

Introduction to the Google Web Toolkit Framework

Google Web Toolkit (GWT) is an open source web development framework that allows developers to easily create high-performance AJAX applications using Java. With GWT, you are able to write your front end in Java, and it compiles your source code into highly optimized, browser-compliant JavaScript and HTML. "Writing web apps today is a tedious and error-prone process. You spend 90% of your time working around browser quirks, and JavaScript's lack of modularity makes sharing, testing, and reusing AJAX components difficult and fragile. It doesn't have to be that way," reads the [Google Web Toolkit site](#).

In this tutorial, you learn how the above principles are applied to real applications. At the same time, you are introduced to NetBeans IDE's support for GWT and you build a simple application that makes use of some of these features.

Contents

- [Setting Up the Environment](#)
 - [Creating the Source Structure of a GWT Application](#)
 - [Examining the Source Structure of a GWT Application](#)
- [Creating an AJAX Random Quote Generator](#)
 - [Generating the Service Stubs](#)
 - [Examining the Generated Classes](#)
 - [Extending the Generated Classes](#)
 - [Customizing the Look and Feel](#)
- [Conclusion](#)
- [See Also](#)

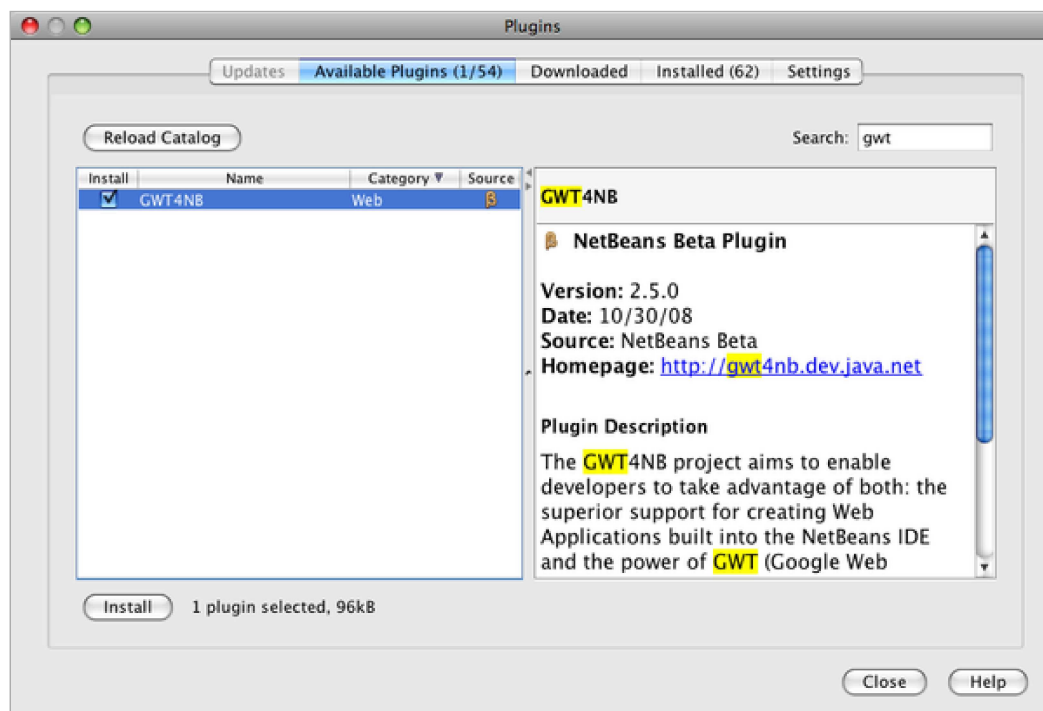


To complete this tutorial, you need the following software and resources.

Software or Resource	Version Required
NetBeans IDE	version 6.x Java
Java Development Kit (JDK)	version 5 or 6
GlassFish application server	V2 UR2 or v3 Prelude
or	
Tomcat servlet container	version 6.x
Google Web Toolkit (GWT)	version 1.4 or 1.5
NetBeans GWT plugin	version 2.x

Notes:

- The Java download bundle enables you to optionally install the GlassFish application server and the Apache Tomcat servlet container 6.0.x. You must install one of these to work through this tutorial.
- Rather than downloading the NetBeans GWT plugin from <https://gwt4nb.dev.java.net/>, you can download and install it directly through the IDE's Plugin Manager. Choose Tools > Plugins from the main menu and install the plugin, as shown below:



For more detailed instructions on how to install a framework plugin in the IDE, see: [Adding Support For A Web Framework](#).

- You can [download a sample working application](#) for this tutorial, as well as other applications using GWT.
- For more information on GWT, see <http://code.google.com/webtoolkit/>. For details on support for GWT in the IDE, see <https://gwt4nb.dev.java.net/>. If you are familiar with GWT, you are welcome to contribute code to the GWT plugin project.
- This tutorial follows some of the examples introduced in "Google Web Toolkit: GWT Java AJAX Programming", by Prabhakar Chaganti, published by [Packt Publishing](#), February 2007.

Setting Up the Environment

Begin by using the IDE to generate a basic source structure. Once you have it, you can study it in some detail in order to understand the inner workings of GWT.

