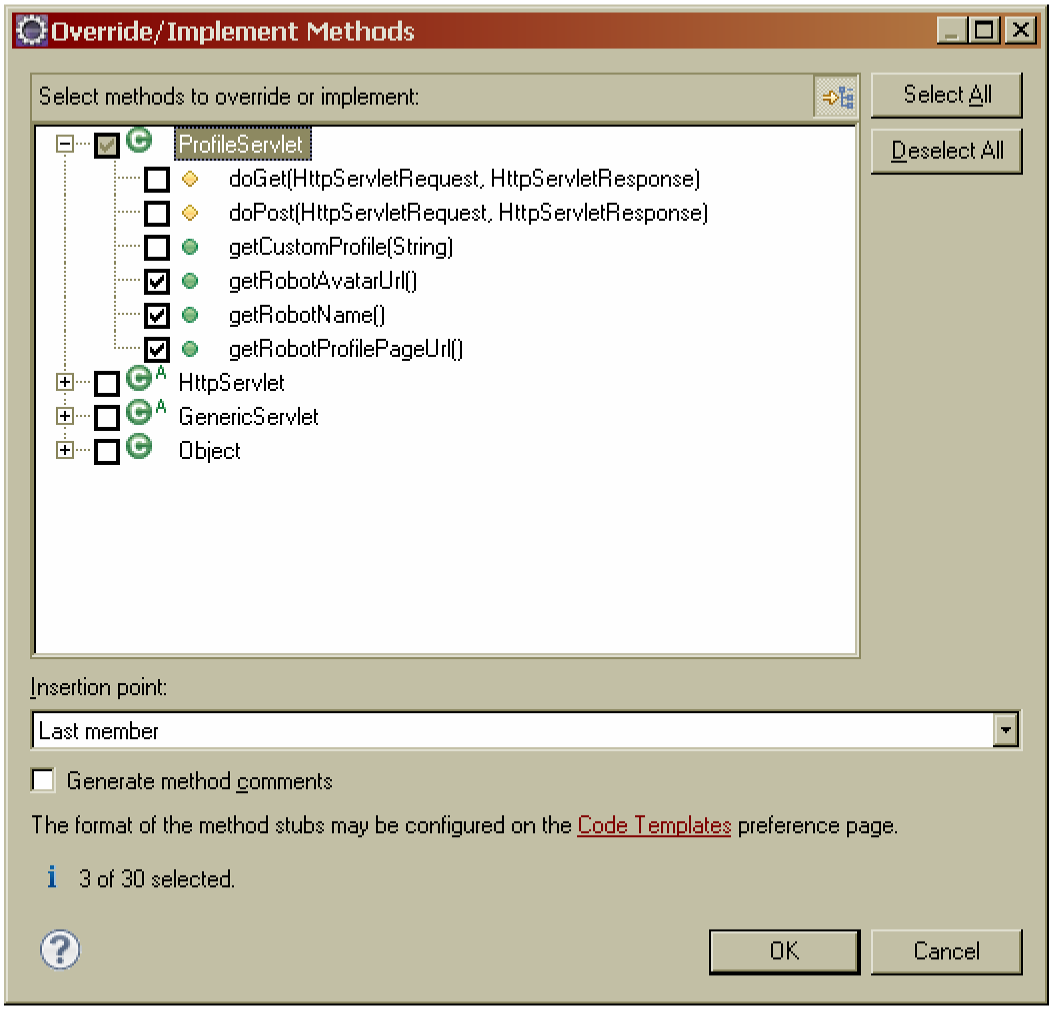
ProfileServlet

ProfileServlet - com.google.wave.api

com.google.wave.api - MyFirstGoogleWave...WEB-INF/lib/wave-robot-api-20090916.jar

3. This will generate a **GAEJRobotProfileServlet.java file.**

The simplest way to generate the stubs for the required methods would be to go to **Source –> Override/Implement** Methods. This will bring up the dialog box as shown below and you only need to select the 3 methods to override as shown:



Select methods to override or implement:

- [✓] ProfileServlet
  - [ ] doGet(HttpServletRequest, HttpServletResponse)
  - [ ] doPost(HttpServletRequest, HttpServletResponse)
  - [ ] getCustomProfile(String)
  - [✓] getRobotAvatarUrl()
  - [✓] getRobotName()
  - [✓] getRobotProfilePageUrl()
- [ ] HttpServlet
- [ ] GenericServlet
- [ ] Object

Insertion point:

Last member

[ ] Generate method comments

The format of the method stubs may be configured on the Code Templates preference page.

3 of 30 selected.

Click on **OK**. This will generate the stubs, which you can then overwrite with the code shown below. The code is easy to understand, all we are doing is providing values for the Name, Avatar(Image) and the Profile Page URL. **Note that for the Avatar, we are providing a file myimage.jpg present in the WAR/_wave folder. You will need to copy an appropriate image file for yourself and make sure that it is physically copied to the folder locally in your Eclipse project before you deploy your application.** You should also replace the word **gaejerobot** in the source code below with your Application ID.

```
package com.gaejexperiments.waverobot;

import com.google.wave.api.ProfileServlet;

public class GAEJRobotProfileServlet extends ProfileServlet {

 @Override
 public String getRobotAvatarUrl() {
 return "http://gaejrobot.appspot.com/_wave/myimage.jpg";
 }
 @Override
 public String getRobotName() {
 return "GAEJ Robot";
 }

 @Override
 public String getRobotProfilePageUrl() {
 return "http://gaejrobot.appspot.com";
 }

}
```

Configuring the Profile Servlet

We need to add the **Profile Servlet <servlet/>** and **<servlet-mapping/>** entry to the **web.xml** file. This file is present in the **WEB-INF** folder of the project. The necessary fragment to be added to your **web.xml** file are shown below.

```
<servlet-name>GAEJRobotProfileServlet</servlet-name>
 <servlet-
class>com.gaejexperiments.waverobot.GAEJRobotProfileServlet</servlet-
class>
 </servlet>
 <servlet-mapping>
 <servlet-name>GAEJRobotProfileServlet</servlet-name>
 <url-pattern>/_wave/robot/profile</url-pattern>
 </servlet-mapping>
```

In the above fragment, you will note that url-pattern **/_wave/robot/profile** has to be mapped to the Profile Servlet that you have written. This is because the Google Wave system will invoke this url to get hold of your profile.

Deploying the Application

To deploy the application, you will need to first create your **Application ID**. The Application Identifier can be created by logging in at **http://appengine.google.com** with your Google Account. You will see a list of application identifiers already registered under your account (or none if you are just getting started). To create a new Application, click on the Create Application button and provide the Application Identifier as requested. Please note this name down since you will be using it for deployment.

For e.g. I have registered an application identifier named **gaejrobot**.

To deploy the application, follow these steps (they should be familiar to you now):

1. Click on the Deploy Icon in the Toolbar.

2. In the Deploy dialog, provide your Email and Password. Do not click on Deploy button yet.
3. Click on the **App Engine Project settings** link. This will lead you to a dialog, where you need to enter your Application ID [For e.g. my Application Identifier **gaejrobot**]
4. Click on **OK**. You will be lead back to the previous screen, where you can click on the **Deploy** button. This will start deploying your application to the GAEJ cloud. You should see several messages in the Console window as the application is being deployed.
5. Finally, you should see the message "Deployment completed successfully".

GAEJ Robot in Action

Your application is going to be available at the http://**yourapplicationid**.appspot.com. In my case, the application is available at **http://gaejrobot.appspot.com**.

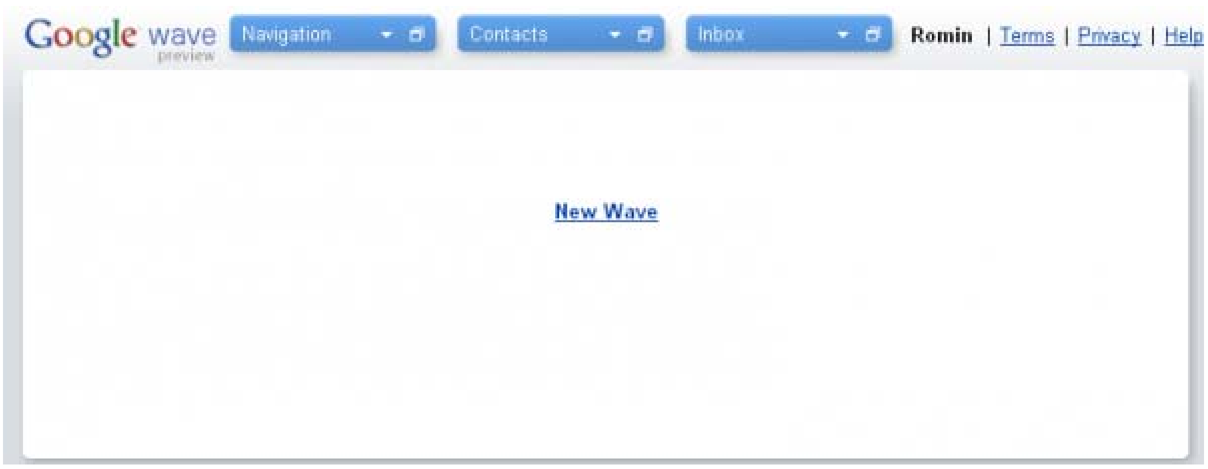
You can test for the presence of your robot capabilities file by simply typing in the following:

http://**yourapplicationid.**appspot.com**/_wave/capabilities.xml** [Replace yourapplicationid with the Application ID that you have]

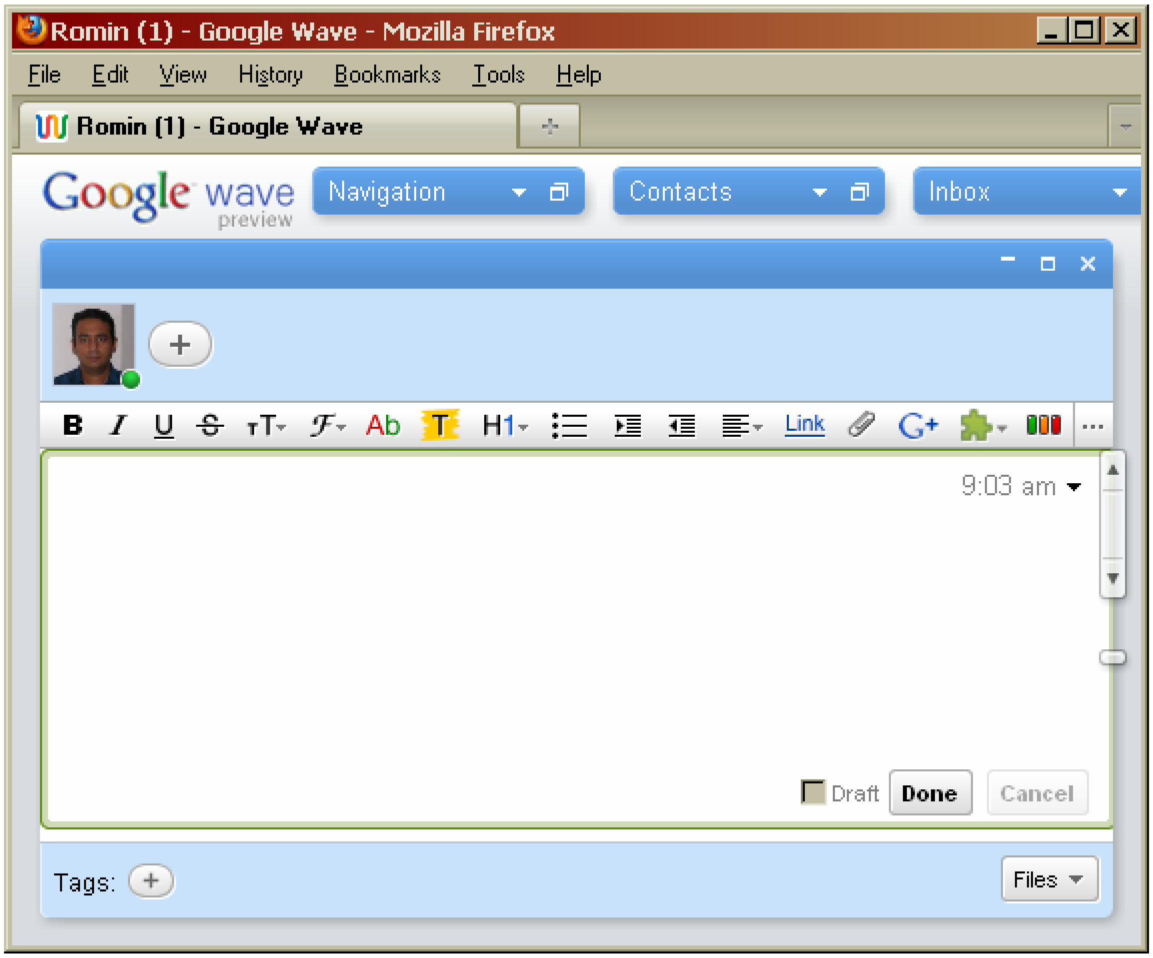
For e.g. when I navigate to the following url (http://gaejrobot.appspot.com/_wave/capabilities.xml) for my robot application, I get the capabilities xml as shown below, which means that the robot is ready and serving if all is right.

```
<w:robot>
<w:capabilities>
<w:capability name="WAVELET_PARTICIPANTS_CHANGED" content="true"/>
<w:capability name="BLIP_SUBMITTED" content="true"/>
</w:capabilities>
<w:version>1</w:version>
</w:robot>
```

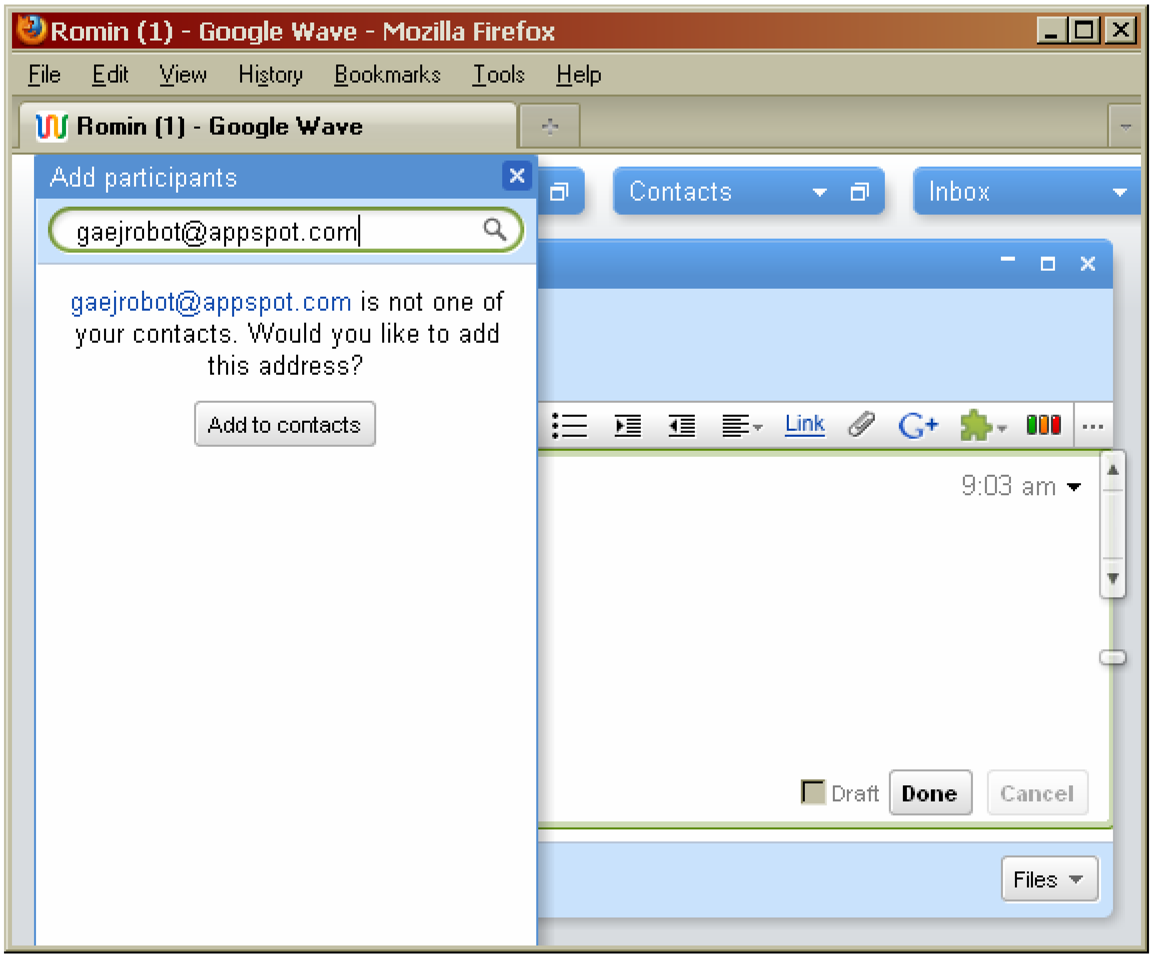
To test out the Robot, you need to launch the Google Wave client and login in with your account by going to http://wave.google.com. On successful login, you will be inside the Wave client from where you can create a new wave by clicking on the **New Wave** link as shown below:



When you do that, currently you are the only participant (myself) as shown in the screen below:



Click on the **+ sign** next to your icon and you can add one or more participants as shown below:



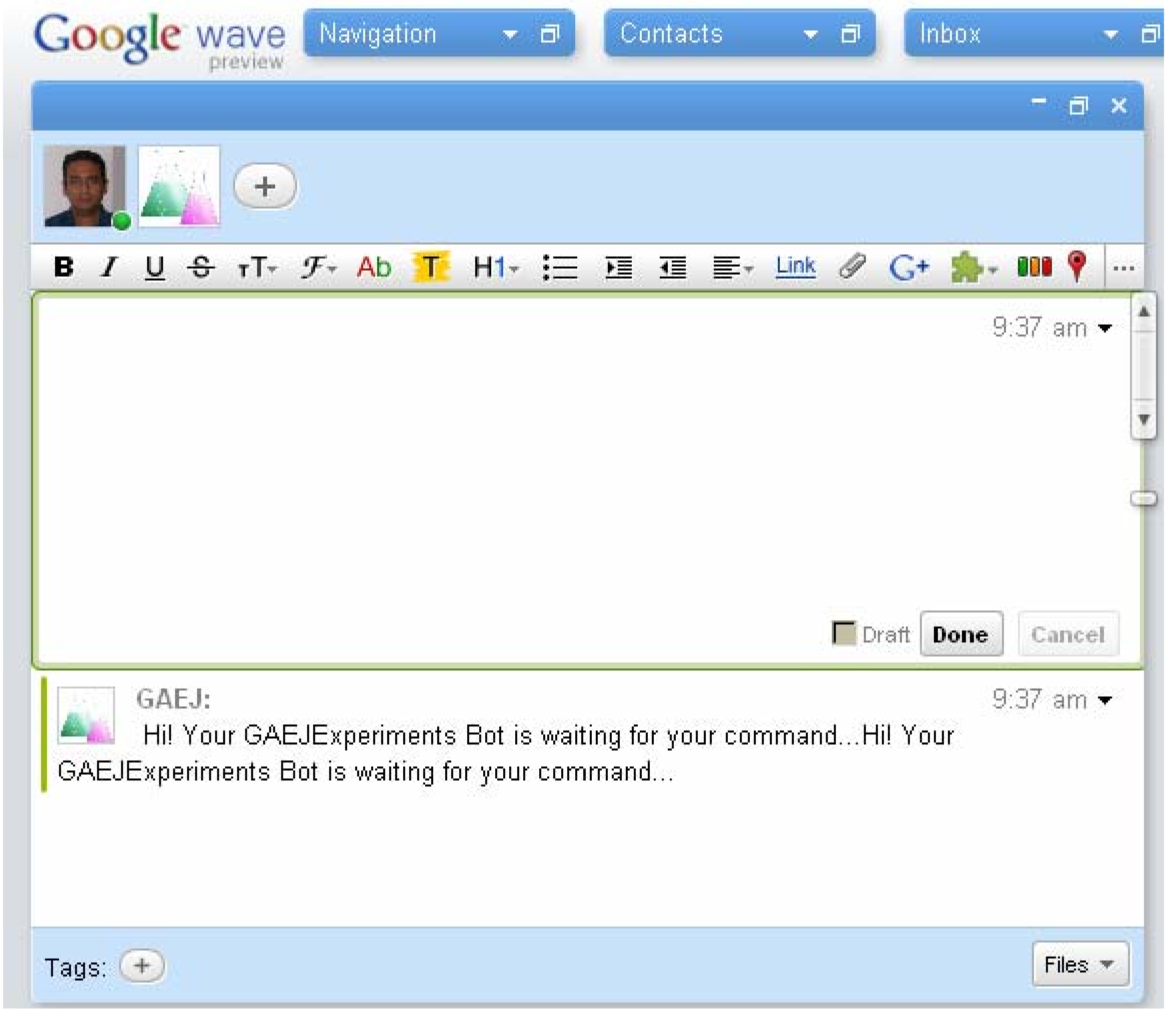
gaejrobot@appspot.com

gaejrobot@appspot.com is not one of your contacts. Would you like to add this address?

Add to contacts

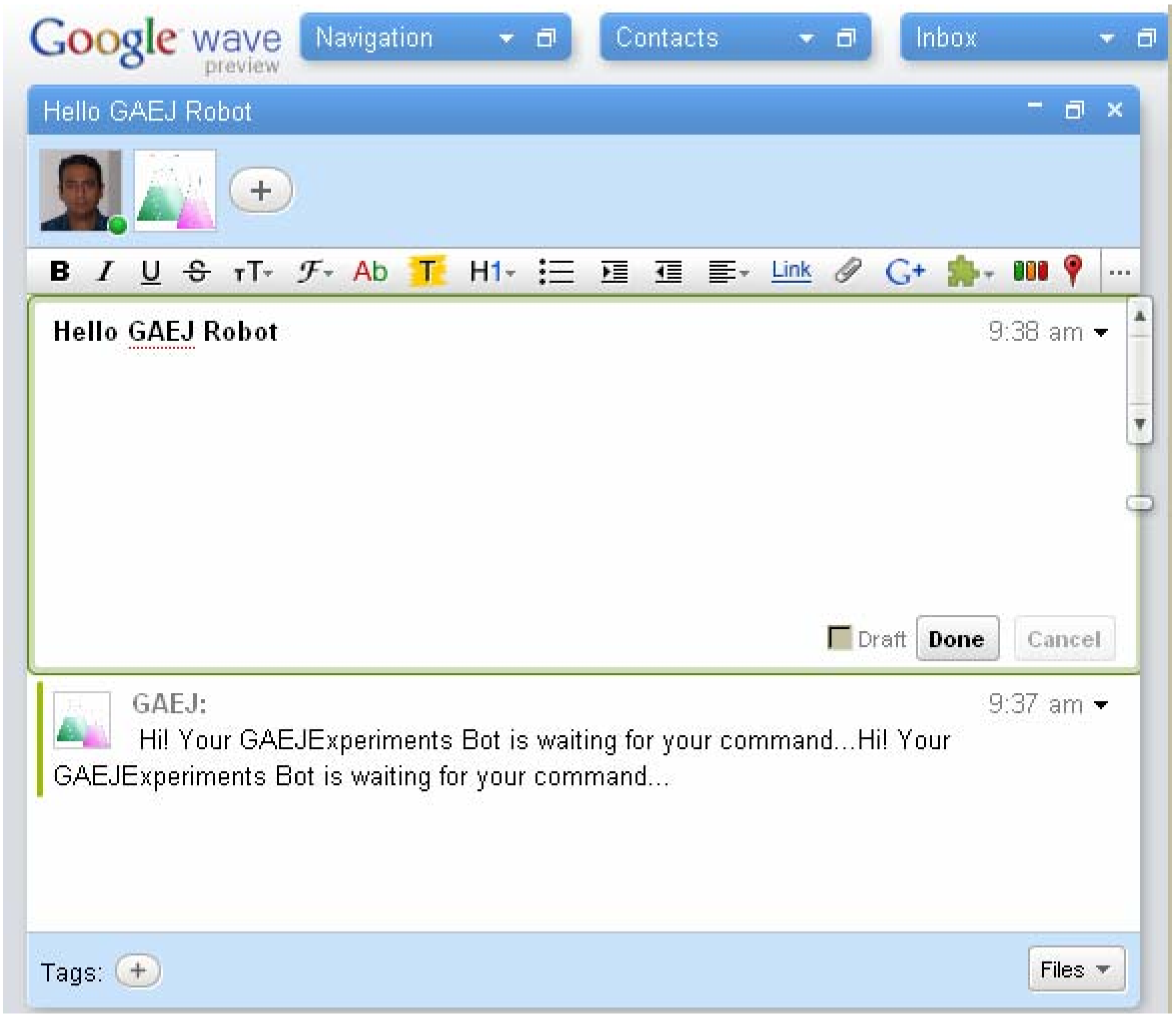
NOTE : Your Google Wave Robot is going to be available at <YOURAPPLICATIONID>@appspot.com , hence I have added **gaejrobot@appspot.com** as that was my application id. But you can replace it with your application id.

If all goes well, you will see your Robot added as a participant (with the icon and all, since the Profile Servlet is invoked behind the scenes by the Google Wave system). **Since the Robot got added, it received a** WAVELET_SELF_ADDED **Event and since we had coded our Robot Servlet to receive that event and print out the greeting message, you see the message saying "Hi! Your GAEJExperiments Bot …..".**



GAEJ:
Hi! Your GAEJExperiments Bot is waiting for your command...Hi! Your GAEJExperiments Bot is waiting for your command...

Next I start typing a message “Hello GAEJ Robot” as shown below. But note that I have still not pressed the **Done** button. Till the **Done** button is pressed, the BLIP_SUBMITTED is not fired.



Hello GAEJ Robot 9:38 am

GAEJ: 9:37 am
Hi! Your GAEJExperiments Bot is waiting for your command...Hi! Your GAEJExperiments Bot is waiting for your command...

Once I click on the **Done** button, the BLIP_SUBMITTED event is fired and our Robot Servlet gets the event. On receiving the event, it simply prints out some metadata and echoes back the message

That is all there is to writing a Google Wave Robot. The example here is not useful as such but the intent of the article is to make sure you get your Robot running and have the right setup in place. Several examples are available over the web in which talented programmers are writing Robots that do clever stuff. Just look around and get inspired.

Going back in Time

In **Episode 2** of this series, we discussed how to write a XMPP Bot that you could add as a friend in XMPP Chat clients like Google Talk, Spark, etc. That bot is not ready to be added to a Google Wave since it does not implement the end-points that are discussed here i.e. /_wave/robot/jsonrpc, etc. So all you will need to do is to follow the steps outlined here and provide these Servlet implementations along with the end-points and capabilities file. Just add them to your previous project and redeploy your Google App Engine application. You should then have a Bot that is both XMPP compliant and also Google Wave compliant.

Parting notes

Google has posted some excellent documentation on Wave extensions over here and I recommend them to be read. Key among them is the design aspects about the extensions which are covered here. It is important to add a parting note that Google Wave is about collaboration. Collaboration is a process in which participants join each other and engage in doing something. And if your Robot is to participate in a collaboration then you need to think out well in advance how it will collaborate with the other participants and how it can add value to the process. It has to do something that brings meaning and utility to the collaboration. Of course one can write a fun Robot or a pretty much useless Robot that I have demonstrated here, but the key thing to take away is the immense power that a Robot could play in a collaborative process. Imagine a Robot querying a backend system and fetching a Purchase Order or an Order just placed at a website. Think of a recruiter asking a Robot to pull up interview slots from a backend system and then presenting that to the candidate to choose. The possibilities are endless and it should be an exciting journey ahead to see the Robots that people come up with.

Episode 8: Using Memcache in your GAEJ applications

Welcome to **Episode 8**. In this episode, we shall cover an important service that is provided in the Google App Engine. The service is called **Memcache** and is used allow applications to manage their data in a cache.

What is a Cache and why do we need one?

As per the Wikipedia definition, a **cache** is a temporary storage area where frequently accessed data can be stored for rapid access. Once the data is stored in the cache, it can be used in the future by accessing the cached copy **rather than re-fetching or recomputing** the original data.

The decision to use a cache in your application should come after carefully determining which operations would be benefit from it. Look at the following scenarios:

1. If you have stored information in a database and which is not updated frequently, then it might make sense to put some of that information in a cache so that if multiple requests come for the same data then you can simply look up the memory cache and retrieve it from there, rather than make repeated database calls that are expensive (and which may return the same data).

2. If you invoke external web services and you determine that the same data could be requested, then a cache would help here too to avoid expensive network calls.

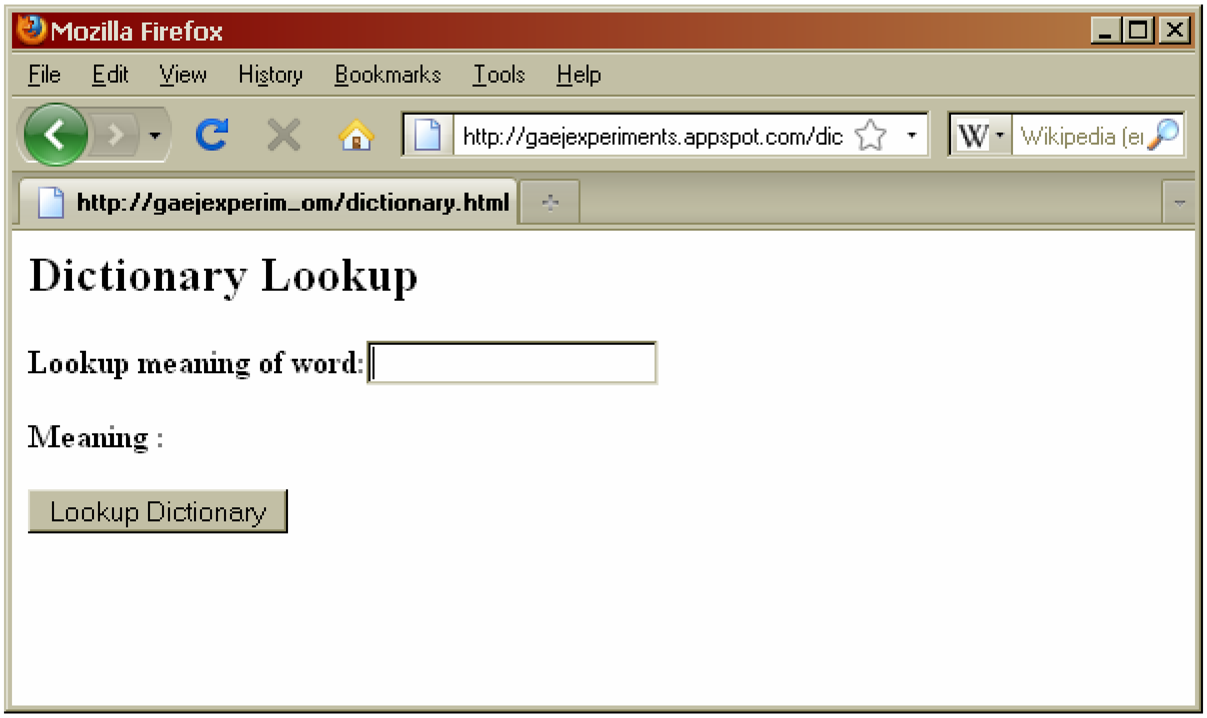
There is a lot of material available on the use of a cache in software applications and I suggest reading from those sources to arrive at a good caching design in your applications. We shall keep our discussion here limited to a simple use case of **using Memcache**, which is a Caching Service provided by Google App Engine and how to implement it quickly in your application.

Before we begin (Important!)

We will be introducing MemCache in an existing application that we have written. This was the Dictionary Service Application that we implemented in **Episode 4**. I strongly urge you to read up that episode and understand what the application was about and have that project ready to make changes accordingly.

To recap, our GAEJ Dictionary application is shown below:

1. Navigate to **http://gaejexperiments.appspot.com/dictionary.html**. This will show a page as shown below:

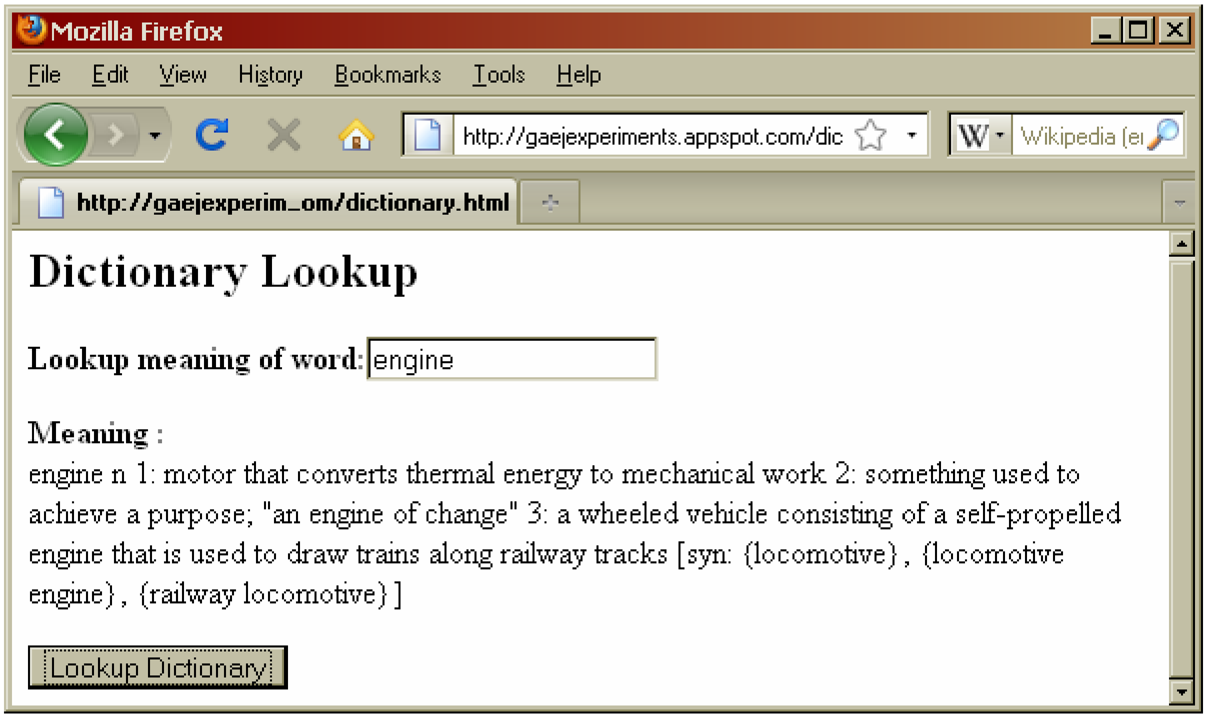

Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://gaejexperiments.appspot.com/dic
Wikipedia (e
http://gaejexperim...om/dictionary.html

Dictionary Lookup

Lookup meaning of word:

Meaning :

2. Enter the word that you wish to lookup the definition for in a dictionary and it will return you the result as shown below:


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://gaejexperiments.appspot.com/dic
Wikipedia (e
http://gaejexperim...om/dictionary.html

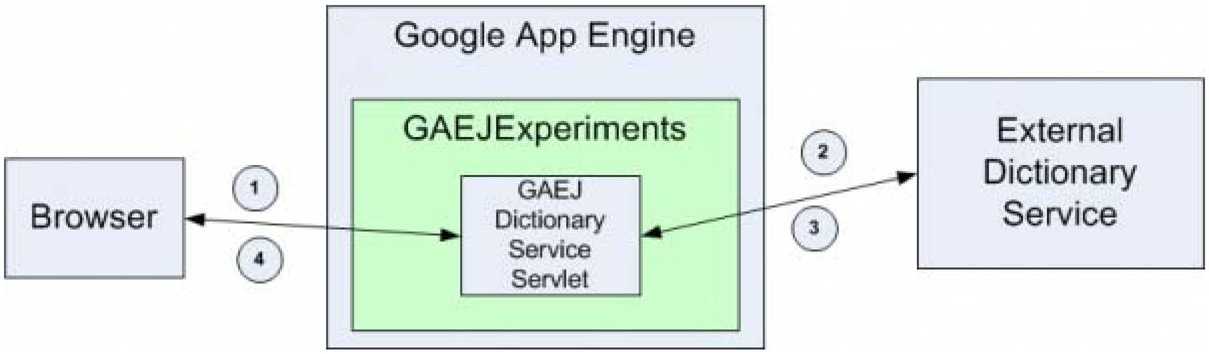
Dictionary Lookup

Lookup meaning of word: engine

Meaning :
engine n 1: motor that converts thermal energy to mechanical work 2: something used to achieve a purpose; "an engine of change" 3: a wheeled vehicle consisting of a self-propelled engine that is used to draw trains along railway tracks [syn: {locomotive}, {locomotive engine}, {railway locomotive}]

Introducing a Cache

The request/response flow for the above Dictionary application is explained below via the diagram shown:



The steps were as follows:

1. The user makes a request to the GAEJ Experiments application by sending a request for the word.

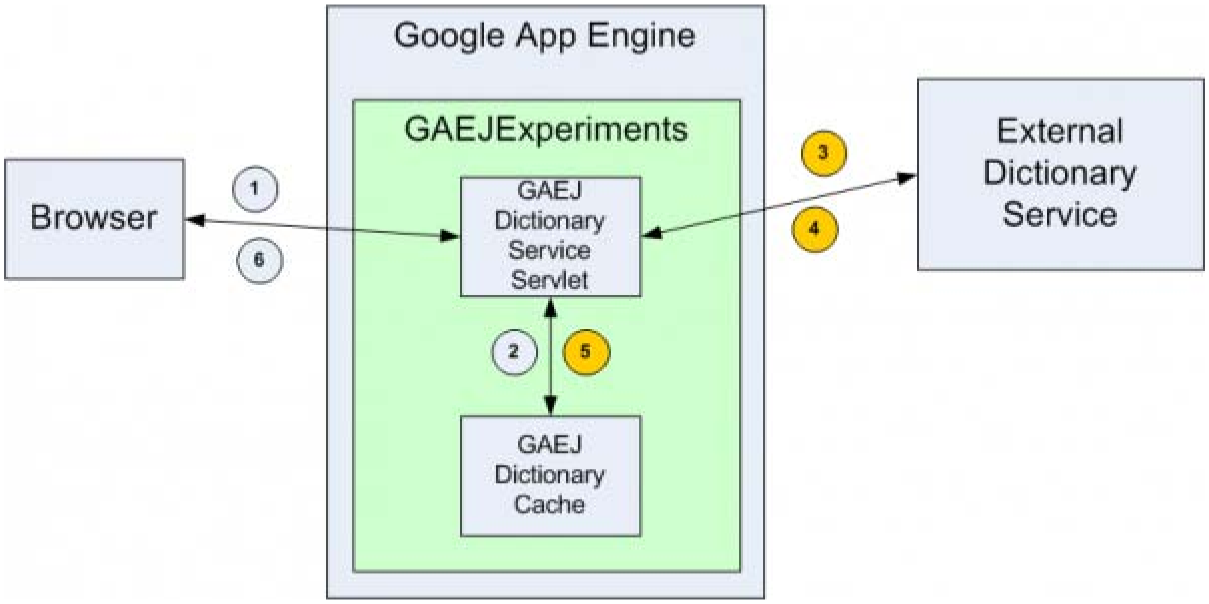
2. The GAEJ Dictionary Service Servlet receives the Request and makes a call to an External Dictionary Service hosted at (**http://services.aonaware.com/DictService/DictService.asmx**).

3. The Response from the Dictionary Service i.e. the definition of the word is received by the GAEJ Dictionary Service Servlet.

4. The Response is sent back to the user application.

The flow is straightforward and typically found in most applications. Now, what happens **if several users make a request for the same word**? As per the application flow, each request from the user will result in a call to an external service for the definition, irrespective of whether it was the **same word or not**. This is wasteful in terms of network resources and the application response too. Coupled with the fact that the definition of a word is not going to change overnight 😃, it would be nice to return back the definition from the GAEJExperiments application itself, if the word definition had been looked up already. **Enter a cache!**

So, what we are introducing in our application flow now is a cache called GAEJ Dictionary Cache and the modified application flow is shown below:



The steps now would be as follows:

1. The user makes a request to the GAEJ Experiments application by sending a request for the word.

2. The GAEJ Dictionary Service Servlet receives the Request and checks if the word and its definition is **already present in the Cache. If it is present in the Cache, then we short circuit and go to Step 6.**

Optional Steps (3,4,5)
3. If the Servlet does not find the definition in the Cache, then it makes a call to the External Dictionary Service hosted at (**http://services.aonaware.com/DictService/DictService.asmx**).

4. The Response from the Dictionary Service i.e. the definition of the word is received by the GAEJ Dictionary Service Servlet.

5. The Servlet **puts** the word and its definition **in the Cache**, **so that future requests to determine if the word/definition is present in the Cache are fulfilled.**

6. The Response is sent back to the user application.

To summarize, we introduced the Cache that functions as follows:

- All word definitions looked up from the External Service are placed in the Cache.
- If a word is already present in the Cache, then it is returned from the Cache itself and an external network call is saved.

The Memcache Service API

A cache is typically implemented as a Map. A Map is a generic data structure that contains a key and its value. You look up the Cache by specifying the key and if it is found, then the value associated with that key is returned. What I am describing here is an over simplification of what a Cache is. There is a lot more to a Cache implementation then just putting a value and getting a value. But we shall keep it simple here.

The Memcache Service API is simple enough to use and there is good documentation available on it **here**. The Memcache Service implements JSR-107 (JCache Interface). The JCache classes are present in the **javax.cache** package and that it what you will use.

At a high level, all you need to do is follow these steps:

1. Get a handle to the Cache implementation:

The snippet of code to do that (reproduced from the Documentation) is shown here:

```
import java.util.Collections;
import javax.cache.Cache;
import javax.cache.CacheException;
import javax.cache.CacheFactory;
import javax.cache.CacheManager;

Cache cache;

try
{
CacheFactory cacheFactory =
CacheManager.getInstance().getCacheFactory();
cache = cacheFactory.createCache(Collections.emptyMap());
}
catch (CacheException e)
{
// ...
}
```

The code is simple. We get a handle to the **CacheFactory** instance and then create a **Cache**. Notice that we are creating a Map i.e. an empty Map. Once we have the Map, then all we need to do is play around with the **(key,value)** pairs.

2. Put a value or get a value from the Cache

Shown below is how we would use the cache in a simple manner. We invoke the **put(key,value)** method on the **javax.cache.Cache** instance. Similarly, to extract a value, we need to invoke the **cache.get(key)** value. It will return us the value if found, which we can then typecast to the appropriate class.

```
String key;      // The Word
String value;    // The Definition of the Word

// Put the value into the cache.
cache.put(key, value);

// Get the value from the cache.
value = (String) cache.get(key);
```

GAEJDictionaryCache.java

Let us first discuss a utility class that I have written that encapsulates the Caching API. I have made this class a **singleton** and it is called **GAEJDictionaryCache**. The source code is shown below:

```
package com.gaejexperiments.networking;

import java.util.Collections;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.cache.Cache;
import javax.cache.CacheException;
import javax.cache.CacheFactory;
import javax.cache.CacheManager;

public class GAEJDictionaryCache {
 public static final Logger _log =
Logger.getLogger(GAEJDictionaryCache.class.getName());

 private static GAEJDictionaryCache _instance;
 private Cache cache;

 private GAEJDictionaryCache() {
 try {
 CacheFactory cacheFactory =
CacheManager.getInstance().getCacheFactory();
 cache = cacheFactory.createCache(Collections.emptyMap());
 }
 catch (CacheException e) {
 //Log stuff
 _log.log(Level.WARNING, "Error in creating the Cache");
 }
 }

 public static synchronized GAEJDictionaryCache getInstance() {
 if (_instance==null) {
 _instance = new GAEJDictionaryCache();
 }
 return _instance;
 }
 public String findInCache(String word) {
 if (cache.containsKey(word)) {
 return (String)cache.get(word);
 }
 else {
 return null;
 }
 }

 public void putInCache(String word, String definition) {
 cache.put(word,definition);
 }
}
```

Let us discuss the key parts of the code:

1. The Singleton design pattern should be obvious over here and the application needs to use the **getInstance()** method to obtain a handle to this singleton.
2. The constructor of this class is private and called only once. In that an instance of the Cache is created.
3. There are two utility methods written : **findInCache** and **putInCache**.
4. The application invokes **findInCache** by providing a key value. If the key is found, the value is written via the **cache.get(key)** method. For our dictionary application, the key is the word that you wish to look up the definition for.
5. If the application wants to put a key,value record into the Cache, then it invokes the **putInCache(…)** method that takes the key and the value. For our dictionary application, the key is the word and the value is the definition of the word.

By encapsulating the MemCache Service API in this fashion, you can create a reusable class that takes care of Cache API details. You could then improve upon this class and provide advanced Cache features and APIs and reuse it in all your future GAEJ applications.

