Database Access in GWT – The Missing Tutorial

A programmer's guide on integrating MySQL into Google Web Toolkit Applications using the Eclipse Programming Environment

Bastian Tenbergen

bastian@tenbergen.org

http://www.tenbergen.org

Human-Computer Interaction MA Program

Department of Psychology

Department of Computer Science

State University of New York, College at Oswego

Oswego, NY, USA

1. Abstract.

This tutorial explains the principle method of adding MySQL support to dynamic GWT Web Applications using asynchronous callback with Remote Procedure Call providers. Even though there are already plenty information on this topic available on the Internet, a detailed, step-by-step explanation seems to be missing so far. The mission of this tutorial is to provide such an explanation in a more detailed fashion than the GWT Examples [3] provide.

2. Table of Content.

3. Preamble.

This tutorial is my attempt to summarize my findings in search the Internet on how to do so, along with massive findings that I discovered myself in "taming" the beast of GWT and MySQL. I neither claim this document to be correct, nor flawlessly correct. If you notice any glitches, please contact me for revisions.

4. Introduction.

Why this tutorial? Isn't there enough info on this on the Intertubes? Well, yes. And no. There are a few good sources, especially the ones from the makers [3], but none really explains the special nuances, like that the DB driver has to live in a Servlet. In fact, that is the most important take-home message: The actual database queries are handled by a **Servlet** and you need to make **RPC calls**, using a **callback handler** to have the client side GUI (recall that the GWT application is executed client side in the browser, not on the server!) talk to the database. The mission of this tutorial is hence to show the necessary steps needed to integrate MySQL with GWT applications and explain what happens at each step.

5. Assumptions.

I will assume that you have an understanding of how to do GWT in first place. At least you should know that GWT is a Java front-end for dynamic JavaScript/AJAX Web Applications, that you the main class implements the EntryPoint interface and all the other stuff to get a very basic GWT App working in your browser. I will also assume that you know basic Java, like classpaths, dependencies and such. I also will assume that you have some dynamic web server, like Tomcat, at your disposal, and that you can use the world's largest knowledge base [4] to find out how to configure it.

6. The Missing Tutorial.

In this little tutorial, we will implement a simple thing – a LoginScreen that will allow you to enter a user name and a password, and authenticate this combination. Assuming, of course, you have a table in the database that lists a user name, with a password. Let's just assume your GWT application has just this one class, LoginScreen, that implements EntryPoint and hence has an public void onModuleLoad() method.

6.1. The Short Version. For the impatient.

Do the following steps and it will work (assuming your GWT is up and running and all you have to do is implement the DB connection):

1. Add your DB connector JAR to your Tomcat's classpath.
2. Add **TWO** interfaces to your project's client package that specify all the methods you can call from within your GWT user interface: One for synchronous callback, one for asynchronous callback. In those, specify the methods that will provide DB queries for you.
3. Add a DB connection class to your project's server package that will handle all the DB queries, just like with any other Java application. This class must extend RemoteServlet and implement your synchronous interface from point 2.
4. Add a RPC provider that implements the asynchronous interface to the GWT application component that needs to query the DB. Also add a nested class to that GUI component that implements AsyncCallback. This handler will do stuff with your GUI depending on the query result.
5. Now you can call the method specified by your interfaces using the RPC provider and applying the callback handler.
6. Create some serializable objects on your client side - in case you want to send more than primitive types (including Strings) to and from the database.
7. Stir, serve, and enjoy - you're done :-)

Now for the detailed stuff.

6.2. Necessary Software.

Of course, you will need GWT [5]. Download it [6] and install it, of course. I strongly recommend to use some form of Integrated Development Environment (IDE) for your development. In fact, without one, you are kind of screwed. Get yourself Eclipse [7], version 3.4 or higher, also known as Ganymede [8]. Make sure to get the Java Enterprise version of Eclipse, as this comes with all the goodies for Web Programming - even with said dynamic Web Servers. The easiest way to develop GWT stuff using Eclipse is getting a GWT focused plug-in. I recommend Cypal Studio [9] - it is free, and the documentation is just what you need, even though the docs [10] are somewhat sparse. Follow their how-to on getting started integrating Cypal Studio and making your first GWT App. You will also need a database you can use - get WAMP [11], as it comes with all you need for that. MySQL, a good free database server and some PHP-based management software. Make sure MySQL in WAMP is running while you develop. Of course, you can use some remote MySQL server, too, like the one your ISP provides you with.

6.3. Add MySQL to Java.

Next, let's take a look on what you have to do to get everything done right. You need to make sure that you have your database driver - get the MySQL Java DB Connector [12] that fits to the MySQL version that came with your WAMP. This should be a JAR file, which you have to add to your classpath. **Careful now!** You need to add it to your Eclipse classpath *and* your Tomcat classpath! Otherwise Eclipse won't let you access the MySQL API and/or Tomcat can't find the DB driver later on. Here is how to do it: Remember that Eclipse specifies a classpath for each project individually. Right-click on your project, click on properties, then on Java Build Path. Click on Add External Jar... and find the MySQL DB Connector JAR file to the project from your file system. This will allow you to have access to the MySQL Java Connector API from within Eclipse - this way the auto completion and syntax checking and all that nice IDE stuff works. While you are at it, in the same window, make sure Cypal Studio added the file "gwt-servlet.jar" and "gwt-user.jar" libraries to the build path of your project - for the same reason ;-) Next, check that the WebContent folder in your project has the same jar added to it. Do so by expanding the WebContent folder, navigate to WEB-INF, then lib. Into there, you can just drag-and-drop the MySQL Connector JAR file from your file system - Eclipse will do the behind the scenes adding yourself.