- [Creating the Source Structure of a GWT Application](#)
- [Examining the Source Structure of a GWT Application](#)

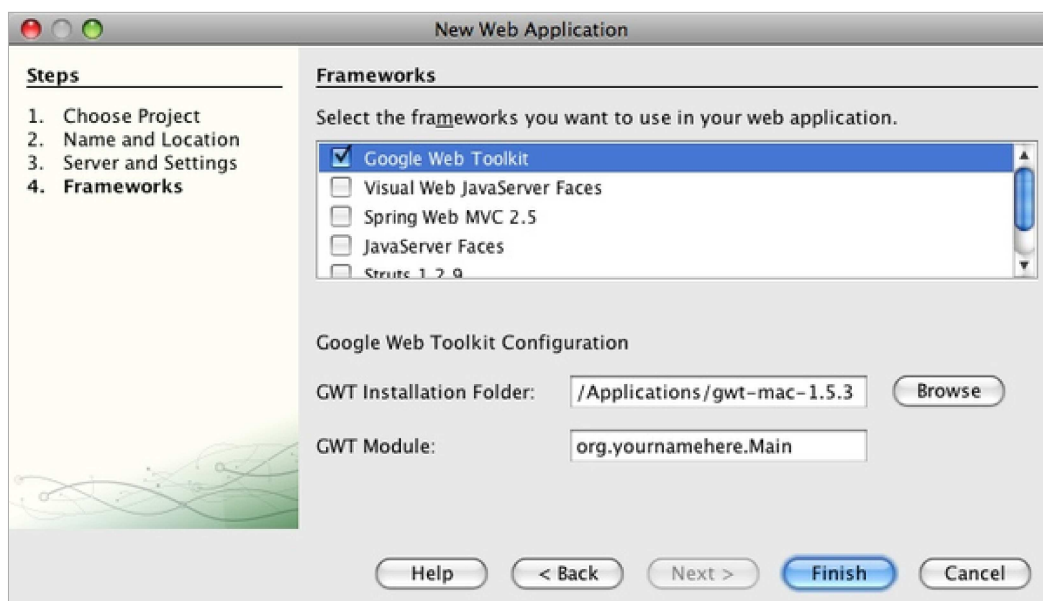
Creating the Source Structure of a GWT Application

The source structure of the application you create must include the GWT JAR files, the GWT module project configuration file, as well as some standard artifacts such as the Java entry point. Since you are using an IDE, you should not need to manually create all these files. Instead, let a wizard do the work for you. Specifically, the final panel of the New Web Application wizard is very useful in the context of creating a GWT application.

1. Choose File > New Project (Ctrl-Shift-N; ⌘-Shift-N on Mac). Under Categories, select Web (or Java Web). Under Projects, select Web Application. Click Next.
2. In step 2, Name and Location, type HelloGWT in Project Name. You can also specify the location of the project by typing in a path on your computer in Project Location field. Click Next.
3. In the Server and Settings step, select any server that you have registered in the IDE. If you have included Tomcat or GlassFish when installing the IDE, they display in the drop-down list.

To register a server in the IDE, click Add to open the Add Server Instance wizard.

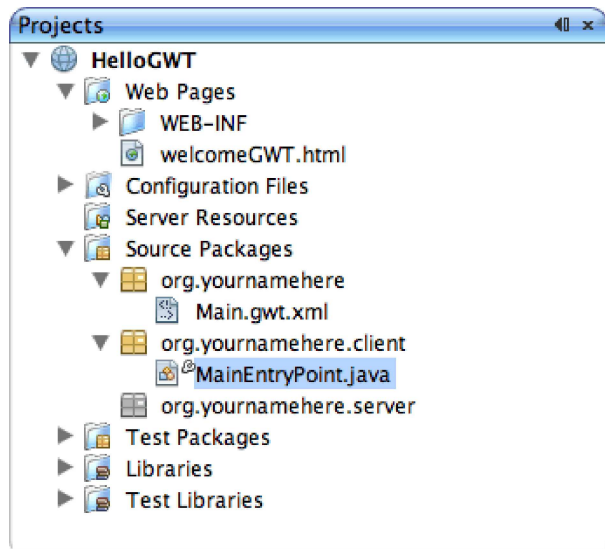
4. Specify the Java version you are using. GWT version 1.4 does not support Java EE 5, so if you are using this version you must also set the Java EE Version to 1.4. Otherwise, for example, Java EE 5 annotations will cause compilation errors. Click Next.
5. In the Frameworks step, choose GWT, as shown here:



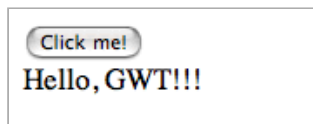
The fields in the panel above provide the following:

- **GWT Installation Folder:** Specify the path to the folder where you installed the Google Web Toolkit at the start of this tutorial. If an incorrect path is specified, a red error message displays and you are not able to complete the wizard.
 - **GWT Module:** Specifies the name and location of the project module that the IDE will generate when you complete the wizard. The project module is an XML file that configures a GWT application. For example, it is used to specify the class instantiated by GWT when the module is loaded. Note that this field in the wizard also determines the main package of the application. By default, the main package is org.yournamehere and the project module is Main. For purposes of this tutorial, leave the defaults unchanged.
6. Click Finish.

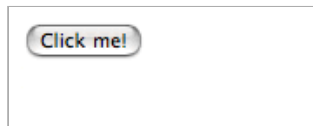
The IDE creates the HelloGWT project. The project contains all of your sources, libraries, and project metadata, such as the project's Ant build script. The project opens in the IDE. You can view its file structure in the Files window (Ctrl-2; ⌘-2 on Mac) and its logical structure in the Projects window (Ctrl-1; ⌘-1 on Mac):



Right-click the project and choose Run. The application is built and a web archive (WAR) is created. It is deployed to the server. The server starts, if it is not running already. Your computer's default browser opens and the welcome page of the application is displayed:



Click the button and the text below it disappears:



In the next section, you explore each of the generated files in detail and examine how the simple application above was created.

Examining the Source Structure of a GWT Application

The IDE's New Web Application wizard created several source files. Take a look at the files and see how they relate to each other within the context of a GWT application.