Modifying the Original GAEJDictionaryService.java class

All that remains now is for us to modify the existing **GAEJDictionaryService.java** class by introducing the appropriate cache usage. The modified code is shown below and I suggest to look at the comment **//MEMCACHE** to make it easier for you to see what I have changed.

```
package com.gaejexperiments.networking;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.StringReader;
import java.net.URL;
import java.util.logging.Logger;

import javax.servlet.ServletException;
import javax.servlet.http.*;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathFactory;

import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;

@SuppressWarnings("serial")
public class GAEJDictionaryService extends HttpServlet {
 public static final Logger _log =
Logger.getLogger(GAEJDictionaryService.class.getName());
 public void doGet(HttpServletRequest req, HttpServletResponse resp)
 throws IOException {

 String strCallResult = "";
 resp.setContentType("text/plain");
 try {

 //Extract out the word that needs to be looked up in the Dictionary
Service
 String strWord = req.getParameter("word");

 //Do validations here. Only basic ones i.e. cannot be null/empty
 if (strWord == null) throw new Exception("Word field cannot be empty.");

 //Trim the stuff
 strWord = strWord.trim();
 if (strWord.length() == 0) throw new Exception("Word field cannot be
empty.");

 //MEMCACHE
 //First get a handle to the Cache
 GAEJDictionaryCache _cache = GAEJDictionaryCache.getInstance();

 //Determine if the value is present in the Cache
 String strWordDefinition = _cache.findInCache(strWord);

 //If the word/definition is present in the Cache, return that
straightaway, no need for external network call
 if (strWordDefinition != null) {
 //Return the definition
 _log.info("Returning the Definition for ["+strWord+"]"+" from
Cache.");
```

```
 strCallResult = strWordDefinition;
 }
 else {
 _log.info("Invoking the External Dictionary Service to get Definition
for ["+strWord+"]");
 //Make the Network Call
 String strDictionaryServiceCall =
"http://services.aonaware.com/DictService/DictService.asmx/Define?word=
";
 strDictionaryServiceCall += strWord;
 URL url = new URL(strDictionaryServiceCall);
 BufferedReader reader = new BufferedReader(new
InputStreamReader(url.openStream()));
 StringBuffer response = new StringBuffer();
 String line;

 while ((line = reader.readLine()) != null) {
 response.append(line);
 }
 reader.close();

 strCallResult = response.toString();

 DocumentBuilderFactory builderFactory =
DocumentBuilderFactory.newInstance();
 DocumentBuilder builder = builderFactory.newDocumentBuilder();
 Document doc = builder.parse(new InputSource(new
StringReader(strCallResult.toString())));

 XPathFactory factory = XPathFactory.newInstance();
 XPath xpath = factory.newXPath();
 XPathExpression expr =
xpath.compile("//Definition[Dictionary[Id='wn']]/WordDefinition/text()"
);

 Object result = expr.evaluate(doc, XPathConstants.NODESET);
 NodeList nodes = (NodeList) result;
 for (int i = 0; i < nodes.getLength(); i++) {
 strCallResult = nodes.item(i).getNodeValue();
 }

 //MEMCACHE
 //Need to check depending on your logic if the values are good
 //Currently we will assume they are and put it in the cache
 //For e.g. if the word is not found, the Dictionary Service gets the
word back.
 //So you could use that logic if you want.
 _cache.putInCache(strWord, strCallResult);
 }

 resp.getWriter().println(strCallResult);

 }
 catch (Exception ex) {
 strCallResult = "Fail: " + ex.getMessage();
 resp.getWriter().println(strCallResult);
 }
 }

 @Override
 public void doPost(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException, IOException {
 doGet(req, resp);
 }

}
```

Let us discuss the modified flow in brief:

1. We first extract out the **word** request parameter and do some basic validation to make sure that it is not empty.
2. We get the handle to the Cache by calling our utility class that we just wrote.

```
      //First get a handle to the Cache
      GAEJDictionaryCache _cache = GAEJDictionaryCache.getInstance();
```

3. We check if the word is present in the cache. The findInCache method will return null if not found.

```
//Determine if the value is present in the Cache
 String strWordDefinition = _cache.findInCache(strWord);
```

4. If the definition is returned from the cache, then we simply return that and no network call is made.
5. If the definition is not found, then the network call is made, the response stream is parsed out for the definition. **And the most important step is to put this definition in the cache, so that the findInCache(…) method will get the definition, the next time another request for the same word is made.**

```
_cache.putInCache(strWord, strCallResult);
```

Try it out

You can try out the modified Dictionary Service application at: **http://gaejexperiments.appspot.com/dictionary.html**. The first time that you search for a word, it might take some time to get back the definition but once present in the cache, it is returned much faster. Do note, that if the same word has been looked up by another user, then the word and its definition will be present in the cache.

Cache Design considerations

A cache if implemented well will save your application from repeated resource intensive operations and also improve application response times significantly. Of course, simply putting in a cache comes with risks. Some of the factors, you need to take into consideration are:

1. Determine precisely which operations in your application would benefit from a cache.

2. Analyse if the data that you are putting in a cache changes frequently or not. If it changes frequently, then it might negate the benefits of a cache.

3. Control the size of the cache and prevent it from becoming too large or unmanageable. The way we have implemented it, the cache will keep growing and we are leaving it to the App Engine run time to truncate our cache due to memory limitations if any.

4. Take into consideration several other factors like what is an interval across which you want to refresh the cache, cache expiration policies, etc.

In short, monitoring the cache and tuning its parameters is required from any production worthy application.

Moving on

This brings Episode 8 to an end. Hope you enjoyed reading it. If you have been following the series so far, I would appreciate if you can give me some feedback on how the series is faring so far. Thank you for your time.

Episode 9: Using the Cron Service to run scheduled tasks

Welcome to **Episode 9**. In this episode, we shall be looking at how you can run background tasks in your GAEJ Application. By background Task, I mean any piece of code that you would like to run at a scheduled time and independent of the user interaction.There are several examples of such tasks. For e.g. :

- Hourly/Daily/Weekly/Monthly backup of data
- End of the day report generation to report any errors, transactions, etc.
- Sending an email at the end of the day (or once a day) with some information to subscribers. For e.g. News sites.

If you have written a few web applications, you would definitely have come across more scenarios like that.

In this episode, we shall cover the following:

1. What is a Cron Job?
2. How to schedule a Cron Job?
3. Write a simple Cron Job that prints a single statement
4. Configure, execute and monitor the Cron Job execution

Let’s go!

What is a Cron Job? When would you need one?

I will use information liberally from Wikipedia over here to explain some of the core concepts. You can refer to the Cron page at Wikipedia if you want.

The word ‘cron’ is short for Chronograph. A Cron is a time-based job scheduler. It enables our application to **schedule** a job to run automatically at a certain time or date. A Job (also known as a Task) is any module that you wish to run. This module can perform system maintenance or administration, though its general purpose nature means that it can be used for other purposes, such as connecting to the Internet and downloading email.

Examples include:

- Taking a daily backup of data via a scheduled task and moving the file to another server. (Runs once daily)
- Sending an email every week to your subscribers. (Runs once weekly)
- Clearing the log files at the end of every day (Runs once daily)
- Remind yourself of a wonderful weekend coming up, every Friday at 5:00 PM (Runs once a week on a Friday at 5:00 PM)

The Google App Engine provides a service called the [Cron Service] that helps us do two fundamental things:

1. Allows your application to schedule these tasks.
2. Execute these tasks based on their schedule.

What does a Cron Job look like? And how do I schedule one?

A Cron Job is nothing but a **URL** that is invoked by the Google App Engine infrastructure at its scheduled execution time. To write a Cron Job, you need to do the following:

1. Write a **Java Servlet** and configure it in the **web.xml**. Note down the **URL** where the servlet can be invoked. The URL is the **<url-pattern>** mentioned in the **<servlet-mapping>** for your Servlet configuration in web.xml. For e.g. the URL is the <url-pattern> element specified in the segment of the web.xml that is shown below:

```
<servlet>
 <servlet-name>GAEJCronServlet</servlet-name>
 <servlet-class>com.gaejexperiments.cron.GAEJCronServlet</servlet-
class>
 </servlet>

<servlet-mapping>
 <servlet-name>GAEJCronServlet</servlet-name>
 <url-pattern>/cron/mycronjob</url-pattern>
 </servlet-mapping>
```

2. Create a **cron.xml** file that specifies one or more Cron Jobs (Scheduled Tasks) that you want to execute. A sample for the above Cron Job is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<cronentries>
 <cron>
 <url>/cron/mycronjob</url>
 <description>Put your Cron Job description here</description>
 <schedule>Put Cron Job Schedule here</schedule>
 </cron>
</cronentries>
```

The **cron.xml** file tells Google App Engine about the Cron Jobs that are scheduled by your application. This file resides in the **WEB-INF** directory of your application and is copied to the App Engine cloud when you deploy the application. The following points are important about the **cron.xml** file:

1. Each Cron Job configured in your application is defined in a **<cron/>** element. So there can be one or more **<cron/>** elements.

2. The above **<cron/>** element has the following 3 elements that defines the Job.
    - **<url/>** specifies where the Google App Engine can invoke your Cron Job. This is nothing but the **Servlet URL** that you defined in the **web.xml** file that we saw earlier.The Servlet URL will point to your Servlet which contains the Cron Job implementation.
    - **<description/>** is a simple text based description of what your Cron Job does. It does not influence any aspect of the execution and is used for display purposes when you look at your application configuration via the App Console.
    - **<schedule/>** is the time when your Job has to be executed. This is where you specify if your job is to be run daily, once every hour, on Friday at 5:00 PM, etc. It is completely dependent on when you wish to execute this job. However, you must follow some rules and they are specified in the documentation on **Scheduling Format**. I **strongly** recommend you to read it up to understand various ways of specifying the schedule. Some of the examples are: “every 1 minute”, “every 12 hours”, “every friday 17:00″ and so on.

Develop a simple Cron Job

The first thing to do is to create a **New Google Web Application** Project. Follow these steps:

1. Either click on **File –> New –> Other** or **press Ctrl-N** to create a new project. Select Google and then Web Application project. Alternately you could also click on the New Web Application Project Toolbar icon as part of the Google Eclipse plugin.

2. In the New Web Application Project dialog, **deselect the Use Google Web Toolkit** and give a name to your project. I have named mine **GAEJExperiments**. I suggest you go with the same name so that things are consistent with the rest of the article, but I leave that to you. **In case you are following the series, you could simply use the same project and skip all these steps altogether. You can simply go to the next part i.e. the Servlet code.**

3. Click on Finish. This will generate the project and also create a sample Hello World Servlet for you. But we will be writing our own Servlet.

GAEJCronServlet.java

Our Cron Job is going to be very simple. It is simply going to print out a statement in the log file that says that it is getting executed. The Cron Service of Google App Engine automatically will invoke this Servlet when its scheduled time to execute has arrived. So all we need to do is code out Servlet. The code is shown below:

```
package com.gaejexperiments.cron;

import java.io.IOException;
import java.util.logging.Logger;

import javax.servlet.ServletException;
import javax.servlet.http.*;

@SuppressWarnings("serial")
public class GAEJCronServlet extends HttpServlet {
 private static final Logger _logger =
Logger.getLogger(GAEJCronServlet.class.getName());
 public void doGet(HttpServletRequest req, HttpServletResponse resp)
 throws IOException {

 try {
 _logger.info("Cron Job has been executed");

 //Put your logic here
 //BEGIN
 //END
 }
 catch (Exception ex) {
 //Log any exceptions in your Cron Job
 }
 }

 @Override
 public void doPost(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException, IOException {
 doGet(req, resp);
 }
}
```

The code is straightforward to understand. It has **doGet()** and **doPost()** methods. And you will find in the **doGet()** method, that we simply log with an INFO level, that the Cron Job has been executed. In fact, your actual Job implementation should go in here as indicated by the comments. So whether you are invoking a backend database, or sending a consolidated email report, etc should all go in here.

All that remains is to now tell the App Engine via configuration about your Servlet (**via web.xml**) and create the **cron.xml** file in which you will mention your Cron Job.

Configure the Cron Job

As mentioned, we need to configure the Servlet in the **web.xml** and also specify it in the **cron.xml** file. Let us look at that now:

Configuring the Servlet

We need to add the **<servlet/>** and **<servlet-mapping/>** entry to the **web.xml** file. This file is present in the **WEB-INF** folder of the project. The necessary fragment to be added to your **web.xml** file are shown below. Please note that you can use your own namespace and servlet class. Just modify it accordingly if you do so.

```
<servlet>
  <servlet-name>GAEJCronServlet</servlet-name>
  <servlet-class>com.gaejexperiments.cron.GAEJCronServlet</servlet-
class>
</servlet>

<servlet-mapping>
  <servlet-name>GAEJCronServlet</servlet-name>
  <url-pattern>/cron/gaejcronjob</url-pattern>
</servlet-mapping>
```

Specifying the Cron Job (cron.xml)

The **cron.xml** for our application will contain only one Cron Job. And here we specify the Servlet URL along with the schedule. Notice that I have chosen to execute this Cron job every 2 minutes. But you are free to experiment if you like with different **Schedule Formats**. This files needs to be created in the **WEB-INF** folder of your project.

```
<?xml version="1.0" encoding="UTF-8"?>
<cronentries>
 <cron>
 <url>/cron/gaejcronjob</url>
 <description>GAEJExperiments Cron Job that simply announces that it
got invoked.</description>
 <schedule>every 2 minutes</schedule>
 </cron>
</cronentries>
```

Deploy the Application

To deploy the application, follow these steps (they should be familiar to you now. I am assuming that you already have the Application ID with you):

1. Click on the Deploy Icon in the Toolbar.
2. In the Deploy dialog, provide your Email and Password. Do not click on Deploy button yet.
3. Click on the **App Engine Project settings** link. This will lead you to a dialog, where you need to enter your Application ID [For e.g. my Application Identifier **gaejexperiments**]
4. Click on **OK**. You will be lead back to the previous screen, where you can click on the **Deploy** button. This will start deploying your application to the GAEJ

cloud. You should see several messages in the Console window as the application is being deployed.

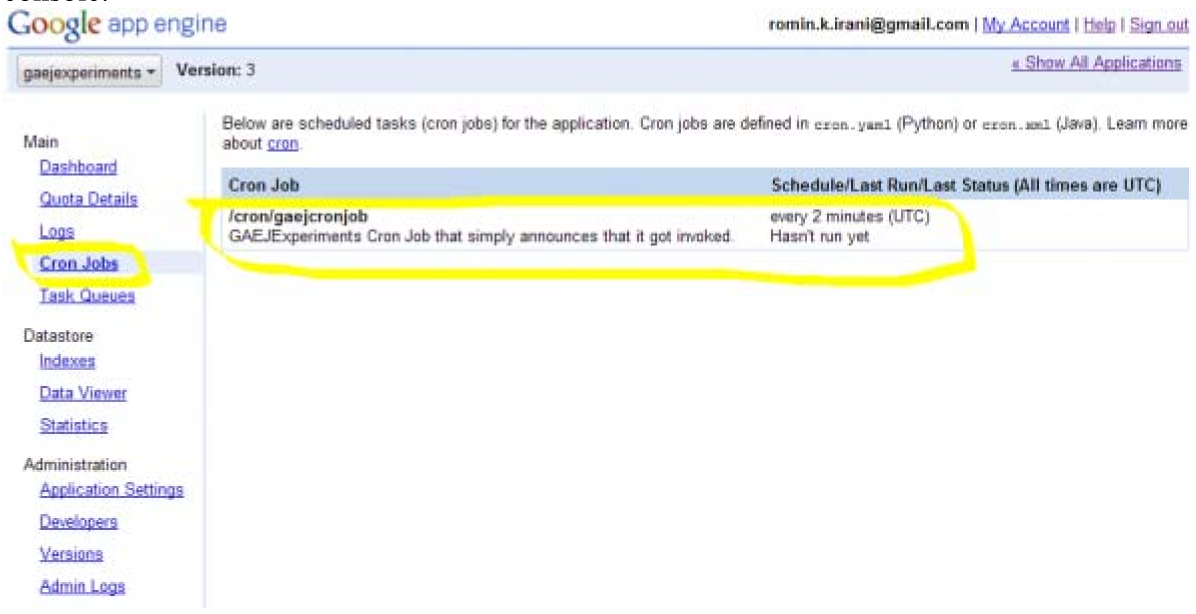
5. Finally, you should see the message "Deployment completed successfully".

We can now check if the Google App Engine got our Cron Job correctly configured and verify if it is getting executed at the schedule that we have configured it to.

Monitoring the Cron Job

You can use the App Engine console to verify if your Cron Job is executing well or not. To do that, perform the following steps:

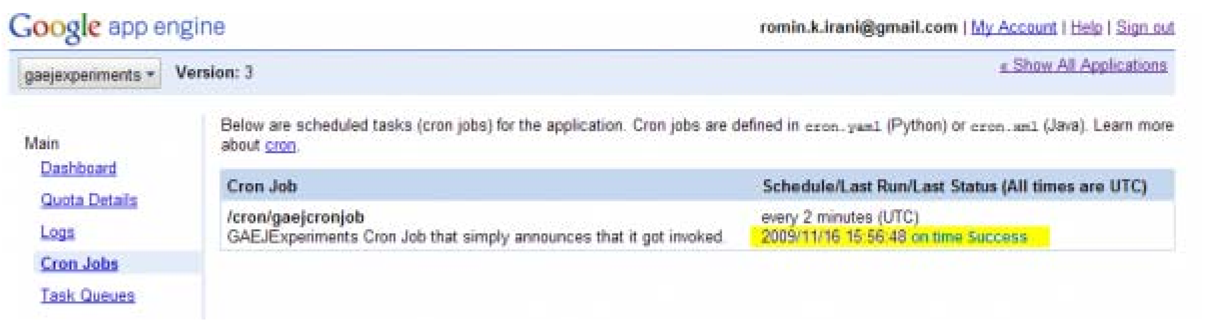
1. Go to http://appengine.google.com and log in with your account.
2. You will see a list of applications registered. Click on the application that you just deployed. In my case, it is **gaejexperiments.**
3. When you click on a particular application, you will be taken to the **Dashboard** for that application, which contains a wealth of information around the requests, quotas, logs, versions, etc.
4. Verify that the Cron Jobs that you specified in the **cron.xml** have been configured successfully for the application by clicking **Cron Jobs,** visible under **Main.** For our application that we deployed, here is the screen shot from the App Engine console:



Below are scheduled tasks (cron jobs) for the application. Cron jobs are defined in cron.yaml (Python) or cron.xml (Java). Learn more about cron.

Cron Job	Schedule/Last Run/Last Status (All times are UTC)
/cron/gaejcronjob GAEJExperiments Cron Job that simply announces that it got invoked.	every 2 minutes (UTC) Hasn't run yet

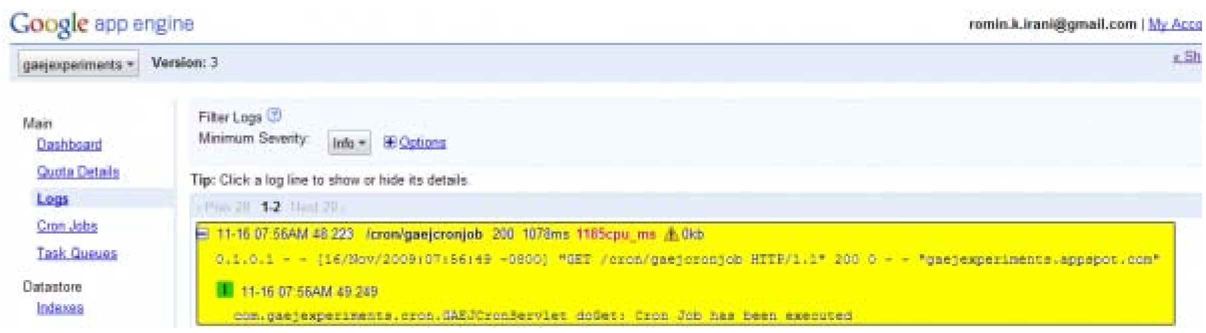
You will notice that the Cron Job has not yet run as the console indicates. Every time that the job is executed, this column is updated with the last date time stamp that the Job executed along with its status. Since we have configured our Job to run every 2 minutes, I waited for 2 minutes and then the job executed itself and when I refreshed the Cron Jobs page, the status was updated as shown below:



Below are scheduled tasks (cron jobs) for the application. Cron jobs are defined in cron.yaml (Python) or cron.xml (Java). Learn more about cron.

Cron Job	Schedule/Last Run/Last Status (All times are UTC)
/cron/gaejcronjob GAEJExperiments Cron Job that simply announces that it got invoked.	every 2 minutes (UTC) 2009/11/16 15:56:48 on time Success

You can also click on the **Logs** link. This will display the application log. And all your application log statements that you code using the Logger class can be visible here. By default, the severity level is set at ERROR and we can change that to INFO and you should be able your log statements that had the log level of INFO. This was the log level at which we had logged the statement in our Java Servlet (Cron Job). Shown below is a screen shot of the log when the Cron Job was fired once.



Minimum Severity: Info

Tip: Click a log line to show or hide its details.

```
11-16 07:56AM 48.223 /cron/gaejcronjob 200 1078ms 1185cpu_ms 0kb
  0.1.0.1 - - [16/Nov/2009:07:56:49 -0800] "GET /cron/gaejcronjob HTTP/1.1" 200 0 - - "gaejexperiments.appspot.com"
  I 11-16 07:56AM 49.249
    com.gaejexperiments.cron.GAEJCronServlet doGet: Cron Job has been executed
```

Conclusion

This concludes Episode 9 of this series in which you learn how to schedule tasks in your Google App Engine applications. These background tasks that can be scheduled at a certain time and which are executed by the Cron Service are an indispensable part of several web applications that are deployed today. If you ever wish to do repeated tasks in your application without any user intervention like sending emails, crawling web sites, taking database backups, etc, then writing a Cron Job and scheduling it for execution is a key feature that you can utilize while deploying your application to Google App Engine.

There is a lot more to Cron Jobs and I suggest to read up the documentation.

Episode 10: Using the Task Queue Service

Welcome to **Episode 10**. In this episode, we shall cover the experimental **Task Queue Service** in Google App Engine. This is an experimental service, which means that it could undergo change in its core functionality from all respects like methods, package names, etc. In any case, we can safely expect that it shall get confirmed in some form or the other in a future release of the Google App Engine.

This episode is sort of a natural progression on our last episode, which covered the **Cron Service**. To reiterate, the Cron Service was used for background jobs that you wish to perform outside of a user request. They are not typically initiated by a user but instead configured by your application to perform certain periodic tasks like summary reports at end of the day, daily backups, etc. The **Task Queue Service** provides a facility to process tasks in the background which are :

- Typically initiated by the user.
- They can be created internally by the application also to break down a background job into smaller tasks.

Some of the examples of performing a Task in the background include:

1. A user signs up at the website and submits his/her information. On receiving the request, a simple message is displayed back to the user that his/her subscription request has been received and they will hear soon about the status. At the same time, a **task** can be created that can process this users request, perform some verification and then send out an email to the user if the subscription is setup successfully. So you could have done either of the following:

- Receive the user details and create a Task. This single task will take care of creating the required records in the database and send out an email.
- Receive the user details and create a Task. This task will simply create a record in the database and then create another task to send out an email.
- And so on.

2. An online shopping site could accept the order from a buyer. A task is then created in the system that will process the order. As part of the order processing, once shipping is done, another task is told to update the shipping status. Similarly when the logistics provider gives updates on the shipment locations, a task could be launched to update the shipment route of the package.

To summarize, any background (asynchronous) task can be launched when a user initiates the event or when an application or business event occurs. All these tasks are put into one or more user defined or default queues and are executed by the Google App Engine infrastructure on behalf of your application.

What does a Task constitute? What is a Queue ? Who executes it ?

The official documentation of **Tasks** is excellent and I suggest reading that in detail. In this episode I will cover just about enough for you to get started on Tasks and then dig deeper into them depending on your needs. So let us first understand what is the basic information that I need let the Google App Engine know about my task. I will take the liberty here to describe all the key concepts via an example that we will build in this application.

We wish to implement the following flow in our application:

1. A user wishes to sign up for your newsletter by providing an email id in a web form provided at our site.
2. The user enters the email id and clicks on sign up (a button).
3. The request is sent to a Servlet that accepts the email id and **creates a Task.** The response is sent back to the user thanking them for their interest.
4. The Task is then executed independently by the Google App Engine infrastructure and our code inside the Task i.e. checking if the email id is used, etc is verified and then an email is sent.

So, as you can see we have decoupled the background task (in step 4) from the sign up process (step 1, step 2 & step3).

It should now be straightforward to define the key elements:

1. **Task** : This is the unit of work that we wish to perform. In fact, the actor that will be performing this task is Google App Engine. So when we create a Task, we need to provide a standard way for the Google App Engine to **invoke tasks** and pass them their **payload**. It should now be straightforward to see that when we create a task, all we need to tell GAEJ is the URL (where to invoke the task) and the parameterized payload (data). This can be done in a standard fashion that you know. The URL is nothing but the servlet URL that will invoke the servlet that implements the Task. And the parameterized data is nothing but request parameters passed into your servlet so that it can execute accordingly. The data in this case will be nothing but the email id of the user who wants to sign up for your newsletter.

2. **Queue** : All Tasks when they are created are placed in a queue. They are then executed by the Google App Engine. To help us manage and categorize our tasks, you can define your queues by giving them appropriate names. For e.g. myqueue, emailqueue, etc. What this helps you to do is to place your tasks in the appropriate queue. Few points to note about queues (refer to the **documentation** for finer details):

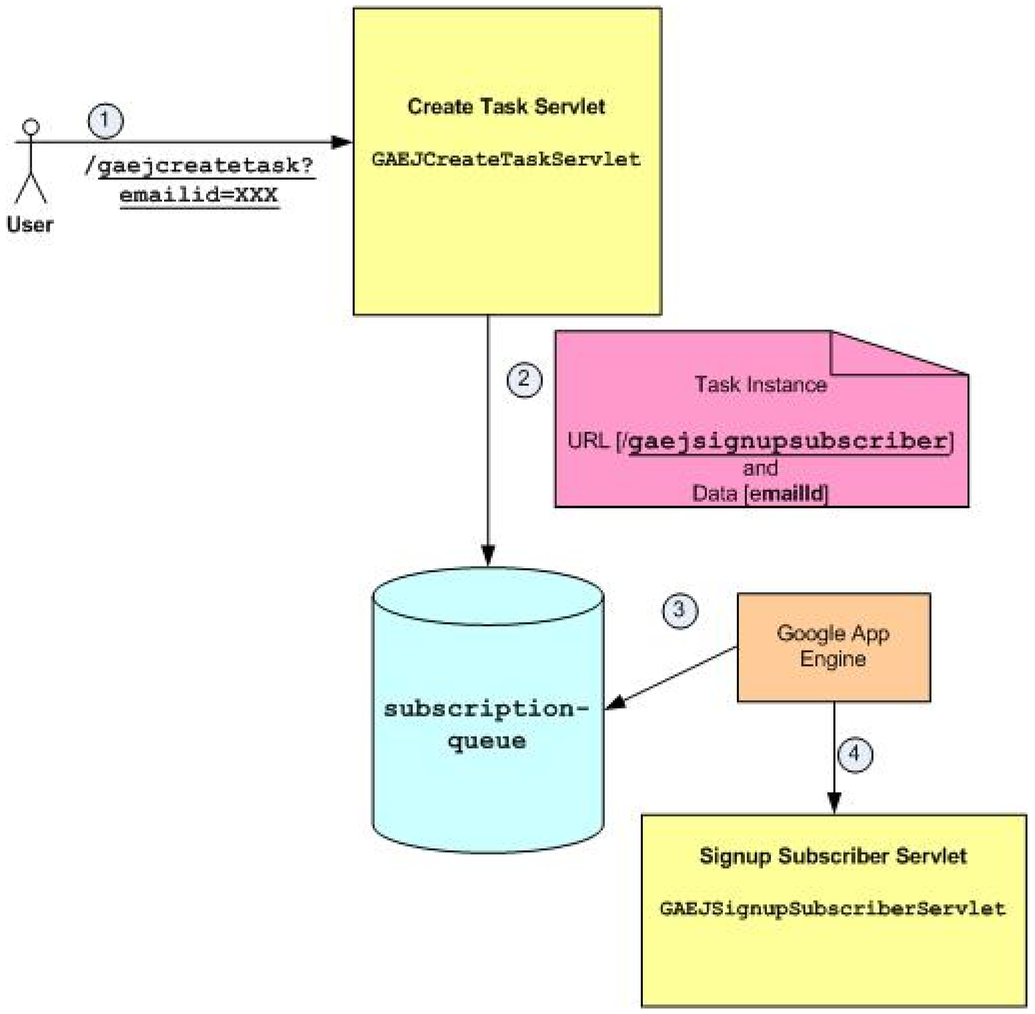
- All queues in your application are defined in a file named **queue.xml** that is present in the WEB-INF folder of your application.

- Each queue has a unique name and you can control the rate at which tasks are executed in this queue by the Google App Engine. If you do not specify a rate, then the default rate is 5 tasks/second.
- There is a default queue for your application named **'default'** and if you can chose to add your tasks to the default queue. Alternately, you can add them to your application defined queue.

I believe this should be sufficient for us to begin developing a simple flow that will demonstrate how to define a task queue, create a task and then see it getting executed by the Google App Engine.

Task Queue in Action

The diagram below shows what we will implement in the next few sections. The use case is that of an user that wishes to subscribe to our newsletter and how we shall break up the process into the request processing and then the task processing.


1
/gaejcreatetask?emailid=XXX
User
Create Task Servlet
GAEJCreateTaskServlet
2
Task Instance
URL [/gaejsignupsubscriber] and Data [emailId]
subscription-queue
3
Google App Engine
4
Signup Subscriber Servlet
GAEJSignupSubscriberServlet

Let us break down the above flow into the steps given below and what we shall be developing at each step:

1. In **Step 1**, the user visits a web page and enters his/her email id to sign up for the newsletter. We shall not be developing a web page over here to keep things simple. Instead we shall be testing it out by directly invoking a servlet that accepts the email id of the user. This is the same thing that you would have normally done by hooking up the action of the HTML form to that of this servlet.

2. In **Step 2,** we will be looking at a **Servlet** whose existence is to do some verification and then **create a Task for background processing**. This servlet (**GAEJCreateTaskServlet**) will create a Task in a queue called **subscription-queue.** As we covered earlier, to create a Task, we need to provide two pieces of information to the Google App Engine so that it can execute it. They are the URL and the Data. The URL (**/gaejsignupsubscriber**) is going to be that of a Servlet (**GAEJSignupSubscriberServlet**) that shall be doing the core task. And the data will be what the servlet needs to complete the task. In our case, the data is the emailid request parameter.

3. In **Step 3,** Google App Engine automatically scans the queues for any tasks that are queued up and picks them up for execution. In our case, it will find a Task instance and execute it by invoking the URL (**/gaejsignupsubscriber**) and passing it the relevant data i.e. **emailid**

4. Finally in **Step 4,** our Servlet (**GAEJSignupSubscriberServlet)** is invoked and it will complete its task. To keep things simple in our example, it will currently only print out a message. But it should be obvious that core logic associated with the task would have gone into the Servlet here. For our specific case, it would have involved checking if the user has not signed up already, creating a database record and then sending off a welcome email.

Implementing the above flow

To summarize the above steps in terms of what we have to code, here is the list:

1. Code the **GAEJCreateTaskServlet** that will accept the request parameter and create a Task instance in the subscription-queue.

2. Code the **GAEJSignupSubscriberServlet** that will be invoked by Google App Engine automatically. We will currently only print out a log statement because the intent is to demonstrate the whole sequence.

3. Configure our queue **(subscription-queue)** in a file named **queue.xml**. This file needs to be placed in the WEB-INF folder of your application.

4. Configure our **GAEJCreateTaskServlet** and **GAEJSignupSubscriberServlet** in the **web.xml** file.

Finally, we can execute our application and use the local development server to see the application in action. Users can optionally even deploy it to the Google App Engine cloud if they wish.

So let us get started.

GAEJCreateTaskServlet.java

This servlet accepts our request for subscription. We shall invoke it via the following url : **http://appurl/gaejcreatetask?emailid=XYZ**.

```
package com.gaejexperiments.taskqueue;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.*;

import com.google.appengine.api.labs.taskqueue.Queue;
import com.google.appengine.api.labs.taskqueue.QueueFactory;
import com.google.appengine.api.labs.taskqueue.TaskOptions;

@SuppressWarnings("serial")
public class GAEJCreateTaskServlet extends HttpServlet {
 public void doGet(HttpServletRequest req, HttpServletResponse resp)
 throws IOException {

 String strCallResult = "";
 resp.setContentType("text/plain");
 try {
 //Extract out the To, Subject and Body of the Email to be sent
 String strEmailId = req.getParameter("emailid");

 //Do validations here. Only basic ones i.e. cannot be null/empty

 if (strEmailId == null) throw new Exception("Email Id field cannot be
empty.");

 //Trim the stuff
 strEmailId = strEmailId.trim();
 if (strEmailId.length() == 0) throw new Exception("Email Id field cannot
be empty.");
 //Queue queue = QueueFactory.getDefaultQueue();
 Queue queue = QueueFactory.getQueue("subscription-queue");
 queue.add(TaskOptions.Builder.url("/gaejsignupsubscriber").param("emai
lid",strEmailId));
 strCallResult = "Successfully created a Task in the Queue";
 resp.getWriter().println(strCallResult);
 }
 catch (Exception ex) {
 strCallResult = "Fail: " + ex.getMessage();
 resp.getWriter().println(strCallResult);
 }
 }

 @Override
 public void doPost(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException, IOException {
 doGet(req, resp);
 }
}
```

The code listed below is straightforward to understand. It does the following:

1. Extracts out the request parameter (**emailid)** and does some basic validation on it.

2. It gets a handle to the **subscription-queue** through the following statement:

```
Queue queue = QueueFactory.getQueue("subscription-queue");
```

3. It adds a Task to the above queue by providing a Task URL (**/gaejsignupsubscriber**) and Data (**emailid parameter**). It uses a helper class **TaskOptions.Builder** to help create the instance of the task. As you can see it provides a url and then the param. The task is created by invoking the add method on the queue.

```
queue.add(TaskOptions.Builder.url("/gaejsignupsubscriber").param("email
id",strEmailId));
```

4. For the readers information, I have shown a commented out line

```
//Queue queue = QueueFactory.getDefaultQueue();
```

which shows how to get the handle to the default queue in case you wish to place your tasks in the default queue itself.

5. Do not that all the Task Queue classes are experimental and are present in the **com.google.appengine.api.labs.taskqueue** package. This could change in the future.

GAEJSignupSubscriberServlet.java

This servlet contains the core task logic. This will be invoked by Google App engine if it finds any tasks present in the **subscription-queue**. If any tasks are there, it will pick them up and invoke the URL mentioned in the Task and pass to it the data present in the Task instance. The code shown below is straightforward, it simply logs a statement saying that it got invoked and it also logs the email id.

```
package com.gaejexperiments.taskqueue;

import java.io.IOException;
import java.util.logging.Logger;

import javax.servlet.ServletException;
import javax.servlet.http.*;

@SuppressWarnings("serial")
public class GAEJSignupSubscriberServlet extends HttpServlet {
 private static final Logger _logger =
Logger.getLogger(GAEJSignupSubscriberServlet.class.getName());
 public void doGet(HttpServletRequest req, HttpServletResponse resp)
 throws IOException {

 String strCallResult = "";
 resp.setContentType("text/plain");
 try {
 String strEmailId = req.getParameter("emailid");
 _logger.info("Got a Signup Subscriber Request for Email ID : " +
strEmailId);
 //
 // PUT YOUR TASK CODE HERE
 //
 strCallResult = "SUCCESS: Subscriber Signup";
 _logger.info(strCallResult);
 resp.getWriter().println(strCallResult);
 }
 catch (Exception ex) {
 strCallResult = "FAIL: Subscriber Signup : " + ex.getMessage();
 _logger.info(strCallResult);
 resp.getWriter().println(strCallResult);
 }
 }

 @Override
 public void doPost(HttpServletRequest req, HttpServletResponse resp)
 throws ServletException, IOException {
 doGet(req, resp);
 }

}
```

queue.xml

All queues are configured in a file named **queue.xml**. Google App Engine provides a default queue. This queue is aptly named “**default**“. But in case you need to define your own queues, which is what we are going to do, we need to define them in a file called **queue.xml**. This file is placed in the **WEB-INF** directory of your application. You can also override settings of the **default** queue by defining it in the file and providing your own values.

Take a look at the queue.xml shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<queue-entries>
 <queue>
 <name>default</name>
 <rate>5/s</rate>
 </queue>
 <queue>
 <name>subscription-queue</name>
 <rate>5/s</rate>
 </queue>
</queue-entries>
```

In the above configuration, you will find that we have defined our own queue named "**subscription-queue**". There is also another element that we have defined for the **<queue>** called **<rate>**. This element determines the rate at which you tell Google App Engine to execute tasks. If you do not specify a rate, then the default execution rate is 5 tasks per second. In the above file, we have provided the expression as **"5/s"**, which reads as 5 per second. Other examples of **<rate>** expressions are **1000/d** (One thousand per day), etc. I suggest to read up the documentation for more examples.

You will also find that we have defined the **default** queue and we can change the rate if we want. But I have left it as is.

Please make sure that the above file (queue.xml) is present in the WEB-INF folder at the time of deploying the application.

Configuring the Servlets (web.xml)

We need to add the **<servlet/>** and **<servlet-mapping/>** entry to the **web.xml** file. This file is present in the **WEB-INF** folder of the project. The necessary fragment to be added to your **web.xml** file are shown below. Please note that you can use your own namespace and servlet class. Just modify it accordingly if you do so. We are defining here both our servlets.

```
<servlet>
 <servlet-name>GAEJCreateTaskServlet</servlet-name>
 <servlet-
class>com.gaejexperiments.taskqueue.GAEJCreateTaskServlet</servlet-
class>
 </servlet>
 <servlet>
 <servlet-name>GAEJSignupSubscriberServlet</servlet-name>
 <servlet-
class>com.gaejexperiments.taskqueue.GAEJSignupSubscriberServlet</servle
t-class>
 </servlet>
 <servlet-mapping>
 <servlet-name>GAEJSignupSubscriberServlet</servlet-name>
```

```
<url-pattern>/gaejsignupsubscriber</url-pattern>
</servlet-mapping>
<servlet-mapping>
<servlet-name>GAEJCreateTaskServlet</servlet-name>
<url-pattern>/gaejcreatetask</url-pattern>
</servlet-mapping>
```

Task Execution in Action

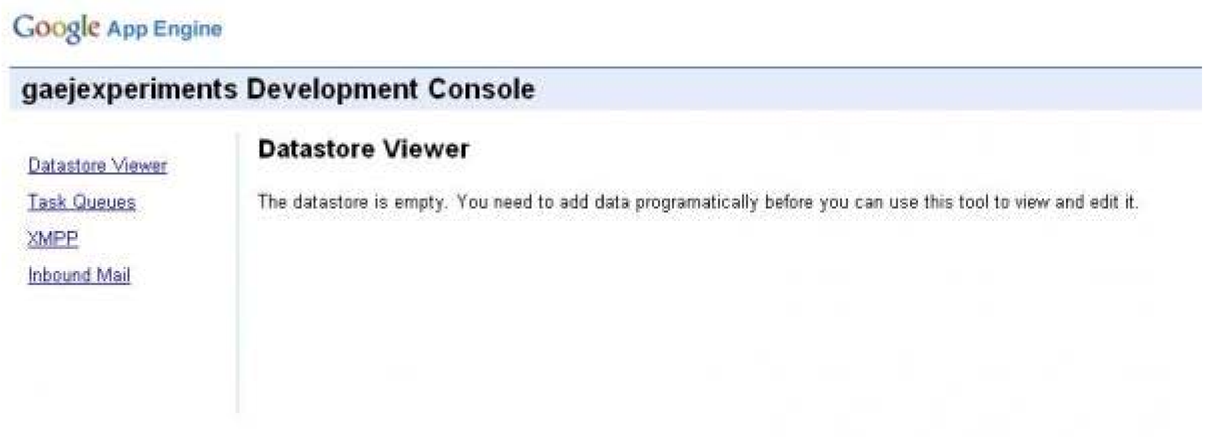
I am assuming that you have already created a new Google Web Application Project and have created the above Servlets, web.xml and queue.xml respectively. For a change, we shall be running this episode within our local development server only.

So assuming that all is well, we will run our application, by right-clicking on the project and selecting **Run As –> Web Application.** Once you see the following message shown below, then the local server is ready to accept requests.

```
Nov 24, 2009 5:11:33 AM com.google.apphosting.utils.jetty.JettyLogger
info
INFO: jetty-6.1.x
Nov 24, 2009 5:11:40 AM com.google.apphosting.utils.jetty.JettyLogger
info
INFO: Started SelectChannelConnector@127.0.0.1:8080
The server is running at http://localhost:8080/
```

Follow the steps given below:

1. Launch the browser on your local machine and navigate to **http://localhost:8080/_ah/admin**. This is the administrative console of the local development server.



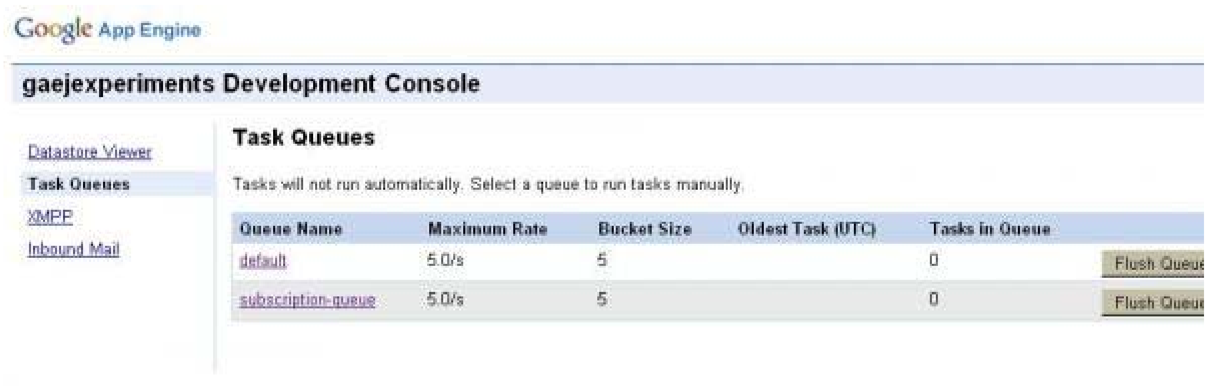
gaejexperiments Development Console

Datastore Viewer

The datastore is empty. You need to add data programatically before you can use this tool to view and edit it.

2. Click on **Task Queues** to view the current task queues that are configured in your application. You should see the screen shown below, which shows that we have configured two task queues: **default** and **subscription-queue.**

You will notice that both of the queues currently do not have any tasks since we have not created any.



Task Queues

Tasks will not run automatically. Select a queue to run tasks manually.

Queue Name	Maximum Rate	Bucket Size	Oldest Task (UTC)	Tasks in Queue	
default	5.0/s	5		0	Flush Queue
subscription-queue	5.0/s	5		0	Flush Queue

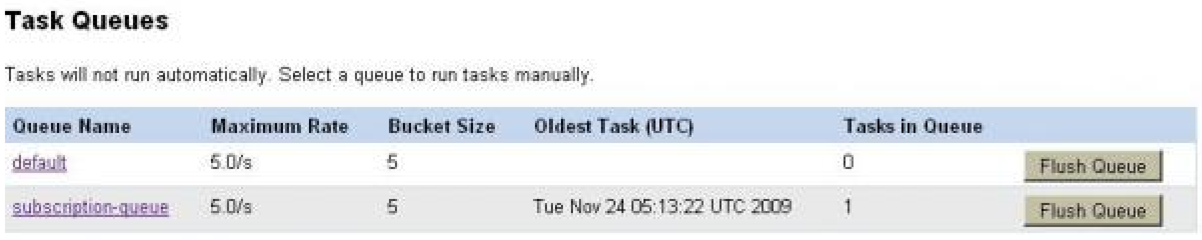
3. The next step is to create a task in the **subscription-queue**. To do that, all we need to do is invoke the following url :

http://localhost:8080/gaejcreatetask?emailid=romin@rocketmail.com

This invokes our GAEJCreateTaskServlet that we have configured. It will create the sample task in the **subscription-queue** and we will see a message as shown below in the browser:

"Successfully created a Task in the Queue"

4. Click on the Task Queues link again, you will find that there will now be **1 task** listed in the **subscription-queue** as shown below:

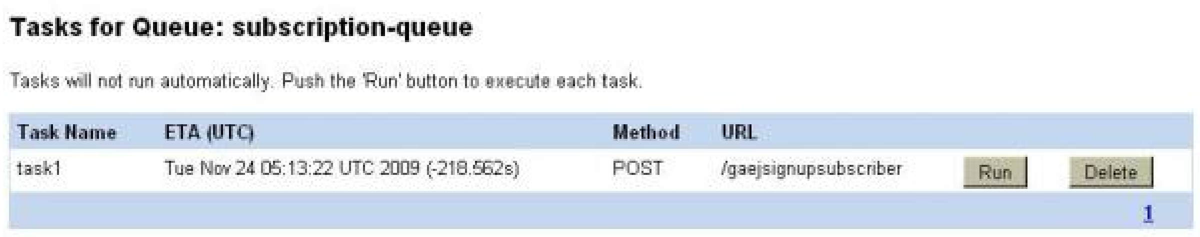


Task Queues

Tasks will not run automatically. Select a queue to run tasks manually.

Queue Name	Maximum Rate	Bucket Size	Oldest Task (UTC)	Tasks in Queue	
default	5.0/s	5		0	Flush Queue
subscription-queue	5.0/s	5	Tue Nov 24 05:13:22 UTC 2009	1	Flush Queue

5. Click on the **subscription-queue** link. This will display the task instance as shown below:

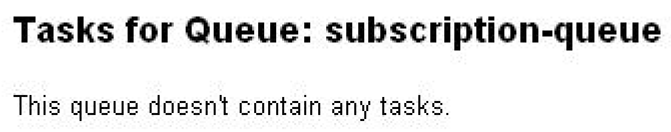


Tasks for Queue: subscription-queue

Tasks will not run automatically. Push the 'Run' button to execute each task.