- **Project module:** The `Main.gwt.xml` file, in the application's main package, is an XML file that holds the complete application configuration needed by a GWT project. The default project module generated by the wizard looks as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<module>
  <inherits name="com.google.gwt.user.User"/>
  <entry-point class="org.yournamehere.client.MainEntryPoint"/>
  <!-- Do not define servlets here, use web.xml -->
</module>
```

The elements in the default project module are as follows:

- **inherits:** Specifies modules inherited by this module. In this simple case, we only inherit the functionality provided by the `User` module, which is built into the GWT framework. When your application becomes more complex, module inheritance lets you reuse pieces of functionality in a quick and efficient way.
- **entry-point:** Refers to the class that will be instantiated by the GWT framework when the module is loaded.
- **Application Entry Point:** The `MainEntryPoint` class is the entry point to the application, as specified in the project module, outlined above. It extends the `EntryPoint` class, and when the GWT module is loaded by the GWT framework, this class is instantiated and its `onModuleLoad()` method is automatically called. The default entry point generated by the wizard looks as follows:

```

package org.yournamehere.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.Widget;

public class MainEntryPoint implements EntryPoint {

    /** Creates a new instance of MainEntryPoint */
    public MainEntryPoint() {
    }

    /**
     * The entry point method, called automatically by loading a module
     * that declares an implementing class as an entry-point
     */
    public void onModuleLoad() {
        final Label label = new Label("Hello, GWT!!!");
        final Button button = new Button("Click me!");

        button.addClickListener(new ClickListener(){
            public void onClick(Widget w) {
                label.setVisible(!label.isVisible());
            }
        });

        RootPanel.get().add(button);
        RootPanel.get().add(label);
    }
}

```

The `onModuleLoad()` method in the default entry point class adds the following to the application:

- **Label:** A new `com.google.gwt.user.client.ui.Label` is created, displaying the text `Hello, GWT!!!`. The label is added to the root panel by means of the final line of code, `RootPanel.get().add(label)`.
- **Button:** A new `com.google.gwt.user.client.ui.Button` is created, displaying the text `Click me!` together with a button listener, provided by a `com.google.gwt.user.client.ui.ClickListener`. The button listener specifies that when a button is clicked, the label is hidden. The button is added to the root panel by means of the second to last line of code, `RootPanel.get().add(button)`.
- **Host Page:** The generated HTML page, called `welcomeGWT.html`, that loads the application. The `web.xml` file uses the `welcome-file` element to specify that the host page is the initial page displayed in the browser when the application is deployed. The default host page generated by the wizard looks as follows:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <meta name='gwt:module' content='org.yournamehere.Main=org.yournamehere.Main'>
    <title>Main</title>
  </head>
  <body>
    <script language="javascript" src="org.yournamehere.Main/org.yournamehere.Main.n
  </body>
</html>

```

The elements in the default host page are as follows:

- **meta:** Points to the application's project directory. This tag provides the link between the HTML page and the application.
- **script:** Imports code from the GWT framework's JavaScript file. This file contains the code required to bootstrap the GWT framework. It uses the configuration in the project module and then dynamically loads the JavaScript created by compiling the entry point to present the application. The JavaScript file is generated by the GWT framework when you run the application in hosted mode or when you compile the application.

Creating an AJAX Random Quote Generator

In this section, you display a random quote on the web page. This example application familiarizes you with the various components of a GWT application. The random quote is to be selected from a list of quotes stored on the server. Every second the application retrieves the random quote provided by the server and displays it on the web page in true AJAX style, that is, without the user needing to refresh the page.


In the process of creating this functionality, you make use of a GWT RPC ([Remote Procedure Call](#)) service.

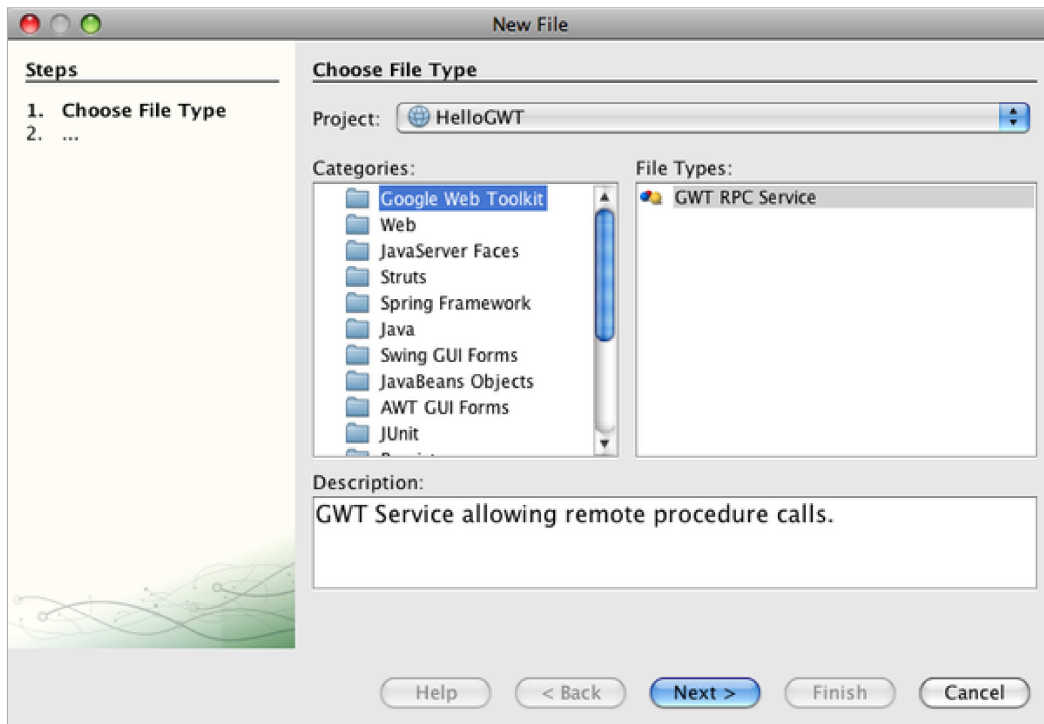
Note: A GWT *service* is not the same as the more general *web service* concept. Specifically, GWT services are not related to the Simple Object Access Protocol (SOAP).