Task Name	ETA (UTC)	Method	URL		
task1	Tue Nov 24 05:13:22 UTC 2009 (-218.562s)	POST	/gaejsignupsubscriber	Run	Delete

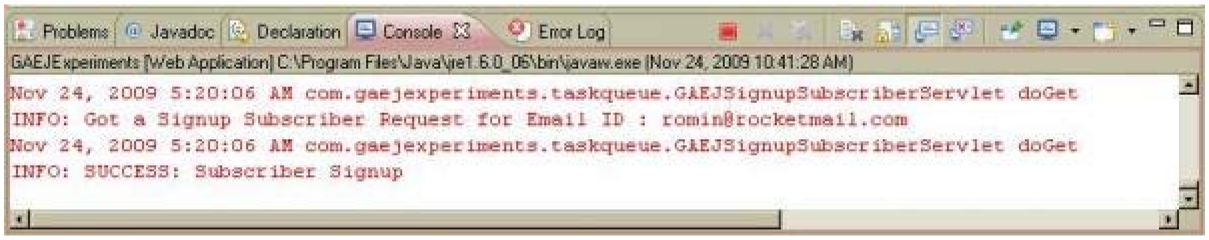
6. Since this is the development server, no automatic execution of tasks (this will occur in Google App Engine cloud) will take place and you have to run them manually, by clicking on the **Run** button. Click that. It will display that there are no more tasks present if the task executes successfully.



Tasks for Queue: subscription-queue

This queue doesn't contain any tasks.

7. To check that our task has executed successfully, we can visit the Console tab in Eclipse and you will see the success log as shown below:



```
GAEJExperiments [Web Application] C:\Program Files\Java\jre1.6.0_06\bin\javaw.exe (Nov 24, 2009 10:41:28 AM)
Nov 24, 2009 5:20:06 AM com.gaejexperiments.taskqueue.GAEJSignupSubscriberServlet doGet
INFO: Got a Signup Subscriber Request for Email ID : romin@rocketmail.com
Nov 24, 2009 5:20:06 AM com.gaejexperiments.taskqueue.GAEJSignupSubscriberServlet doGet
INFO: SUCCESS: Subscriber Signup
```

Moving on

In this episode, we saw how to split up a process into individual tasks and assign them to the Google App Engine for execution. What we have demonstrated here is a simple flow and the specifics of the configuration. I encourage you to try it out in your applications and as an exercise deploy it to the Google App Engine and monitor it there via the Administration Console.

Do keep in mind that this API is experimental and is likely to change drastically. At the same time, if you have any feedback, it would be nice to pass it along to the Google App Engine team.

Episode 11: Develop Simple Google Wave Robots using the WadRobotFramework

Welcome to Episode 11 of this series. We saw in earlier Episode # 7, how you can write a Google Wave Robot. Please go through that tutorial to get familiar with the basics. This episode will be focused on writing a Google Wave Robot using a framework called the WadRobotFramework. This framework aims to help you develop your Robot quickly so that you can focus on the logic of your Robot, while letting the framework do the heavy lifting. Sure, you can do your programming with the Google Wave Robot API as explained in Episode 7, but this framework as we shall see wraps the details nicely so that you get to focus on your Robot logic.

What is the WadRobotFramework?

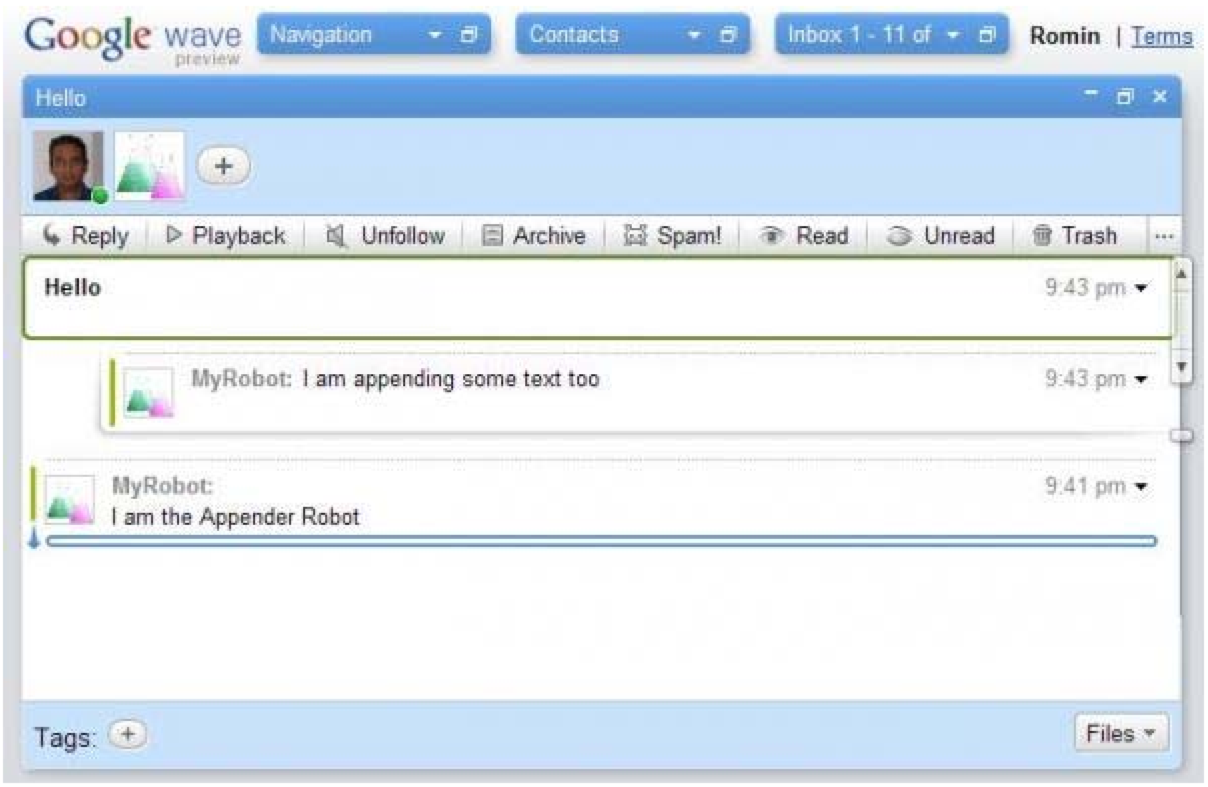
The **WadRobotFramework** is available at **http://code.google.com/p/wadrobotframework.** This framework is the work of Jerome BATON, who has stated that the goal of the framework is to make writing Google Wave Robots in Java much simpler.

The framework goes ahead and defines two kinds of Robots: **Simple** and **Advanced**.

Simple Robots

Simple Robots are typically those that **can react to a new Blip**. This kind of Robots simply wait till a Blip is submitted. And on submission of the Blip, they will append a message to the Blip. It could be as simple as just analyzing the text of the blip and appending some new information or simply a fun sort of a Robot, which simply prints a random message.

An example of a this is shown below and which we will build in this episode:



Hello 9:43 pm

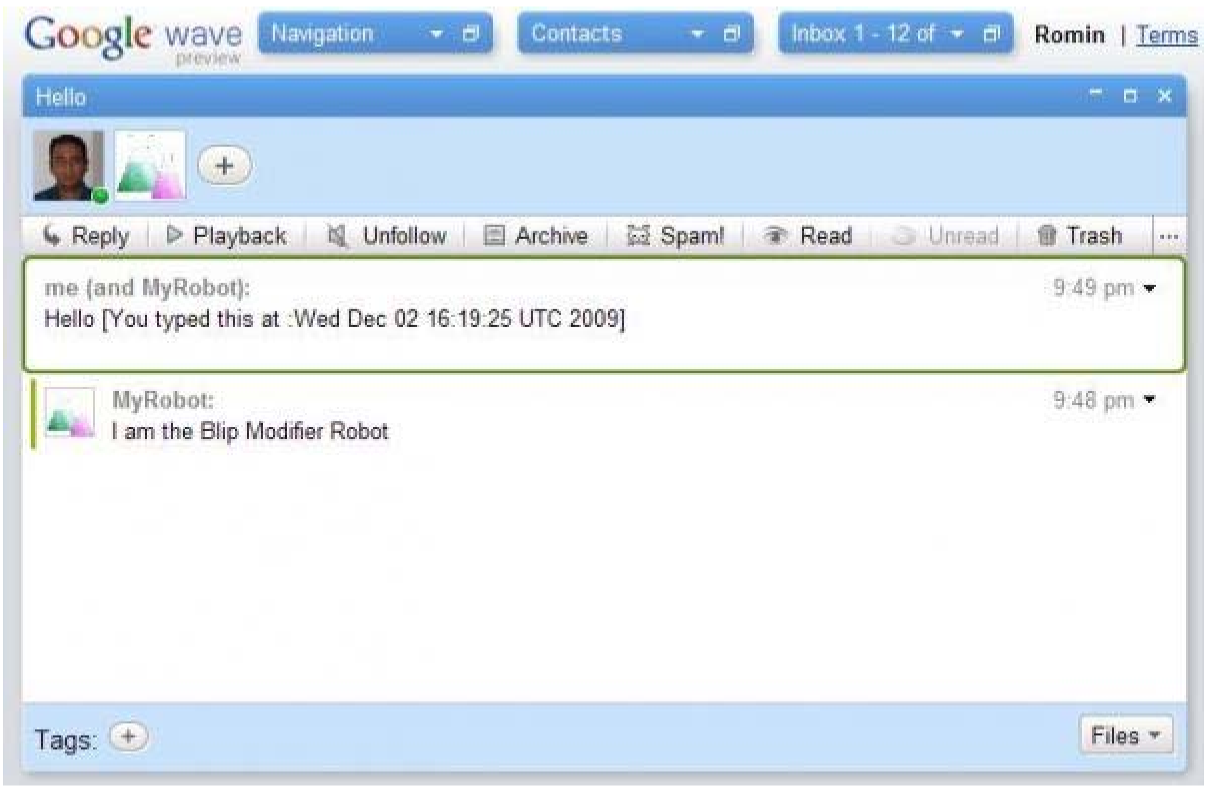
MyRobot: I am appending some text too 9:43 pm

MyRobot:
I am the Appender Robot 9:41 pm

In the above wave, there are two participants (myself and the Robot). The Robot announces itself when it is added to the Wave. Additionally, whenever I submit a Blip (for e.g. Hello), it simply appends a new blip with some text.

Simple Robots are also those **that can modify the Blip Text** instead of appending a new blip. For e.g. They could read the blip text and filter out the profanities, do automated spell checks or simply add some information to the existing Blip Text itself. In other words, it modifies the Blip Text.

Shown below is an example of a Simple Robot that modifies the Blip Text:



me (and MyRobot): 9:49 pm
Hello [You typed this at :Wed Dec 02 16:19:25 UTC 2009]

MyRobot: 9:48 pm
I am the Blip Modifier Robot

In the above wave, there are two participants (myself and the Robot). The Robot announces itself when it is added to the Wave. Additionally, whenever I submit a Blip (for e.g. Hello), it simply modifies the Blip Text by mentioning when I submitted the Blip text.

There is also a 3rd thing that the Simple Robots could do, which is very useful too. They can act in a **stealth** manner and simply do their work quietly. For e.g. whether you wish to write a Robot that appends a Blip or modifies a submitted Blip, you can simply chose not to append or modify but instead get the Blip text and do your work quietly. For e.g. count the words in the background, etc.

The WadRobotFramework also supports Advanced Robots, which are particularly interesting and will be covered in the Next Episode.

Advanced Robots are those that can react to commands in the Blips. Here are some samples of Advanced Robots and how they would react to commands from Blips:

1. A character Counting Advanced Robot:

Your submitted Blip Text contains : “Here is some text in the blip. Please count the length of this message for me. {mycountingrobot:count}”.

You can write an advanced Robot (mycountingrobot) that knows how to count the length of the message. So it gets notified when it there is a command **(count)** in the Blip. Your Advanced Robot can then count out the characters and then either append or modify a blip as needed.

2. A Tweeting Robot:

Your submitted Blip Text contains the following text : "{mytweetingrobot:tweet}

You can write an advanced Robot (mytweetingrobot) that knows how to tweet the message to Twitter. So it gets notified when it there is a command **(tweet**) in the Blip. Your Advanced Robot can then append a blip or modify a blip saying that the message has been tweeted.

The best part of it all is that you could combine all of this into a single Robot that can respond to one or more commands. For example, take a look at *Today's Special* Robot (see http://ppandhi.wordpress.com/2009/11/08/todays-special-robot/) that can respond to more than one command. It can give you the quotes, day in history, word of the day, cricket score, your daily horoscope by simply responding to the command that you type in. With the WadRobotFramework, we shall see in the next episode how you can write such an Advanced Robot that responds to different commands. If you cannot wait, take a look at the documentation at the site (Advanced Robots).

Since this episode will be focusing on Simple Robots, let us recap what we know so far: The WadRobotFramework supports Simple Robots in two flavours: The **BlipAppenderRobot** and the **BlipModifierRobot**. The **BlipAppenderRobot** is used to **simply append to a Blip**. The **BlipModifierRobot** is used to **modify a submitted Blip**.

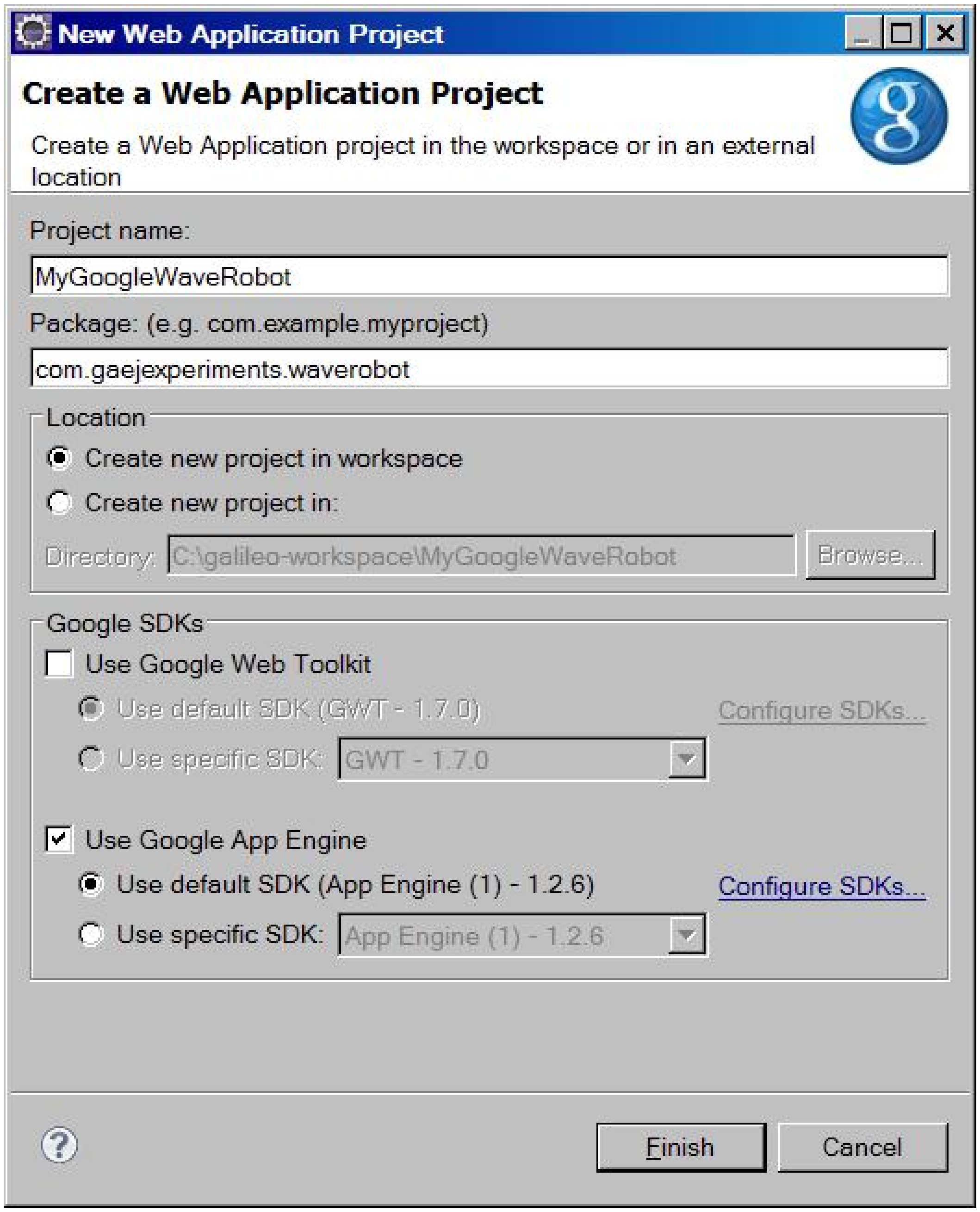
Simple, isnt it? Lets develop and deploy our Simple Robots using the WadRobotFramework.

Create a New Project

We need to create a New Project first. Follow the steps below:

1. Either click on **File –> New –> Other** or press **Ctrl-N** to create a new project. Select Google and then Web Application project. Alternately you could also click on the New Web Application Project Toolbar icon as part of the Google Eclipse plugin.

2. In the New Web Application Project dialog, **deselect** the **Use Google Web Toolkit** and give a name to your project. I have named mine MyGoogleWaveRobot and I suggest you go with the same name so that things are consistent with the rest of the article (but if you wish to name it something else, that is fine too). The Screenshot is shown below:


New Web Application Project

Create a Web Application Project

Create a Web Application project in the workspace or in an external location

Project name: MyGoogleWaveRobot

Package: (e.g. com.example.myproject) com.gaejexperiments.waverobot

Location

[✓ Create new project in workspace]

Create new project in:

Directory: C:\galileo-workspace\MyGoogleWaveRobot

Google SDKs

Use Google Web Toolkit

[✓ Use default SDK (GWT - 1.7.0)]

Use specific SDK: GWT - 1.7.0

[✓ Use Google App Engine]

[✓ Use default SDK (App Engine (1) - 1.2.6)]

Use specific SDK: App Engine (1) - 1.2.6

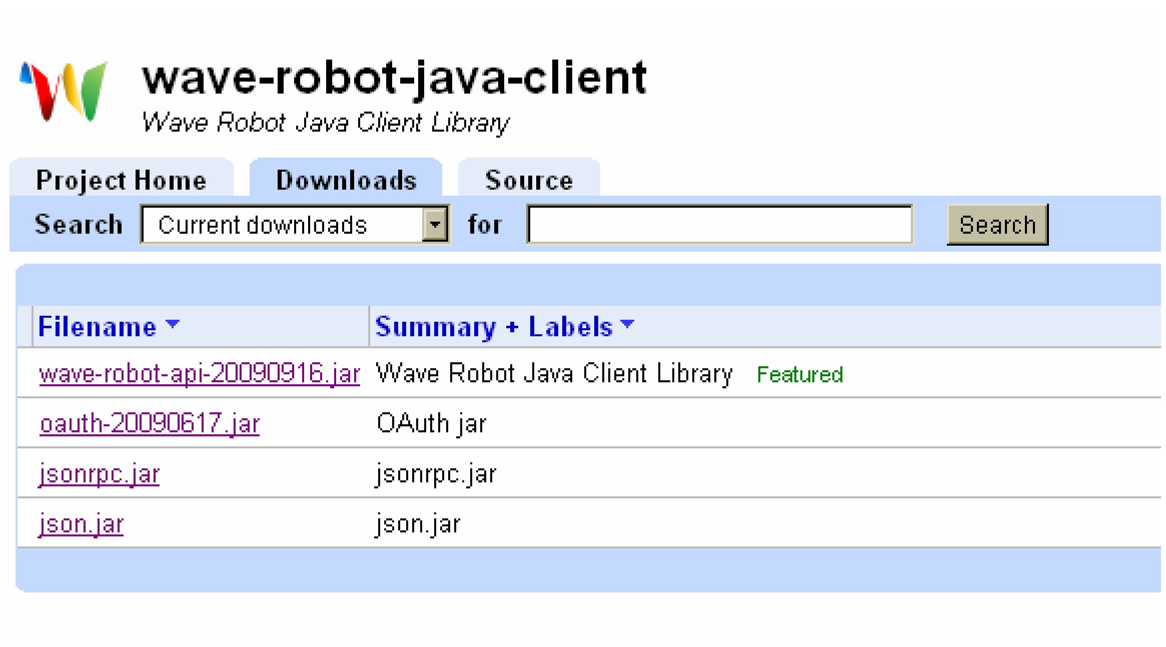
3. Click on Finish. This will generate the project and also create a sample Hello World Servlet for you. But we will be writing our own Servlet. **So I suggest that you can delete the Servlet Java class and the mappings made in the web.xml or you can leave it for now since we are going to write our own.**

Adding Google Wave Robot JAR files and WadRobotFramework JAR file to your Project Path

Since we are going to be writing a Wave Robot, we need some additional files on the client side. These additional files (JAR files) are required for the additional Wave API's and also for deployment in your **WEB-INF\lib** folder, so that they are correctly deployed and available to the run-time engine. These JAR files do not ship along with the Google Eclipse plugin, so you will need to download them for a website. The Google code website for the JAR files is:

http://code.google.com/p/wave-robot-java-client/downloads/list

The web page when you navigate to the above URL is shown below:



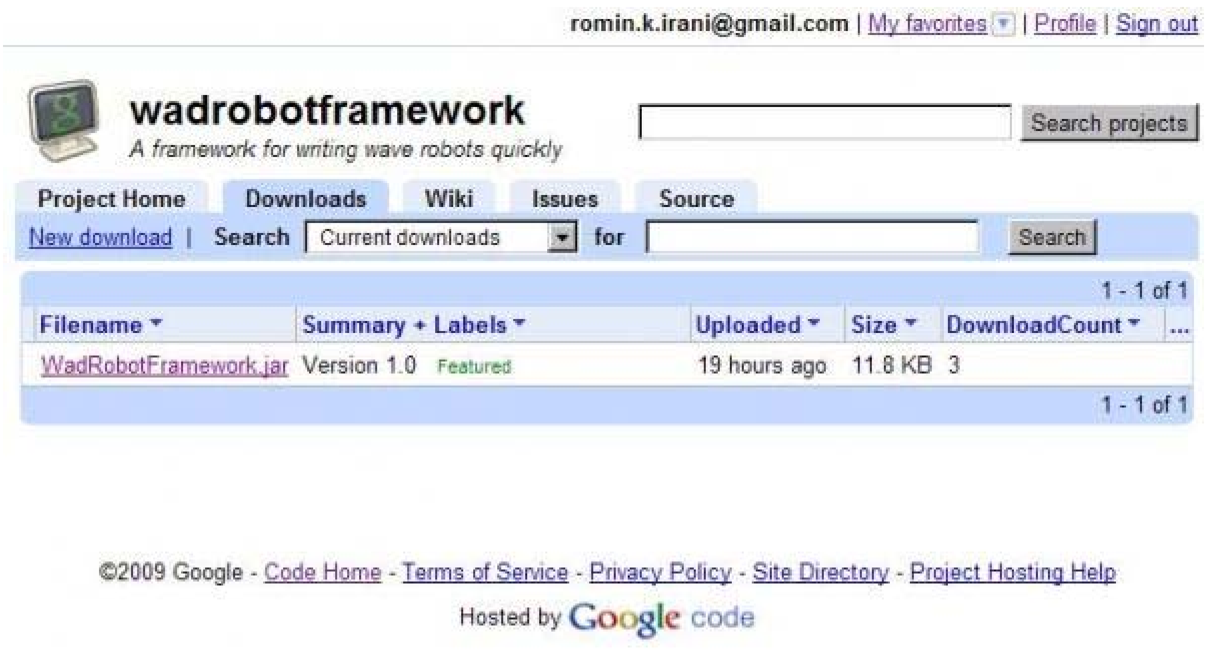
Filename	Summary + Labels
wave-robot-api-20090916.jar	Wave Robot Java Client Library Featured
oauth-20090617.jar	OAuth jar
jsonrpc.jar	jsonrpc.jar
json.jar	json.jar

Download all the above files to your machine.

The **WadRobotFramework JAR file** is available at the following location :

http://code.google.com/p/wadrobotframework/downloads/list

The web page when you navigate to the above URL is shown below:


romin.k.irani@gmail.com | My favorites | Profile | Sign out

wadrobotframework
A framework for writing wave robots quickly

1 - 1 of 1

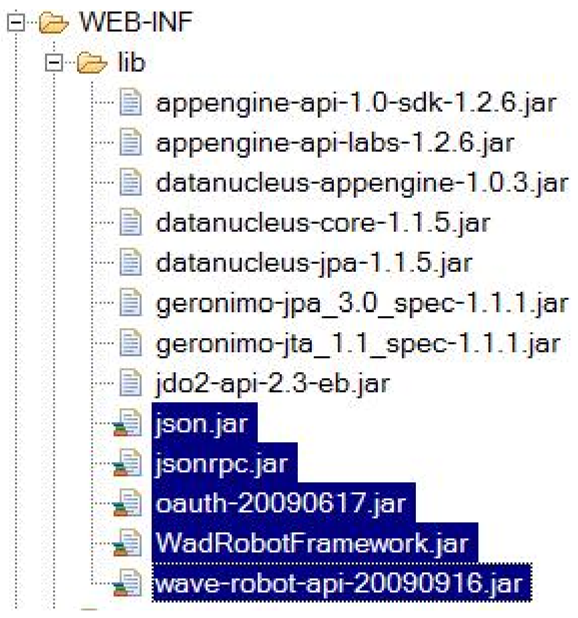
Filename	Summary + Labels	Uploaded	Size	DownloadCount	...
WadRobotFramework.jar	Version 1.0 Featured	19 hours ago	11.8 KB	3	

1 - 1 of 1

©2009 Google - Code Home - Terms of Service - Privacy Policy - Site Directory - Project Hosting Help
Hosted by Google code

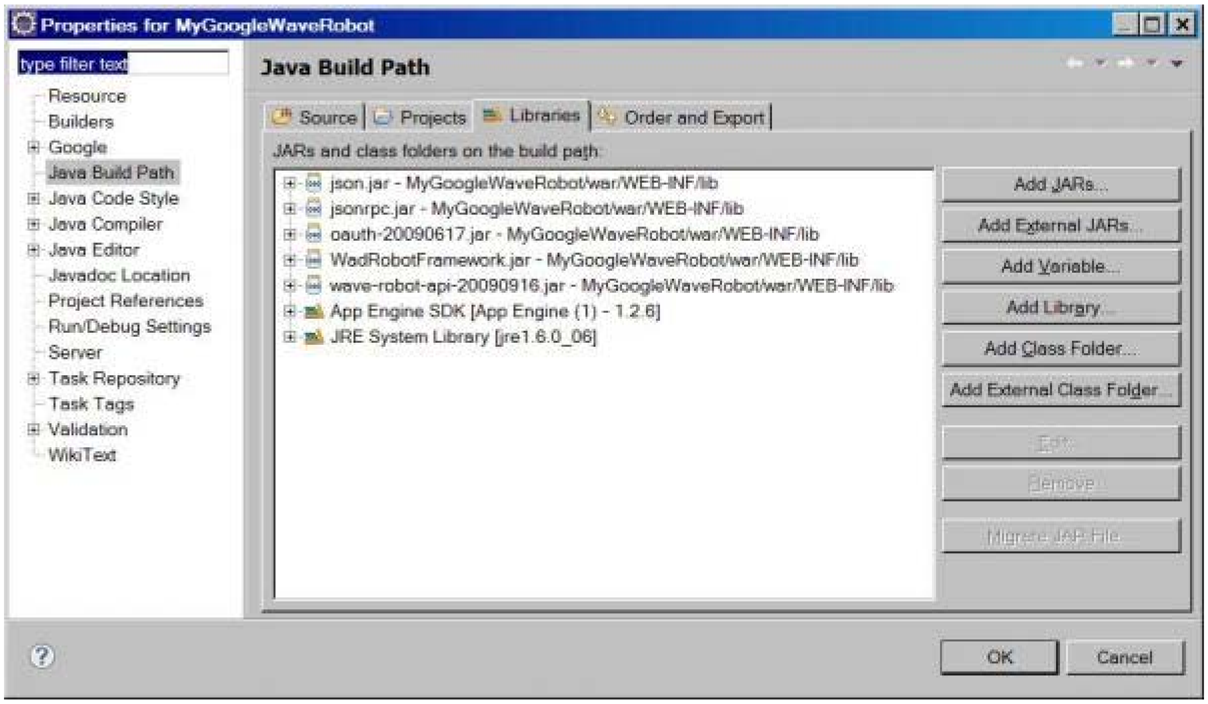
Once you have downloaded the files, follow these steps to setup your Project Build Path and runtime correctly.

1. Copy all the **5 JAR** files to the **WEB-INF\lib** folder of your Eclipse Project. After copying you should see the files as shown in the project hierarchy below:


```
WEB-INF
  lib
    appengine-api-1.0-sdk-1.2.6.jar
    appengine-api-labs-1.2.6.jar
    datanucleus-appengine-1.0.3.jar
    datanucleus-core-1.1.5.jar
    datanucleus-jpa-1.1.5.jar
    geronimo-jpa_3.0_spec-1.1.1.jar
    geronimo-jta_1.1_spec-1.1.1.jar
    jdo2-api-2.3-eb.jar
    json.jar
    jsonrpc.jar
    oauth-20090617.jar
    WadRobotFramework.jar
    wave-robot-api-20090916.jar
```

2. Right-click on the Project in the Project Hierarchy. Select **Properties** and then **Java Build Path.** Click on **Add JARs** and then select the 5 JAR files from your Project **WEB-INF\lib** folder.

3. Your Project Build Path should like the screenshot below.



JARs and class folders on the build path:

- json.jar - MyGoogleWaveRobot/war/WEB-INF/lib
- jsonrpc.jar - MyGoogleWaveRobot/war/WEB-INF/lib
- oauth-20090617.jar - MyGoogleWaveRobot/war/WEB-INF/lib
- WadRobotFramework.jar - MyGoogleWaveRobot/war/WEB-INF/lib
- wave-robot-api-20090916.jar - MyGoogleWaveRobot/war/WEB-INF/lib
- App Engine SDK [App Engine (1) - 1.2.6]
- JRE System Library [jre1.6.0_06]

Click on OK to proceed. This completes your Build Path setup with the **Google Wave Robot JAR** and **WadRobotFramework JAR** files.

Writing the Simple Robot: MyAppenderRobot : MyAppenderRobot.java

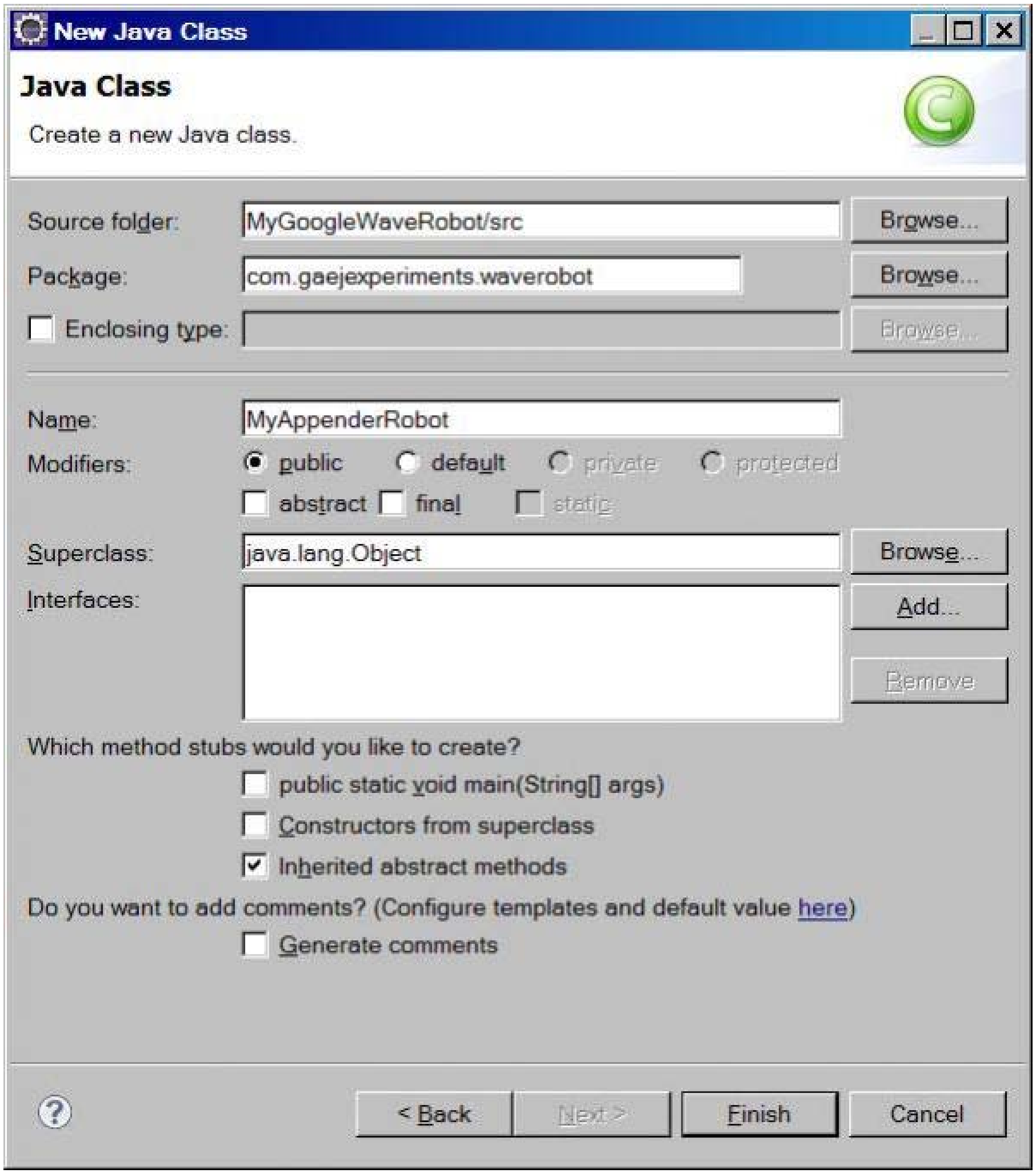
Let us first create our Simple Robot Java class based on the WadRobotFramework. This Simple Robot is the one that will react to a new Blip. So all we will do is to make the robot append to the blip, when the blip is submitted. The Robot demonstrated here will be straightforward and you can easily modify it to make it react in your own way.

As discussed, this is known as the BlipAppenderRobot and so all we need to do is to extend the BlipAppenderRobot class in the WadRobotFramework and provide our simple implementation.

The steps are straightforward and given below. All we need to do is write our class that extends the **org.wadael.waverobotfrmwrk.simple.BlipAppenderRobot** class and provide an implementation for the **getTextToAppend** method.

Follow these steps:

1. Create a new Java class within the same package. The New Java Class dialog is shown below. I have named the class **MyAppenderRobot** as shown below. Click on the **Browse** button to select a Super Class.


New Java Class

Java Class

Create a new Java class.

Source folder: MyGoogleWaveRobot/src

Package: com.gaejexperiments.waverobot

☐ Enclosing type:

Name: MyAppenderRobot

Modifiers: ● public ○ default ○ private ○ protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

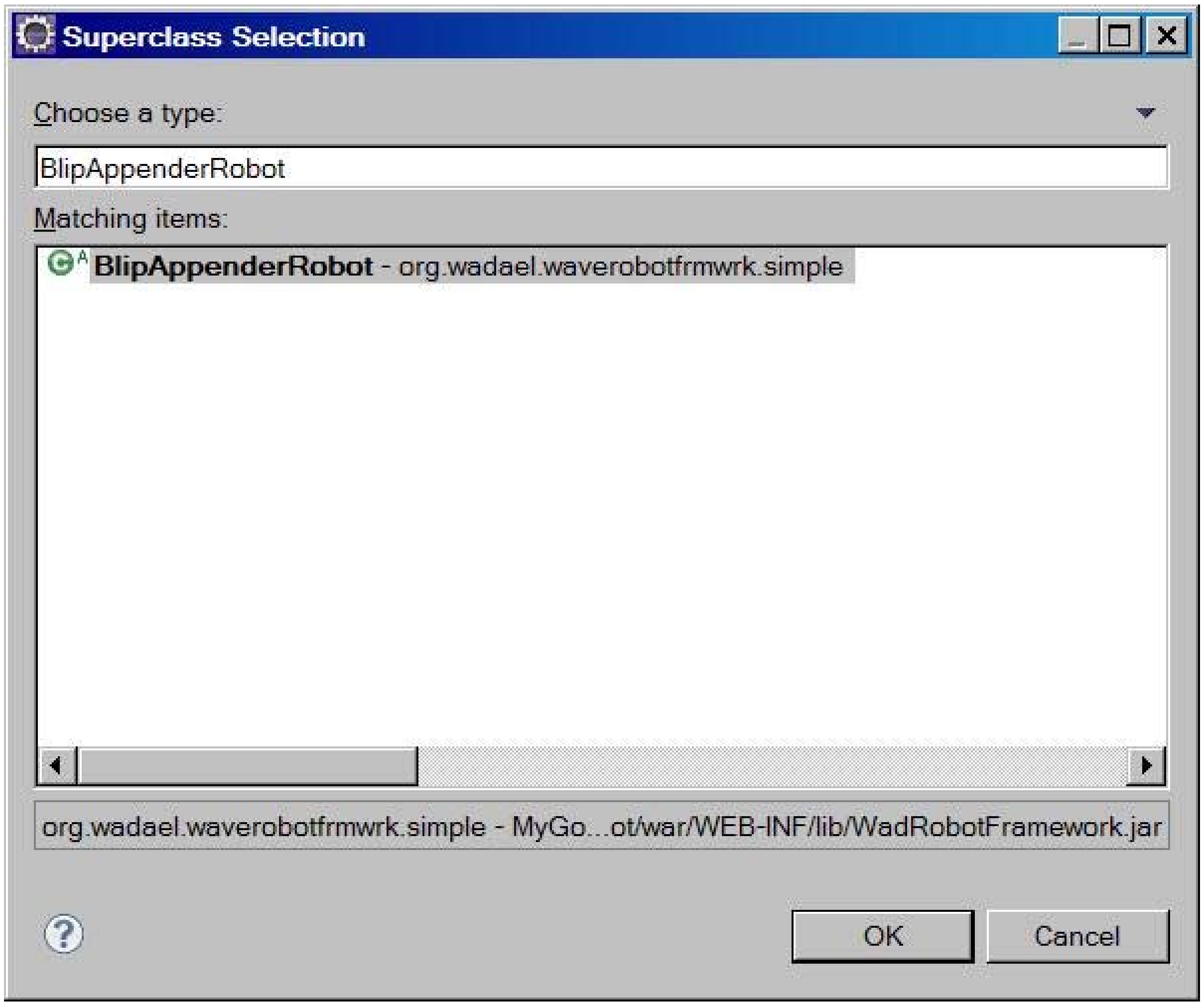
☐ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

2. In the **Superclass Selection** dialog shown below, type the word BlipAppenderRobot (some part of it is also OK as the screenshot shows below) in the **Choose a type** field as shown. This will bring up the correct **Matching item**s i.e. **org.wadael.waverobotfrmwrk.simple.BlipAppenderRobot**. Click on **OK.**



BlipAppenderRobot

BlipAppenderRobot - org.wadael.waverobotfrmwrk.simple

org.wadael.waverobotfrmwrk.simple - MyGo...ot/war/WEB-INF/lib/WadRobotFramework.jar

This will generate the code and you simply replace it with the following code listing given below:

```
package com.gaejexperiments.waverobot;

import org.wadael.waverobotfrmwrk.simple.BlipAppenderRobot;

public class MyAppenderRobot extends BlipAppenderRobot {

@Override
protected String getTextToAppend(String msg) {
```

```
return "I am appending some text too";
}

@Override
protected String getRobotSelfIntroduction() {
return "I am the Appender Robot";
}

}
```

Let us go through the code now:

1. We have extended the BlipAppenderRobot since all we want to do in this Robot is to react to the blip and append our own Blip

2. All we need to do as part of extending the BlipAppenderRobot class is to implement the **getTextToAppend** method. This method gets passed one parameter msg that is the blip text that was submitted. So in all essence, you could inspect what the text was in the blip submitted and then react to it. The implementation simply returns the string that it wants to append. The WadRobotFramework's **BlipAppenderRobot** will take care of creating a Blip and appending it to the current blip.

3. We also implement an optional method named **getRobotSelfIntroduction**. This method returns a string and you can return something that identifies the addition of your Robot to the Wave. So when your Robot is added as a participant and if you have implemented the **getRobotSelfIntroduction** method, then it will display this message out. It is sort of announcing to the other wave participants about your presence.

That is all there is to implemented the MyAppenderRobot. If you would have studied the earlier tutorial on writing a Google Wave Robot, you would have noticed that the WadRobotFramework has done away with all the Event processing that you had to handle yourself and also shielded you from methods that you need to know to create a blip and append to it.

Configuring the MyAppenderRobot in web.xml

We need to add the **MyAppenderRobot in the <servlet/>** and **<servlet-mapping/>** entry to the **web.xml** file. This file is present in the **WEB-INF** folder of the project. The necessary fragment to be added to your **web.xml** file are shown below.

```
<servlet>
<servlet-name>MyAppenderRobot</servlet-name>
<servlet-class>com.gaejexperiments.waverobot.MyAppenderRobot</servlet-
class>
</servlet>
<servlet-mapping>
<servlet-name>MyAppenderRobot</servlet-name>
<url-pattern>/_wave/robot/jsonrpc</url-pattern>
```

```
</servlet-mapping>
```

In the above fragment, you will note that url-pattern **/_wave/robot/jsonrpc** has to be mapped to the Robot Servlet that you have written. This is because the Google Wave system will invoke this url to communicate with your Robot using its protocol.

Creating the MyAppenderRobot capabilities.xml files

We need an additional file to describe the capabilities of the Robot that we have written. This file is called the **capabilities.xml** and it needs to reside in a certain location. You need to create a **_wave** directory inside of the **war** directory of your project. The location of this file is **war/_wave** directory.

You will need to create the **_wave** directory and create the **capabilities.xml** file over there. The capabilities file shown below is pretty straightforward and is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<w:robot xmlns:w="http://wave.google.com/extensions/robots/1.0">
  <w:capabilities>
    <w:capability name="BLIP_SUBMITTED" content="true" />
  </w:capabilities>
  <w:version>1</w:version>
</w:robot>
```

There are two elements of interest over here.

The **<capability>** element is of particular interest and we have registered for the BLIP_SUBMITTED event. This is important because it informs Google Wave that whenever a BLIP is submitted by a participated, then our Robot needs to be notified of it. The WadRobotFramework will notify the Robot by invoking the **getTextToAppend** method.

The other element is the **<version>** element. If you change any capabilities in your robot, then it is recommended that before you deploy, you change the version value over here, so that Google Wave can detect that there is a newer version and hence it can then query for your modified capabilities if any.

Writing our MyAppenderRobot Profile Servlet (not required but nice)

This is not a required step but it would be good practice to do so to make your Robot look more professional. A Profile Servlet is used to tell the following about your Robot:

1. A Name for your Robot

2. A custom image for your Robot

3. A profile page for your Robot (a URL)

If you provide these, then the Google Wave client is able to retrieve them and set it for your Robot when it is added as a participant. This makes the Robot look more professional.

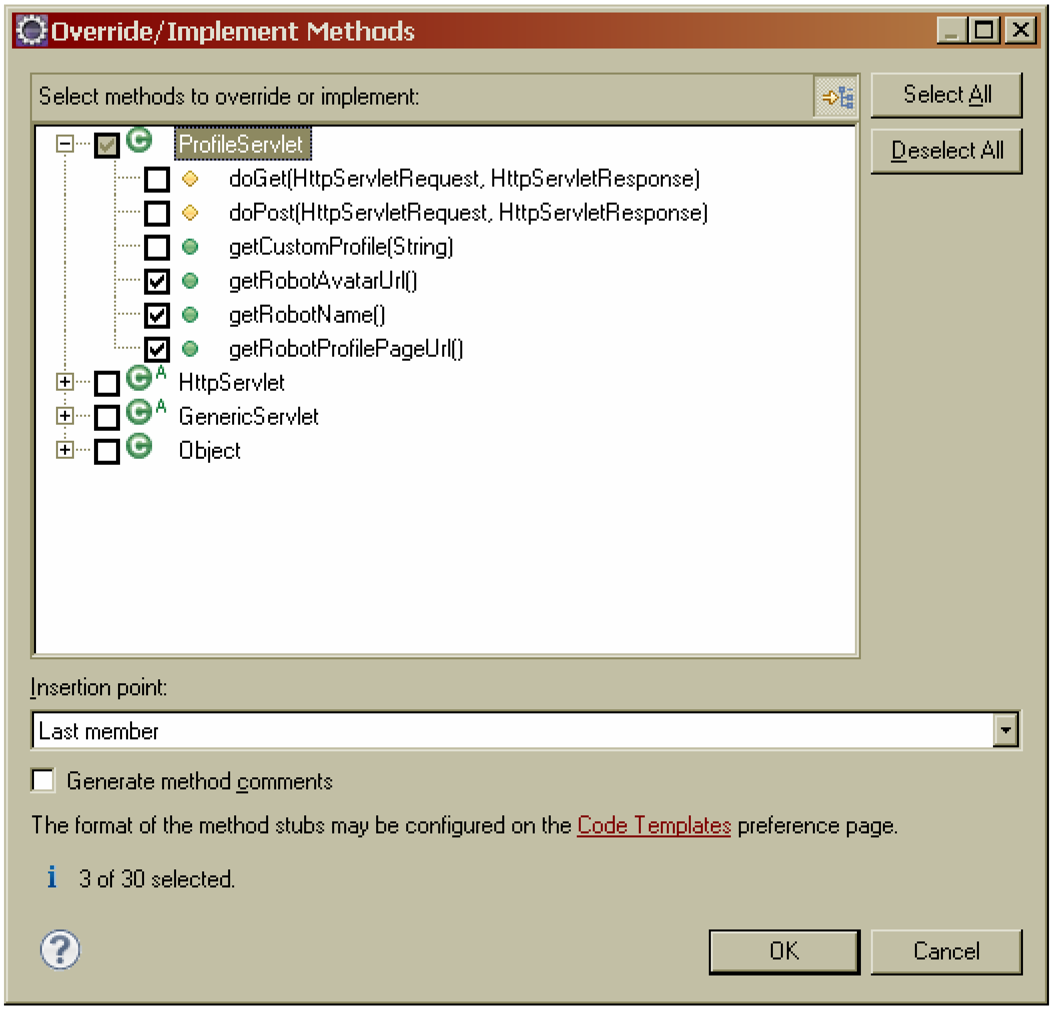
This profile information needs to be provided by you by writing a Profile Servlet. The Profile Servlet is nothing but extending the **com.google.wave.api.ProfileServlet** class and providing simple implementations for the overwritten methods.

Follow these steps to write the Profile Servlet:

1. Create a new Java class within the same package. Name the classs **MyRobotProfileServlet** and mention the Superclass as **com.google.wave.api.ProfileServlet** class. Click on **OK**.

2. This will generate a **MyRobotProfileServlet.java file.**

The simplest way to generate the stubs for the required methods would be to go to **Source –> Override/Implement** Methods. This will bring up the dialog box as shown below and you only need to select the 3 methods to override as shown:



The format of the method stubs may be configured on the Code Templates preference page.

3 of 30 selected.

Click on **OK**. This will generate the stubs, which you can then overwrite with the code shown below. The code is easy to understand, all we are doing is providing values for the Name, Avatar(Image) and the Profile Page URL. **Note that for the Avatar, we are providing a file myimage.jpg present in the WAR/_wave folder. You will need to copy an appropriate image file for yourself and make sure that it is physically copied to the folder locally in your Eclipse project before you deploy your application.**

I have used my AppId as myinfoagent but you can replace it with your AppId.