- [Generating the Service Stubs](#)
- [Examining the Generated Classes](#)
- [Extending the Generated Classes](#)
- [Customizing the Look and Feel](#)

Generating the Service Stubs

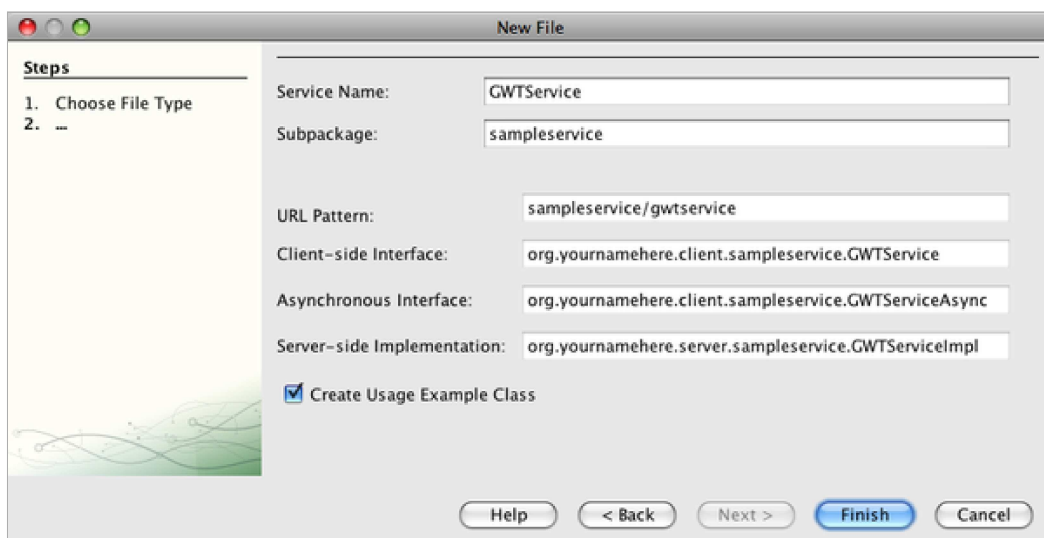
The NetBeans GWT plugin provides a wizard for creating an [RPC](#) service. The wizard generates the basic service classes for you. In this subsection, you are introduced to this wizard.

1. Click the New File icon in the IDE (). In the New File wizard, the Google Web Toolkit category shows a file template named GWT RPC Service:



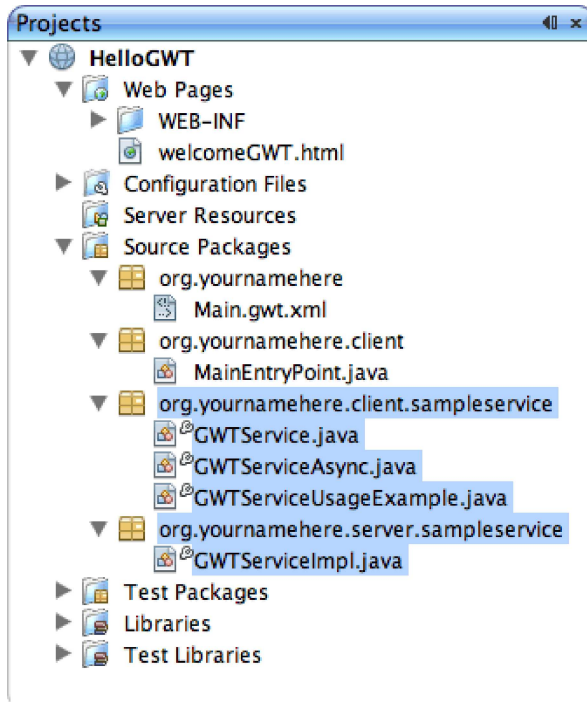
Click Next.

2. Optionally, fill in a subpackage where the files that will be generated will be stored. For purposes of this tutorial, type `sampleservice` as the Subpackage field:



Note: By leaving the Create Usage Example Class option selected in this step, you allow the IDE to generate the `GWTServiceUsageExample` class, which can be used to invoke the service.

- Click Finish. The files listed in the New File wizard (shown in the above screenshot) are generated, and the Projects window automatically updates to reflect changes:



Examining the Generated Classes

The GWT RPC Service wizard creates several source files. Here, look at the files and see how they relate to each other within the context of a GWT service.

For an extended description of GWT service classes, see [Creating Services](#).

- **GWTService:** The client-side definition of the service. This interface extends the [RemoteService](#) tag interface.

```
package org.yournamehere.client.sampleservice;

import com.google.gwt.user.client.rpc.RemoteService;

public interface GWTService extends RemoteService {
    public String myMethod(String s);
}
```

- **GWTServiceImpl:** The servlet that implements the GWTService interface and provides the functionality for retrieving a random quote via RPC.

```
package org.yournamehere.server.sampleservice;

import com.google.gwt.user.server.rpc.RemoteServiceServlet;
import org.yournamehere.client.sampleservice.GWTService;

public class GWTServiceImpl extends RemoteServiceServlet implements GWTService {

    public String myMethod(String s) {
        // Do something interesting with 's' here on the server.
        return "Server says: " + s;
    }

}
```

- **GWTServiceAsync:** An asynchronous interface, which is based on the original GWTService interface. It provides a callback object that enables the asynchronous communication between server and client.

```
package org.yournamehere.client.sampleservice;

import com.google.gwt.user.client.rpc.AsyncCallback;

public interface GWTServiceAsync {
    public void myMethod(String s, AsyncCallback callback);
}
```

- **GWTServiceUsageExample:** The sample user interface generated as a test client. It can be used to invoke the service.

```
package org.yournamehere.client.sampleservice;

import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.rpc.ServiceDefTarget;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.VerticalPanel;
import com.google.gwt.user.client.ui.Widget;

public class GWTServiceUsageExample extends VerticalPanel {
    private Label lblServerReply = new Label();
    private TextBox txtUserInput = new TextBox();
    private Button btnSend = new Button("Send to server");

    public GWTServiceUsageExample() {
        add(new Label("Input your text: "));
        add(txtUserInput);
        add(btnSend);
        add(lblServerReply);

        // Create an asynchronous callback to handle the result.
        final AsyncCallback callback = new AsyncCallback() {
            public void onSuccess(Object result) {
                lblServerReply.setText((String)result);
            }

            public void onFailure(Throwable caught) {
                lblServerReply.setText("Communication failed");
            }
        };

        // Listen for the button clicks
        btnSend.addClickListener(new ClickListener(){
            public void onClick(Widget w) {
                // Make remote call. Control flow will continue immediately and later
                // 'callback' will be invoked when the RPC completes.
                getService().myMethod(txtUserInput.getText(), callback);
            }
        });
    }

    public static GWTServiceAsync getService(){
        // Create the client proxy. Note that although you are creating the
        // service interface proper, you cast the result to the asynchronous
        // version of
        // the interface. The cast is always safe because the generated proxy
        // implements the asynchronous interface automatically.
        GWTServiceAsync service = (GWTServiceAsync) GWT.create(GWTService.class);
        // Specify the URL at which our service implementation is running.
        // Note that the target URL must reside on the same domain and port from
        // which the host page was served.
        //
        ServiceDefTarget endpoint = (ServiceDefTarget) service;
        String moduleRelativeURL = GWT.getModuleBaseURL() + "sampleservice/gwtservice";
        endpoint.setServiceEntryPoint(moduleRelativeURL);
        return service;
    }
}
```

Now, modify the entry point class to invoke the service by instantiating a GWTServiceUsageExample object.

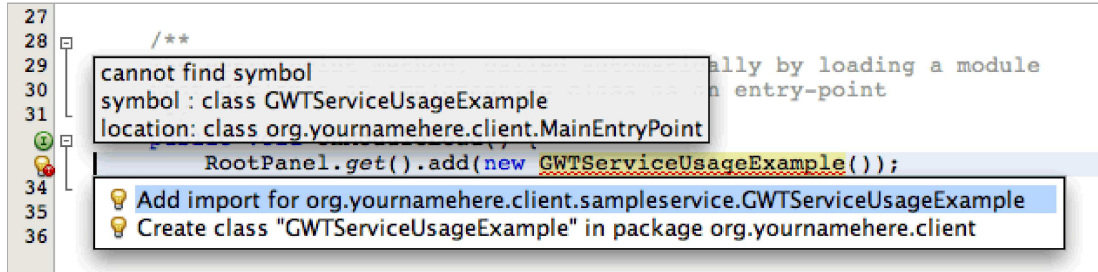
1. In MainEntryPoint.java, change the onModuleLoad() method to the following:


```

public void onModuleLoad() {
    RootPanel.get().add(new GWTServiceUsageExample());
}

```

Note: After modifying the `onModuleLoad()` method, you need to add an import statement to the `sampleservice.GWTServiceUsageExample` class. To do so, click on the lightbulb that displays in the left column where the `GWTServiceUsageExample()` is listed in the Source Editor and choose Add Import for `org.yournamehere.client.sampleservice.GWTServiceUsageExample`:



2. In the Projects window, right-click the project node and choose Run. The server starts, if it is not running already. The project is recompiled and deployed to the server. The browser opens to display a text field. Type in a message and click the button. A label appears with the message you sent:

Input your text:

Send to server

Server says: All you need is love

You have successfully tested the generated service. In the next section, you can do something useful with it, now that you know it works.

Extending the Generated Classes

In this section, you tweak and extend the classes that were examined in the previous subsection. At the end of this subsection, you will have completed the AJAX random quote generator.

1. In the `GWTService` class, remove the `String` parameter from `myMethod()` so that the interface is as follows:

```

public interface GWTService extends RemoteService {
    public String myMethod();
}

```

2. Do the same in the asynchronous service, so that the interface is like this:

```

public interface GWTServiceAsync {
    public void myMethod(AsyncCallback callback);
}

```

3. In `GWTServiceImpl`, implement the interface as follows:


```

public class GWTServiceImpl extends RemoteServiceServlet implements GWTService {

    private Random randomizer = new Random();
    private static final long serialVersionUID = -15020842597334403L;
    private static List quotes = new ArrayList();

    static {
        quotes.add("No great thing is created suddenly - Epictetus");
        quotes.add("Well done is better than well said - Ben Franklin");
        quotes.add("No wind favors he who has no destined port - Montaigne");
        quotes.add("Sometimes even to live is an act of courage - Seneca");
        quotes.add("Know thyself - Socrates");
    }

    public String myMethod() {
        return (String) quotes.get(randomizer.nextInt(4));
    }

}

```

Note: Right-click anywhere in the Source Editor and choose Fix Imports to let the IDE create the correct import statements. When you do so, make sure to select `java.util.Random` instead of `com.google.gwt.user.client.Random`:



- Copy the `getService()` method from the generated usage example class (`GWTServiceUsageExample`) and paste it into the entry point class (`MainEntryPoint`).

```

public static GWTServiceAsync getService() {
    // Create the client proxy. Note that although you are creating the
    // service interface proper, you cast the result to the asynchronous
    // version of
    // the interface. The cast is always safe because the generated proxy
    // implements the asynchronous interface automatically.
    GWTServiceAsync service = (GWTServiceAsync) GWT.create(GWTService.class);
    // Specify the URL at which our service implementation is running.
    // Note that the target URL must reside on the same domain and port from
    // which the host page was served.
    //
    ServiceDefTarget endpoint = (ServiceDefTarget) service;
    String moduleRelativeURL = GWT.getModuleBaseURL() + "sampleservice/gwtservice";
    endpoint.setServiceEntryPoint(moduleRelativeURL);
    return service;
}

```

- Change the `onModuleLoad()` method in the entry point class to the following:

```

/**
 * The entry point method, called automatically by loading a module
 * that declares an implementing class as an entry-point
 */

public void onModuleLoad() {

    final Label quoteText = new Label();

    Timer timer = new Timer() {

        public void run() {
            //create an async callback to handle the result:
            AsyncCallback callback = new AsyncCallback() {

                public void onFailure(Throwable arg0) {
                    //display error text if we can't get the quote:
                    quoteText.setText("Failed to get a quote");
                }

                public void onSuccess(Object result) {
                    //display the retrieved quote in the label:
                    quoteText.setText((String) result);
                }
            };
            getService().myMethod(callback);
        }
    };

    timer.scheduleRepeating(1000);
    RootPanel.get().add(quoteText);
}

```

Note: Right-click in the Source Editor and choose Fix Imports in order to let the IDE create the correct import statements. When you do so, make sure to select `com.google.gwt.user.client.Timer`, instead of `java.util.Timer`.

6. Delete the `GWTServiceUsageExample` class. To do so, right-click the class node in the Projects window and choose Delete.
7. Run the project. When the application is deployed and the browser opens, you see a new quote received from the server, every other second:

Sometimes even to live is an act of courage - Seneca

In the next section, you apply a stylesheet to change the look and feel of the quotes.

Customizing the Look and Feel

In this section, you attach a stylesheet to the HTML host page. You also refer to it in the entry point class. Specifically, you need to set the style name of the label in the entry point class to the name of the style in the stylesheet. At runtime, GWT connects the style to the label and displays a customized label in the browser.

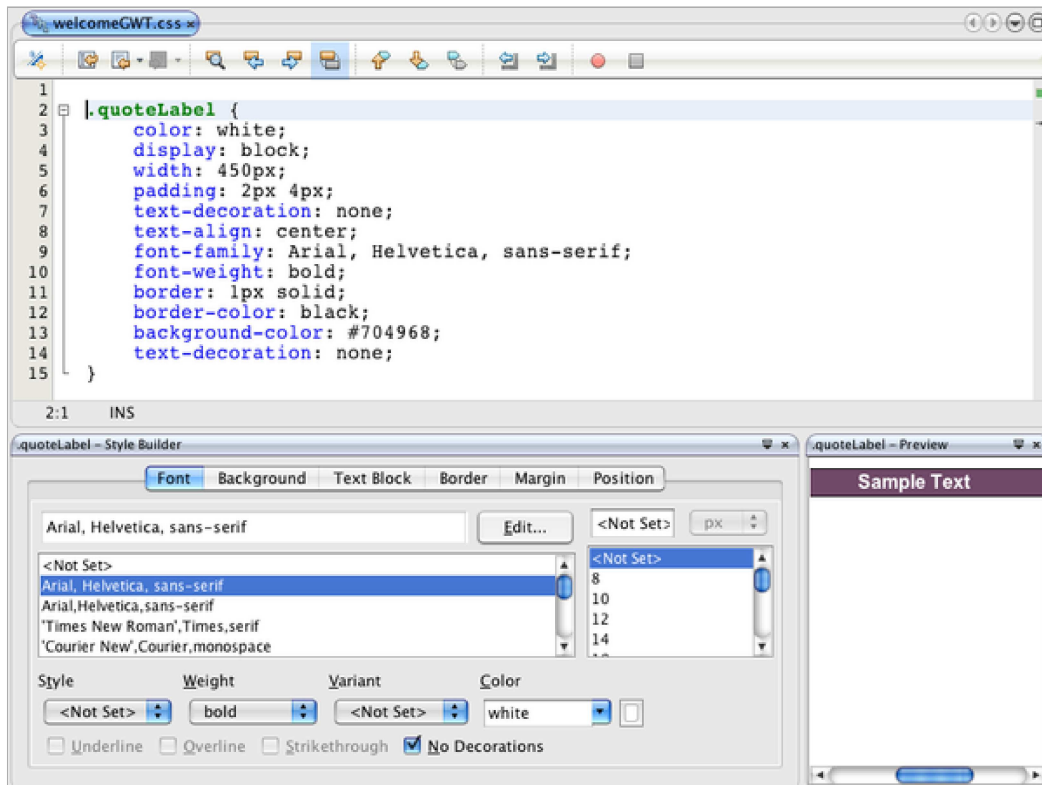
1. Create a stylesheet called `welcomeGWT.css`. To create the file, right-click the Web Pages node in the Projects window and choose New > Other.
2. In the New File wizard that displays, choose Other under Categories then choose Cascading Style Sheet under File Types. When you complete the wizard, the new empty file opens in the Source Editor.
3. Create the following `quoteLabel` selector for the new stylesheet:

```

.quoteLabel {
    color: white;
    display: block;
    width: 450px;
    padding: 2px 4px;
    text-decoration: none;
    text-align: center;
    font-family: Arial, Helvetica, sans-serif;
    font-weight: bold;
    border: 1px solid;
    border-color: black;
    background-color: #704968;
    text-decoration: none;
}

```

The stylesheet editor should now show the following:



To display the CSS Preview and Style Builder, choose Window > Other from the main menu.