```
package com.gaejexperiments.waverobot;

import com.google.wave.api.ProfileServlet;

public class MyRobotProfileServlet extends ProfileServlet {

@Override
public String getRobotAvatarUrl() {
return "http://myinfoagent.appspot.com/_wave/myimage.jpg";
}

@Override
public String getRobotName() {
return "MyRobot";
}

@Override
public String getRobotProfilePageUrl() {
return "http://myinfoagent.appspot.com";
}

}
```

Configuring the Profile Servlet

We need to add the **Profile Servlet <servlet/>** and **<servlet-mapping/>** entry to the **web.xml** file. This file is present in the **WEB-INF** folder of the project. The necessary fragment to be added to your **web.xml** file are shown below.

```
<servlet>
<servlet-name>MyRobotProfileServlet</servlet-name>
<servlet-
class>com.gaejexperiments.waverobot.MyRobotProfileServlet</servlet-
class>
</servlet>
<servlet-mapping>
<servlet-name>MyRobotProfileServlet</servlet-name>
<url-pattern>/_wave/robot/profile</url-pattern>
</servlet-mapping>
```

In the above fragment, you will note that url-pattern **/_wave/robot/profile** has to be mapped to the Profile Servlet that you have written. This is because the Google Wave system will invoke this url to get hold of your profile.

Deploying the Application

To deploy the application, you will need to first create your **Application ID**. The Application Identifier can be created by logging in at **http://appengine.google.com** with your Google Account. You will see a list of application identifiers already registered under your account (or none if you are just getting started). To create a new Application, click on the Create Application button and provide the Application Identifier as requested. Please note this name down since you will be using it for deployment.

For e.g. I have registered an application identifier named **myinfoagent**.

To deploy the application, follow these steps (they should be familiar to you now):

1. Click on the Deploy Icon in the Toolbar.
2. In the Deploy dialog, provide your Email and Password. Do not click on Deploy button yet.
3. Click on the **App Engine Project settings** link. This will lead you to a dialog, where you need to enter your Application ID [For e.g. my Application Identifier **myinfoagent**]
4. Click on **OK**. You will be lead back to the previous screen, where you can click on the **Deploy** button. This will start deploying your application to the GAEJ cloud. You should see several messages in the Console window as the application is being deployed.
5. Finally, you should see the message "Deployment completed successfully".

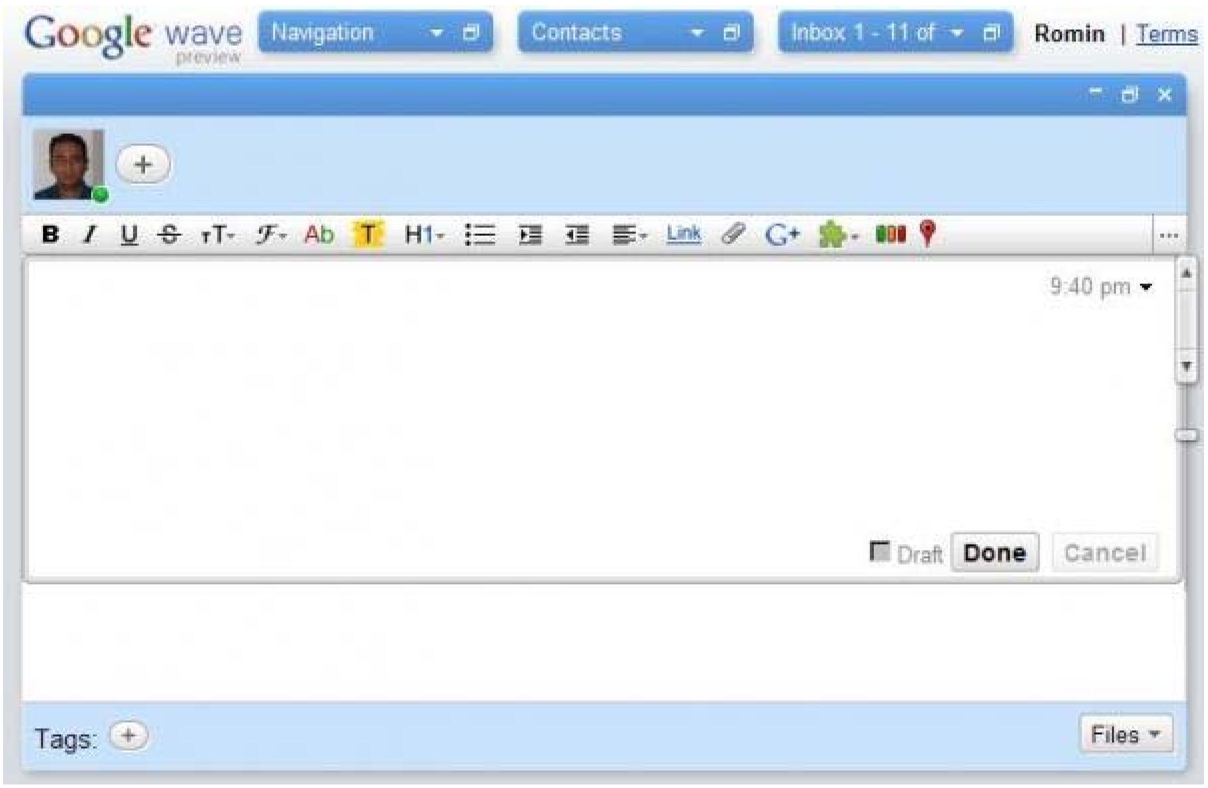
MyAppenderRobot in Action

Your application is going to be available at the http://**yourapplicationid**.appspot.com. In my case, the application is available at **http://myinfoagent.appspot.com**.

You can test for the presence of your robot capabilities file by simply typing in the following:

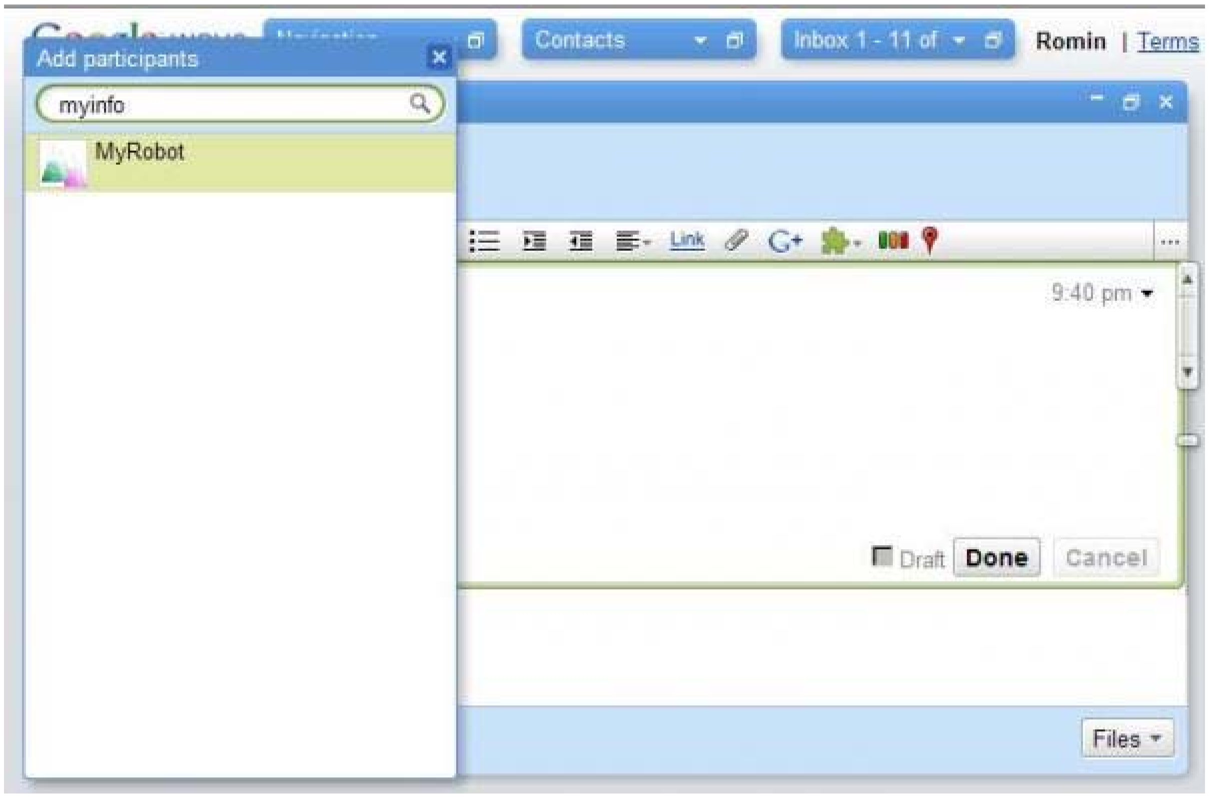
http://**yourapplicationid.**appspot.com**/_wave/capabilities.xml** [Replace yourapplicationid with the Application ID that you have] and you should see the capabilities.xml file being served up.

To test out the Robot, you need to launch the Google Wave client and login in with your account by going to http://wave.google.com. On successful login, you will be inside the Wave client from where you can create a new wave by clicking on the **New Wave** link. When you do that, currently you are the only participant (myself) as shown in the screen below:

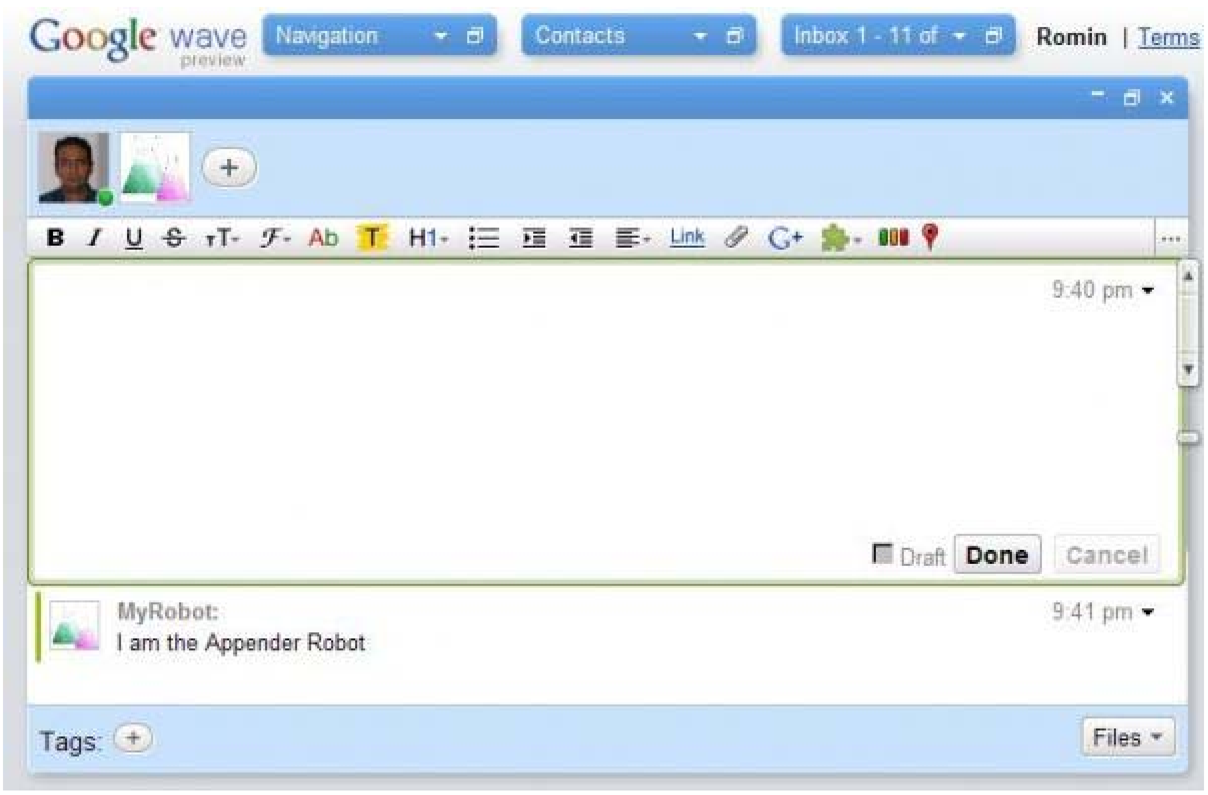


Click on the **+ sign** next to your icon and you can add one or more participants as shown below:

NOTE : Your Google Wave Robot is going to be available at <YOURAPPLICATIONID>@appspot.com , hence I have added **myinfoagent@appspot.com** as that was my application id. But you can replace it with your application id.

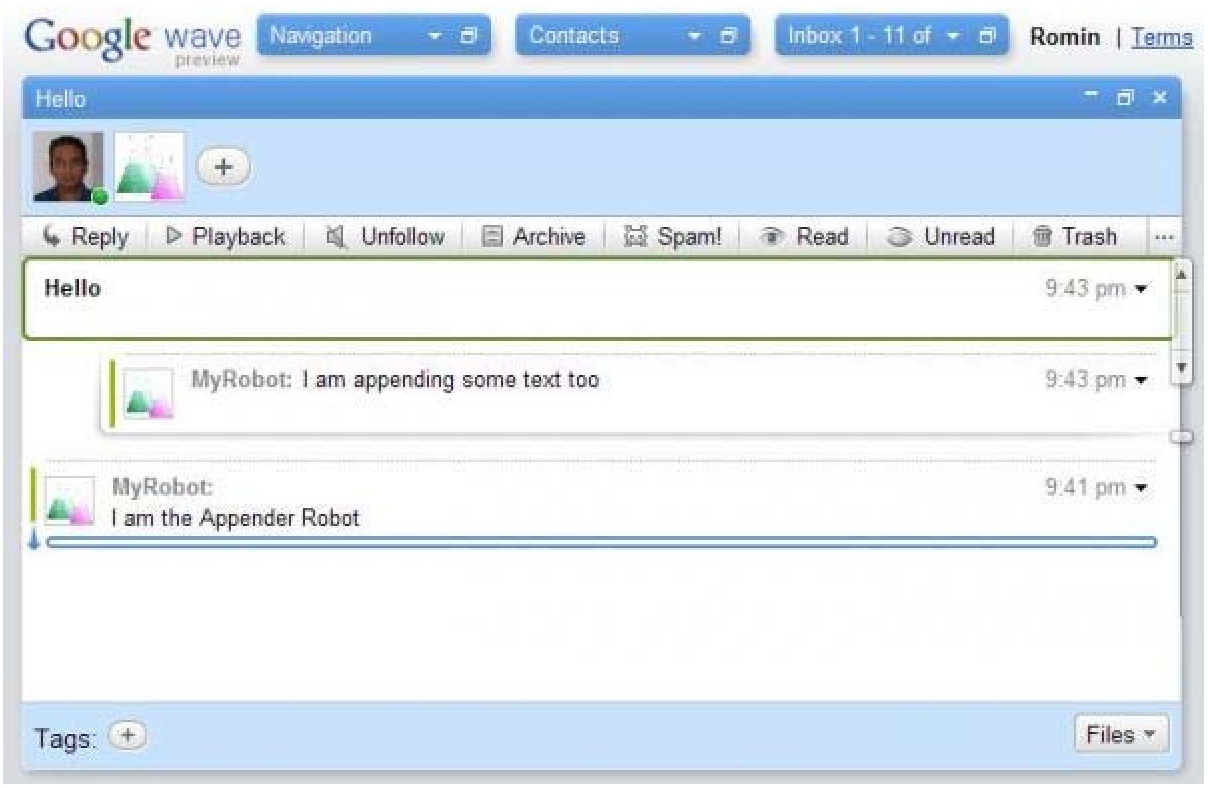


If all goes well, you will see your Robot added as a participant (with the icon and all, since the Profile Servlet is invoked behind the scenes by the Google Wave system). It announces itself and this was done via the implementation that we provided in the **getRobotSelfIntroduction** method.



MyRobot:
I am the Appender Robot

Now I type a message “Hello” as shown below and click on the **Done** button, the BLIP_SUBMITTED event is fired and our Robot gets the event i.e. the **getTextToAppend** method is invoked on our Robot Java class. The method simply appends a blip as shown below:



Hello 9:43 pm

MyRobot: I am appending some text too 9:43 pm

MyRobot:
I am the Appender Robot 9:41 pm

Writing another Simple Robot: MyBlipModifierRobot : MyBlipModifierRobot.java

We saw how easy it was to create our Simple Robot that simply appended a blip to the wave conversation when a blip was submitted. Let us now write another Simple Robot that will modify the Blip Text when the blip is submitted.

As discussed, this is known as the **BlipModifierRobot** and so all we need to do is to extend the **BlipModifierRobot** class in the **WadRobotFramework** and provide our simple implementation.

The steps are straightforward and given below. All we need to do is write our class that extends the **org.wadael.waverobotfrmwrk.simple.BlipModifierRobot** class and provide an implementation for the **modifyBlipText** method.

Follow these steps:

1. Create a new Java class within the same package. Name it **MyBlipModifierAppenderRobot** as shown below and mention the superclass as **org.wadael.waverobotfrmwrk.simple.BlipModifierRobot**

This will create the MyBlipModifierRobot.java file and you can replace it with the following source:

```
package com.gaejexperiments.waverobot;

import java.util.Date;
import org.wadael.waverobotfrmwrk.simple.BlipModifierRobot;

public class MyBlipModifierRobot extends BlipModifierRobot {

@Override
protected String modifyBlipText(String originalBlipText) {
return originalBlipText + " [You typed this at :" + new Date().toString()
+ "]";
}

@Override
protected String getRobotSelfIntroduction() {
return "I am the Blip Modifier Robot";
}

}
```

Let us go through the code now:

1. We have extended the BlipModifierRobot since all we want to do in this Robot is to modify the Blip that was submitted.

2. All we need to do as part of extending the BlipModifierRobot class is to implement the **modifyBlipText** method. This method gets passed one parameter **originalBlipText** that is the blip text that was submitted. So in all essence, you could inspect what the text was in the blip submitted and then modify it as needed . The implementation above appends the text “You typed this at [DateTimeStamp]” to the originalBlipText and simply returns that string. The WadRobotFramework’s **BlipModifierRobot** will take care of modifying the Blip Text.

3. We also implement an optional method named **getRobotSelfIntroduction**. This method returns a string and you can return something that identifies the addition of your Robot to the Wave. So when your Robot is added as a participant and if you have implemented the **getRobotSelfIntroduction** method, then it will display this message out. It is sort of announcing to the other wave participants about your presence.

That is all there is to implemented the **MyBlipModifierRobot**. If you would have studied the earlier tutorial on writing a Google Wave Robot, you would have noticed that the WadRobotFramework has done away with all the Event processing that you had to handle yourself and also shielded you from methods that you need to know to create a blip and append to it.

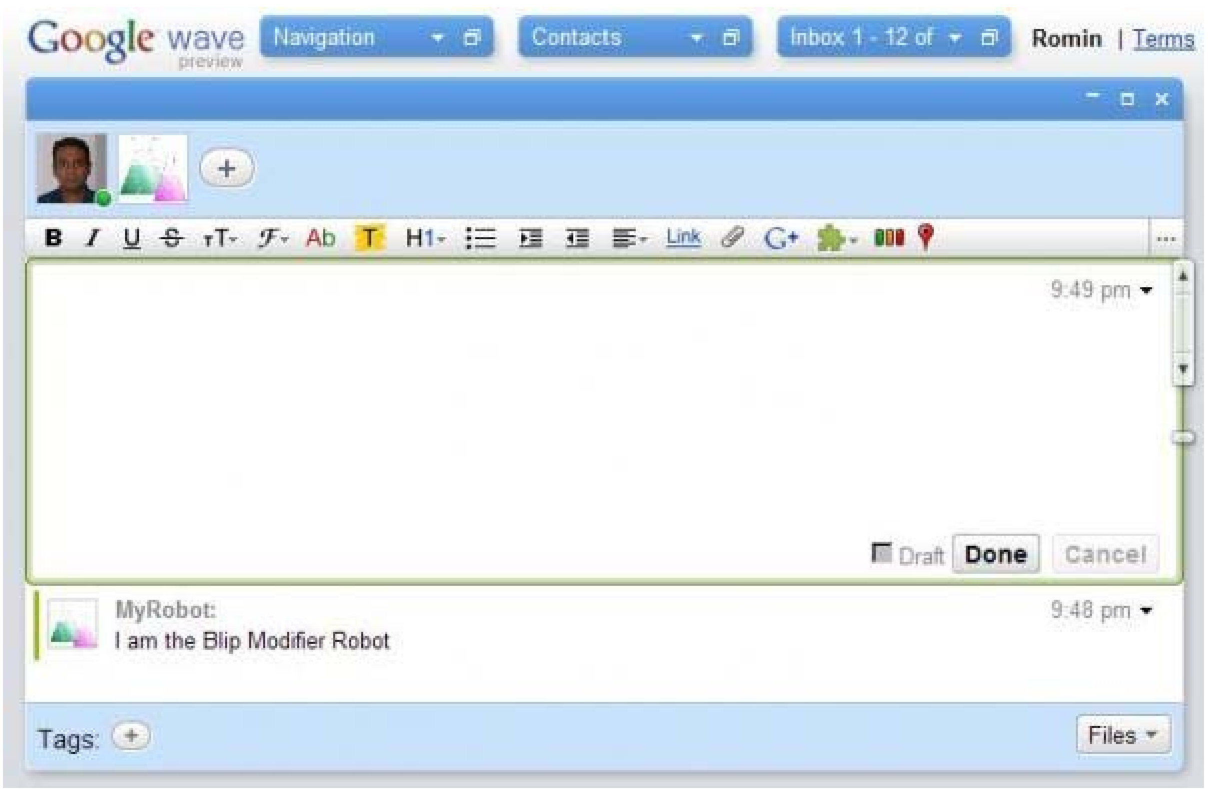
Configuring the MyBlipModifierRobot in web.xml

We need to add the **MyBlipModifierRobot in the <servlet/>** and **<servlet-mapping/>** entry to the **web.xml** file. This file is present in the **WEB-INF** folder of the project. The necessary fragment to be added to your **web.xml** file are shown below.

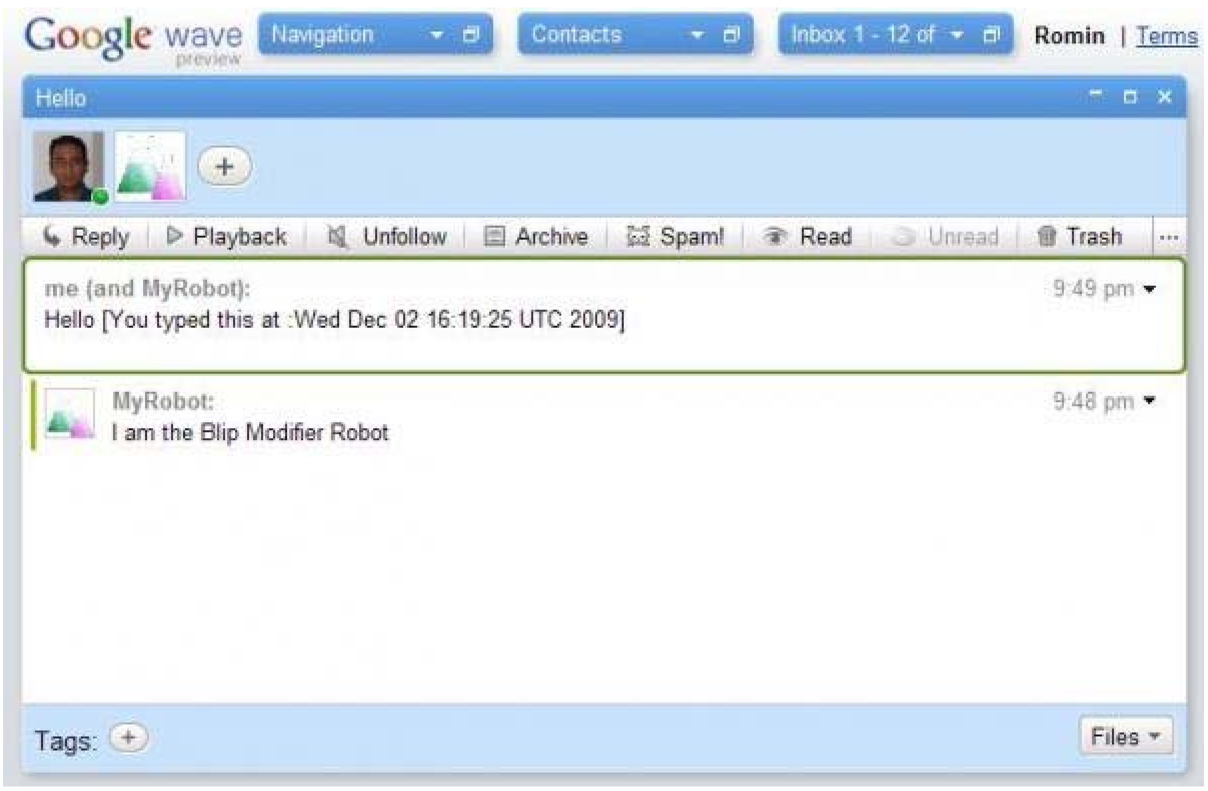
```
<servlet>
 <servlet-name>MyBlipModifierRobot</servlet-name>
 <servlet-
class>com.gaejexperiments.waverobot.MyBlipModifierRobot</servlet-class>
 </servlet>
 <servlet-mapping>
 <servlet-name>MyBlipModifierRobot</servlet-name>
 <url-pattern>/_wave/robot/jsonrpc</url-pattern>
 </servlet-mapping>
```

If you wish to check out this Robot in action, you can do so with the same project. Simply replace the previous Robot's (MyAppenderRobot) servlet entries with the one shown above. And deploy the application following the same steps in the **Deploying the Application** section. Leave the Profile Servlet and the capabilities.xml file as is because the same applies in this case too.

Once you deploy the MyBlipModifierRobot, you should see it in action as shown below. The first screen shows the MyBlipModifierRobot announcing itself and then the second screen shows how it modified the blip text that I submitted by appending the TimeStamp to it.



MyRobot:
I am the Blip Modifier Robot



me (and MyRobot):
Hello [You typed this at :Wed Dec 02 16:19:25 UTC 2009]

MyRobot:
I am the Blip Modifier Robot

Conclusion

This concludes Episode 11 of the series. We saw how easy it was to write Simple Google Wave Robots using the WadRobotFramework. Several Google Wave Robots that are out there simply do one of the things that we have seen in this episode i.e. modify a blip’s contents or append a blip in reaction to a blip submitted. You can get going with such Robots by using this framework which hides you from the underlying Google Wave API.

In the next episode, we shall see how to write an Advanced Google Wave Robot that can implement several commands. Till then, unleash those Simple Robots!

Episode 12 : Writing an Advanced Google Wave Robot using WadRobotFramework

Welcome to **Episode 12** of the series. This episode is an extension of the earlier episode where we saw how to write Simple Google Wave Robots using the WadRobotFramework. I strongly recommend that you have completed the earlier episode and have got comfortable with the WadRobotFramework since this episode builds on the earlier one.

To recap, the WadRobotFramework distinguishes between 2 kinds of Robots and I summarize it again over here.

Simple Robots : These robots we covered in the earlier episode and saw how you can write a simple robot to react to a blip by appending a new blip (**BlipAppenderRobot**) or even modify the Blip Text (**BlipModifierRobot**).

Advanced Robots : These are of main focus in this article and I reproduce from the earlier episode the text so that you understand what Advanced Robots are first. The definition of Advanced Robots is per the WadRobotFramework and it is not meant to indicate this is the final definition of it.

Advanced Robots are those that can react to **instructions** (or commands) in the Blips. Here are some samples of Advanced Robots and how they would react to commands from Blips:

1. A character Counting Advanced Robot:

Your submitted Blip Text contains : “Here is some text in the blip. Please count the length of this message for me.” {mycountingrobot:count}.

You can write an advanced Robot (**mycountingrobot**) that knows how to count the length of the message. So it gets notified when it there is a **instruction (**count**)** in the Blip. Your Advanced Robot can then count out the characters and then either append or modify a blip as needed.

2. A Tweeting Robot:

Your submitted Blip Text contains the following text : “{mytweetingrobot:tweet} Some text to tweet”

You can write an advanced Robot (**mytweetingrobot**) that knows how to tweet the message to Twitter. So it gets notified when it there is a **instruction (**tweet) in the Blip. Your Advanced Robot can then append a blip or modify a blip saying that the message has been tweeted.

The best part of it all is that you could combine all of this into a single Robot that can respond to one or more commands. For example, take a look at *Today's Special* Robot (see http://ppandhi.wordpress.com/2009/11/08/todays-special-robot/) that can respond to more than one command. It can give you the quotes, day in history, word of the day, cricket score, your daily horoscope by simply responding to the command that you type in.

So for example, you could write a robot and give it commands like this:

1. {myrobot:doCommand1}

2. {myrobot:doCommand2}

3. {myrobot:doCommandN} and so on.

In this article, we are going to see exactly how to achieve the above command **Robot** that will delegate its work to different **workers** who are responsible for executing the command i.e. doing the work.

Let us get a few definitions in place first:

1. The Command Robot: This is the main class of your Robot and you need to extend the **org.wadael.waverobotfrmwrk.advanced.WithWorkersRobot**. You need to have an identifier for your robot, which is a unique ID for your Robot. Let us call it **GAEJRobot.**

2. Each Command Robot is capable of following instructions or commands. These instructions are executed by the Workers.

3. A Worker is a Command implementation that performs a certain logic. For e.g. fetching a stock quote, getting a word of a day, sending a Tweet, sending an email, etc. Each worker will indicate what instruction or command it obeys.

As an example, say you want to write an Advanced Robot class (**WithWorkersRobot**) whose identifier is named **GAEJRobot** that can responds to the following two commands:

a. SendTweet
b. GiveWordOfTheDay

So, you will implement two Workers and register (add) them to the **GAEJRobot** class. The two worker classes will be :

- **SendTweetWorker** which says that listens to an instruction named **tweet** and it will implement its core logic in the **doWork()** method.
- **GiveWordOfTheDayWorker** which says that it listens to an instruction named **wotd** and it will implement its core logic in the **doWork()** method.

Now, in your Wave Conversation, you can give the following text in the Blip (of course after adding the Robot as a participant).

1. {GAEJRobot:tweet}

2. {GAEJRobot:wotd}

Voila! The WadRobotFramework will then do the heavy lifting for you. It roughly works as follows:

- When you submit a Blip, it will scan the Blip Text for the identifier and see if it matches itself.
- If yes, it will scan out the instructions and invoke the doWork() method of the Worker Robot that implements the instruction.

This is not all. The WadRobotFramework has thought about **parameters** that you may need to pass to your Robot. For e.g. consider the following fictitious instruction that you need to give to a Stock Quote Robot.

{StockQuoteRobot:getquote GOOG} or {StockQuoteRobot:getquote GOOG,MSFT,ADBE,ORCL,IBM}

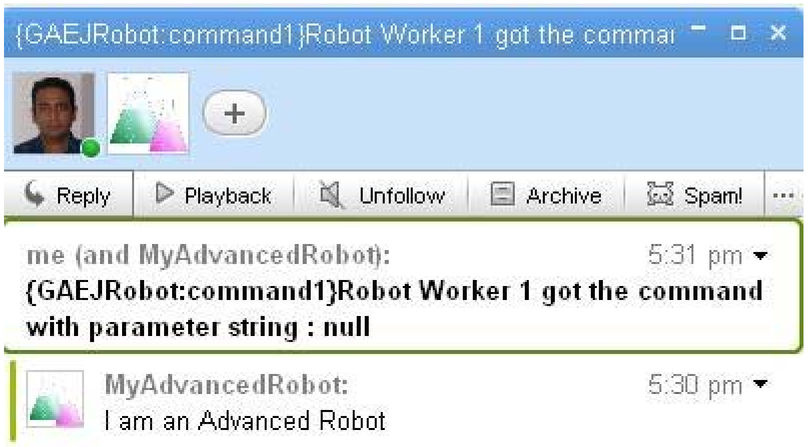
In short the format is as follows:

{RobotIdentifier:Instruction<space>[parameters]}

So in the above two examples the parameter GOOG and the parameter string "GOOG,MSFT,ADBE,ORCL,IBM" will be passed to the **doWork()** method of your **RobotWorker** that has registered with the **Advanced Robot** and who implements the ***getquote*** **instruction**. Please read this last statement clearly with the highlighted words as the key pieces required to build out an Advanced Robot.

Simple yet powerful and it opens up a wide range of Robots that you can start writing today. So let me get started and demonstrate the basic flow to get an Advanced Robot up and running which can accept 1 or more instructions. The Robot does not do anything specific except for simply demonstrating the flow. Readers are expected to extend it with their ideas brimming inside their minds.

To understand what we will build, it helps to take a look at the running robot as shown below:



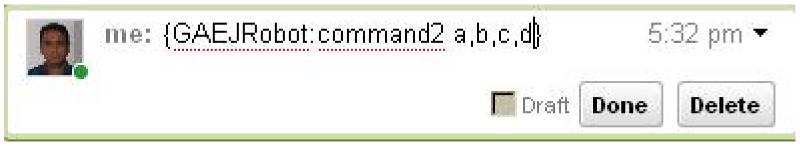
me (and MyAdvancedRobot): 5:31 pm
{GAEJRobot:command1}Robot Worker 1 got the command with parameter string : null

MyAdvancedRobot: 5:30 pm
I am an Advanced Robot

You will notice that I have added my robot called **MyAdvancedRobot**. The identifier for the Robot is **GAEJRobot** and the Robot has two workers (**Worker1** and **Worker2**) registered with it, which implement the instructions **command1** and **command2** respectively.

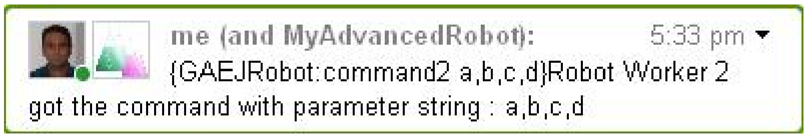
Now when I submit the text **{GAEJRobot:command1}** , the doWork() method of the Worker1 is invoked. It simply accepts the command and prints out that it received the command with no parameters passed to it.

Similarly, look at the wave conversation below:



me: {GAEJRobot:command2 a,b,c,d} 5:32 pm

Here I give the command2 to the GAEJRobot and I am also passing a string of parameters. When I click the **Done** button, the doWork() method of the Worker2 is invoked. It simply accepts the command and prints out that it received the command with the parameter string. Now, it could have processed the parameters and performed its logic accordingly. This demonstrates how the wiring is done by WadRobotFramework to make your life easier in writing Advanced Google Wave Robots.



me (and MyAdvancedRobot): 5:33 pm
{GAEJRobot:command2 a,b,c,d}Robot Worker 2 got the command with parameter string : a,b,c,d

Let us start writing code now. I will assume that you are fairly conversant now with the typical project structure and requirements of writing a Google Wave Robot. If not, I suggest that you first go through these tutorials in the following order:

- **Episode 7** : Writing your First Google Wave Robot

- **Episode 11:** Develop Simple Google Wave Robots using the WadRobotFramework

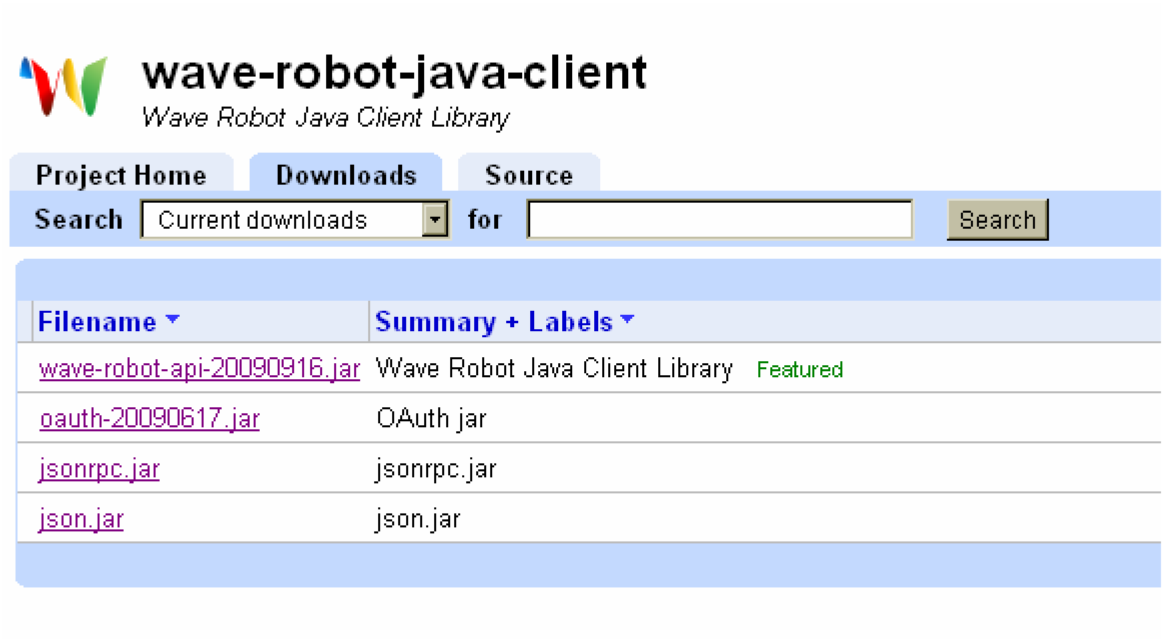
Project Setup

Create a New Project or use the same project **MyGoogleWaveRobot** that we used in the earlier **Episode 11**.

If you are creating a **New project**, then please make sure that you download all the JARs from the following two locations:

http://code.google.com/p/wave-robot-java-client/downloads/list

The web page when you navigate to the above URL is shown below:



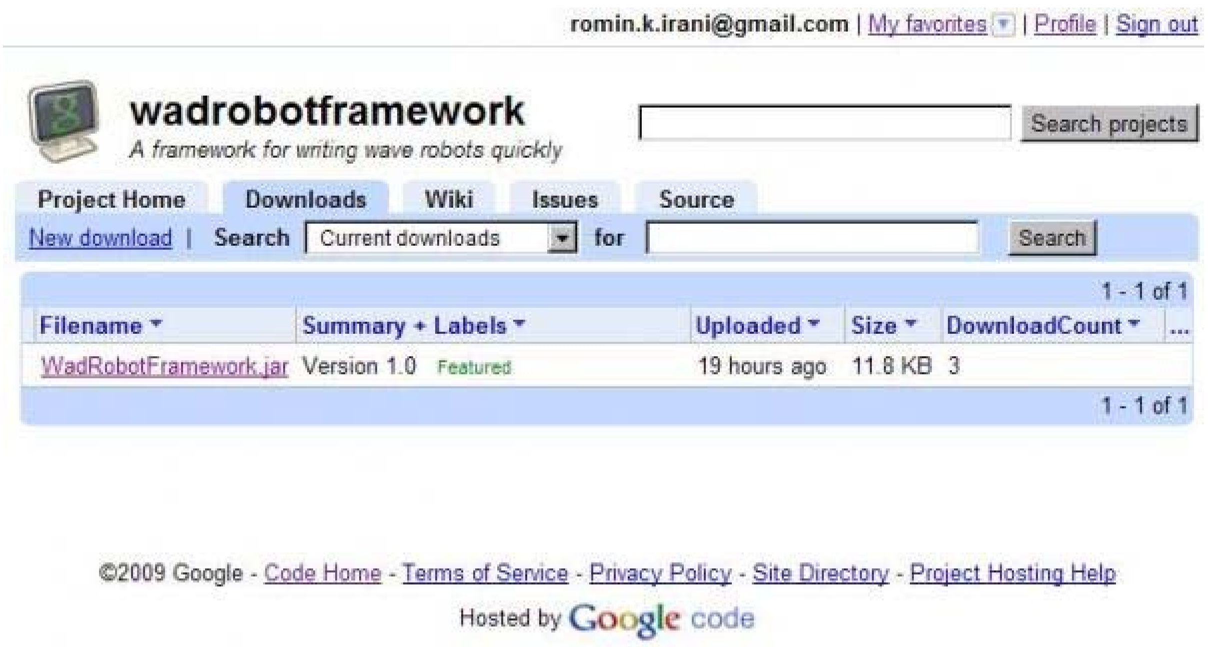
Filename	Summary + Labels
wave-robot-api-20090916.jar	Wave Robot Java Client Library Featured
oauth-20090617.jar	OAuth jar
jsonrpc.jar	jsonrpc.jar
json.jar	json.jar

Download all the above files to your machine.

The **WadRobotFramework JAR file** is available at the following location :

http://code.google.com/p/wadrobotframework/downloads/list

The web page when you navigate to the above URL is shown below:


romin.k.irani@gmail.com | My favorites | Profile | Sign out

wadrobotframework
A framework for writing wave robots quickly

Project Home | Downloads | Wiki | Issues | Source

New download | Search Current downloads for

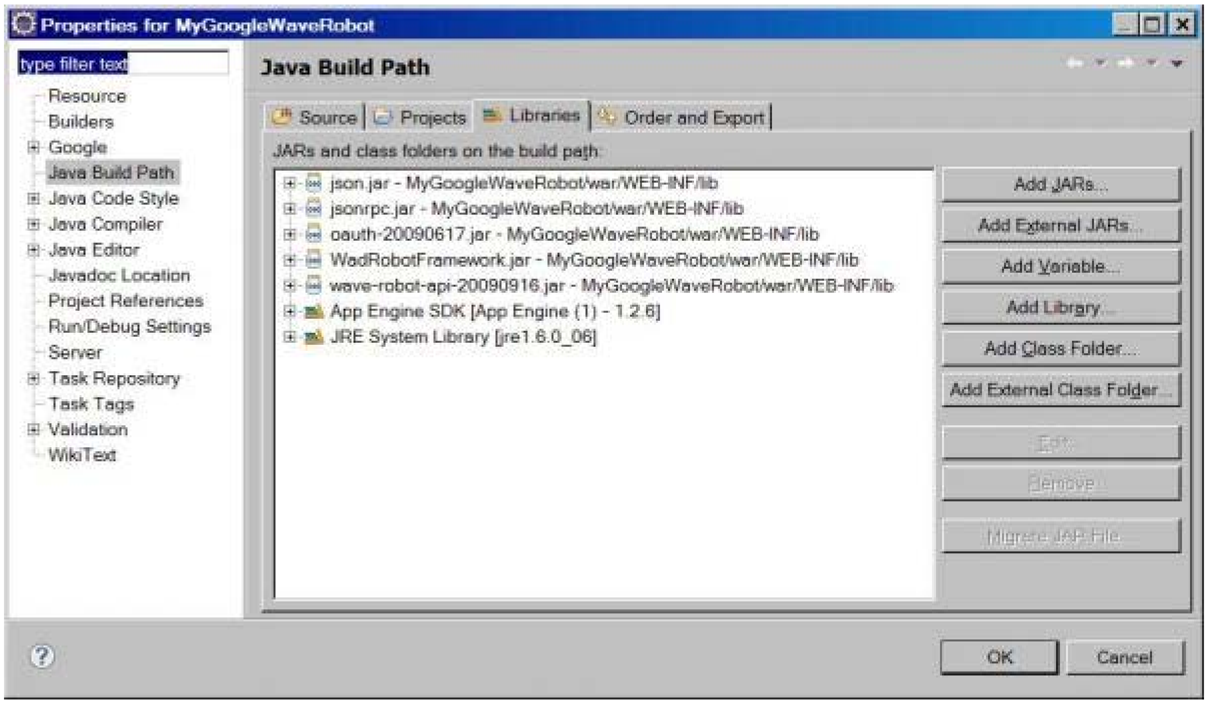
1 - 1 of 1

Filename	Summary + Labels	Uploaded	Size	DownloadCount
WadRobotFramework.jar	Version 1.0 Featured	19 hours ago	11.8 KB	3

1 - 1 of 1

©2009 Google - Code Home - Terms of Service - Privacy Policy - Site Directory - Project Hosting Help
Hosted by Google code

Download the JAR file to your machine. Once you have downloaded the **5 JAR files**, make sure that they are copied to the \WEB-INF\lib folder of your project and that the Project Build Path is also setup with the JAR files as shown below:


Properties for MyGoogleWaveRobot

Java Build Path

JARs and class folders on the build path:

- json.jar - MyGoogleWaveRobot/war/WEB-INF/lib
- jsonrpc.jar - MyGoogleWaveRobot/war/WEB-INF/lib
- oauth-20090617.jar - MyGoogleWaveRobot/war/WEB-INF/lib
- WadRobotFramework.jar - MyGoogleWaveRobot/war/WEB-INF/lib
- wave-robot-api-20090916.jar - MyGoogleWaveRobot/war/WEB-INF/lib
- App Engine SDK [App Engine (1) - 1.2.6]
- JRE System Library [jre1.6.0_06]

Writing the Advanced Robot : MyAdvancedRobot.java

The first step is to create a Java class. Call it MyAdvancedRobot.java. The source code is listed below:

```
package com.gaejexperiments.waverobot;

import org.wadael.waverobotfrmwrk.advanced.WithWorkersRobot;

public class MyAdvancedRobot extends WithWorkersRobot {

 public MyAdvancedRobot() {
 super();
 //This will process 'command1'
 addRobotWorker(new Worker1());

 //This will process 'command2'
 addRobotWorker(new Worker2());
 }

 @Override
 public String getRobotIdentifier() {
 return "GAEJRobot";
 }

 @Override
 protected String getUsage() {
 return "Advanced Robot commands : command1 and command2";
 }

 @Override
 protected String getRobotSelfIntroduction() {
 return "I am an Advanced Robot";
 }

}
```

Let us dissect the code now:

1. Our Advanced Robot class **MyAdvancedRobot** extends the **WithWorkersRobot** class, which is required for creating the Advanced Robots with the WadRobotFramework.

2. The **WithWorkersRobot** class constructor uses a method called **addRobotWorker**(…) to add one or more workers to it. Remember each worker will execute or do its job as per the instruction that it obeys. So we are going to have two workers : Worker1 and Worker2 which we are adding to our AdvancedRobot. We will get to the listing of the Worker1 and Worker2 later but it is sufficient to keep in mind, that **Worker1** class will perform the work required with **command1** instruction is given in the Blip and **Worker2** class will perform the work required when **command2** instruction is given in the Blip. To recap, as you add more workers, you will need to add them here in the constructor using the **addRobotWorker** method.

3. The getRobotIdentifier() method is used to return the string that is the unique identifier of your Robot. This is used to distinguish Robots in the same Wave. The identifier if you recollect is going to be used by the other participants in the wave to give instructions to your Robot. As mentioned, the format in the Blip to give an instruction to your robot will be like this:

{RobotIdentifier:Instruction<space>[parameters]}

Each Instruction is implemented by a Worker. For e.g. command1 will be implemented by Worker1 and command2 will be implemented by Worker2.

So to invoke Worker1, we have to say {**GAEJRobot**:**command1**} in the Blip and submit it. Hence we return **GAEJRobot** in the **getRobotIdentifier()** method and this will be the unique way in which we identify this Robot

4. The **getUsage()** method is used to return a string that will be displayed to the participant when they type **/help** in the start of the Blip. This is useful to give a list of possible instructions that your Advanced Robot implements. In our case here, we are implementing two instructions and hence we have returned some string. But you can give an elaborate help string stating detailed command text, sample parameters, etc.

5. Finally, we have the **getRobotSelfIntroduction()** method. This is not mandatory but it is nice to announce to the particpants when you (Robot) gets added to the wave as a participant. Simply return a String that you would like to announce to the other (existing) participants in the Wave.

Implementing the Workers

We are now going to implement the Workers i.e. Worker1 and Worker2. The code is identical for both of them and it is listed below:

Worker1.java

```java
package com.gaejexperiments.waverobot;

import org.wadael.waverobotfrmwrk.advanced.RobotWorker;

import com.google.wave.api.Blip;
import com.google.wave.api.Event;
import com.google.wave.api.RobotMessageBundle;

public class Worker1 implements RobotWorker {

 public String getInstruction() {

 return "command1";
 }

 public boolean doWork(RobotMessageBundle bundle, Blip blip, Event evt,
String params) {
 blip.getDocument().append("Robot Worker 1 got the command with
parameter string : " + params);
 return true;
 }

 public String getDescription() {
 return "Robot Worker 1";
 }
}
```

Worker2.java

```java
package com.gaejexperiments.waverobot;

import org.wadael.waverobotfrmwrk.advanced.RobotWorker;

import com.google.wave.api.Blip;
import com.google.wave.api.Event;
import com.google.wave.api.RobotMessageBundle;

public class Worker2 implements RobotWorker {

 public boolean doWork(RobotMessageBundle bundle, Blip blip, Event evt,
String params) {
 blip.getDocument().append("Robot Worker 2 got the command with
parameter string : " + params);
 return true;
 }
 public String getDescription() {
 return "Robot Worker 2";
 }

 public String getInstruction() {

 return "command2";
 }

}
```

Let us go through the code of one of them and you will be able to understand it:

1. To recap, a Worker implements an instruction or a single command. Each Worker class needs to implement the **RobotWorker** interface in the WadRobotFramework.

2. It needs to implement the getInstruction() method which returns a String. This is the instruction that the Worker will obey or perform. In our case, the command1 is being done by Worker1 class and the command2 is being done by Worker2 class respectively. So when someone submits {GAEJRobot:**command1**} in the Blip, the doWork() implementation of the **Worker1** class will be invoked and if they submit {GAEJRobot:**command2**} in the Blip, the doWork() implementation of the **Worker2** class will be invoked.

3. It needs to implement the **doWork()** method. This method is the heart or the main implementation of the Worker. Here you will place all your processing logic. Notice that since this is an Advanced Robot, it is assumed that you would even like to make use of the Google Wave API classes directly. So you are passed in instances of **RobotMessageBundle**, **Blip** and **Event** classes. The last parameter passed is **params** and it represents any parameters passed to the robot.

You will notice in the implementation that we have done for the Worker, that we simply Append to the Blip Text saying that the Worker got its command and notice that we also print out the Parameter String. So if you logic depends on the values of the parameters passed, you can parse out the parameters here itself and perform your logic.

That is all we need to do as far as writing Java code is concerned. Of course we have the other mandatory files that we need to create, which we will cover quickly now:

Configuring the MyAppenderRobot in web.xml

We need to add the **MyAdvancedRobot in the <servlet/>** and **<servlet-mapping/>** entry to the **web.xml** file. This file is present in the **WEB-INF** folder of the project. The necessary fragment to be added to your **web.xml** file are shown below.