4. Attach the stylesheet to the HTML page (welcomeGWT.html). At the same time, add some text to introduce the application to the user. The new parts of the HTML page are highlighted below in **bold**:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<html>
  <head>
    <meta name='gwt:module' content='org.yournamehere.Main=org.yournamehere.Main'>
    <title>Main</title>
    <link rel="stylesheet" type="text/css" href="welcomeGWT.css">
  </head>

  <body>
    <script language="javascript" src="org.yournamehere.Main/org.yournamehere.Main.no

    <p>This is an AJAX application that retrieves a random quote from
      the Random Quote service every second. The data is retrieved
      and the quote updated without refreshing the page!</p>

  </body>
</html>

```

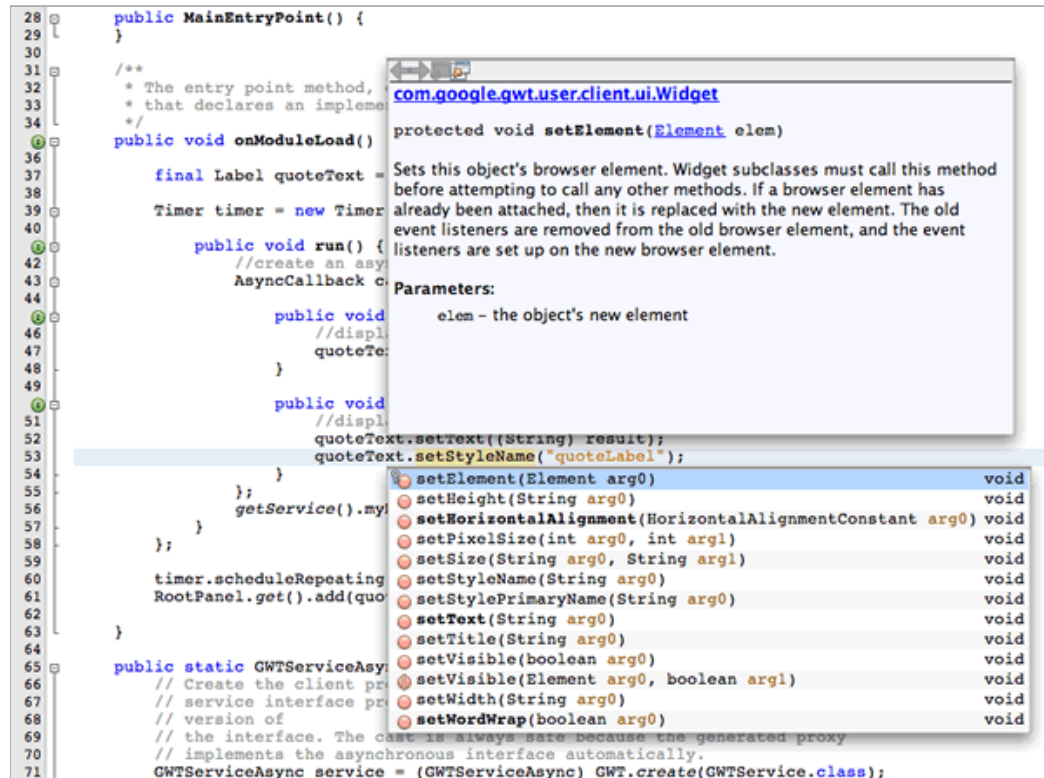
5. In the entry point class (MainEntryPoint.java), specify that, upon success, the style defined in the stylesheet should be applied to the label. The new line is highlighted in **bold** below:

```

public void onSuccess(Object result) {
    //display the retrieved quote in the label:
    quoteText.setText((String) result);
    quoteText.setStyleName("quoteLabel");
}