```
<servlet>
<servlet-name>MyAdvancedRobot</servlet-name>
```

```
<servlet-class>com.gaejexperiments.waverobot.MyAdvancedRobot</servlet-
class>
</servlet>
<servlet>
<servlet-name>MyRobotProfileServlet</servlet-name>
<servlet-
class>com.gaejexperiments.waverobot.MyRobotProfileServlet</servlet-
class>
</servlet>
<servlet-mapping>
<servlet-name>MyAdvancedRobot</servlet-name>
<url-pattern>/_wave/robot/jsonrpc</url-pattern>
</servlet-mapping>
<servlet-mapping>
<servlet-name>MyRobotProfileServlet</servlet-name>
<url-pattern>/_wave/robot/profile</url-pattern>
</servlet-mapping>
```

Notice that we also have the ProfileServlet configured here, which is a good and recommended thing to have for your Robot. The ProfileServlet class is covered in the next section.

ProfileServlet.java

The implementation is straightforward and contains the ApplicationId that I have used for my AdvancedRobot. You can replace it with your id.

```
package com.gaejexperiments.waverobot;

import com.google.wave.api.ProfileServlet;

public class MyRobotProfileServlet extends ProfileServlet {

@Override
public String getRobotAvatarUrl() {
return "http://myinfoagent.appspot.com/_wave/myimage.jpg";
}

@Override
public String getRobotName() {
return "MyAdvancedRobot";
}

@Override
public String getRobotProfilePageUrl() {
return "http://myinfoagent.appspot.com";
}

}
```

Creating the MyAdvancedRobot capabilities.xml files

We need an additional file to describe the capabilities of the Robot that we have written. This file is called the **capabilities.xml** and it needs to reside in a certain location. You need to create a **_wave** directory inside of the **war** directory of your project. The location of this file is **war/_wave** directory.

You will need to create the **_wave** directory and create the **capabilities.xml** file over there. The capabilities file shown below is pretty straightforward and is shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<w:robot xmlns:w="http://wave.google.com/extensions/robots/1.0">
 <w:capabilities>
 <w:capability name="BLIP_SUBMITTED" content="true" />
 </w:capabilities>
 <w:version>1</w:version>
</w:robot>
```

Deploying the Application

To deploy the application, you will need to first create your **Application ID**. The Application Identifier can be created by logging in at **http://appengine.google.com** with your Google Account. You will see a list of application identifiers already registered under your account (or none if you are just getting started). To create a new Application, click on the Create Application button and provide the Application Identifier as requested. Please note this name down since you will be using it for deployment.

For e.g. I have registered an application identifier named **myinfoagent**.

To deploy the application, follow these steps (they should be familiar to you now):

1. Click on the Deploy Icon in the Toolbar.
2. In the Deploy dialog, provide your Email and Password. Do not click on Deploy button yet.
3. Click on the **App Engine Project settings** link. This will lead you to a dialog, where you need to enter your Application ID [For e.g. my Application Identifier **myinfoagent**]
4. Click on **OK**. You will be lead back to the previous screen, where you can click on the **Deploy** button. This will start deploying your application to the GAEJ cloud. You should see several messages in the Console window as the application is being deployed.
5. Finally, you should see the message "Deployment completed successfully".

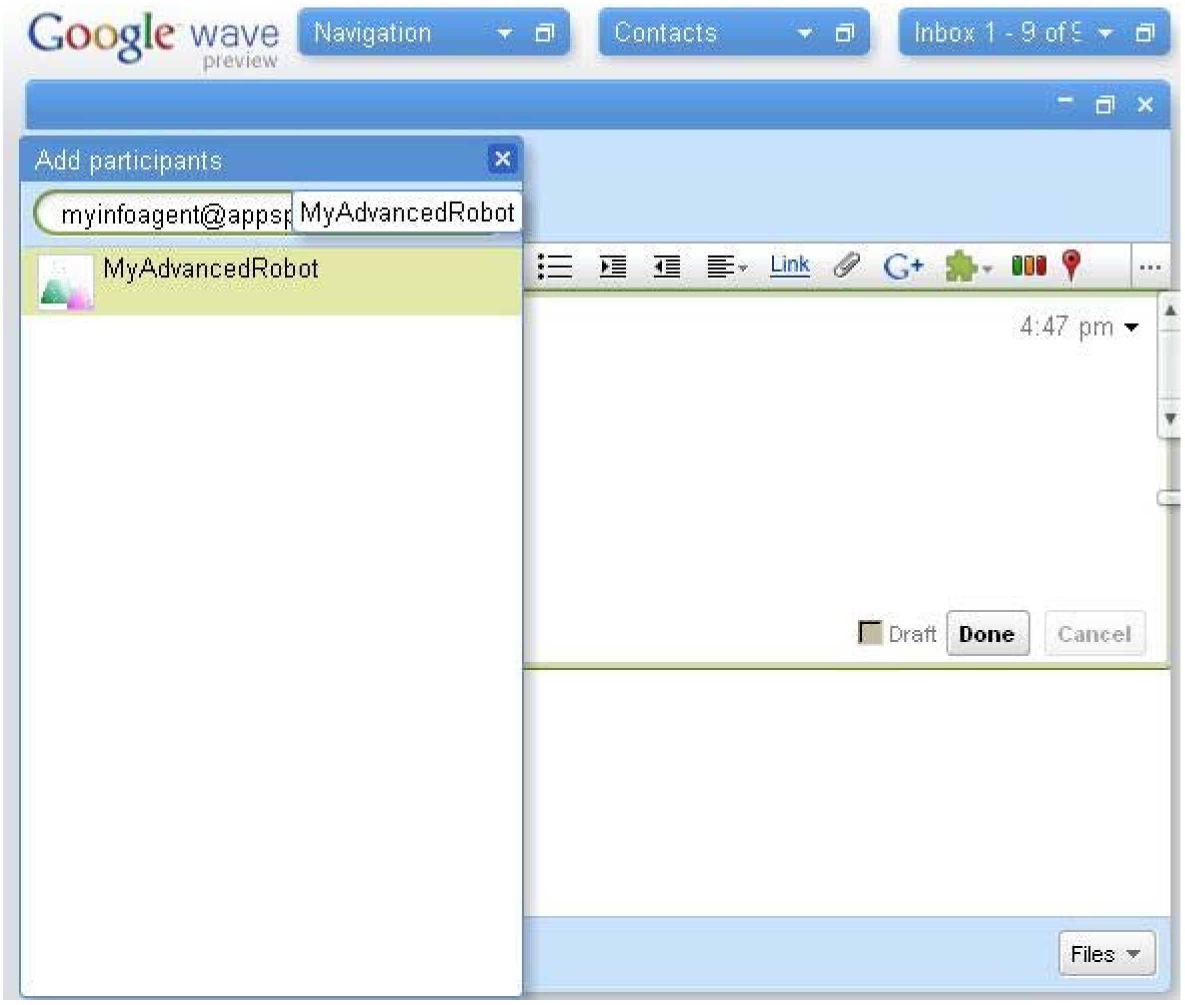
MyAdvancedRobot in Action

Your application is going to be available at the http://**yourapplicationid**.appspot.com. In my case, the application is available at **http://myinfoagent.appspot.com**.

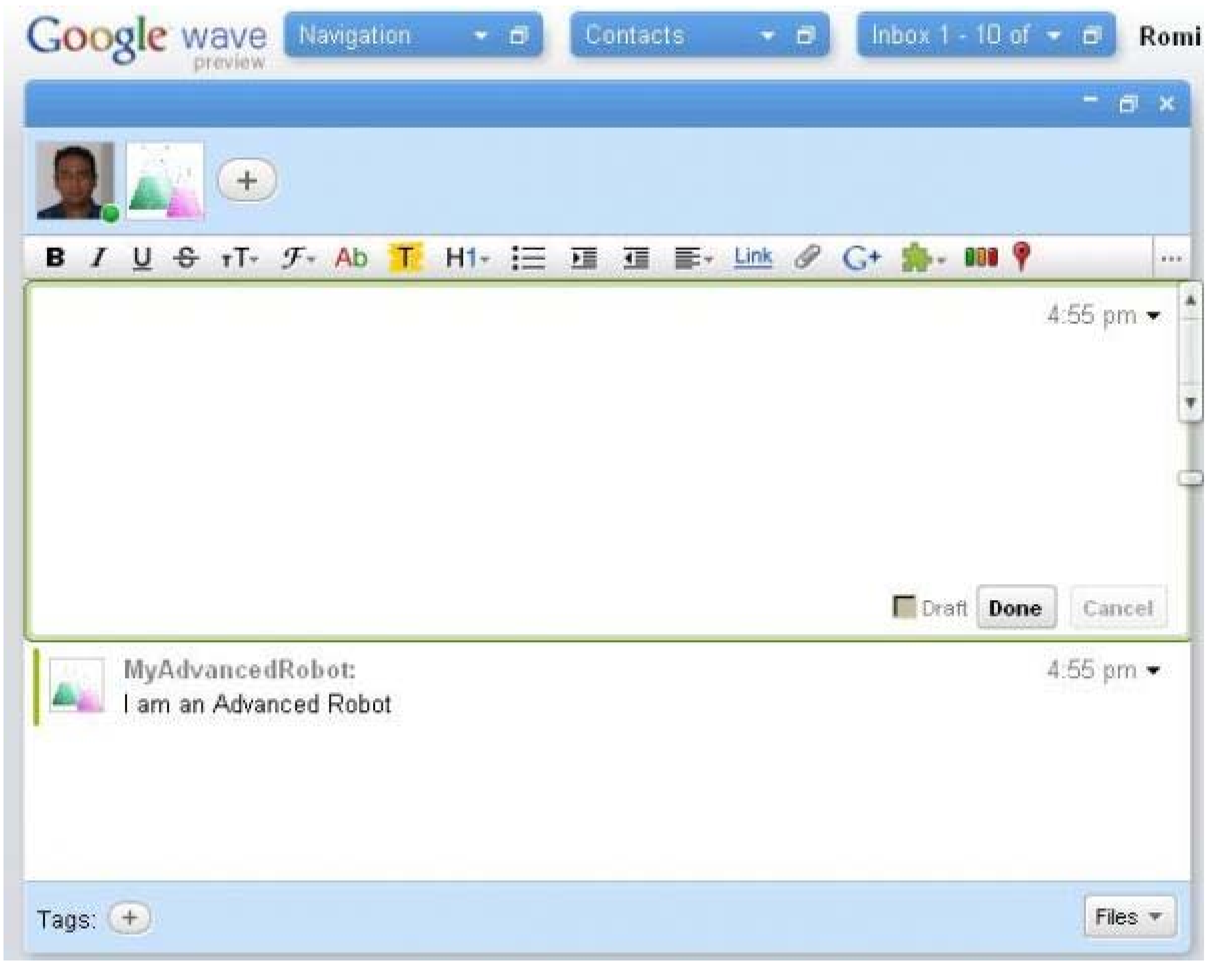
You can test for the presence of your robot capabilities file by simply typing in the following:

http://**yourapplicationid.**appspot.com**/_wave/capabilities.xml** [Replace yourapplicationid with the Application ID that you have] and you should see the capabilities.xml file being served up.

To test out the Robot, you need to launch the Google Wave client and login in with your account by going to http://wave.google.com. On successful login, you will be inside the Wave client from where you can create a new wave by clicking on the **New Wave** link. I then add the **myinfoagent@appspot.com**, which is our AdvancedRobot to the Wave as a participant as shown below:



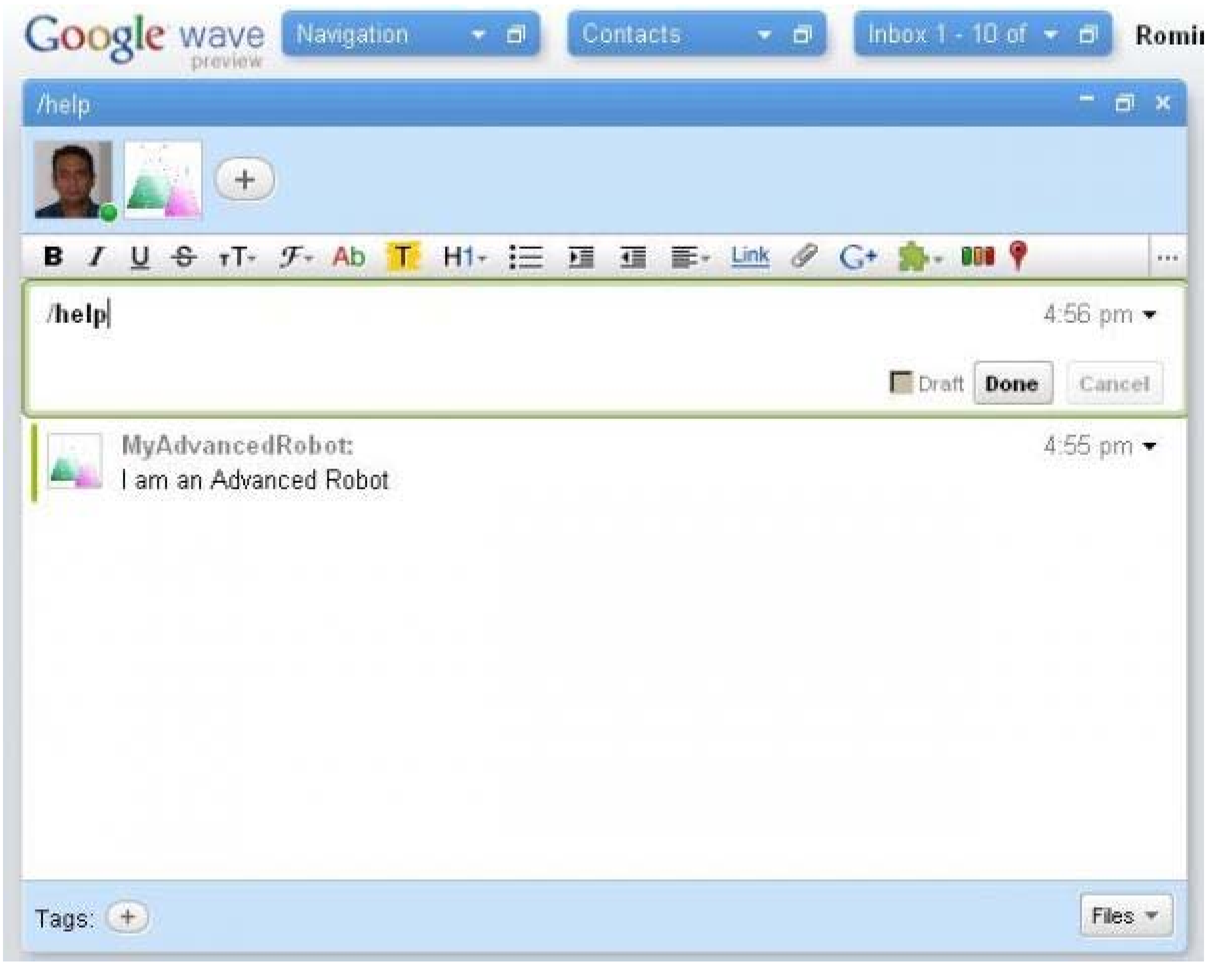
On addition, the AdvancedRobot will announce itself to the Participants. This is the method **getRobotSelfIntroduction()** that we implemented in the **MyAdvancedRobot.java** class. The output is shown below:



MyAdvancedRobot:
I am an Advanced Robot

4:55 pm

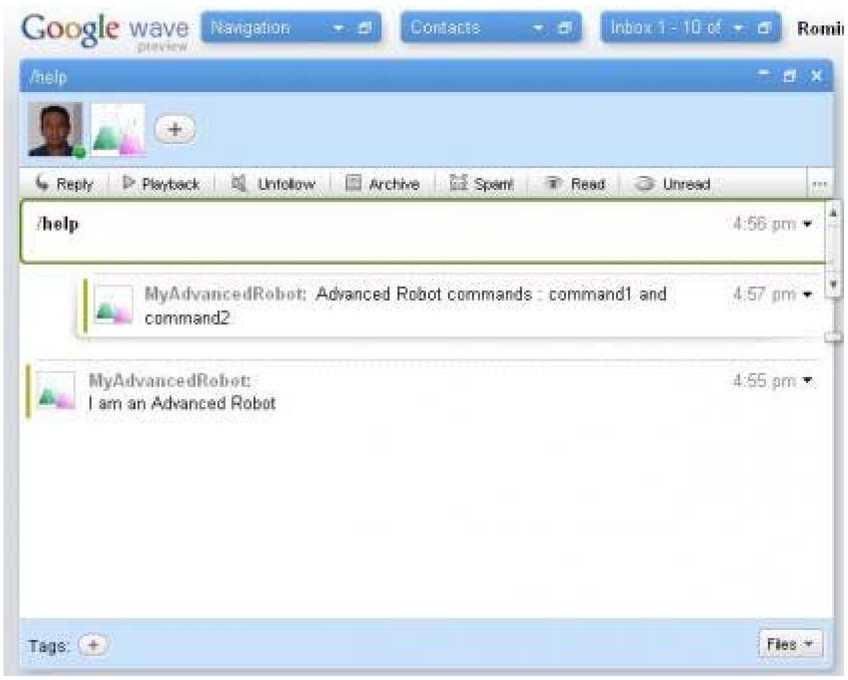
Now, we type in the message **/help** in the beginning of the Blip and submit it as shown below:



/help 4:56 pm

MyAdvancedRobot: 4:55 pm
I am an Advanced Robot

When submitted, the method **getUsage()** of the MyAdvancedRobot class is invoked and it displays the commands and any help instruction that you provided in there.

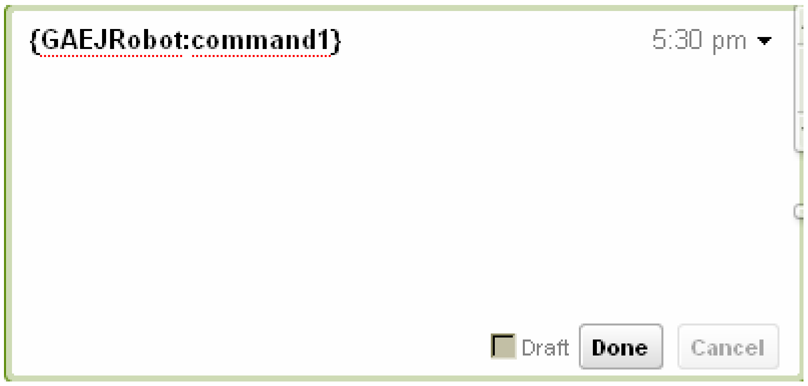


/help 4:56 pm

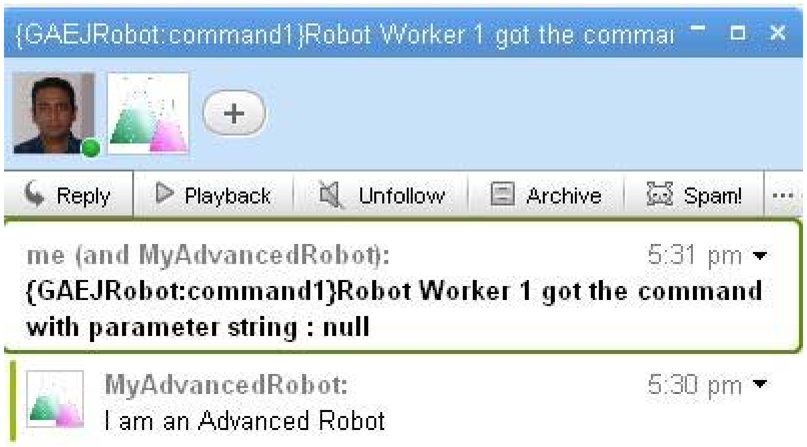
MyAdvancedRobot: Advanced Robot commands : command1 and command2 4:57 pm

MyAdvancedRobot:
I am an Advanced Robot 4:55 pm

The next thing we do is give a command to the first Worker as shown below and click on the **Done** button.


{GAEJRobot:command1} 5:30 pm

This will invoke the **doWork()** method on the **Worker1** class since this class has indicated that it implements **command1** as mentioned in the **getInstruction()** method of the class. The response of the Worker is shown below:

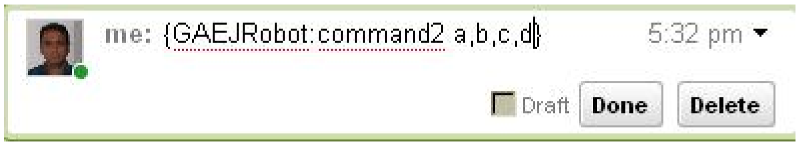


me (and MyAdvancedRobot): 5:31 pm
{GAEJRobot:command1}Robot Worker 1 got the command with parameter string : null

MyAdvancedRobot: 5:30 pm
I am an Advanced Robot

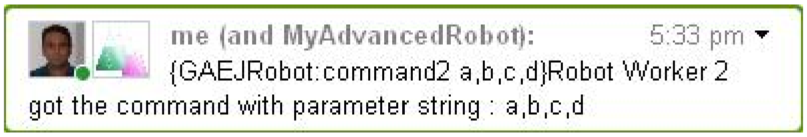
As you can see we did not pass any parameters to it, so it printed out null.

Now, let us invoke the second Worker as shown below and notice that we are passing parameters to it now:



me: {GAEJRobot:command2 a,b,c,d} 5:32 pm

When we click on **Done**, the **doWork()** method on the **Worker2** class since this class has indicated that it implements **command2** as mentioned in the **getInstruction()** method of the class. The response of the Worker is shown below:



me (and MyAdvancedRobot): 5:33 pm
{GAEJRobot:command2 a,b,c,d}Robot Worker 2 got the command with parameter string : a,b,c,d

Notice that the parameter string was now passed and successfully displayed. You can easily parse out the String and perform your logic.

Conclusion

This concludes the 2-part tutorial on creating Simple and Advanced Robots using the WadRobotFramework. The framework takes away much of the API that you need to know to get going with the Wave API and instead lets you focus on the core of our Robot logic. The framework is still is its early stages and would do with some feedback from you.

Episode 13: Using the Blobstore Java API

Welcome to **Episode 13** of this series. It has been a while since the last episode and to compensate I thought why not cover something that has just been released. In keeping with that line, this episode will cover the recently released Blobstore API in the 1.3.0 release of the Google App Engine SDK.

Using the Blobstore API, we shall cover how to build an application that mimics several sites that allow you to upload a picture and send a Tweet. The Tweet will then contain a link back to the image that you uploaded. Blobstore API now makes all this possible along with several other applications that you could envision like file sharing, document management, etc.

What is the Blobstore API?

The Blobstore API allows your application to store blobs of information, where each blob can be upto 50MB in size. These blobs are typically large files like video and images. This is different from the datastore API that does not allow for saving objects upto this large size. The API documention for the Blobstore API is provided here.

The Blobstore API provides two types of functions:

- An ability to **upload** and **save** the blob automaticallyThe BlobstoreService which is provided by the **com.google.appengine.api.blobstore.BlobstoreService** allows us to specify a URL where users can upload their large files. You can think of this url as the action element in the HTML form. The implementation at this URL is internal to the BlobstoreService. But what it does is significant. It will extract out the file contents that you uploaded and store it as a Blob in the database. Each blob that is stored in the database is associated with the a Blob Key. This Blob key is then provided to your url and you can then use the Blob Key to do anything within your application. In our case, we form a url that is tweeted to the users who can then view the picture that we uploaded.

- An ability to **serve** or **retrieve** the blob.The BlobstoreService also provides an ability to serve or retrieve the blob that was saved successfully. It will provide the blob as a response that could then use as a source in an <img> element for example. All you need to do is provide it the Blob Key and the response stream and in return, it will provide the content properly encoded as per its type that you could then use.

It will be clear as we work out the application in the next few sections.

Note: The Blobstore API is enabled in your application only if you enable “billing” for your applications. What this means is that you can try out the API with your local development server but if you deploy the same on the App Engine cloud, your application will not be able to use the API if it has not been enabled for “billing”. So keep that in mind before leaping with joy that your new application enabled with large file uploads, etc is now ready to start functioning in the cloud. In case you try to use any Blobstore APIs without enabling billing in your application, you will get a **FeatureNotSupported** exception and the text as: **“The Blobstore API will be enabled for this application once billing has been enabled in the admin console.”**

Prerequisites

Google announced the release of version 1.3.0 of the App Engine SDK for both Python and Java. We need to set our development environment with this release of the SDK since the Blobstore API was released in this version.

So the first thing that you need to do is to download the latest release 1.3.0 of the App Engine SDK and upgrade your Eclipse development environment with that. There are two ways you can do that:

a. Use the Update features of Eclipse to download and install the latest SDK

b. Download the SDK from the following link. Make sure you download the Java SDK. Further to the download, inside of your Eclipse IDE, go to **Windows -> Preferences** and then **Google -> App Engine SDK**. Click on Add and provide the path to the expanded 1.3.0 distribution on your machine.

I had covered in detail how to upgrade your SDK for your Eclipse IDE when new releases come out. This was covered in **Episode 5**, in which we upgraded from 1.2.5 -> 1.2.6. Readers are advised to take a look at it in case they are not familiar. The process remains the same.

Tweet My Picture in Action

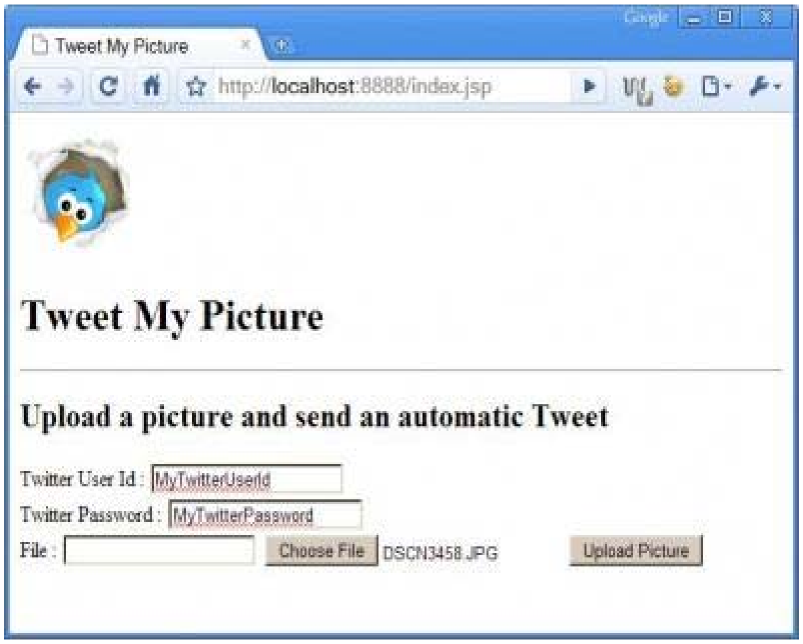
The application that we are going to develop is called **Tweet My Picture**. There are several sites now that allow you to upload a picture to their site and then send a Tweet. The Tweet contains a simple description and the url to the picture. An example is **TwitPic**. I thought the Blobstore API now actually makes an application of this kind way too easy to create, so why not demonstrate that. We will not be creating a polished looking application but rather demonstrating the API and showing that it works. Remember that all the instructions later on deploying and running this application will happen in the local development server and not the live App Engine site because I need to enable billing in order to utilize the Blobstore Service. But the best part is that the local development server works fine, so we can do our complete development and testing on it. Then you are the best judge of writing your own application that utilizes the Blobstore API and making it a billable application when you go live.

Before we get down to code, we should see the end result.

Assuming that I have started my local development server, I navigate to the **index.jsp** page that is shown below:

This will display a simple page, where you provide the following information:

- Your Twitter UserId and Password
- The image file that you wish to upload. Simply click the **Choose File** button and select a file to upload.



Tweet My Picture

Upload a picture and send an automatic Tweet

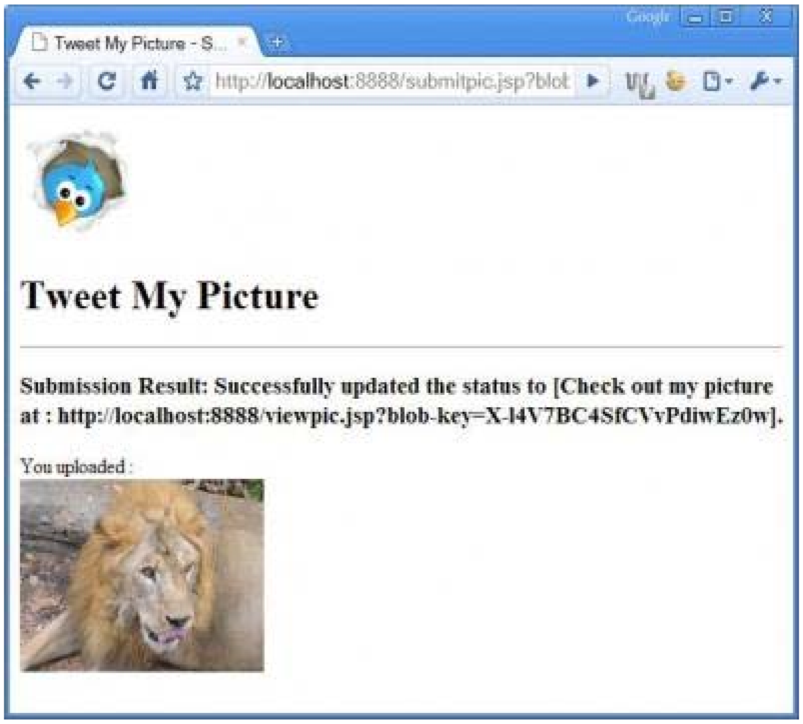
Twitter User Id : MyTwitterUserId
Twitter Password : MyTwitterPassword
File : [Choose File] DSCN3458.JPG [Upload Picture]

Then when you click the **Upload Picture** button, what happens is the following:

1. The file contents that you uploaded will be stored by the Blobstore API into the datastore automatically for you. It will provide you with the Blob Key for the blob that was saved.

2. We use the Blob Key to create a simple Tweet message containing the link. And then we use **Twitter4J** to send the Tweet.

3. Finally a page is displayed indicating the status of the Tweet, the Tweet contents and it also shows the image that you uploaded. This page is shown below:

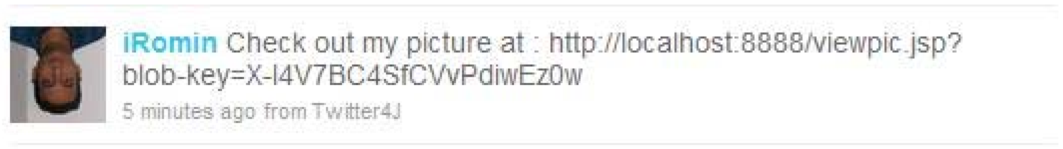


Tweet My Picture

Submission Result: Successfully updated the status to [Check out my picture at : http://localhost:8888/viewpic.jsp?blob-key=X-l4V7BC4SfCVvPdiwEz0w].

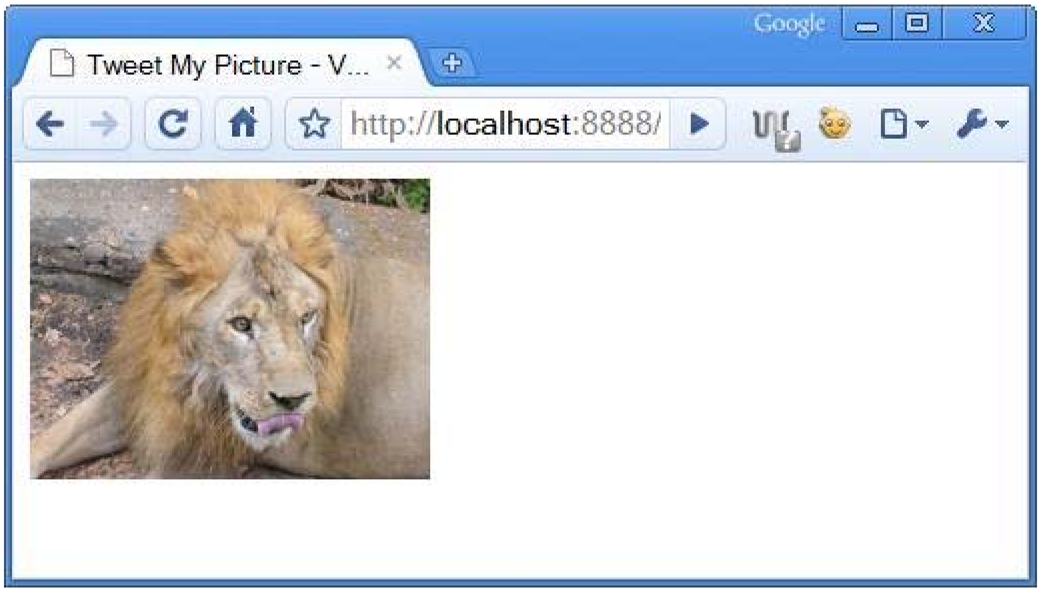
You uploaded :

Now, switch to your Twitter client and you will notice that the Tweet has been posted. A sample screenshot is shown below.The url is pointing to the localhost but that is because I wish to do my testing locally and I have not enabled billing for my live application at the appspot.com domain.



iRomin Check out my picture at : http://localhost:8888/viewpic.jsp?blob-key=X-l4V7BC4SfCVvPdiwEz0w
5 minutes ago from Twitter4J

Clicking on the Tweet link, takes you back to an page that shows you the picture as below:

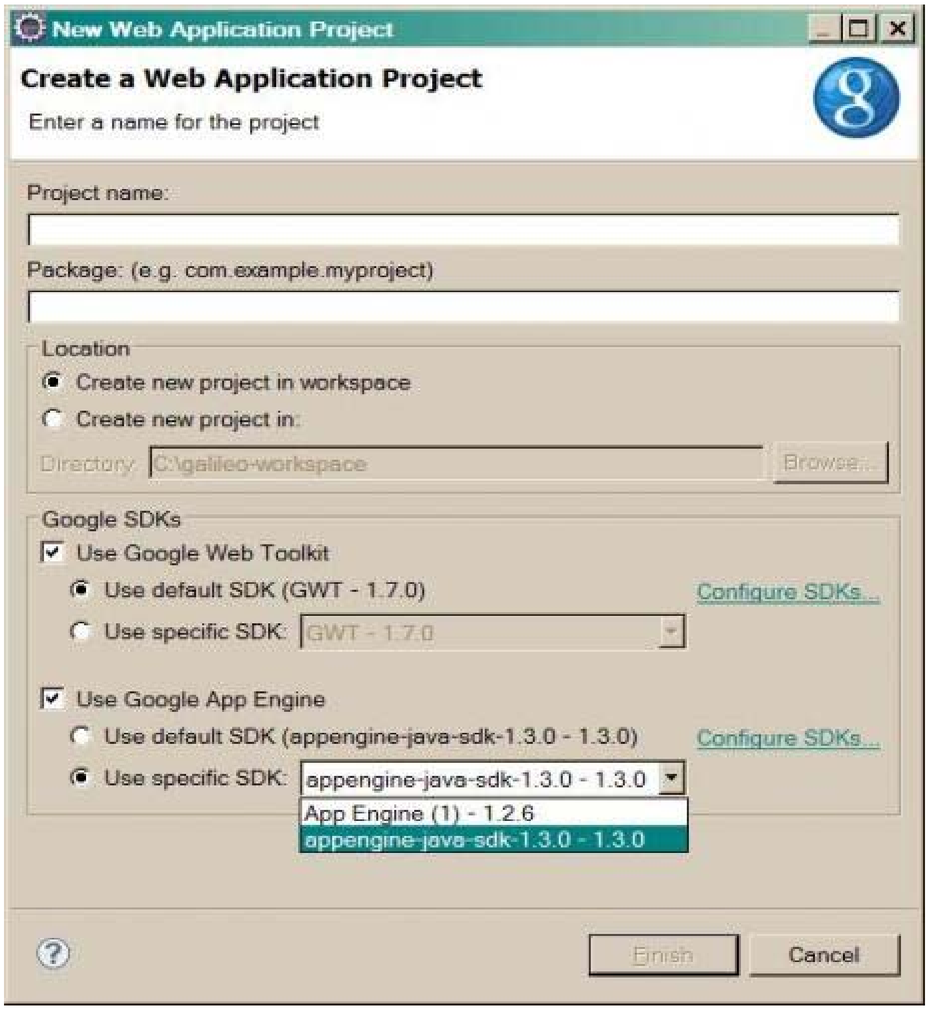


Nice, isn't it? Do not limit yourself only to this example. You can now start thinking on creating serious document management, file sharing, etc applications and use the App Engine infrastructure to host it. So let's get coding now.

Implementing Tweet My Picture

The first thing to do is to create a **New Google Web Application** Project. Follow these steps:

1. Either click on **File –> New –> Other** or **press Ctrl-N** to create a new project. Select Google and then Web Application project. Alternately you could also click on the New Web Application Project Toolbar icon as part of the Google Eclipse plugin. **Please make sure that you have installed the latest 1.3.0 version of the App Engine SDK and are using that as the API for this version.** Shown below is the New Project screenshot where you should see an additional SDK 1.3.0 and need to select that.


New Web Application Project

Create a Web Application Project

Enter a name for the project

Project name:

Package: (e.g. com.example.myproject)

Location

(● Create new project in workspace) ○ Create new project in:

Directory: C:\galileo-workspace

Google SDKs

☑ Use Google Web Toolkit

(● Use default SDK (GWT - 1.7.0)) ○ Use specific SDK: GWT - 1.7.0

☑ Use Google App Engine

○ Use default SDK (appengine-java-sdk-1.3.0 - 1.3.0) (● Use specific SDK: appengine-java-sdk-1.3.0 - 1.3.0)

App Engine (1) - 1.2.6

appengine-java-sdk-1.3.0 - 1.3.0

2. In the New Web Application Project dialog, **deselect the Use Google Web Toolkit** and give a name to your project. I have named mine **GAEJExperiments**. I suggest you go with the same name so that things are consistent with the rest of the article, but I leave that to you. **In case you are following the series, you could simply use the same project and skip all these steps altogether. You can simply go to the next part i.e. the Servlet code.**

3. Click on **Finish**. This will generate the project and also create a sample Hello World Servlet for you. But we will be writing our own Servlet.

Adding Twitter4J Jar to the Classpath and WEB-INF\lib folder

Since we will be using the excellent Twitter4J library for posting to Twitter, you need to download the latest version of the JAR file. Download it from here. I have used **twitter4j-2.0.10.jar**. You need to copy this file to the **WEB-INF\lib** folder of the project. Also add the JAR file to the **Build Classpath** of the Eclipse project.

index.jsp

The first page is the index.jsp page. This is a simple page that will have a form that accepts the Twitter UserId/Password along with the file to be uploaded. The code is shown below. Just a quick note that all the JSP files will go into the **war** folder of the project.

```
<%@ page
import="com.google.appengine.api.blobstore.BlobstoreServiceFactory" %>
<%@ page import="com.google.appengine.api.blobstore.BlobstoreService"
%>

<%
    BlobstoreService blobstoreService =
BlobstoreServiceFactory.getBlobstoreService();
%>
<html>
    <head>
        <title>Tweet My Picture</title>
    </head>
    <body>
        <img src="tweet.png"/>
        <h1>Tweet My Picture</h1>
        <hr/>
        <h2>Upload a picture and send an automatic Tweet</h2>
        <form action="<%= blobstoreService.createUploadUrl("/upload")
%>" method="post" enctype="multipart/form-data">
            Twitter User Id : <input type="text"
name="twitter_userid"/><br/>
            Twitter Password : <input type="p"
name="twitter_password"/><br/>
            File :
            <input type="text" name="filename"/>
            <input type="file" name="myTweetPic"/>
            <input type="submit" value="Upload Picture"/>
        </form>
</html>
```

Let us look at the main points in the code above:

1. It has a HTML <form/> that accepts the twitter userid and password. We also have an <input/> element of type **file**, that allows the user to select the file from his machine to be uploaded.

2. The **action** attribute of the FORM is where you should pay attention. The Form action as you know submits the request parameters to a particular server side resource like a Servlet/JSP, etc. The action here is created via a helper function provided by the Blobstore API itself. Notice that first we are getting an instance of the Blobstore service via the BlobstoreServiceFactory as shown below:

BlobstoreService blobstoreService = BlobstoreServiceFactory.getBlobstoreService();

Then we are using a helper function called **blobstoreService.createUploadURL("SUCCESS_PATH")** for the action. Let us spend some time on that since it is important and will help us understand what the BlobstoreService does behind the scenes.

If you launch your local development server and navigate to the index.jsp page and do a view source on the HTML, you will find that the action that is generated by the **createUploadURL(…)** method looks something like this:

action="/_ah/upload/agp0d2VldG15cGljchsLEhVfX0Jsb2JVcGxvYWRTZXNzaW9uX18YEgw". You may wonder where did our SUCCESS_PATH i.e. /upload go away. Let me explain:

When you do an upload, the request hits an internal URL in the App Engine infrastructure that will do the following:

1. The BlobstoreService implementation inside of AppEngine will extract out the Blob and save it for you automatically in the datastore. You do not have to do anything special. It does this for the all the Blobs that are present in the **HTTP Request** stream.

2. On successfully saving these blobs into the stores, what it has with itself now are one or more name value pairs i.e. a Map. The name is the request parameter name for your blob. For e.g. in our index.jsp, the file that we uploaded had the name of **myTweetPic** i.e. the form element. And the value will be the Blob Key that you can use to retrieve the Blob. It will put this map into the Request stream i.e. it will augment the Request stream with this map and then invoke your SUCCESS_PATH url. The SUCCESS_PATH url is the end point or in our case the Servlet.

This Servlet can then extract out each of the Blobs that were saved by inspecting the Map and do further action. In our case, we will be posting a link to Twitter. This ervlet is what 0we shall see next.

Upload Servlet

The Upload Servlet code is listed below.

```
package com.gaejexperiments.twitter;

import java.io.IOException;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.mail.Session;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import twitter4j.Status;
import twitter4j.Twitter;

import com.google.appengine.api.blobstore.BlobKey;
import com.google.appengine.api.blobstore.BlobstoreService;
import com.google.appengine.api.blobstore.BlobstoreServiceFactory;

@SuppressWarnings("serial")
public class Upload extends HttpServlet {
 private final static Logger _logger =
Logger.getLogger(Upload.class.getName());
 private BlobstoreService blobstoreService =
BlobstoreServiceFactory.getBlobstoreService();



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

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

        if (blobKey == null) {
            res.sendRedirect("/");
        }
        else {
         String strStatus = "";
         try {
          String twitter_userid   =
(String)req.getParameter("twitter_userid");
          String twitter_password =
(String)req.getParameter("twitter_password");
          _logger.info("UserId:"+twitter_userid + " ,
Password:"+twitter_password);
          Twitter twitter = new
Twitter(twitter_userid,twitter_password);
```

```
          String strTweet = "Check out my picture at : " + "<a
href="http://localhost:8888/viewpic.jsp?blob-
key=&quot;+blobKey.getKeyString">http://localhost:8888/viewpic.jsp?blob
-key="+blobKey.getKeyString</a>();
             Status status = twitter.updateStatus(strTweet);
             strStatus = "Successfully updated the status to [" +
status.getText() + "].";
             _logger.info(strStatus);
         }
         catch (Exception ex) {
          strStatus = "Could not update Twitter Status : " +
ex.getMessage();
          _logger.log(Level.WARNING,strStatus);
         }
         finally {
             res.sendRedirect("/submitpic.jsp?blob-
key="+blobKey.getKeyString() + "&status="+strStatus);
         }
         }
    }
}
```

Let us go through the main points in the Upload Servlet code:

1. The Servlet implementation is the **doPost()** method. This method was invoked by the Blobstore Service after the blobs were successfully saved in the datastore.

2. As mentioned, the Blobstore Service augments the Request stream with a Map of successfully saved Blobs. It also provides a helper method that we can use to extract out this Map instance from the request stream. The code is shown below:

Map<String, BlobKey> blobs = blobstoreService.getUploadedBlobs(req);

3. The next thing to do is to get the Blob Key for our saved Blob. The Blob Key is unique and will be used to retrieve the Blob as we shall see later on in the **viewpic.jsp** code. The Key that we will use is the same as the input parameter name for our file blob that we provided in the FORM i.e. index.jsp.

BlobKey blobKey = blobs.get("myTweetPic");

4. We do a simple check to verify if the blobKey instance is not null. If not, we create a status update for twitter giving a url that points to the **viewpic.jsp** page which is passed the **blob-key** request parameter as shown below:

String strTweet = "Check out my picture at : " + "http://localhost:8888/viewpic.jsp?blob-key="+blobKey.getKeyString();

5. Finally, we using Twitter4J to post out tweet using the twitter id and password that we provided.

The status of our upload and tweet is then displayed by navigating to the **submitpic.jsp** page that simply shows the status and the picture that we uploaded.

submitpic.jsp

This JSP file is invoked by our Upload Servlet to display the status of our Tweet to twitter and the image that we uploaded as discussed in the previous section. You will notice an interesting thing over here and that is the **<img/>** element. The **source** attribute of the **<img/>** element is a servlet residing at an endpoint /serve. All it needs is the blob-key request parameter. We shall see this servlet in a moment.

```
<%@ page
import="com.google.appengine.api.blobstore.BlobstoreServiceFactory" %>
<%@ page import="com.google.appengine.api.blobstore.BlobstoreService"
%>

<html>
    <head>
        <title>Tweet My Picture - Submission</title>
    </head>
    <body>
        <img src="tweet.png"/>
        <h1>Tweet My Picture</h1>
        <hr/>
        <h3>Submission Result: <%=request.getParameter("status")%></h3>
        <%
           String blobKey = (String)request.getParameter("blob-key");
           if (blobKey != null) {%>
                You uploaded : <br/>
       <img width="200" height="150" src="<%="<a
href="http://localhost:8888/serve?blob-
key=&quot;+blobKey">http://localhost:8888/serve?blob-key="+blobKey</a>
%>">
           <%}%>
    </body>
</html>
```

Serve Servlet

This Servlet uses another helper function of the BlobStoreService. The method is called **serve** and it takes two parameters as shown in the listing below. The blob Key and the HTTP response stream. Note that we pass it the Blob Key that we got after saving our blob. What the serve method will do is that it will automatically retrieve the Blob from the Datastore using the Blob Key that was provided to it. Once it retrieves the Blob successfully, it will take the content of the Blob, set the correct MIME types and insert that into the **HTTP Response** stream. You can the use the Response stream to assign it as a source for the **HTML <img/>** element. We did exactly that in the submitpic.jsp file and we do the same in the **viewpic.jsp** file that will be invoked when someone clicks on the Twitter post.

```
package com.gaejexperiments.twitter;

import java.io.IOException;

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

import com.google.appengine.api.blobstore.BlobKey;
import com.google.appengine.api.blobstore.BlobstoreService;
import com.google.appengine.api.blobstore.BlobstoreServiceFactory;

@SuppressWarnings("serial")
public class Serve extends HttpServlet {
 private BlobstoreService blobstoreService =
BlobstoreServiceFactory.getBlobstoreService();

 public void doGet(HttpServletRequest req, HttpServletResponse resp)
throws IOException {
        BlobKey blobKey = new BlobKey(req.getParameter("blob-key"));
        blobstoreService.serve(blobKey, resp);
    }
}
```

viewpic.jsp

As discussed earlier, the Tweet that we post has the following format:

Check out my picture at http://localhost:8888/viewpic.jsp?blob-key=A_BLOB_KEY

This will invoke the **viewpic.jsp** file with a request parameter named **blob-key** which contains the Blob key value. So all we have to do i to create a **<img/>** element, whose **source** attribute will be served via the /serve endpoint as shown below. Simple isn't it ?

```
<%@ page
import="com.google.appengine.api.blobstore.BlobstoreServiceFactory" %>
<%@ page import="com.google.appengine.api.blobstore.BlobstoreService"
%>

<%
    BlobstoreService blobstoreService =
BlobstoreServiceFactory.getBlobstoreService();
    String blobKey = (String)request.getParameter("blob-key");
%>
<html>
    <head>
        <title>Tweet My Picture - View</title>
    </head>
    <body>
    <img width="200" height="150" src="<%="<a
href="http://localhost:8888/serve?blob-
key=&quot;+blobKey">http://localhost:8888/serve?blob-key="+blobKey</a>
%>">
    </body>
</html>
```

Configuring the Servlets

We need to add both the **Upload Servlet** and the **Serve Servlet** with their appropriate entries for **<servlet/>** and **<servlet-mapping/>** in the **web.xml** file. This file is present in the **WEB-INF** folder of the project. The necessary fragment to be added to your **web.xml** file are shown below.

```
 <servlet>
  <servlet-name>Upload</servlet-name>
  <servlet-class>com.gaejexperiments.twitter.Upload</servlet-class>
 </servlet>
 <servlet>
  <servlet-name>Serve</servlet-name>
  <servlet-class>com.gaejexperiments.twitter.Serve</servlet-class>
 </servlet>
 <servlet-mapping>
  <servlet-name>Upload</servlet-name>
  <url-pattern>/upload</url-pattern>
 </servlet-mapping>
 <servlet-mapping>
  <servlet-name>Serve</servlet-name>
  <url-pattern>/serve</url-pattern>
 </servlet-mapping>
```

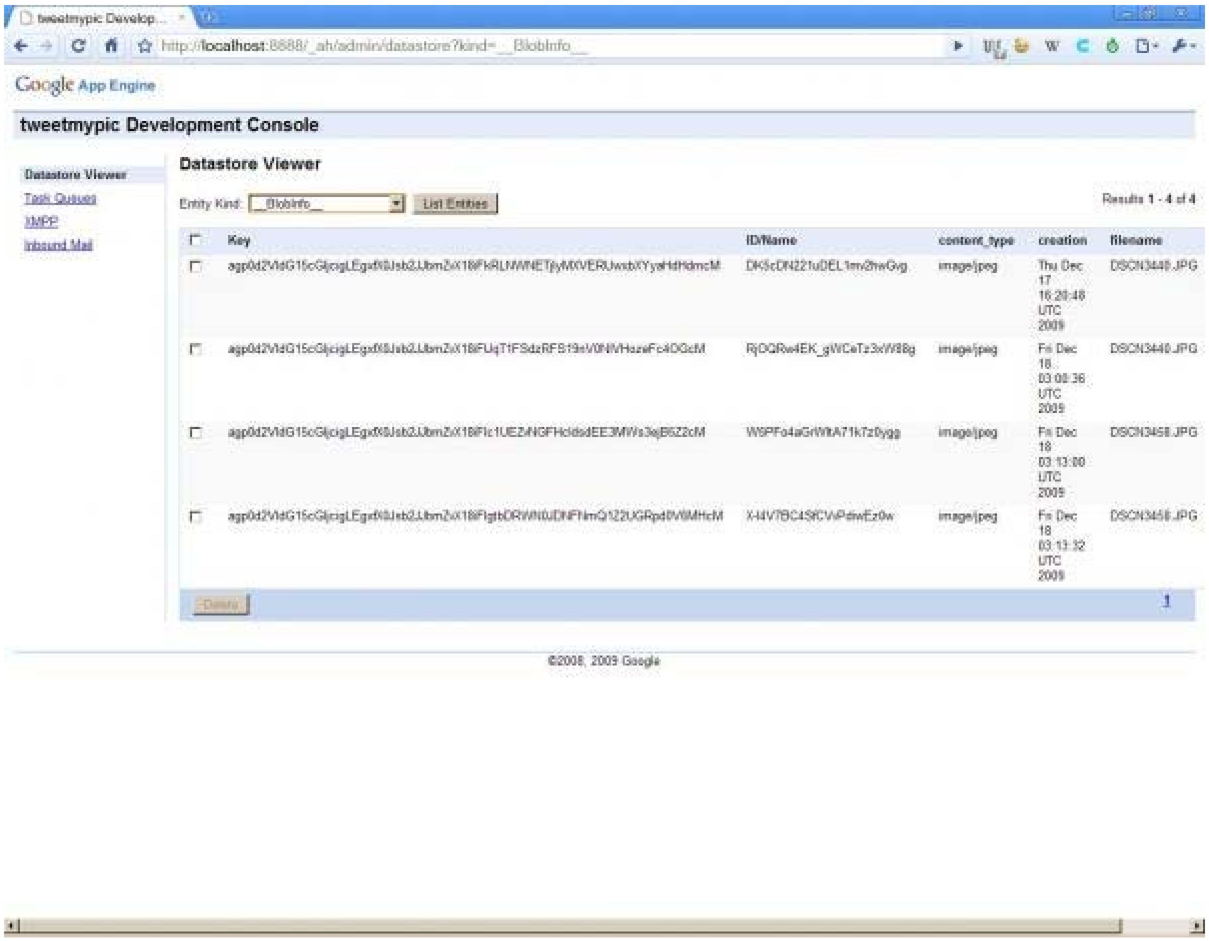
Deploying the application

Since we are going to be running this application locally itself, we simply need to validate it against the local development server. Assuming that the development is complete, simply run the Web Application from your Eclipse IDE and using a browser, navigate to http://localhost:8888/index.jsp. In my case, the port is 8888, but it could be different for you.

Please make sure that you are connected to the Internet and do have a valid Twitter userid and password.

Refer to the section above on **Tweet My Picture** in action for the flow. I will not repeat it here again.

An interesting thing that you could look at is to navigate to **http://localhost:8888/_ah/admin console.** Go to the DataStore viewer. If you have uploaded several images successfully, you will find instances of the Entity : __BlobInfo__ getting created. Click on the **List Entities** button to see the entities that you have uploaded so far. The **ID/Name** column is the Blob Key for your Blob. A sample screenshot from my development server instance is shown below:



http://localhost:8888/_ah/admin/datastore?kind=__BlobInfo__

tweetmypic Development Console

Datastore Viewer

Entity Kind: __BlobInfo__ List Entities

Results 1 - 4 of 4

Key	ID/Name	content_type	creation	filename
agp0d2VldG15cGljcigLEgxfX0Jsb2JJbmZvX18iFkRLNWNETjIyMXVERUwxbXYyaHdHdmcM	DK5cDN221uDEL1mv2hwGvg	image/jpeg	Thu Dec 17 16:20:48 UTC 2009	DSCN3440.JPG
agp0d2VldG15cGljcigLEgxfX0Jsb2JJbmZvX18iFUqT1FSdzRFS19nV0NIVHozeFc4OGcM	RjOQRw4EK_gWCeTz3xW88g	image/jpeg	Fri Dec 18 03:08:36 UTC 2009	DSCN3440.JPG
agp0d2VldG15cGljcigLEgxfX0Jsb2JJbmZvX18iFlc1UEZvNGFHcldsdEE3MWs3ejB6Z2cM	W9PFo4aGrWtA71k7z0ygg	image/jpeg	Fri Dec 18 03:13:00 UTC 2009	DSCN3458.JPG
agp0d2VldG15cGljcigLEgxfX0Jsb2JJbmZvX18iFlgtbDRWN0JDNFNmQ122UGRpd0V6MHcM	X-l4V7BC4SfCVvPdwEz0w	image/jpeg	Fri Dec 18 03:13:32 UTC 2009	DSCN3458.JPG

©2008, 2009 Google

Conclusion

This brings the episode to an end. In this episode we saw the Blobstore API service that has been introduced in version 1.3.0 of the App Engine SDK. The Blobstore API service is experimental at this point in time and could change but it gives us enough today to see how to create application that require blobs of large sizes like video and images. This episode showed a simple implementation called Tweet My Picture but the possibilities are endless. Do keep in mind that if you have to use Blobstore Service in a live application hosted at the appspot.com domain, then you will need to enable “billing” for your application.

Till the next episode, have fun with the Blobstore Service!

P.S: Just in case you were wondering… the picture of the lion was taken at a distance of 2 feet away from it in the National Park, Mumbai.

Episode 14: Writing a Chrome Extension powered by App Engine

Welcome to **Episode 14**. In this episode we shall cover how you can get started with writing Google Chrome Extensions. We shall also drive the Extension via our App Engine application. We have already seen in earlier episodes how your App Engine applications participated within Google Wave, Google Talk, etc. So this episode takes it further and shows how you can write a Google Chrome Extension whose data will be provided by our App Engine Application.

Google Chrome Extension

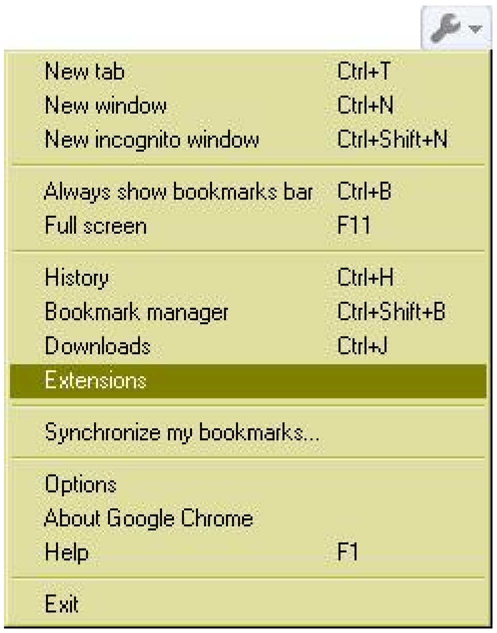
Google Chrome browser since its release has got a fair amount of developer mind share. Recently they addressed extensibility of the Chrome browser by announcing support for writing your own extensions. So the focus is now on the developer to take this extension mechanism, harness it and produce some great extensions that extend the capabilities of the browser. You can read here the official announcement for Extension support over **here**.

Google Chrome Dev Channel version

Note: The first thing that you need to do is download a copy of the Chrome Browser from their Dev Channel because extensions are currently supported there till they come into the mainstream release. So given that, make sure that you download that version of Google Chrome browser.

The Beta channel or the Dev channel is available at :
http://www.chromium.org/getting-involved/dev-channel.

Depending on your Operation System, download the appropriate version and install it. To check that you have extension enabled correctly, launch the Chrome Browser and click on the Settings icon as shown below. You should see the **Extensions** menu item available.

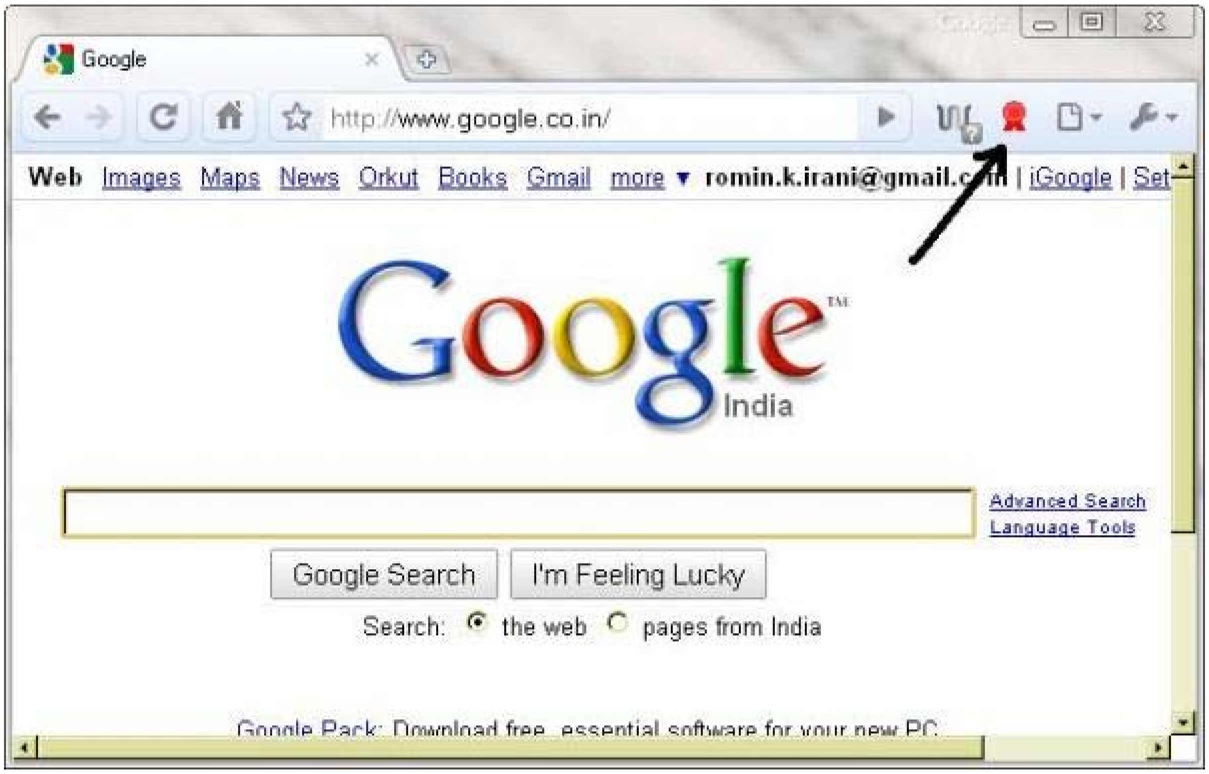


It is a good idea to take a look at the existing Chrome Extensions gallery that lists several excellent extensions that you can start using today inside of your Google Chrome browser. The gallery is available at Gallery : **https://chrome.google.com/extensions**. This page also contains a link where you can submit your extension too for the world to use!

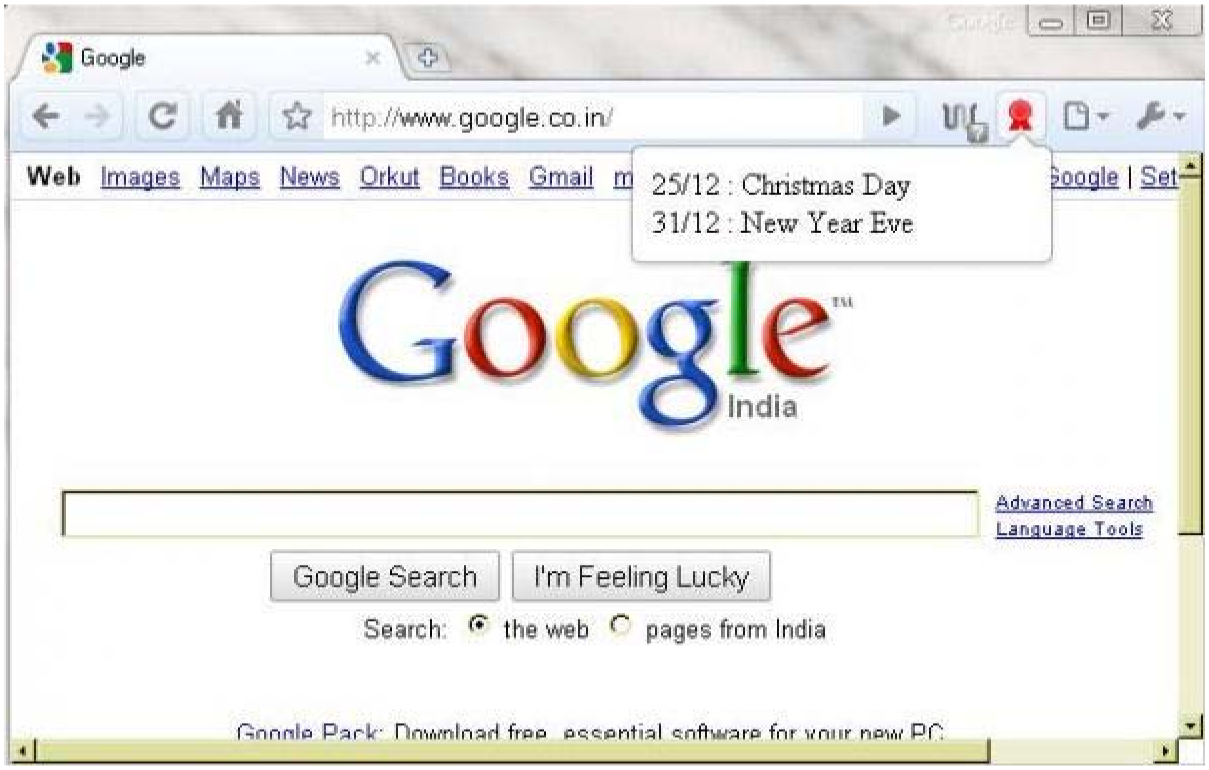
See it in Action

Let us see our Google Chrome Extension that we shall build in this episode. The extension is a simple **Reminder** application. It shows the current Reminders that you have set up for the month. All the data is saved and managed by our App Engine application and the Chrome Extension simply requests and fetches the data for the current month.

Shown below is the Chrome Browser screen with our Extension available as a little red icon in the Toolbar.


Google
http://www.google.co.in/
Web Images Maps News Orkut Books Gmail more ▾ romin.k.irani@gmail.com | iGoogle | Set
Google India
Advanced Search
Language Tools
Google Search
I'm Feeling Lucky
Search: the web pages from India
Google Pack: Download free, essential software for your new PC

When we click the icon, it will fetch the reminders that I have setup in the current month (December) and display them as shown below:



Sounds good? We shall split up the details into two parts:

Part I : The first part will deal with developing/deploying our Server side application. This application will be deployed as a Google App Engine Java application. We shall keep the implementation simple and not do any database operations, etc. The focus is on the second part where we need to understand what it takes to get a simple Chrome Extension up and running, and which is powered by this Google App Engine application.

Part II: The second part will deal with how you can write a Google Chrome Extension. We will consider the bare minimum that you need to do to get your extension working. I will leave the design and look/feel aspects to you.

Alright then, let’s get started.

Developing our Java Web Application : MyReminderApp

The first thing to do is to create a **New Google Web Application** Project. Follow these steps:

1. Either click on **File –> New –> Other** or **press Ctrl-N** to create a new project. Select Google and then Web Application project. Alternately you could also click on the New Web Application Project Toolbar icon as part of the Google Eclipse plugin.

2. In the New Web Application Project dialog, **deselect the Use Google Web Toolkit** and give a name to your project. I have named mine **MyReminderApp**. I suggest you go with the same name so that things are consistent with the rest of the article.

3. Click on **Finish**. This will generate the project and also create a sample Servlet named **MyReminderAppServlet** for you. We shall be modifying this Servlet code appropriately.

Design discussion

Now that the project template is ready, let me discuss in brief what we shall be building:

1. A simple Reminder App that contains a list of Reminders. Each Reminder is kept simple and has 3 attributed only: The day, month and the description. For e.g. [25,12,Christmas Day], [19,12,My Birthday], [1,1,New Year Day], [31,7,Pay Taxes] and so on.

2. So we will define a **Reminder.java** class that simply has getter/setters for the 3 attributes : day, month and description

3. We will then write a **ReminderService.java** class (A Singleton) that implements two simple functions:

- ArrayList<Reminder> **findRemindersByMonth**(String month)
  This method returns the reminders for a particular month. We pass in "1″ and we get reminders defined for January, etc.
- ArrayList<Reminder> **findCurrentMonthReminders**()
  This method returns the reminders for the current month. The method determines which month is currently on and retrieves reminders for that month only. And this is precisely the method that will be invoked when our extension asks for the current reminders.

4. Finally, we shall modify the **MyReminderAppServlet.java** so that it implements a simple REST like interface for retrieving the reminders. If the request stream **contains** a parameter named **month**, then the reminders will be returned for that month by invoking the **findRemindersByMonth** method. If the request stream **does not contain** a parameter named **month**, then the current month reminders will be returned by invoking the **findCurrentMonthReminders**() method. The response will be returned in **XML** format.

Given next are the code listings, that should be straight forward to understand now:

Reminder.java

```
package com.reminderapp;

public class Reminder {

private String day;
private String month;
private String event;

public Reminder(String day, String month, String event) {
super();
this.day = day;
this.month = month;
this.event = event;
}
public String getDay() {
return day;
}
public void setDay(String day) {
this.day = day;
}
public String getMonth() {
return month;
}
public void setMonth(String month) {
this.month = month;
}
public String getEvent() {
return event;
}
public void setEvent(String event) {
this.event = event;
}
}
```

The Reminder class simply has 3 attributes : day , month and year. There are getter/setter for each of these attributes and a constructor that takes in all the 3 attributes. We are keeping at as String for now to keep things simple.

ReminderService.java

```
package com.reminderapp;

import java.util.ArrayList;
import java.util.Calendar;

public class ReminderService {

private static ReminderService _instance = null;

private ArrayList<Reminder> _reminderList = null;
private ReminderService() {
initializeData();
}

public static ReminderService getInstance() {
if (_instance == null) {
_instance = new ReminderService();
}
return _instance;
}

public ArrayList<Reminder> findRemindersByMonth(String month) {
ArrayList<Reminder> _results = new ArrayList<Reminder>();
for (Reminder reminder : _reminderList) {
if (reminder.getMonth().equals(month)) {
_results.add(reminder);
}
}
return _results;
}

public ArrayList<Reminder> findCurrentMonthReminders() {
ArrayList<Reminder> _results = new ArrayList<Reminder>();
String currentMonth = "" +
(Calendar.getInstance().get(Calendar.MONTH)+1);
for (Reminder reminder : _reminderList) {
if (reminder.getMonth().equals(currentMonth)) {
_results.add(reminder);
}
}
return _results;
}

private void initializeData() {
_reminderList = new ArrayList<Reminder>();
Reminder R1 = new Reminder("1","1","New Year Day");
Reminder R2 = new Reminder("26","1","Republic Day");
Reminder R3 = new Reminder("15","8","Independence Day");
Reminder R4 = new Reminder("25","12","Christmas Day");
Reminder R5 = new Reminder("31","12","New Year Eve");

_reminderList.add(R1);
_reminderList.add(R2);
_reminderList.add(R3);
_reminderList.add(R4);
_reminderList.add(R5);

}

}
```

The ReminderService is a singleton class. And we are initializing some data here via the **initializeData**() call. We are not complicating things by using a database here but this is something that should ideally be there in a real application. As you can notice we add 5 sample records and for the current month i.e. December (12), we have two events. The two methods **findRemindersByMonth** and **findCurrentMonthReminders** are straightforward in which it goes through the reminderList and matches the records as appropriate.

MyReminderAppServlet.java

```
package com.reminderapp;

import java.io.IOException;
import java.util.ArrayList;

import javax.servlet.http.*;

@SuppressWarnings("serial")
public class MyReminderAppServlet extends HttpServlet {
public void doGet(HttpServletRequest req, HttpServletResponse resp)
throws IOException {
resp.setContentType("text/xml");
//resp.getWriter().println("Hello, world");

String month = (String)req.getParameter("month");
ArrayList<Reminder> _results = null;
if (month == null) {
_results = ReminderService.getInstance().findCurrentMonthReminders();
}
else {
_results = ReminderService.getInstance().findRemindersByMonth(month);
}

StringBuilder SB = new StringBuilder();
SB.append("<?xml version=\"1.0\"?>");
SB.append("<ReminderList>");
for (Reminder reminder : _results) {
SB.append("<Reminder>"+reminder.getDay()+"/"+reminder.getMonth()+" : " +
reminder.getEvent()+"</Reminder>");
}
SB.append("</ReminderList>");
resp.getWriter().println(SB.toString());
}
}
```

Let us go through the code for the MyReminderAppServlet in brief:

1. It implements the doGet(…) method and inspects if a request parameter named month is passed or not.

2. If the month parameter is passed, its value is extracted and the findRemindersByMonth method is invoked. Else the findCurrentMonthReminders is invoked.

3. The result is then formatted into a simple XML structure and then it is returned back in the response stream.

Configuring the Servlet

We need to make sure that the entry for **<servlet/>** and **<servlet-mapping/>** in the **web.xml** file. is there for MyReminderAppServlet. This file is present in the **WEB-INF** folder of the project. The necessary fragment to be added to your **web.xml** file are shown below.

```
<servlet>
<servlet-name>MyReminderApp</servlet-name>
<servlet-class>com.reminderapp.MyReminderAppServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>MyReminderApp</servlet-name>
<url-pattern>/myreminderapp</url-pattern>
</servlet-mapping>
```

Deploying the Application

To deploy the application, you will need to first create your **Application ID**. The Application Identifier can be created by logging in at **http://appengine.google.com** with your Google Account. You will see a list of application identifiers already registered under your account (or none if you are just getting started). To create a new Application, click on the Create Application button and provide the Application Identifier as requested. Please note this name down since you will be using it for deployment.

For e.g. I have registered an application identifier named **itsareminder**.

To deploy the application, follow these steps (they should be familiar to you now):

1. Click on the Deploy Icon in the Toolbar.
2. In the Deploy dialog, provide your Email and Password. Do not click on Deploy button yet.
3. Click on the **App Engine Project settings** link. This will lead you to a dialog, where you need to enter your Application ID [For e.g. my Application Identifier **itsareminder**]
4. Click on **OK**. You will be lead back to the previous screen, where you can click on the**Deploy** button. This will start deploying your application to the GAEJ cloud. You should see several messages in the Console window as the application is being deployed.
5. Finally, you should see the message “Deployment completed successfully”.

Testing our Reminder Service

To test our Reminder Service, we can use the http://<AppId>.appspot.com/myreminderapp URL. You can replace it with your AppId and endpoint as appropriate to your application. For e.g. I have hosted it at http://itsareminder.appspot.com/myreminderapp and putting this in the browser returns me the Reminders that are set for the current Month i.e. Month of December. I get the following response XML:

```
<ReminderList><Reminder>25/12 : Christmas Day</Reminder><Reminder>31/12
: New Year Eve</Reminder></ReminderList>
```

Now that the development of the Reminder Service is complete, we can move on to the next part i.e. developing the Chrome Extension.

Developing the Reminder Service Chrome Extension

The **official documentation** on developing a Chrome Extension is quite good and you can refer it for more details. I will keep things to a minimum here and cover only what is required to get the Chrome Extension up and running. Follow these steps:

1. Create a folder on your machine. For e.g. I have created the following folder on my machine in C:\. It is **c:\MyReminderExtension.** All the necessary files that we create for the extension will reside in this folder.

2. To create an extension, we need at the minimum **3 files**. The three files are an icon file for your extension. This icon will be visible in the toolbar as we saw earlier. The second file is a **manifest file** called **manifest.json**. The manifest provides important metadata about the extension like name,version, permissions, action, etc. And the **final file** is the **html file** in which you will keep the core of your extensions code written in Javascript.

3. So the first thing that I do is to create a manifest file for the extension. The manifest.json file for the Reminder Service is shown below:

```
{
"name": "My Reminder Extension",
"version": "1.0",
"description": "Reminders based on month",
"browser_action": {
"default_icon": "myreminder.png",
"popup": "reminder.html"
},
"permissions": [
"http://itsareminder.appspot.com/"
]
}
```

As you can see this is pretty simple. There is the standard name/version/description that you can give as per your preferences. The **browser_action** has two important attributes. The default_icon refers to the icon file that contains the icon for our extension. So this means that we will have the **myreminder.png** file present in the same folder. The **popup** attribute indicates which **HTML** file to be invoked when the icon is clicked. The **permissions** attribute indicates which domain the extension will be accessing to retrieve the data. In our case it is the http://itsareminder.appspot.com domain at which we hosted our application.

4. As shown above, we will create a file named myreminder.png in the same folder i.e. C:\MyReminderExtension.

5. Finally, we have the **reminder.html** file that will contain the code of the extension. It is listed below:

```
<style>
body {
min-width:200px;
overflow-x:hidden;
}

</style>

<script>
var req = new XMLHttpRequest();
req.open(
"GET",
"http://itsareminder.appspot.com/myreminderapp",
true);
req.onload = showReminders;
req.send(null);

function showReminders() {
var reminders = req.responseXML.getElementsByTagName("Reminder");

for (var i = 0, reminder; reminder = reminders[i]; i++) {
var reminderDIV = document.createElement("div");
reminderDIV.innerHTML = reminder.textContent;
document.body.appendChild(reminderDIV);
}
}

</script>
```

The code above is focussed around the <script> tag. It requires knowledge about AJAX and the XMLHttpRequest object along with some Javascript basics. The core pieces are mentioned below:

1. We use the XMLHttpRequest to make a call to our Reminder Service. The end point is where our servlet is present to serve the requests i.e. http://itsareminder.appspot.com/myreminderapp.

2. When we get the response XML, we call the **showReminders()** method. This method simply parses the <Reminder> tags and for each tag, it creates a **<div>** element in which the text content is the Reminder Event Text returned. Its appends each of the **<div/>** elements to the document.body as shown.

Simple isnt it? Now let us see how to install this extension into your Chrome browser to test it out.

Installing the Reminder Chrome Extension

At this point in time, I will assume that you have the following 3 files created in the C:\MyReminder folder:

- manifest.json
- myreminder.png
- reminder.html

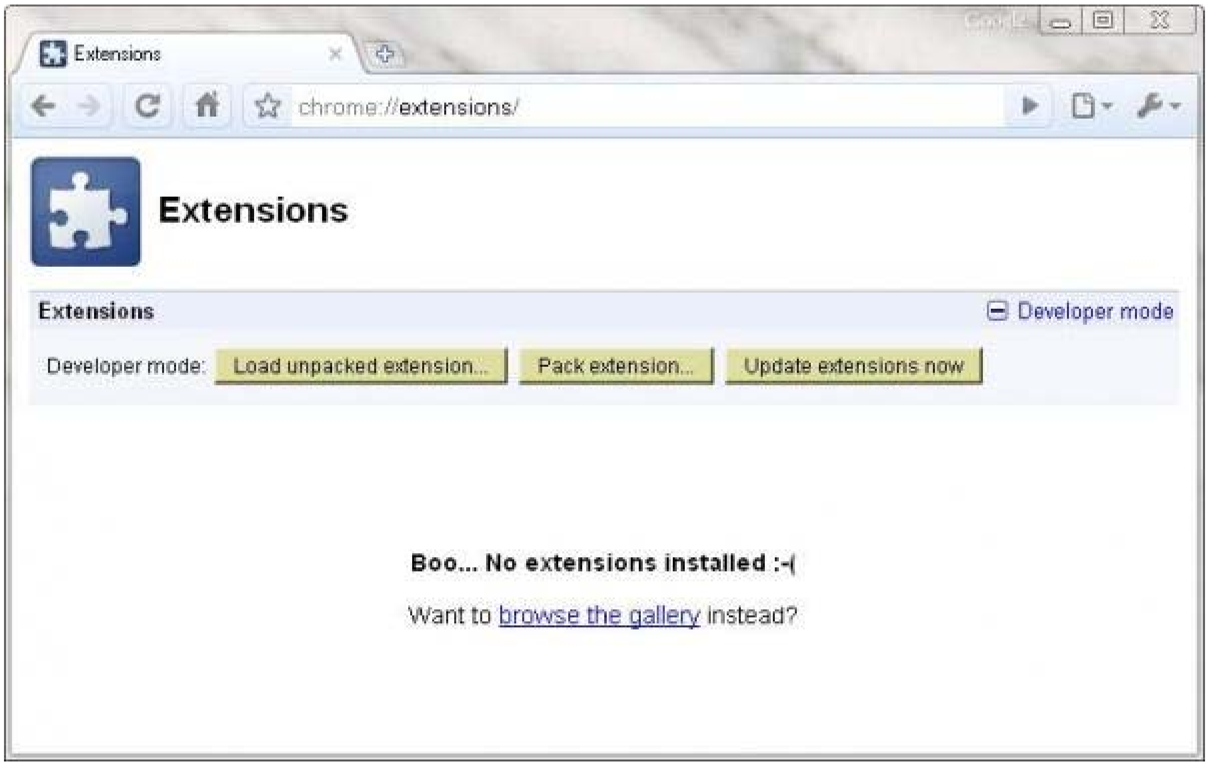
You could have created different files but just remember that you need the icon, manifest and your html file that implements the Chrome extension code.

Once you have that, follow these steps:

1. Launch Google Chrome Browser. You need to go the Extensions page that allows you to view the current extensions installed and from where you can either install new extensions/reload/uninstall,etc. There are two ways to go to that page:

- In the address, type in **chrome://extensions**
- From the settings option as mentioned in the start of this article, click on the **"Extensions"** menu option.

2. You should see a screen that looks something like this:

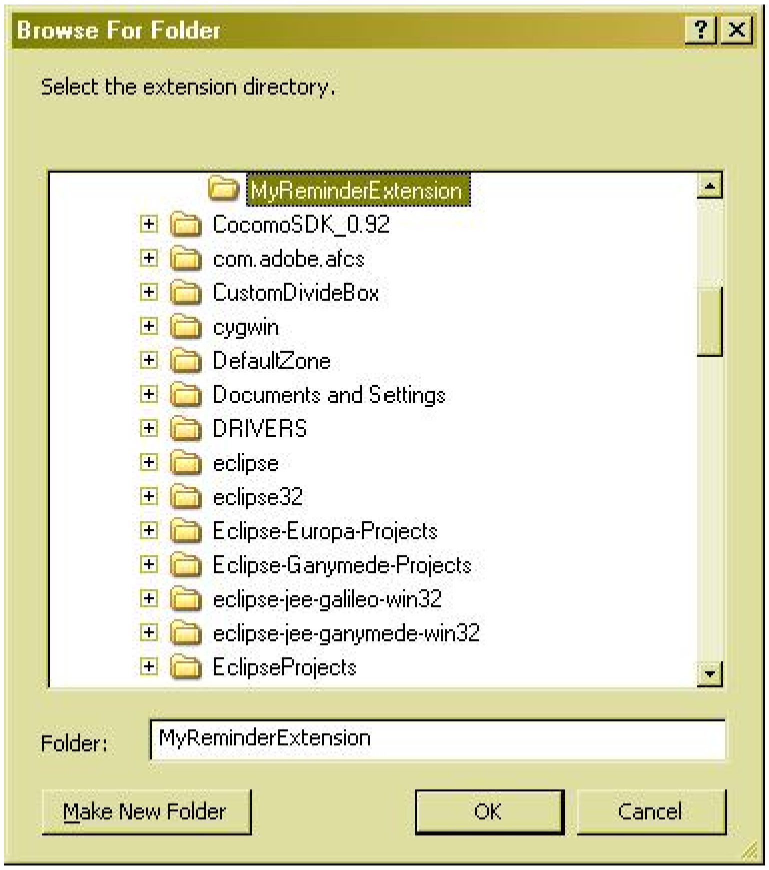

Extensions
chrome://extensions/
Extensions
Extensions
Developer mode
Developer mode: Load unpacked extension... Pack extension... Update extensions now

Boo... No extensions installed :-(

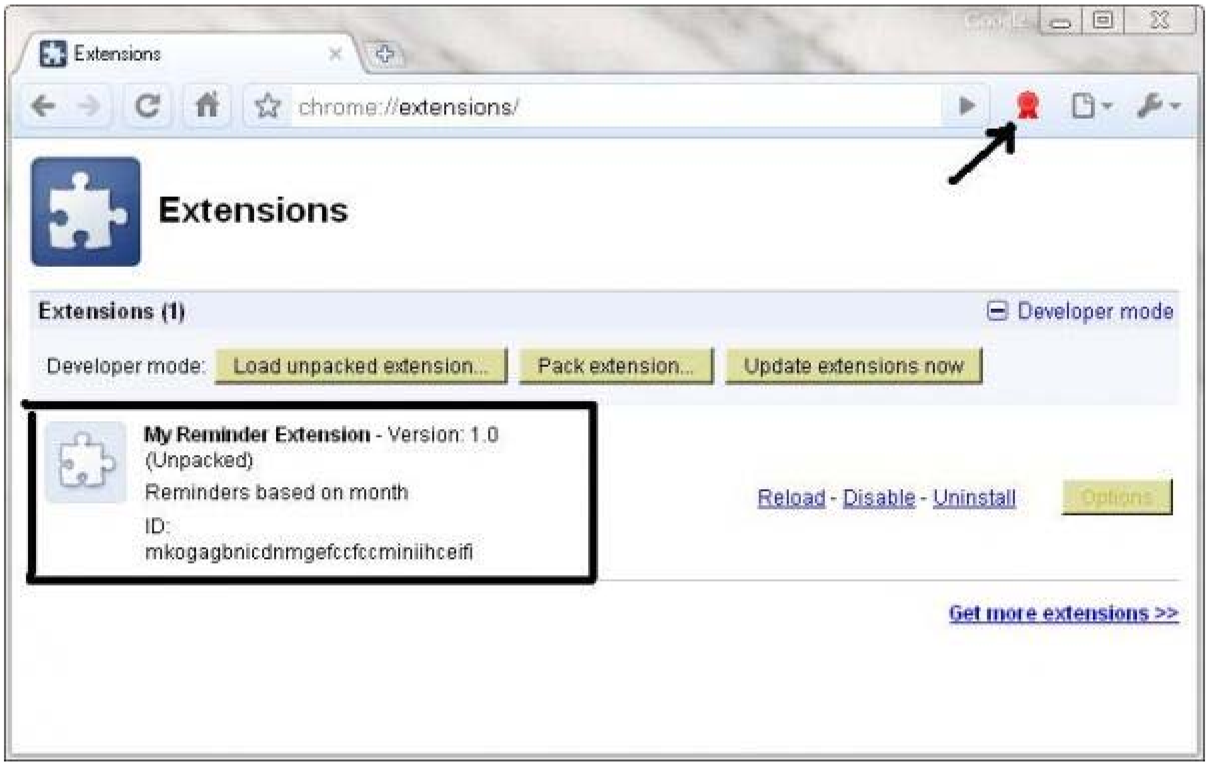
Want to browse the gallery instead?

What this means is that there are no extensions currently installed in my Chrome browser. If there were any, they would be listed over here.