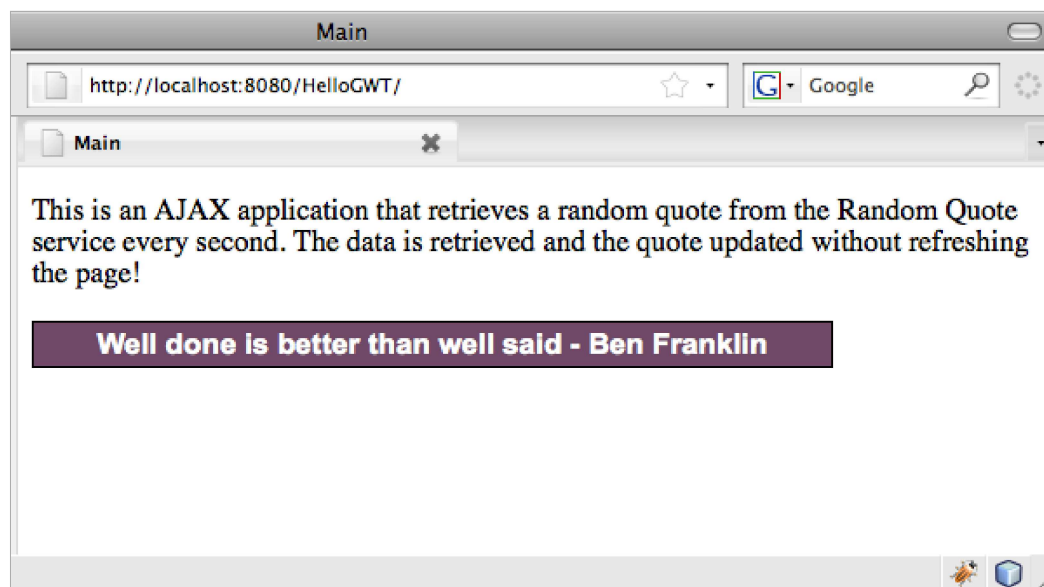
```

As you code, notice that code completion can help you, suggesting ways of completing the code and showing you the related Javadoc:



You can invoke code completion suggestions by pressing Ctrl-Space.

6. In the Projects window, right-click the project node and choose Run. This time, the label is shown with a custom style, using the stylesheet you created in this subsection:



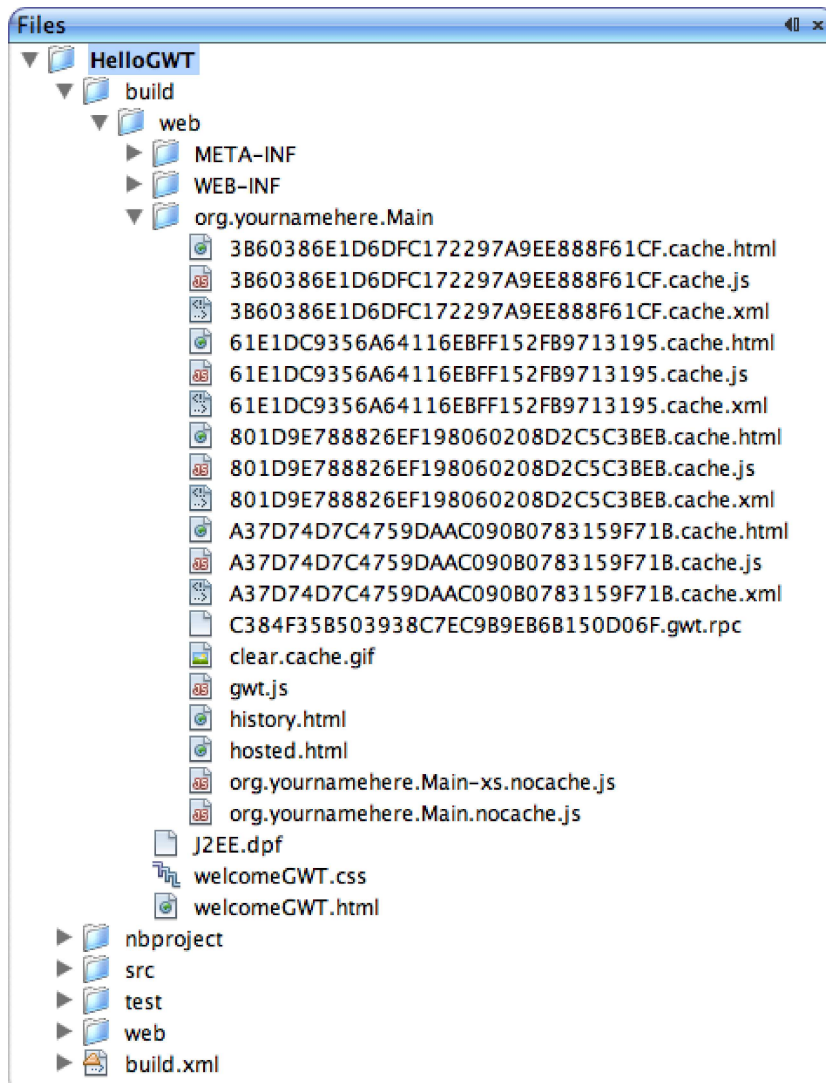
Conclusion

In this tutorial, you have learned the following:

- What a typical application source structure looks like in a Google Web Toolkit application.

- How Google Web Toolkit artifacts relate to each other.
- How to set up the IDE to use the Google Web Toolkit.
- What the tools are that are available to you in the IDE, specifically for using the Google Web Toolkit.

Finally, open the Files window (Ctrl-2) and expand the build folder. (If the build folder is not present, you need to build the project again in order to have the IDE regenerate the build folder.) You should see something like this:



This folder is generated automatically by GWT when the application is compiled. The folder consists of a ready-to-deploy version of the client application. It contains, among other files, the following:

- **gwt.js**: A generated JavaScript file that contains bootstrap code for loading and initializing the GWT framework.
- **history.html**: An HTML file that provides history management support.
- **xxx-cache.html**, **xxx-cache.js**, and **xxx-cache.xml**: An HTML, JavaScript, and XML file are generated per supported browser. These contain the code generated by the compilation of the Java source files in the client and server packages.

In conclusion, because the GWT framework handles browser-related code generation, as well as the creation of the lower-level XMLHttpRequest API code, you can take advantage of the framework to focus on the functionality that you want your applications to provide. Hence, as stated in the introduction, GWT lets you avoid the headaches associated with browser compatibility while simultaneously letting you offer users the same dynamic, standards-compliant experience that the Web 2.0 world typically provides. As this tutorial demonstrated, you can apply the GWT framework to write your complete front end in Java, because you know that you can let the GWT compiler convert Java classes to browser-compliant JavaScript and HTML. And, as also demonstrated, the IDE provides a complete set of tools for making all this easy and efficient, without the need to hand-code a GWT application's basic infrastructure.

[Send Us Your Feedback](#)

See Also

This concludes the Introduction to the Google Web Toolkit Framework tutorial. For related and more advanced material, see the following resources:

- [Using Google Web Toolkit \(GWT\) and NetBeans for Building AJAX Applications](#)

- [NetBeans Google Web Toolkit Project Page](#)
- [Google Web Toolkit Developer's Guide](#)
- [GWT4NB under the hood](#)
- [Google Web Toolkit Blog](#)