3. Click on the **Load unpacked extension** button. This will bring up a dialog box from where you can navigate to the folder where your extension files are present. In our case, it was the **C:\MyReminderExtension** and so I navigate to that and click on **OK** as shown below:

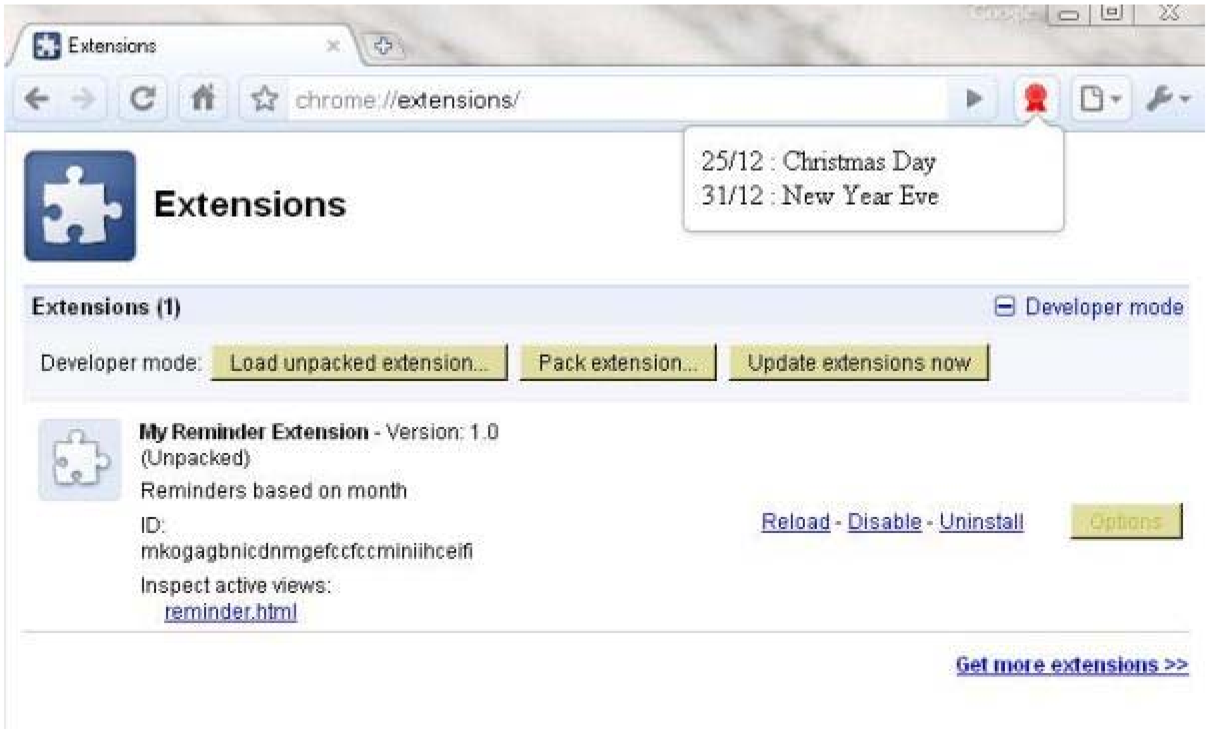


4. This will install the Chrome Extension in the browser as shown below:



My Reminder Extension - Version: 1.0
(Unpacked)
Reminders based on month
ID:
mkogagbnicdnmgefccfccminiihceifi

5. Finally, we can test out the browser by clicking on the icon and it will display the list of Reminders for the current month.



25/12 : Christmas Day
31/12 : New Year Eve

My Reminder Extension - Version: 1.0
(Unpacked)
Reminders based on month
ID:
mkogagbnicdnmgefccfccminiihceifi
Inspect active views:
reminder.html

If you have newer versions, all you need to do is simply click on **Reload** option for the extension. Alternately, you can also **Uninstall** and load it again.

Some points to note:

1. You have not completely installed this extension. What this means is that if you close the Chrome Browser and launch it again, this Extension will not be visible.

2. The next step would be to visit the official documentation at http://code.google.com/chrome/extensions/getstarted.html and also go to the gallery and submit your Extension. The gallery is present at : https://chrome.google.com/extensions and you will find a link to submit your extension. Click that and follow the steps to submit your extension. All you will need to provide from your side will be a ZIP file of the folder that we mentioned above.

Conclusion

This concludes the final episode of the year 2009. In this episode we covered how to write a Google Chrome Extension and power it via a Google App Engine application.

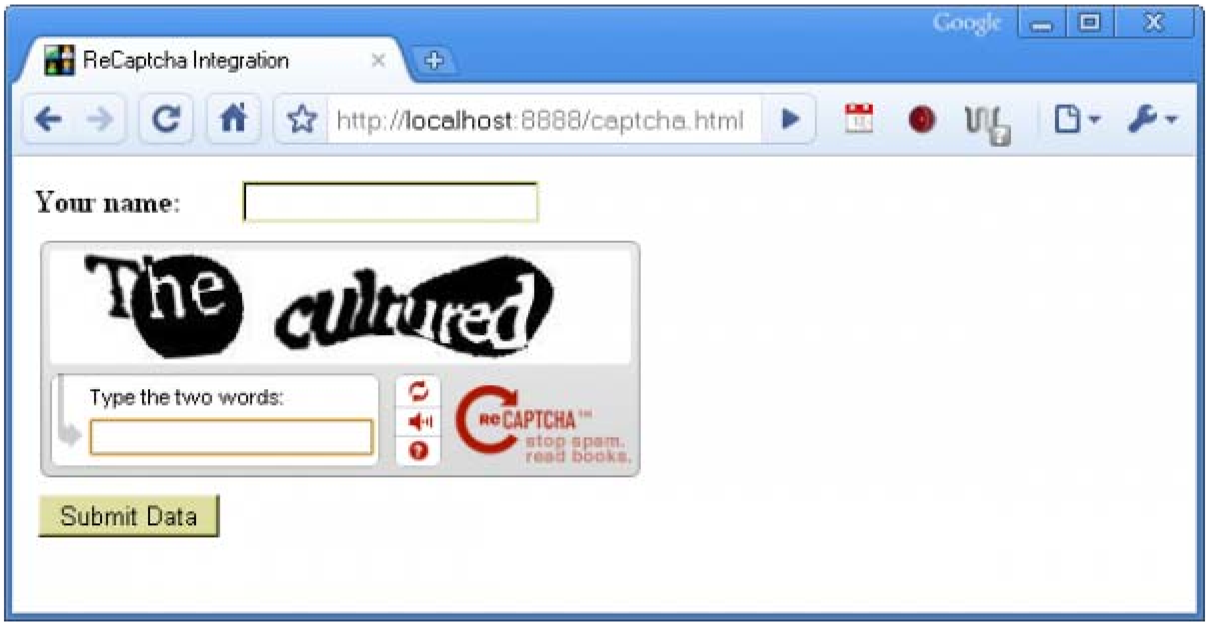
I hope that the new year takes you further into writing Cloud applications using Google App Engine. I will continue to explore tools/languages/APIs/extensions that work in coherence with the Google App Engine. The journey or might I say “Experiments” are far from over. Have a great 2010 and thanks for reading this blog. Its been a joy to share my experiments with all of you.

Episode 15: Using a CAPTCHA in your Google App Engine Application

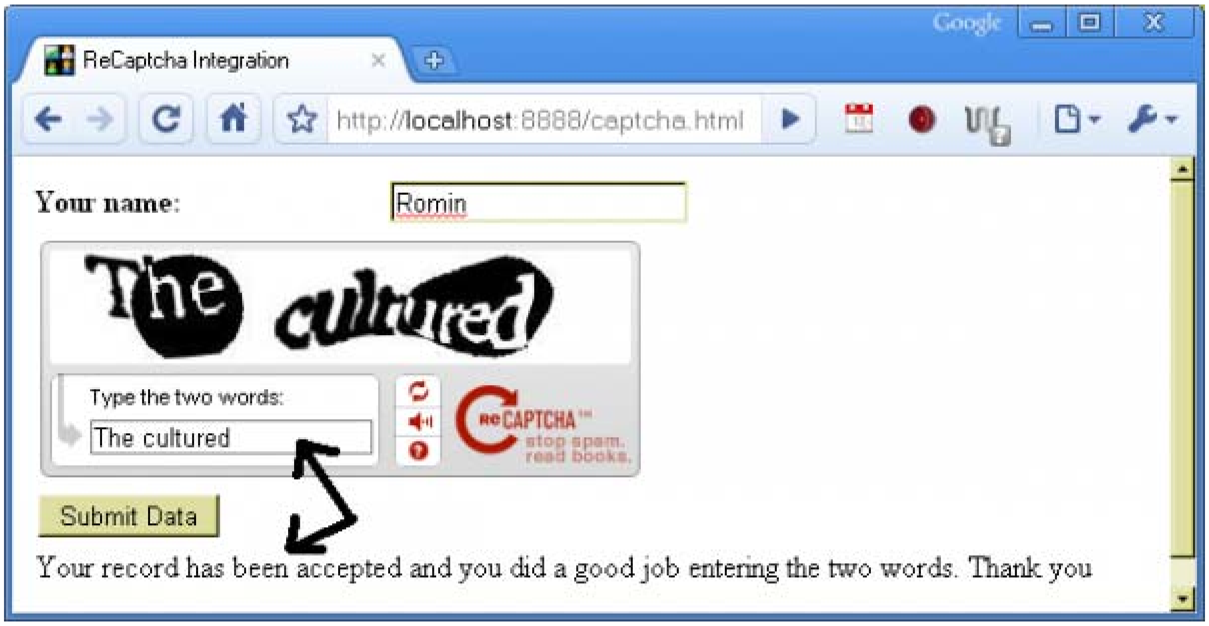
Welcome to **Episode 15**. In this episode, we shall cover how to incorporate a **CAPTCHA** in your application. The **CAPTCHA** solution that I will be demonstrating over here is the ReCAPTCHA project found **here**. To quote from its Wikipedia entry, a CAPTCHA is a type of challenge-response test used in computing to ensure that the response is not generated by the computer. Usually, one sees one or two words shown to us that we need to enter and they need to match before our request submission is accepted by the server.

See it in Action

Let us first see the application that we shall be developing in its final form. The main screen is a dummy form that accepts one field and displays the CAPTCHA to the user as shown below:



If the CAPTCHA response is entered **correctly** and the submit button is clicked, the response is sent to the server. The server side validates the CAPTCHA challenge-response and will display an “all is well” message as shown below:

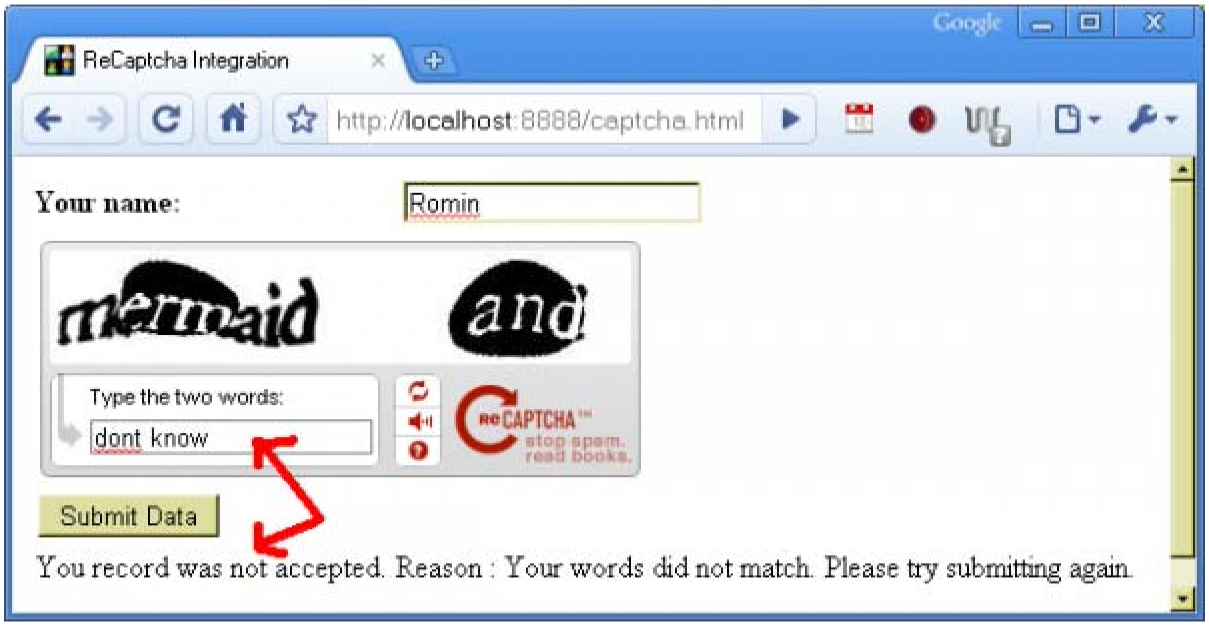


Your name: Romin

The cultured

Your record has been accepted and you did a good job entering the two words. Thank you

If the CAPTCHA response is **not entered correctly**, the server side will display an "all is not well" message as shown below:



Your name: Romin

dont know

You record was not accepted. Reason : Your words did not match. Please try submitting again.

ReCAPTCHA Project

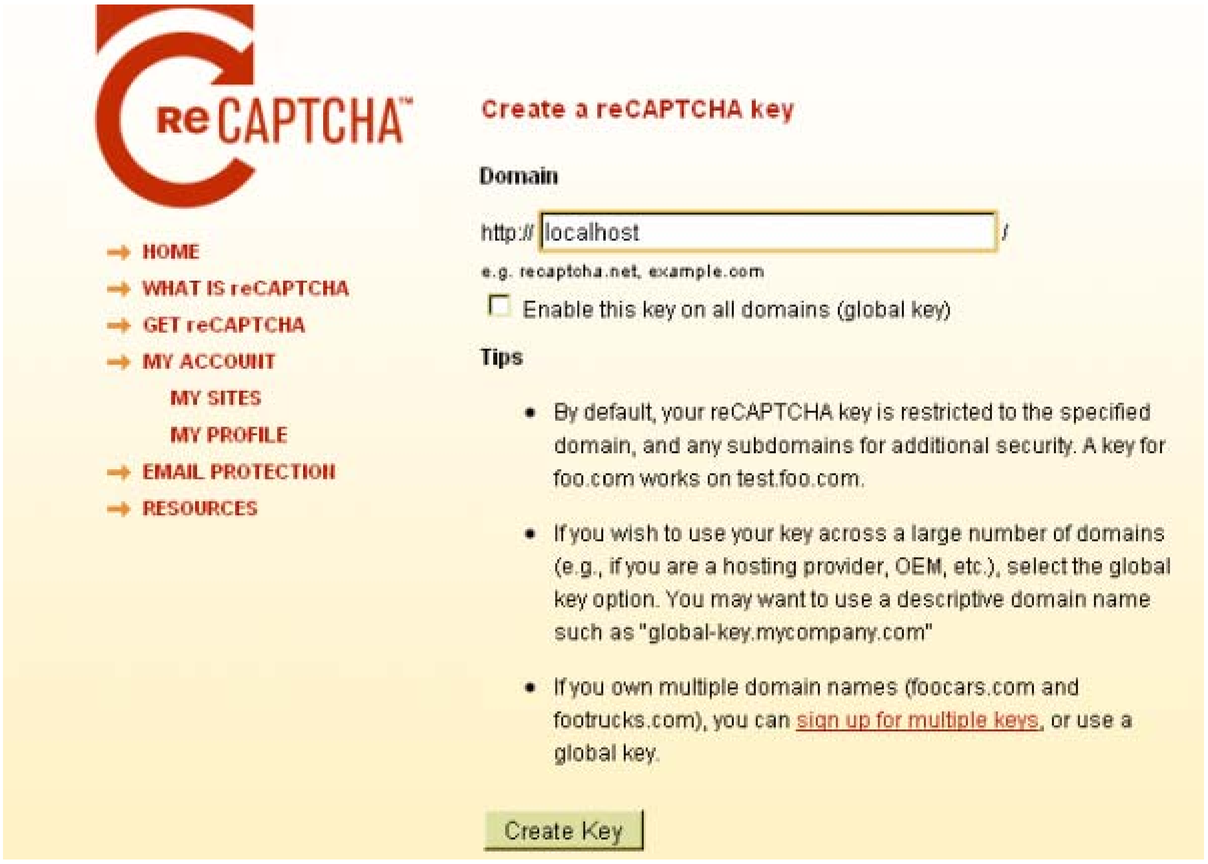
We shall be using the ReCAPTCHAProject present at http://recaptcha.net/. The ReCAPTCHAproject is comprehensive and provides support for a variety of server side platforms, hence I have gone with it. I do not have a particular preference for it, except that most people recommended this to me and hence I decided to try it out. I was able to integrate it easily and quickly into a recent App Engine application that is currently live, so I can vouch for its ease of integration and which is what I shall demonstrate in this experiment.

To get started with ReCAPTCHA, we need to do 2 things:

1. Get your private/public keys for the site name hat your application will be running on.

Follow these steps:

1. First get yourself a ReCAPTCHAaccount. Go to http://recaptcha.net/whyrecaptcha.html and click on the **Sign up Now!** button.
2. This will bring up a login form, but you click on the "New to reCaptcha? Sign up" link. Here you can register by giving a username/password/email . Click on the Sign up now button.
3. On successful registration, you will signed in automatically and led to a **Create a reCAPTCHA key** page. Here you need to enter the site name that you will be using ReCAPTCHAon. Remember you can get ReCAPTCHAkeys for more than 1 site and you can manage it all from your account itself. For e.g. you can get a key for localhost since that will be what we need to test on first before deploying to the Google App Engine cloud. And then you will atleast need one based on the Application ID of your Google App Engine application. So for e.g. if my application ID is **gaejexperiments,** then my full site url is **http://gaejexperiments.appspot.com**. So you will need to enter **gaejexperiments.appspot.com** in the **Domain** form field.
4. Click on **Create Key**. For e.g. shown below is the series of screens that I got when I wanted **keys** for **localhost. You will need to do the same if you wish to test the code first on your local system i.e. localhost.**



Create a reCAPTCHA key

Domain

http:// localhost /

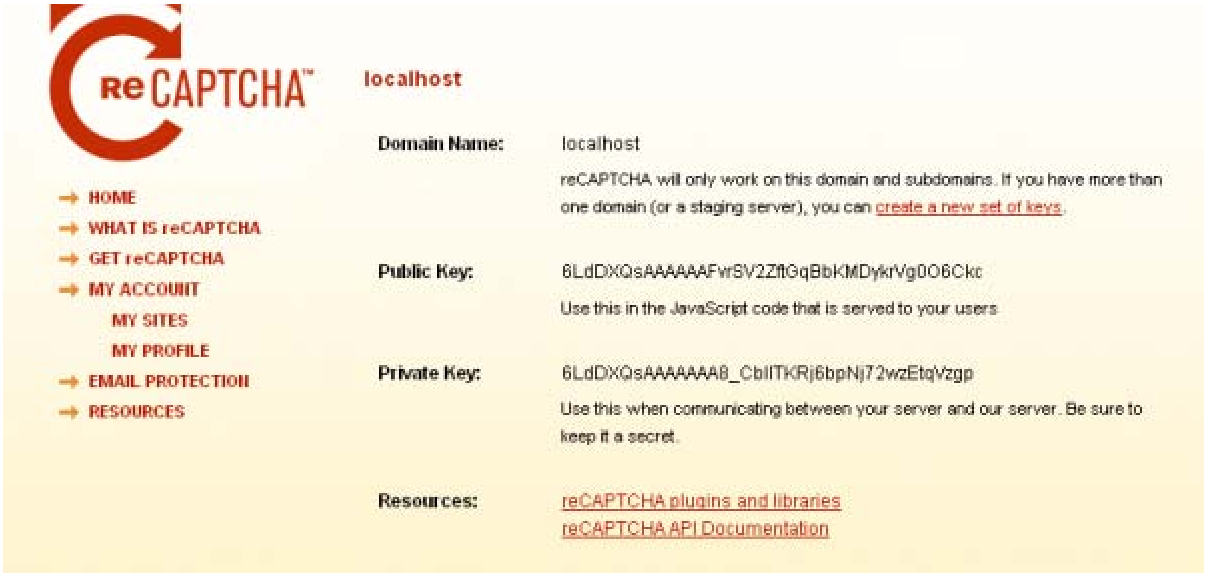
e.g. recaptcha.net, example.com

☐ Enable this key on all domains (global key)

Tips

- By default, your reCAPTCHA key is restricted to the specified domain, and any subdomains for additional security. A key for foo.com works on test.foo.com.
- If you wish to use your key across a large number of domains (e.g., if you are a hosting provider, OEM, etc.), select the global key option. You may want to use a descriptive domain name such as "global-key.mycompany.com"
- If you own multiple domain names (foocars.com and footrucks.com), you can sign up for multiple keys, or use a global key.

Then when you click **Create Key,** you will be presented the **Public Key** and **Private Key** as shown below:



localhost

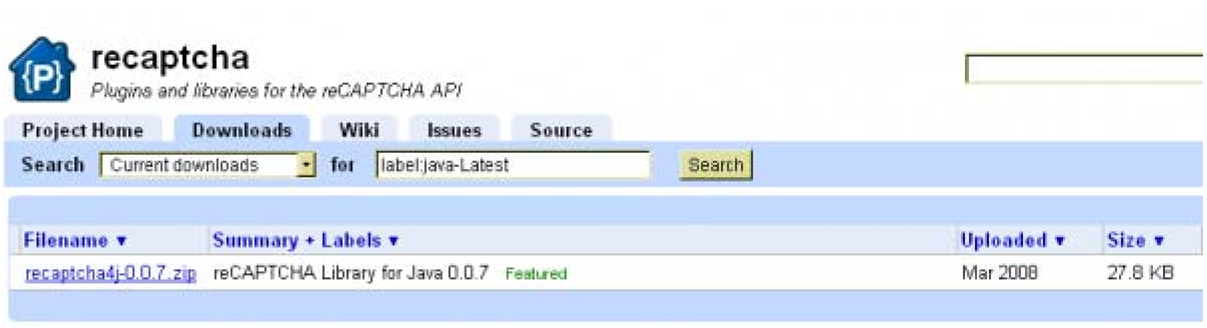
Domain Name:	localhost reCAPTCHA will only work on this domain and subdomains. If you have more than one domain (or a staging server), you can create a new set of keys.
Public Key:	6LdDXQsAAAAAAFwSV2ZftGqBbKMDykrVg0O6Ckc Use this in the JavaScript code that is served to your users
Private Key:	6LdDXQsAAAAAAB_CbIITKRj6bpNj72wzEtqVzgp Use this when communicating between your server and our server. Be sure to keep it a secret.
Resources:	reCAPTCHA plugins and libraries reCAPTCHA API Documentation

Please note down the Public and Private Key since you will need to use them in your application. The usage is straightforward and mentioned on the screen and I will be repeat it here:

- Use the Public Key in your HTML page (Javascript) to communicate with the ReCAPTCHA server to get the challenge (image).
- Use the Private Key in your server side Java code that will be used to communicate with the Server to verify the challenge + response submitted by the form in the above step.

2. Download the JAR file that we shall be using at the Server side to verify the CAPTCHA challenge and response.

To do the server side validation, you need to download a JAR file that is available from the following url : http://code.google.com/p/recaptcha/downloads/list?q=label:java-Latest. Go to the URL and you shall be presented with a screen as shown below:



Filename	Summary + Labels	Uploaded	Size
recaptcha4j-0.0.7.zip	reCAPTCHA Library for Java 0.0.7 Featured	Mar 2008	27.8 KB

The JAR file is required and it encapsulates all communication between our application and the ReCAPTCHA Servers.

Download the zip file, expand it in an appropriate folder and you will find the **recaptcha4j-0.0.7.jar** present in that. We will need it later on to be referenced in your App Engine Eclipse Project.

Developing our Application

The first thing to do is to create a **New Google Web Application** Project. Follow these steps:

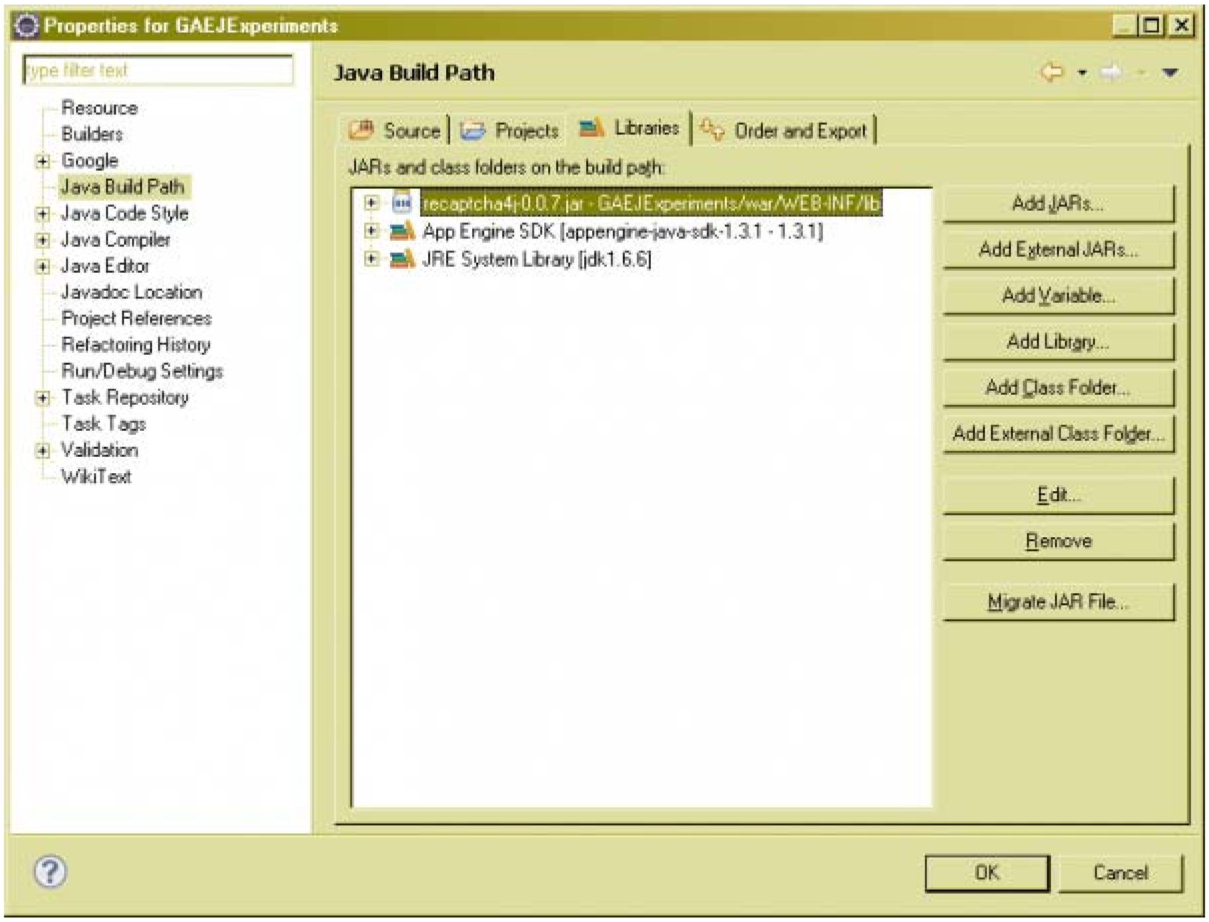
1. Either click on File –> New –> Other or press Ctrl-N to create a new project. Select Google and then Web Application project. Alternately you could also click on the New Web Application Project Toolbar icon as part of the Google Eclipse plugin.
2. In the New Web Application Project dialog, deselect the Use Google Web Toolkit and give a name to your project. I have named mine GAEJExperiments. I suggest you go with

the same name so that things are consistent with the rest of the article, but I leave that to you. **In case you are following the series, you could simply use the same project and skip all these steps altogether. You can go straight to the Servlet Development section.**
3. Click on Finish.

This will generate the project and also create a sample Hello World Servlet for you. But we will be writing our own Servlet.

Add the recaptcha4j-0.0.7.jar file to your Project classpath

The recaptcha4j-0.0.7.jar file that we downloaded needs to be added to the runtime and compile time CLASSPATH of your App Engine Eclipse Project. Follow these steps:
a) Copy **recaptcha4j-0.0.7.jar** to the **WEB-INF\lib** folder of your project. This folder is present in the **war** sub folder of your main project structure.
b) Go to Project properties –> Java Build Path and reference the JAR file (**recaptcha4j-0.0.7.jar**) in the Java Build Path of your application as shown below:



JARs and class folders on the build path:
recaptcha4j-0.0.7.jar - GAEJExperiments/war/WEB-INF/lib
App Engine SDK [appengine-java-sdk-1.3.1 - 1.3.1]
JRE System Library [jdk1.6.6]

The Front end HTML form [captcha.html]

```
<html xmlns="<a
href="http://www.w3.org/1999/xhtml">http://www.w3.org/1999/xhtml</a>"
lang="en" xml:lang="en">
<head>
 <title>ReCaptcha Integration</title>
<script type="text/javascript">
 function PreloadCaptcha() {
  showRecaptcha();
 }


function showRecaptcha() {
    Recaptcha.create("YOUR_PUBLIC_KEY", "dynamic_recaptcha_1", {
          theme: "white",
          callback: Recaptcha.focus_response_field
    });
  }


 var xmlhttp;
 function submitFormData(name)
 {

 //alert("Message");
 xmlhttp=null;
 if (window.XMLHttpRequest)
   {// code for IE7, Firefox, Opera, etc.
   xmlhttp=new XMLHttpRequest();
   }
 else if (window.ActiveXObject)
   {// code for IE6, IE5
   xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
   }
 if (xmlhttp!=null)
   {
   xmlhttp.onreadystatechange=state_Change;
   var url = "postformdata";
   var params = "name="+name+"&recaptcha_challenge_field=" +
Recaptcha.get_challenge() +
"&recaptcha_response_field="+Recaptcha.get_response();
   var status = document.getElementById("status");
   status.innerHTML = "<img src='img/ajax-loader.gif'><b>Submitting
your data. Please wait...</b>";
   xmlhttp.open("POST",url,true);
   xmlhttp.setRequestHeader("Content-type", "application/x-www-form-
urlencoded");
   xmlhttp.setRequestHeader("Content-length", params.length);
   xmlhttp.setRequestHeader("Connection", "close");
   xmlhttp.send(params);
   }
 else
```

```
   {
   alert("Your browser does not support XMLHTTP.");
   }
 }

 function state_Change()
 {
 if (xmlhttp.readyState==4)
   {// 4 = "loaded"
   if (xmlhttp.status==200)
     {
     // 200 = "OK"
    var status = document.getElementById("status");
    status.innerHTML = xmlhttp.responseText;;
    Recaptcha.reload();
    setTimeout(function() {
      status.innerHTML = "";
    }, 3000);
     }
   else {
    var status = document.getElementById("status");
    status.innerHTML = xmlhttp.responseText;;
    Recaptcha.reload();
    setTimeout(function() {
      status.innerHTML = "";
    }, 3000);
   }
   }
 }
 </script>
</head>

<body onload="PreloadCaptcha()">
       <FORM NAME="dataform">
       <TABLE>
     <TR>
     <TD><b>Your name:</b></TD>
     <TD><INPUT NAME="txtName"/></TD>
     </TR>
     <TR>
     <TD colspan="2"><div id="dynamic_recaptcha_1"></div></TD>
     </TR>
     <TR>
                    <TD colspan="2"><INPUT type="button" value="Submit
Data" name="btnSubmitData" onClick="submitFormData(txtName.value);
return true"></TD>
                    </TR>
                    <TR>
     <TD colspan="2"><div id="status"/></TD>
     </TR>
       </TABLE>
       </FORM>
</body>
</html>
```

Let us go through the important parts of the code:

1. The first part to notice is that I have referenced the javascript file for the ReCAPTCHA code as shown :

```
<script type="text/javascript"
src="http://api.recaptcha.net/js/recaptcha_ajax.js"></script>
```

2. There is a javascript function call being made on the **load** of the body (<body onload="**PreloadCaptcha**()">). This invokes the function **showRecaptcha()** that is shown below:

```
Recaptcha.create("YOUR_PUBLIC_KEY", "dynamic_recaptcha_1", {
          theme: "white",
          callback: Recaptcha.focus_response_field
    });
```

The Recaptcha class is available from the javascript file and we use the create method. The first parameter is the PUBLIC_KEY that you get when you registered at ReCaptcha for your site name. If you are testing it locally, this will be the Public Key for your localhost site. The second parameter is a **DIV** element that is present in the form.

The create() method on successful invocation will populate the **DIV element** with the CAPTCHA challenge/response fields.

3. The rest of the code is standard AJAX stuff. The submit button invokes the **submitDataForm()** that does the following:

- It will invoke the URL : /postformdata that will be our servlet that does the verification. The servlet code is discussed in the next section
- It will form the request parameter string as shown below:
  var params = "name="+name+"&recaptcha_challenge_field=" + Recaptcha.get_challenge() + "&recaptcha_response_field="+Recaptcha.get_response();
- **Note the two fields : recaptcha_challenge_field** and **recaptcha_challenge_response_field.** We get the two values from the **Recaptcha** class methods**, get_challenge()** and **get_response() respectively.** The **get_challenge()** is what was provided by the **ReCAPTCHA Server** and the **get_response()** is what was entered by **the user.**
- Finally we do a POST to the server and collect the response. The response is that displayed in another **DIV** element "status".

Please note that the code is just for demonstration purpose and may not represent the best practices or most efficient way of writing JavaScript, AJAX, etc.

The Servlet [PostFormDataServlet.java]

```
package com.gaejexperiments.captcha;

import java.io.IOException;

import javax.servlet.http.*;

//RECAPTCHA
import net.tanesha.recaptcha.ReCaptchaImpl;
import net.tanesha.recaptcha.ReCaptchaResponse;
@SuppressWarnings("serial")
public class PostFormDataServlet extends HttpServlet {

 public void doPost(HttpServletRequest req, HttpServletResponse resp)
   throws IOException {
  resp.setContentType("text/plain");
  String strResponse = "";
  try {

/*
 * CAPTCHA CHECK
 *
 *
 */
   //First validate the captcha, if not -- just get out of the loop
   String challenge = (String)
req.getParameter("recaptcha_challenge_field");
   String response = (String)
req.getParameter("recaptcha_response_field");
   if ((challenge == null) || (response == null)) {
    throw new Exception("Your words did not match. Please try submitting
again.");
   }

      String remoteAddr = req.getRemoteAddr();
      ReCaptchaImpl reCaptcha = new ReCaptchaImpl();

      reCaptcha.setPrivateKey("YOUR_PRIVATE_KEY");

      ReCaptchaResponse reCaptchaResponse =
          reCaptcha.checkAnswer(remoteAddr, challenge, response);

      if (!reCaptchaResponse.isValid()) {
       //RECAPTCHA VALIDATION FAILED
       throw new Exception("Your words did not match. Please try
submitting again.");
```

```
      }

      strResponse = "Your record has been accepted and you did a good
job entering the two words. Thank you";
  }
  catch (Exception ex) {
   strResponse = "You record was not accepted. Reason : " +
ex.getMessage();
  }
  resp.getWriter().println(strResponse);
 }
}
```

Let us go through the important parts in the code:

1. Notice that we are importing 2 classes : net.tanesha.recaptcha.ReCaptchaImpl and net.tanesha.recaptcha.ReCaptchaResponse at the beginning. These are the implementation classes that encapsulate the verification with the ReCAPTCHA Server and the response from the ReCAPTCHA Server respectively.

2. We first extract out the challenge and the response fields as passed by our form.

```
String challenge = (String)
req.getParameter("recaptcha_challenge_field");
String response = (String)
req.getParameter("recaptcha_response_field");
```

3. We also need the Remote IP Address that is making the request.

```

String remoteAddr = req.getRemoteAddr();
```

4. We then instantiate an instance of the ReCaptchaImpl class. We set the **Private Key** that we got during our registration for the site **localhost.** Please use your key over here. And then we make a call to their server by invoking the **checkAnswer** method. The **checkAnswer** method takes 3 parameters : challenge, response and the Remote Address.

```
      ReCaptchaImpl reCaptcha = new ReCaptchaImpl();

      reCaptcha.setPrivateKey("YOUR_PRIVATE_KEY");

      ReCaptchaResponse reCaptchaResponse =
          reCaptcha.checkAnswer(remoteAddr, challenge, response);
```

5. We will receive an instance of the ReCaptchaResponse class. We simply use a utility method **isValid()** to determine if the response entered for the challenge was correct. And depending on that we send back an appropriate response back to the browser, which is then displayed to the user.

Servlet Configuration

To complete our Servlet development, we will also need to add the **<servlet/>** and **<servlet-mapping/>** entry to the **web.xml** file. This file is present in the **WEB-INF** folder of the project. The necessary fragment to be added to your **web.xml** file are shown below. Please note that you can use your own namespace and servlet class. Just modify it accordingly if you do so.

```
<servlet>
<servlet-name>PostFormDataServlet</servlet-name>
<servlet-
class>com.gaejexperiments.captcha.PostFormDataServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>PostFormDataServlet</servlet-name>
<url-pattern>/postformdata</url-pattern>
</servlet-mapping>
```

Running the application locally

I am assuming that you have already created a new Google Web Application Project and have created the above Servlets, web.xml , etc. We shall be running this episode within our local development server only since the keys that I have used are for **localhost**. Before you deploy the application to the actual Google App Engine infrastructure using your Application ID, do remember to get yourself the appropriate public/private ReCAPTCHA keys for your site name.

So assuming that all is well, we will run our application, by right-clicking on the project and selecting **Run As –> Web Application.** Launch the browser on your local machine and navigate to http://localhost:<YourLocalPort>/captcha.html

Conclusion

In this episode, we saw how to incorporate a CAPTCHA in our Google App Engine application. Please remember that you will need to determine if you really need a CAPTCHA in your application. There are some good points mentioned in this **article** on various other techniques that can be used to address the same problems.

Episode 16 : Using the Datastore API

Welcome to **Episode 16**. In this episode we shall cover basic usage of the Datastore API in Google App Engine. The Datastore API is used to persist and retrieve data. Google App Engine uses the BigTable database as its underlying datastore and provides abstraction for using it via the Datastore API. There are currently two options available to developers i.e. via the Java Data Objects (JDO) and Java Persistence Architecture (JPA) APIs.

In this episode, we shall cover the following items:

- Persist a simple record to the datastore using JDO. The intention is to limit it to a single record in this episode and not address relationships and complex structures. That could be the basis of future episodes.
- Retrieve the persisted records by firing some queries. In the process, we shall see how to create parameterized queries and execute them.
- Discuss some nuances about indexes and what steps you need to do to make sure that the same application that you use locally will work fine when deployed to the Google App Engine cloud.

The underlying theme will not be to present a comprehensive tutorial about the Datastore API. There are some excellent references available for that. The **official documentation** and well as the **GAE Persistence Blog**. The focus will be on getting up and running with the Datastore API ASAP and seeing it work when deployed to the cloud.

What we shall build

In this episode, we shall build the following:

1. Create a simple Object that we shall persist to the underlying datastore. The Object will be a **Health Report** and will have about 4-5 attributes that we would like to save.
2. Write the **Save** method that takes an instance of the above Health Report Record and persists it using the JDO API.
3. Write a **Search** method that will query for several Health Reports using several filter parameters.
4. Look at the **datastore_indexes.xml** file that is required when you deploy the application to the cloud.

Please note that the focus will be on the server side and not on building a pretty GUI. All server side actions will be invoked via a REST like request (HTTP GET) — so that we can test the functionality in the browser itself.

Developing our Application

The first thing to do is to create a **New Google Web Application** Project. Follow these steps:

1. Either click on File –> New –> Other or press Ctrl-N to create a new project. Select Google and then Web Application project. Alternately you could also click on the New Web Application Project Toolbar icon as part of the Google Eclipse plugin.
2. In the New Web Application Project dialog, deselect the Use Google Web Toolkit and give a name to your project. I have named mine GAEJExperiments. I suggest you go with the same name so that things are consistent with the rest of the article, but I leave that to you. **In case you are following the series, you could simply use the same project and skip all these steps altogether. You can go straight to the Servlet Development section.**
3. Click on Finish.

This will generate the project and also create a sample Hello World Servlet for you. But we will be writing our own Servlet.

Few things to note first:

Quite a few things are enabled for you by default as far as the database support is concerned. They are as follows:

a. Several JAR files are added to the CLASSPATH by default. Take a look and you will see several JARs *jpa*.jar, *datanucleus*.jar, etc.
b. In the **src/META-INF** folder, you will find a **jdoconfig.xml** file. There is a default Persistence Manager Factory class in that that we shall be using in the rest of the article. For the purposes of this article we do not need to do anything to this file.
c. GAEJ uses the DataNucleus library to abstract the BigTable store. The DataNucleaus library provides the JDO and JPA interfaces so that you do not have to deal with the underlying low level API. You will also find a **logging.properties** file present in **war/WEB-INF** folder. You will find several log levels mentioned for the DataNucleus classes. You can tweak them to lower levels like DEBUG/INFO to see more debug level statements of what happens when you are using these APIs. **I have found it very helpful to set the debug levels to DEBUG/INFO especially when facing a problem.**

PMF.java

The first class that we shall write is a simple utility class that shall get us the underlying Persistence Manager factory instance. This class is important since all other methods like saving a record, querying records, etc will work on the instance of the PersistenceManagerFactory.

The code is shown below and wherever we need an instance of the class, we shall simply invoke the **get()** method below:

```
package com.gaejexperiments.db;

import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManagerFactory;

public final class PMF {
    private static final PersistenceManagerFactory pmfInstance =
        JDOHelper.getPersistenceManagerFactory("transactions-
optional");

    private PMF() {}

    public static PersistenceManagerFactory get() {
        return pmfInstance;
    }
}
```

HealthReport.java

Next is the Health Report. As mentioned in the beginning of the article, we shall be saving a Health Report record. The Health Report will have 4 attributes and they are explained below:

1. Key : This is a unique key that is used to persist/identify the record in the datastore. We shall leave its implementation/generation to the Google App Engine implementation.
2. PinCode : This is similar to a ZipCode. This is a string that shall contain the value of the zipcode (Area) where the Health Incident has occured.
3. HealthIncident: This is a string value that contains the health incident name. For e.g. Flu, Cough, Cold, etc. In this simple application — we shall be concerned only with 3 health incidents i.e. Flu, Cough and Cold.
4. Status : This is a string value that specifies if the record is ACTIVE or INACTIVE. Only records with ACTIVE status shall be used in determining any statistics / data reports. We shall set this value to ACTIVE at the time of saving the record.
5. ReportDateTime : This is a Date field that shall contain the date/time that the record was created.

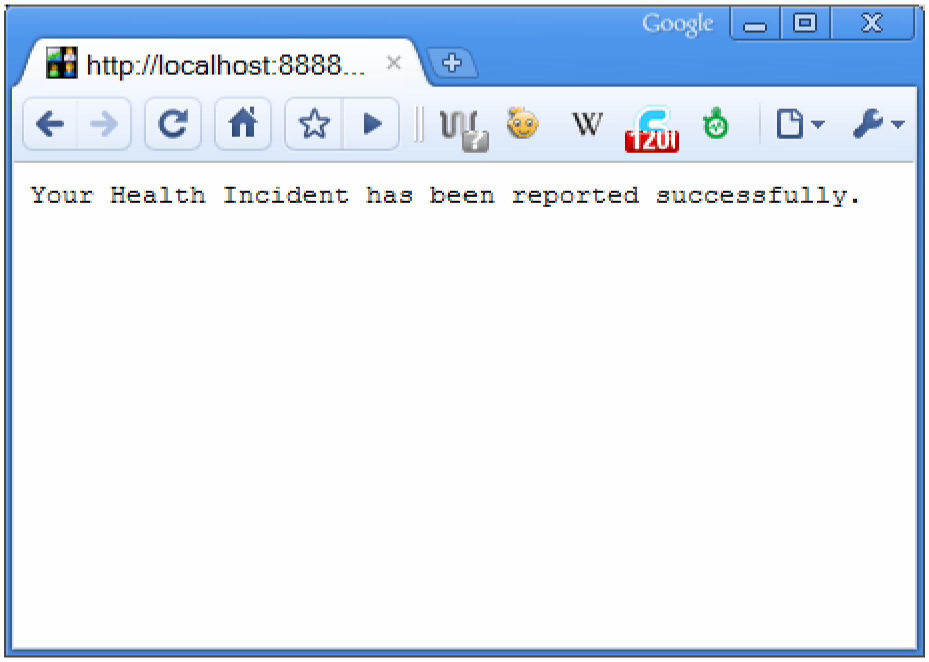
Shown below is the listing for the HealthReport.java class. In addition to the above attributes and getter/setter methods for them, note the following additions to make sure that your class can be persisted using JDO.

1. We need to have a constructor that contains all the fields except for the Key field.
2. All fields that need to be persisted are annotated with the **@Persistent** annotation.
3. The class is declared as being persistable via the **@PersistenceCapable annotation** and we are leaving the identity to the Application.
4. The Primary Key field i.e. Key is declared via the **@PrimaryKey** annotation and we are using an available Generator for the ID instead of rolling our own.

```
package com.gaejexperiments.db;

import java.util.Date;
import com.google.appengine.api.datastore.Key;

import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.IdentityType;
import javax.jdo.annotations.PersistenceCapable;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;

@PersistenceCapable(identityType = IdentityType.APPLICATION)
public class HealthReport {
 @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
    private Key key;

    private String pinCode;
    @Persistent
    private String healthIncident;
    @Persistent
    private String status;
    @Persistent
    private Date reportDateTime;

 public HealthReport(String pinCode, String healthIncident,String
status, Date reportDateTime) {
  super();
  this.pinCode = pinCode;
  this.healthIncident = healthIncident;
  this.status = status;
  this.reportDateTime = reportDateTime;
 }

 public Key getKey() {
  return key;
 }

 public void setKey(Key key) {
  this.key = key;
 }

 public String getPinCode() {
  return pinCode;
 }

 public void setPinCode(String pinCode) {
  this.pinCode = pinCode;
 }

 public String getHealthIncident() {
  return healthIncident;
 }

 public void setHealthIncident(String healthIncident) {
  this.healthIncident = healthIncident;
 }

 public String getStatus() {
  return status;
 }

 public void setStatus(String status) {
  this.status = status;
 }

 public Date getReportDateTime() {
  return reportDateTime;
 }

 public void setReportDateTime(Date reportDateTime) {
  this.reportDateTime = reportDateTime;
 }

}
```

PostHealthIncidentServlet.java

We shall now look at how to persist the above Health Record. Since we are not going to build a UI for it, we shall simply invoke a servlet (HTTP GET) with the required parameters. It would almost be like a FORM submitting these values to the Servlet. Before we write this Servlet code, let us look at how we will invoke it. Given below is a screenshot of the browser where I punch in the URL : http://localhost:8888/posthealthincident?**healthincident**=Flu&**pincode**=400101



```
Your Health Incident has been reported successfully.
```

As you can see, I am running the application on my local development server and invoke the servlet (which we shall see in a while) providing two key parameters **healthincident** and **pincode.** These two parameters are two key fields of the HealthReport class that we saw above. The other fields like ReportDateTime and Status are determined automatically by the application. Similarly the Key value of the record in the underlying datastore will be generated by App Engine infrastructure itself.

Let us now look at the **PostHealthIncidentServlet.java** code shown below:

```
package com.gaejexperiments.db;

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

import javax.servlet.ServletException;
import javax.servlet.http.*;
@SuppressWarnings("serial")
public class PostHealthIncidentServlet extends HttpServlet {
 public static final Logger _logger =
Logger.getLogger(PostHealthIncidentServlet.class.getName());

 @Override
 protected void doGet(HttpServletRequest req, HttpServletResponse resp)
   throws ServletException, IOException {
  doPost(req, resp);
 }

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

```
  resp.setContentType("text/plain");
  String strResponse = "";
  String strHealthIncident = "";
  String strPinCode = "";
  try {

   //DO ALL YOUR REQUIRED VALIDATIONS HERE AND THROW EXCEPTION IF
NEEDED


   strHealthIncident = (String)req.getParameter("healthincident");
   strPinCode = (String)req.getParameter("pincode");

   String strRecordStatus = "ACTIVE";

         Date dt = new Date();
      HealthReport HR = new HealthReport(strPinCode,
                                  strHealthIncident,
                                  strRecordStatus,
                                  dt);
      DBUtils.saveHealthReport(HR);
      strResponse = "Your Health Incident has been reported
successfully.";
  }
  catch (Exception ex) {
   _logger.severe("Error in saving Health Record : " +
strHealthIncident + "," + strPinCode +  " : " + ex.getMessage());
   strResponse = "Error in saving Health Record via web. Reason : " +
ex.getMessage();
  }
  resp.getWriter().println(strResponse);
 }
}
```

The main points of the code are :

1) We extract out the HealthIncident and PinCode request parameters. We do not do any particular validations but you could do all of that depending on your application requirement.
2. We generate the two other field values i.e. Date (ReportDate) and Status (ACTIVE).
3. Finally, we create a new instance of the HealthReport class, providing the values in the constructor. And then call the **DBUtils.saveHealthReport(…)** method that persists the record to the underlying datastore.
4. We display back a successfull message if all is well, which is what was visible in the screenshot above.

Let us look at the DBUtils.java class now. Please note that we have simply separated out the code into this file but you can manage/partition your code in any which way you like.

DBUtils.java

The DBUtils.java source code is listed below. Focus first on the **saveHealthReport()** method which was invoked by our servlet earlier. The other method, we shall come to that later on in the article.

Key Points are :

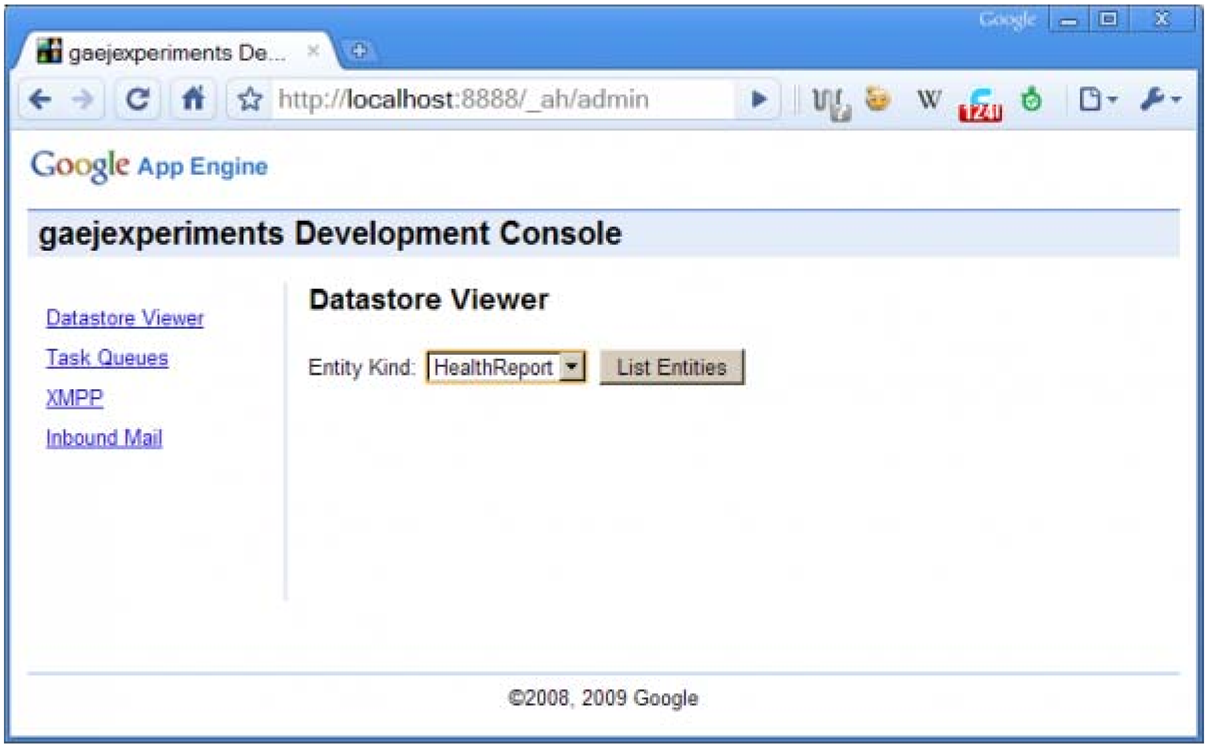
1. The **saveHealthReport()** method first gets the instance of the PersistenceManager through the **PMF.java** class that we wrote earlier.
2. It simply invoke the **makePersistent()** method on it. The makePersistent() method will take as a parameter the object that you want to persist. In our case it is the HealthReport.java class instance that we have created in the servlet. This method will persist the record and in the process also assign it a unique key.
3. Finally, we need to close the PersistenceManager instance by invoking the **close()** method.

The entire code listing is shown below:

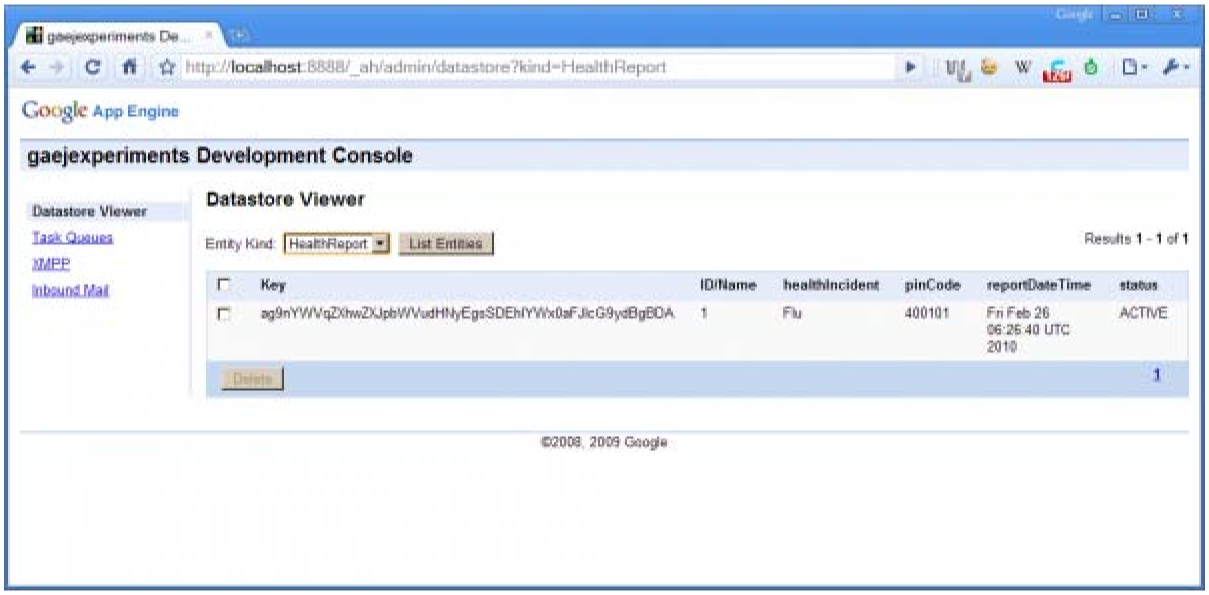
```
package com.gaejexperiments.db;

import java.util.Calendar;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.jdo.PersistenceManager;
import javax.jdo.Query;

public class DBUtils {
 public static final Logger _logger =
Logger.getLogger(DBUtils.class.getName());

 //Currently we are hardcoding this list. But this could also be
retrieved from
 //database
 public static String getHealthIncidentMasterList() throws Exception {
  return "Flu,Cough,Cold";
 }

 /**
  * This method persists a record to the database.
  */
 public static void saveHealthReport(HealthReport healthReport)
   throws Exception {
  PersistenceManager pm = PMF.get().getPersistenceManager();
  try {
   pm.makePersistent(healthReport);
   _logger.log(Level.INFO, "Health Report has been saved");
  } catch (Exception ex) {
   _logger.log(Level.SEVERE,
     "Could not save the Health Report. Reason : "
       + ex.getMessage());
   throw ex;
  } finally {
   pm.close();
  }
 }

 /**
  * This method gets the count all health incidents in an area
(Pincode/Zipcode) for the current month
  * @param healthIncident
  * @param pinCode
  * @return A Map containing the health incident name and the number of
cases reported for it in the current month
  */
 public static Map<String, Integer>
getHealthIncidentCountForCurrentMonth(String healthIncident, String
pinCode) {
  Map<String, Integer> _healthReport = new HashMap<String, Integer>();

  PersistenceManager pm = null;

  //Get the current month and year
  Calendar c = Calendar.getInstance();
  int CurrentMonth = c.get(Calendar.MONTH);
  int CurrentYear = c.get(Calendar.YEAR);

  try {
   //Determine if we need to generate data for only one health Incident
or ALL
   String[] healthIncidents = {};
   if (healthIncident.equalsIgnoreCase("ALL")) {
    String strHealthIncidents = getHealthIncidentMasterList();
    healthIncidents = strHealthIncidents.split(",");
   }
   else {
    healthIncidents =  new String[]{healthIncident};
   }

   pm = PMF.get().getPersistenceManager();
   Query query = null;

   //If Pincode (Zipcode) is ALL, we need to retrieve all the records
irrespective of Pincode
   if (pinCode.equalsIgnoreCase("ALL")) {
    //Form the query
    query = pm.newQuery(HealthReport.class, " healthIncident ==
paramHealthIncident && reportDateTime >= paramStartDate &&
reportDateTime < paramEndDate && status == paramStatus");

    // declare parameters used above
    query.declareParameters("String paramHealthIncident, java.util.Date
paramStartDate, java.util.Date paramEndDate, String paramStatus");
   }
   else {
    query = pm.newQuery(HealthReport.class, " healthIncident ==
paramHealthIncident && pinCode == paramPinCode && reportDateTime >=
paramStartDate && reportDateTime <paramEndDate && status ==
paramStatus");

    // declare params used above
    query.declareParameters("String paramHealthIncident, String
paramPinCode, java.util.Date paramStartDate, java.util.Date
paramEndDate, String paramStatus");
   }

   //For each health incident (i.e. Cold Flu Cough), retrieve the
records

   for (int i = 0; i < healthIncidents.length; i++) {
    int healthIncidentCount = 0;
    //Set the From and To Dates i.e. 1st of the month and 1st day of
next month
    Calendar _cal1 = Calendar.getInstance();
    _cal1.set(CurrentYear, CurrentMonth, 1);
    Calendar _cal2 = Calendar.getInstance();
    _cal2.set(CurrentYear,CurrentMonth+1,1);

    List<HealthReport> codes = null;
    if (pinCode.equalsIgnoreCase("ALL")) {
     //Execute the query by passing in actual data for the filters
     codes = (List<HealthReport>)
query.executeWithArray(healthIncidents[i],_cal1.getTime(),_cal2.getTime
(),"ACTIVE");
    }
    else {
     codes = (List<HealthReport>)
query.executeWithArray(healthIncidents[i], pinCode,
_cal1.getTime(),_cal2.getTime(),"ACTIVE");
    }

    //Iterate through the results and increment the count
    for (Iterator iterator = codes.iterator(); iterator.hasNext();) {
     HealthReport _report = (HealthReport) iterator.next();
     healthIncidentCount++;
    }

    //Put the record in the Map data structure
    _healthReport.put(healthIncidents[i], new
Integer(healthIncidentCount));
   }
   return _healthReport;
  } catch (Exception ex) {
   return null;
  } finally {
   pm.close();
  }
```

```
 }
}
```

Assuming that your application is running, you can view the data records that are being persisted. If you navigate to http://localhost:<YourPort>/_ah/admin in your browser, you will see a screenshot similar to the one shown below:



The screen above shows the Entity types that are currently having some data records. In our case it is the HealthReport entity of which we have saved one record so far. If you click on the **List Entities** button, you can see the records that are currently persisted for the HealthReport Entity Type. A sample screenshot from my system after saving my first record is shown below:



gaejexperiments Development Console

Datastore Viewer

Datastore Viewer
Task Queues
XMPP
Inbound Mail

Entity Kind: HealthReport List Entities

Results 1 - 1 of 1

	Key	ID/Name	healthIncident	pinCode	reportDateTime	status
☐	ag9nYWVqZXhwZXJpbWVudHNyEgsSDEhlYWx0aFJlcG9ydBgBDA	1	Flu	400101	Fri Feb 26 06:26:40 UTC 2010	ACTIVE

Delete 1

©2008, 2009 Google

Go ahead and populate a few more records in the database for different HealthIncidents like Cough, Cold, Flu (only). This is needed so that we can get some more data when we cover how to query persistent data in the next section.

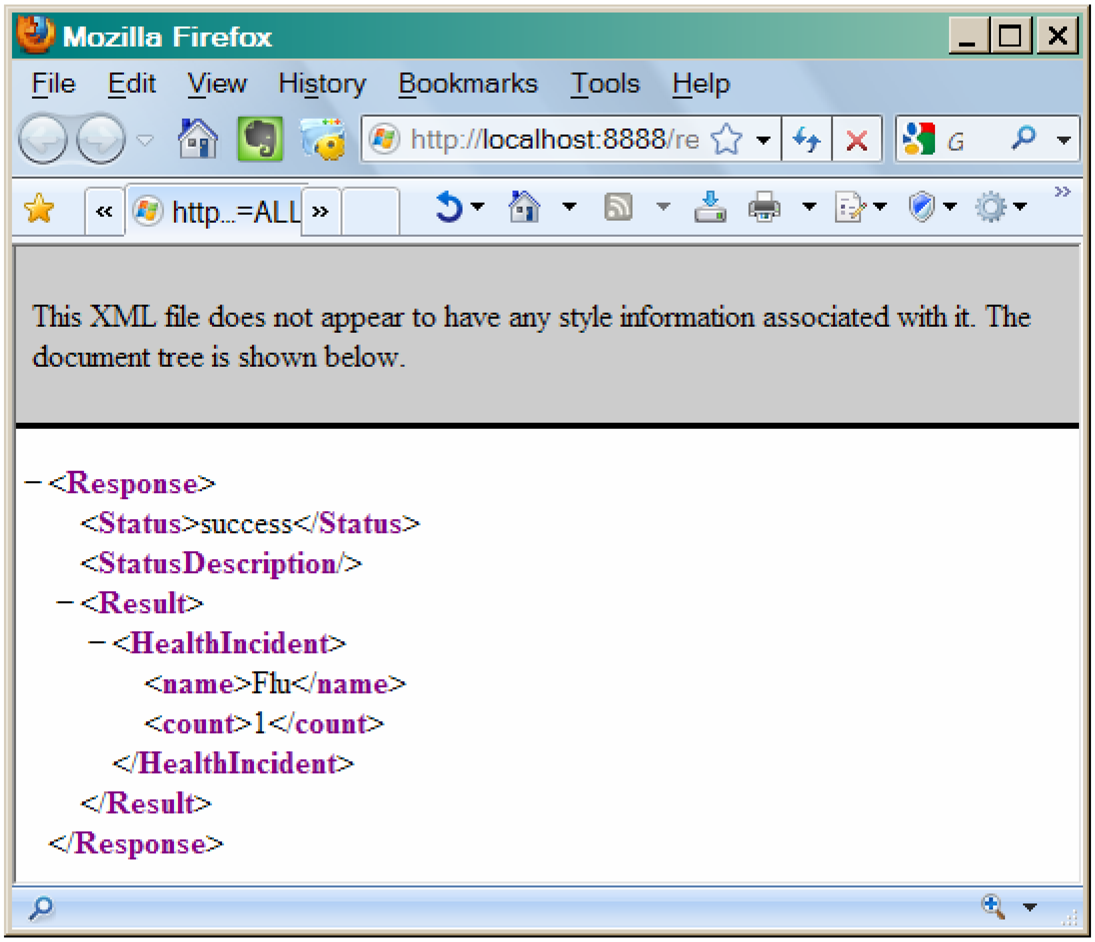
ReportsServlet.java

Before we look at this servlet, let us look at the output that this servlet produces, so that it becomes easier to follow the code later. This is assuming that you have added atleast 4-5 records using the **/posthealthincident** servlet that we covered above.

Shown below is the screenshot of the servlet output when I provide the following url:

http://localhost:8888/reports?type=HEALTHINCIDENTCOUNT_CURRENT_MONTH&healthincident=Flu&pincode=ALL

What we are asking for here is a report that gets all health incidents in the current month (type = HEALTHINCIDENTCOUNT_CURRENT_MONTH) for the healthincident = Flu and where pincode = ALL (irrespective of pincode)

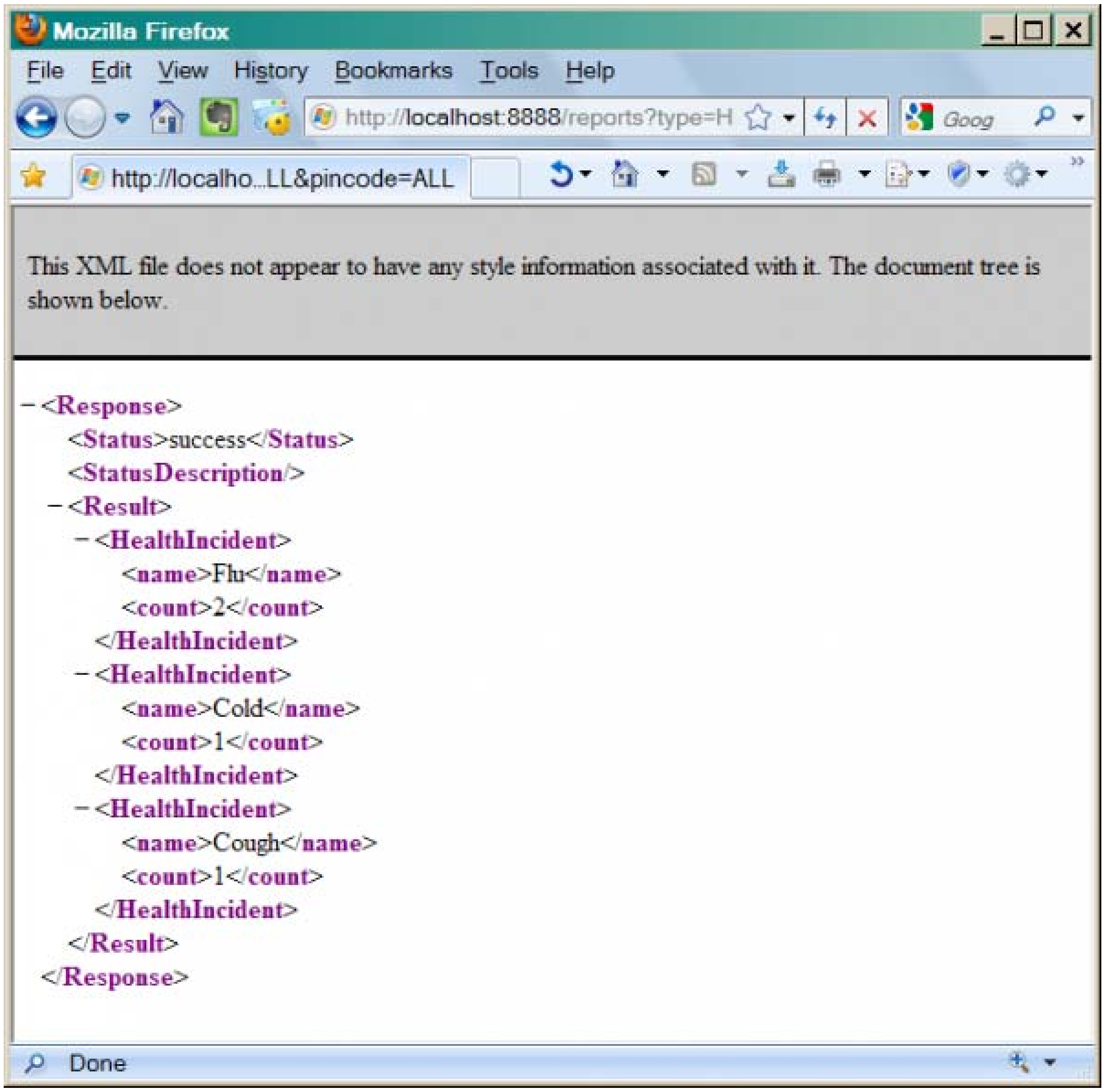


This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
- <Response>
    <Status>success</Status>
    <StatusDescription/>
  - <Result>
    - <HealthIncident>
        <name>Flu</name>
        <count>1</count>
      </HealthIncident>
    </Result>
  </Response>
```

Shown below is the screenshot of the servlet output when I provide the following url:

http://localhost:8888/reports?type=HEALTHINCIDENTCOUNT_CURRENT_MONTH&healthincident=ALL&pincode=ALL
What we are asking for here is a report that gets all health incidents in the current month (type = HEALTHINCIDENTCOUNT_CURRENT_MONTH) for the healthincident = ALL (irrespective of health incident which means all of them) and where pincode = ALL (irrespective of pincode)


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8888/reports?type=H
http://localho...LL&pincode=ALL
Done

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
- <Response>
    <Status>success</Status>
    <StatusDescription/>
  - <Result>
    - <HealthIncident>
        <name>Flu</name>
        <count>2</count>
      </HealthIncident>
    - <HealthIncident>
        <name>Cold</name>
        <count>1</count>
      </HealthIncident>
    - <HealthIncident>
        <name>Cough</name>
        <count>1</count>
      </HealthIncident>
    </Result>
  </Response>
```

So what we have effectively done here is to query the set of Health Records that are present in our database using a variety of parameters (filters). In other words, if we take a SQL like aspect to it, we are saying something like this:

SELECT * FROM HEALTHREPORTS WHERE PINCODE = %1 AND HEALTHINCIDENT = %2 AND REPORTDATE >= %3 AND REPORTDATE < %4 AND STATUS = ACTIVE , etc.

The above SQL statement is just representative of the queries that we want to execute. So let us look at the code for the servlet first.

```
package com.gaejexperiments.db;

import java.io.IOException;
import java.util.Iterator;
import java.util.Map;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
@SuppressWarnings("serial")
public class ReportsServlet extends HttpServlet {
 public void doGet(HttpServletRequest req, HttpServletResponse resp)
   throws IOException {
  resp.setContentType("text/xml");
  String strResult = "";
  String strData   = "";
  try {

   String type = (String)req.getParameter("type");
   if (type == null) {
    strResult =  "No Report Type specified.";
    throw new Exception(strResult);
   }
   else if (type.equals("HEALTHINCIDENTCOUNT_CURRENT_MONTH")) {
    String strHealthIncident =
(String)req.getParameter("healthincident");
    String strPinCode = (String)req.getParameter("pincode");
    Map<String,Integer> _healthReports =
DBUtils.getHealthIncidentCountForCurrentMonth(strHealthIncident,strPinC
ode);
    if (_healthReports == null) {
    }
    else {
     Iterator<String> it = _healthReports.keySet().iterator();
     while (it.hasNext()) {
      String healthIncident = (String)it.next();
      int healthIncidentCount = 0;
      Integer healthIncidentCountObject =
_healthReports.get(healthIncident);
      if (healthIncidentCountObject == null) {
       healthIncidentCount = 0;
      }
      else {
       healthIncidentCount = healthIncidentCountObject.intValue();
      }
      if (healthIncidentCount > 0)
       strData += "<HealthIncident><name>" + healthIncident + "</name>"
+ "<count>" + healthIncidentCount + "</count></HealthIncident>";
     }
    }
    strResult = 
"<Response><Status>success</Status><StatusDescription></StatusDescripti
on><Result>" + strData + "</Result></Response>";
   }
  }
  catch (Exception ex) {
   strResult = "<Response><Status>fail</Status><StatusDescription>"+ 
"Error in executing operation : " + ex.getMessage() + 
"</StatusDescription></Response>";
  }
  resp.getWriter().println(strResult);
 }

} </pre>
<pre>
```

The Servlet code is straightforward:

1. Currently it has only one type of report i.e.
HEALTHINCIDENTCOUNT_CURRENT_MONTH
2. Next we extract out the Pincode and the HealthIncident request parameter values.
3. Then we invoke the **DBUtils.getHealthIncidentCountForCurrentMonth** method. This takes two parameters : pincode and healthincident that we have got from above step.
4. The method will return us a map where each record in the map will contain the key (String) containing the healthincident name and the value containing the count of incidents reported for that month. So something like [{"Flu","20"} , {"Cough", "30"} , {"Cold","10"}]
5. Finally we simply format that into a XML output so that it can be returned to the client. And this is the exact output that we see in the browser.

Analyzing the DBUtils.getHealthIncidentCountForCurrentMonth method

I reproduce here the method from the DBUtils.java class that was listed before:

```
 /**
  * This method gets the count all health incidents in an area 
(Pincode/Zipcode) for the current month
  * @param healthIncident
  * @param pinCode
  * @return A Map containing the health incident name and the number of 
cases reported for it in the current month
  */
 public static Map<String, Integer> 
getHealthIncidentCountForCurrentMonth(String healthIncident, String 
pinCode) {
  Map<String, Integer> _healthReport = new HashMap<String, 
Integer>();

  PersistenceManager pm = null;
  //Get the current month and year
  Calendar c = Calendar.getInstance();
  int CurrentMonth = c.get(Calendar.MONTH);
  int CurrentYear = c.get(Calendar.YEAR);

  try {
   //Determine if we need to generate data for only one health Incident
or ALL
   String[] healthIncidents = {};
   if (healthIncident.equalsIgnoreCase("ALL")) {
    String strHealthIncidents = getHealthIncidentMasterList();
    healthIncidents = strHealthIncidents.split(",");
   }
   else {
    healthIncidents =  new String[]{healthIncident};
   }

   pm = PMF.get().getPersistenceManager();
   Query query = null;

   //If Pincode (Zipcode) is ALL, we need to retrieve all the records
irrespective of Pincode
   if (pinCode.equalsIgnoreCase("ALL")) {
    //Form the query
    query = pm.newQuery(HealthReport.class, " healthIncident ==
paramHealthIncident && reportDateTime >= paramStartDate &&
reportDateTime < paramEndDate && status == paramStatus");

    // declare parameters used above
    query.declareParameters("String paramHealthIncident, java.util.Date
paramStartDate, java.util.Date paramEndDate, String paramStatus");
   }
   else {
    query = pm.newQuery(HealthReport.class, " healthIncident ==
paramHealthIncident && pinCode == paramPinCode && reportDateTime >=
paramStartDate && reportDateTime <paramEndDate && status ==
paramStatus");

    // declare params used above
    query.declareParameters("String paramHealthIncident, String
paramPinCode, java.util.Date paramStartDate, java.util.Date
paramEndDate, String paramStatus");
   }

   //For each health incident (i.e. Cold Flu Cough), retrieve the
records

   for (int i = 0; i < healthIncidents.length; i++) {
    int healthIncidentCount = 0;
    //Set the From and To Dates i.e. 1st of the month and 1st day of
next month
    Calendar _cal1 = Calendar.getInstance();
    _cal1.set(CurrentYear, CurrentMonth, 1);
    Calendar _cal2 = Calendar.getInstance();
    _cal2.set(CurrentYear,CurrentMonth+1,1);
    List<HealthReport> codes = null;
    if (pinCode.equalsIgnoreCase("ALL")) {
     //Execute the query by passing in actual data for the filters
     codes = (List<HealthReport>)
query.executeWithArray(healthIncidents[i],_cal1.getTime(),_cal2.getTime
(),"ACTIVE");
    }
    else {
     codes = (List<HealthReport>)
query.executeWithArray(healthIncidents[i], pinCode,
_cal1.getTime(),_cal2.getTime(),"ACTIVE");
    }

    //Iterate through the results and increment the count
    for (Iterator iterator = codes.iterator(); iterator.hasNext();) {
     HealthReport _report = (HealthReport) iterator.next();
     healthIncidentCount++;
    }

    //Put the record in the Map data structure
    _healthReport.put(healthIncidents[i], new
Integer(healthIncidentCount));
   }
   return _healthReport;
  } catch (Exception ex) {
   return null;
  } finally {
   pm.close();
  }
 }
```

I have attempted to provide comments so that you can follow the code but I will list down the important parts here:

1. We are going to deal with the following classes : **PersistenceManager** and **Query** from the **javax.jdo** package.
2. We get the PersistenceManager instance via the PMF.java class that we wrote earlier.
3. We are using the **Query** class here to first build the query. For e.g.
   query = pm.newQuery(HealthReport.class, " healthIncident == paramHealthIncident && reportDateTime >= paramStartDate && reportDateTime < paramEndDate && status == paramStatus");

What this means is that we are creating a query instance where we wish to get all records for the HealthReport class. Additionally we are passing a **criteria** string. Notice that the lefthand side are the fields of the **HealthReport** class (**healthIncident, reportDateTime, status)** and the right hand side are parameters which will define and then pass the values for to execute the query.

4. We define the parameters next as shown below:
   // declare parameters used above
   query.**declareParameters**("**String** paramHealthIncident, **java.util.Date**

paramStartDate, **java.util.Date** paramEndDate, **String** paramStatus”);
5. Finally we use the query.**executeWithArray(…)** method which takes as parameter an array that contains all the values for the above parameters that you have declared.
6. The **executeWithArray(…)** will return a **List<>** of **HealthReport** class instances that you can then iterate through the populate your result. In our code, we simply compute the total number for each of the health incidents (Flu, Cough, Cold).

Servlet Configuration

To complete our Servlet development, we will also need to add the **<servlet/>** and **<servlet-mapping/>** entry to the **web.xml** file. This file is present in the **WEB-INF** folder of the project. The necessary fragment to be added to your **web.xml** file are shown below. Please note that you can use your own namespace and servlet class. Just modify it accordingly if you do so.

```
 <servlet>
  <servlet-name>PostHealthIncidentServlet</servlet-name>
  <servlet-
class>com.gaejexperiments.db.PostHealthIncidentServlet</servlet-class>
 </servlet>
 <servlet>
  <servlet-name>ReportsServlet</servlet-name>
  <servlet-class>com.gaejexperiments.db.ReportsServlet</servlet-class>
 </servlet>
 <servlet-mapping>
  <servlet-name>PostHealthIncidentServlet</servlet-name>
  <url-pattern>/posthealthincident</url-pattern>
 </servlet-mapping>
 <servlet-mapping>
  <servlet-name>ReportsServlet</servlet-name>
  <url-pattern>/reports</url-pattern>
 </servlet-mapping>
```

Running the application locally

I am assuming that you have already created a new Google Web Application Project and have created the above Servlets, web.xml , etc. So assuming that all is well, we will run our application, by right-clicking on the project and selecting **Run As –> Web Application.** Launch the browser on your local machine and try out the URLs that we have covered so far like :

1. Adding a Health Report record :

http://localhost:8888/posthealthincident?healthincident=Flu&pincode=400101

2. Reports

http://localhost:8888/reports?type=HEALTHINCIDENTCOUNT_CURRENT_MONTH &healthincident=Flu&pincode=ALL

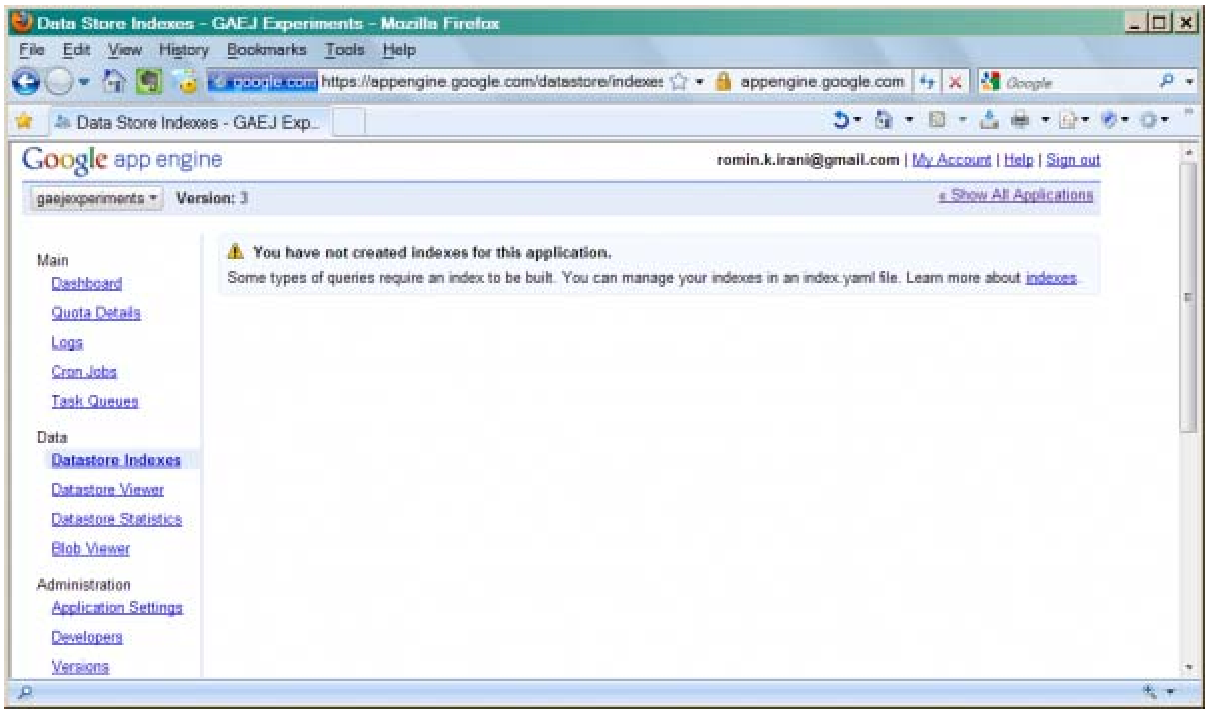
http://localhost:8888/reports?type=HEALTHINCIDENTCOUNT_CURRENT_MONTH &healthincident=ALL&pincode=ALL

Please replace the port 8888 with your local port number.

datastore_indexes.xml

If you run your application locally, you will notice that everything is working fine. **However,** if you deploy this application the Google App Engine, you will not get any results where you query the reports. Why is that? It is because there are **no indexes** defined for your application.

If you visit your Developer Console at http://appengine.google.com and for your particular application, you will find no indexes are defined for the **Datastore indexes** link as shown below:


Data Store Indexes - GAEJ Experiments - Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://appengine.google.com/datastore/indexes
Google app engine
romin.k.irani@gmail.com | My Account | Help | Sign out
gaejexperiments Version: 3
« Show All Applications
Main
Dashboard
Quota Details
Logs
Cron Jobs
Task Queues
Data
Datastore Indexes
Datastore Viewer
Datastore Statistics
Blob Viewer
Administration
Application Settings
Developers
Versions

You have not created indexes for this application.
Some types of queries require an index to be built. You can manage your indexes in an index.yaml file. Learn more about indexes.

What are these indexes ? Indexes are generated for every possible query that you fire in your program. This is Google's way of retrieving your data results efficiently. However when you run in local mode, these indexes are generated for your automatically. If you look in **war/WEB-INF** directory, you will find a directory named **appengine-generated**. Inside this directory is a file that is constantly updated called **datastore-indexes-**

auto.xml. The contents of this file for the reports that we have run so far is shown below:

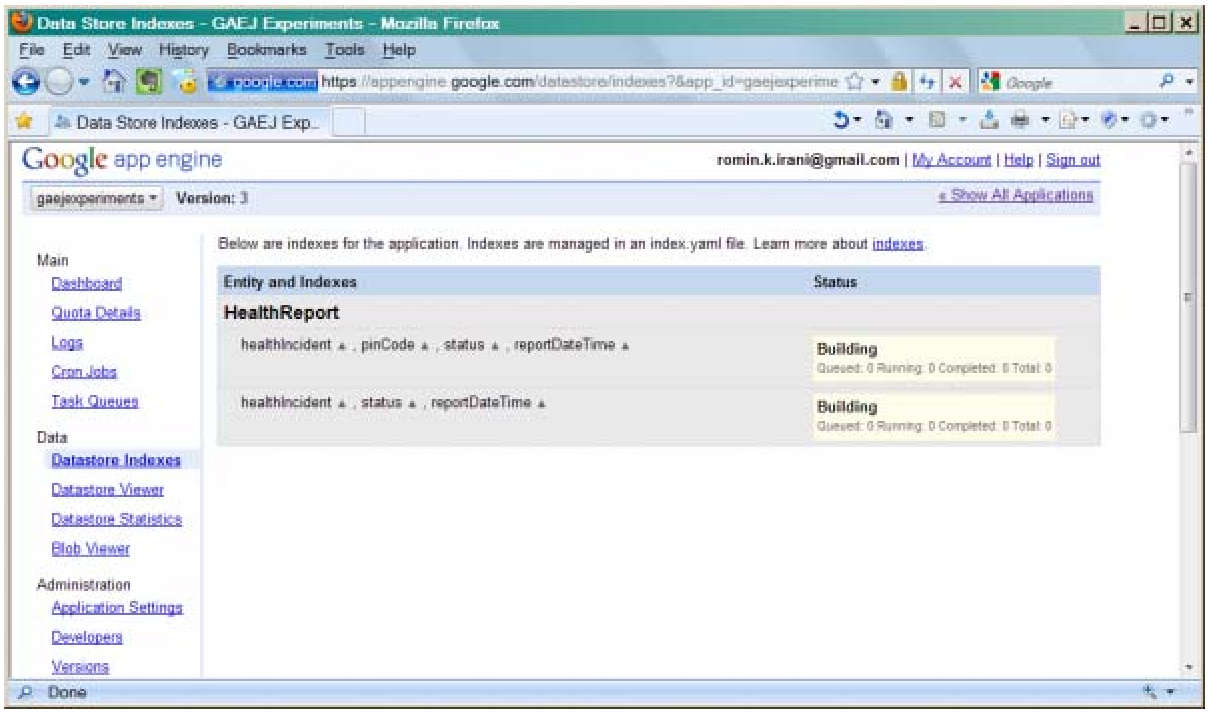
```
<datastore-indexes>



    <datastore-index kind="HealthReport" ancestor="false"
source="auto">
        <property name="healthIncident" direction="asc"/>
        <property name="status" direction="asc"/>
        <property name="reportDateTime" direction="asc"/>
    </datastore-index>



    <datastore-index kind="HealthReport" ancestor="false"
source="auto">
        <property name="healthIncident" direction="asc"/>
        <property name="pinCode" direction="asc"/>
        <property name="status" direction="asc"/>
        <property name="reportDateTime" direction="asc"/>
    </datastore-index>



</datastore-indexes>
```

As you can see, there are two indexes generated for the searches that we fired and each of them contains the parameters that we queried the Health Reports on.

To get your deployed application to function correctly, you will need to copy these indexes to a file named datastore_indexes.xml.

You will need to place this file in the **war/WEB-INF** folder of your application project.

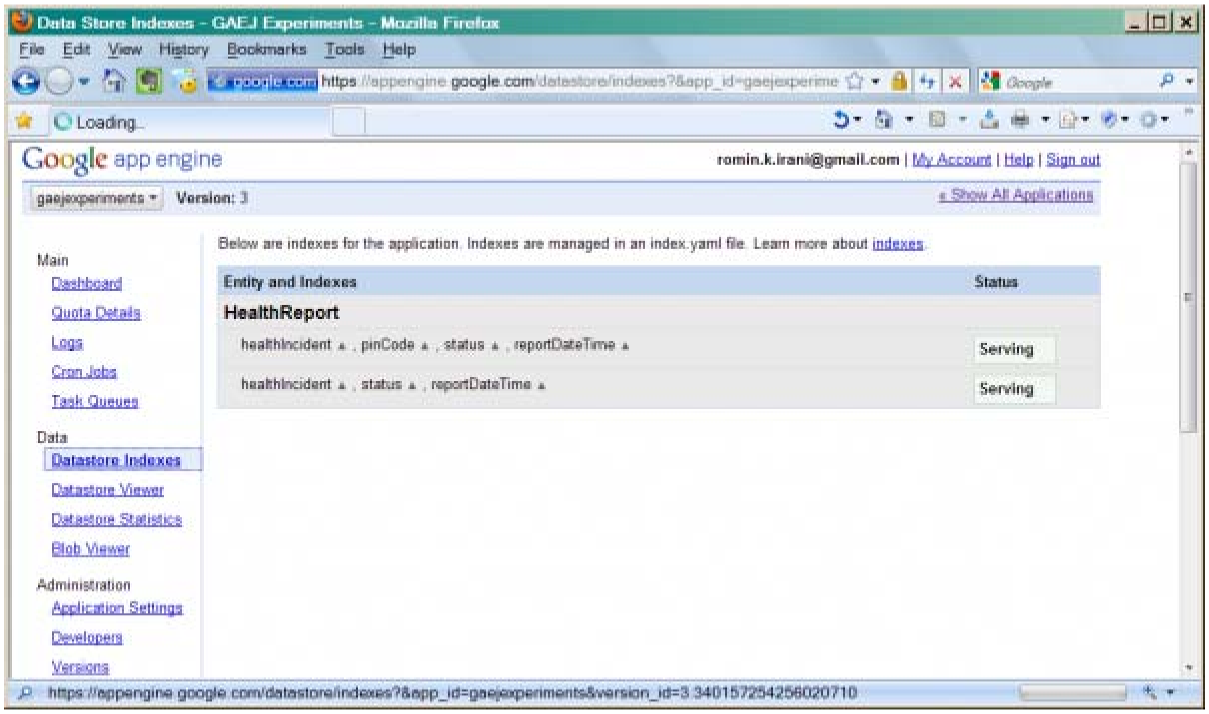
Now deploy your project again. On successful deploy, revist the Developer Console –> Datastore indexes. You will find that the indexes have got created and are buing built as shown below:



Below are indexes for the application. Indexes are managed in an index.yaml file. Learn more about indexes.

Entity and Indexes	Status
HealthReport	
healthIncident ▲ , pinCode ▲ , status ▲ , reportDateTime ▲	Building Queued: 0 Running: 0 Completed: 0 Total: 0
healthIncident ▲ , status ▲ , reportDateTime ▲	Building Queued: 0 Running: 0 Completed: 0 Total: 0

Wait for a while. The building process goes through your current data and then on completion, you will see the screen as shown below (A refresh needed!):


Data Store Indexes - GAEJ Experiments - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Loading...
Google app engine
romin.k.irani@gmail.com | My Account | Help | Sign out
gaejexperiments Version: 3
« Show All Applications
Main
Dashboard
Quota Details
Logs
Cron Jobs
Task Queues
Data
Datastore Indexes
Datastore Viewer
Datastore Statistics
Blob Viewer
Administration
Application Settings
Developers
Versions
https://appengine.google.com/datastore/indexes?&app_id=gaejexperiments&version_id=3.340157254256020710

Below are indexes for the application. Indexes are managed in an index.yaml file. Learn more about indexes.

Entity and Indexes	Status
HealthReport	
healthIncident ▲ , pinCode ▲ , status ▲ , reportDateTime ▲	Serving
healthIncident ▲ , status ▲ , reportDateTime ▲	Serving

Try out the queries now on your deployed application and they should work fine. You can also manual create the datastore_indexes.xml file but the local development server has this nice feature of auto generating out for you, so that you dont end up making mistakes. But remember to upload the updated datastore_indexes.xml as part of your deployment others the queries will silently fail.

Conclusion

We conclude this episode in which we covered how you can persist and query data using the JDO API. I hope that it has given you enough inputs to start incorporating persistence into your applications. It is by no means a simple exercise especially if you are coming in from a SQL world. Not everything SQL-like can be converted as is to a NoSQL world so refer to the several excellent sources available on the web in your persistence journey. I highly recommend the official documentation along with the GAE Persistence Blog mentioned at the beginning of the article.