6.4. Add MySQL configuration to GWT.

Now, you have to tell the myriad of XML configuration files of GWT that you have, in fact, a Servlet that supports MySQL. Well, you will have one, after the next step. Under "Java Resources: src" is all your Java configuration and source file stuff. Navigate to the folder of your project that is named after the package that all your classes are in. This should be the first one down from the "Java Resources: src" folder and look something like `com.yourdomain.projectname` or whatever you have chosen as your project package. In there, you will find not only the client and server sub packages, but also a folder that contains a sub folder public and an XML configuration file, named after your main class. Remember, that in your GWT application, you need some class that implements the `EntryPoint` interface from the `com.google.gwt.core.client` package. We said earlier that this class is `LoginScreen`, so the configuration file you want to look for is `LoginScreen.gwt.xml`. Add the following line to this configuration point:

```
<servlet class="com.yourdomain.projectname.server.MySQLConnection"
path="/MySQLConnection" />
```

The class's package name must be the one you have set for your project. Server is a sub package that is there anyways, because GWT wants it. MySQLConnection may be called something else - this is a class you will write later on. The path argument is rather is arbitrary, but must match up with the RPC initialization inside Java class. More about that later, too.
In the end, you should have some form of configuration XML file like this:
LoginScreen.gwt.xml:

```
<module>
        <!-- Inherit the core Web Toolkit stuff. -->
        <inherits name='com.google.gwt.user.User'/>
        <!-- Specify the app entry point class.  -->
        <entry-point class='com.yourdomain.projectname.client.LoginScreen'/>
    <!-- servlet context - path is arbritray, but must match up with the rpc init
inside java class -->
    <!-- Tomcat will listen for this from the server and waits for rpc request in
this context -->
        <servlet class="com.yourdomain.projectname.server.MySQLConnection"
```

```
path="/MySQLConnection" />
        <inherits name="com.google.gwt.user.theme.standard.Standard"/>
        <inherits name="com.google.gwt.user.theme.chrome.Chrome"/>
        <inherits name="com.google.gwt.user.theme.dark.Dark"/>
</module>
```

6.5. Clean your project.

This will make sure that all your configurations are set-up correctly, re-configure your webserver and compile the GWT project so that it will run properly again. Do so, by clicking in Eclipse's menu bar on Project, then Clean... and select your project. It will clean, compile, and prepare for execution, potentially add missing configurations and repairing ambiguities. You may need to help out a little bit, when Eclipse asks you, but that should be trivial.

6.6. Make the client side be aware of the Servlet.

Now, we need to make some client side preparations for stuff to happen. You need to create **TWO** interfaces. Both must live in the `com.yourdomain.projectname.client` package. You are allowed to put it in a sub ackage, but it must be in client, or the Ninjas will get you. Now, as I said, two interfaces: one for synchronous callback, one for asynchronous callback. Let's say, you call the synchronous interface `DBConnection`. This interface **must** extend the `com.google.gwt.user.client.rpc.RemoteService` interface. In this interface, you may specify all the methods you'd like to use to connect to the database. Let it be one big fat method, that's fine. But I recommend to make one method for every specific update, insert, delete, whatever... this is the better way, I think. You can give it any return type, but for non-primitive types, you will need to create serializable objects. More about that later. Now, ideally, Cypal Studio will have created the asynchronous interface for you and call it `DBConnectionAsync`. As you can see, the asynchronous interface specifies the same methods you have written into the synchronous interface, with two differences: The return type is always void and every method has an additional parameter: an `com.google.gwt.user.client.rpc.AsyncCallback` object. `AsyncCallback` is in fact an interface, but - again - more on that later. Every `AsyncCallback` parameter is equipped with Java Generics, the generic type being the return type of the method in the synchronous version. Don't try to change it - in fact, you need to make sure that what I just explained in fact is automatically done by

Cypal Studio. So go ahead, double-check, that you have two interfaces that are set up in the described way. Done? Good. You hence should have two files like this:

DBConnection.java:

```
public interface DBConnection extends RemoteService {
     public User authenticateUser(String user, String pass);
}
```

DBConnectionAsync.java:

```
public interface DBConnectionAsync {
     public void authenticateUser(String user, String pass, AsyncCallback<User>
callback);
}
```

Note the User object as return value... more on that in the next paragraph. Of course you can - instead of a User object - also just return a Boolean if authentication was successful - your choice. Depends on how you want to use your authentication, I guess.

6.7. Create serializable objects for callbacks.

Also on the client side (i.e. in the `client` package), create one object for every non-primitive return value that you have depicted in the methods in your interfaces. You don't have to do create objects for primitive types, like Boolean, Integer, Double, or anything in the `java.lang`, or `java.util` packages. GWT will do that for you and simply use JavaScript equivalents. You can even use Arrays and `java.util.Vector` with Generics. The only thing you can't use is `java.lang.Thread`, for obvious reasons (wanna know those reasons? This shall be a thought excerise for you. Hint: You can't serialize Threads in plain vanilla Java anyways. You know why?)

GWT is VERY picky [13] with what is serializable and what not, as GWT's idea of serializing an object is significantly different from Java's (and pretty much everyone else's) way of doing so. A class in GWT is serializable, if (according to the website [13]) **one** of the following holds true:

- is primitive, such as `char`, `byte`, `short`, `int`, `long`, `boolean`, `float`, or `double`;
- is `String`, `Date`, or a primitive wrapper such as `Character`, `Byte`, `Short`, `Integer`, `Long`, `Boolean`, `Float`, or `Double`;

- is an array of serializable types (including other serializable arrays);
- is a serializable user-defined class; or
- has at least one serializable subclass

A user-defined class is serializable, if **all** of the following are true (according to the website [13], again):

1. it is assignable to IsSerializable or Serializable, either because it directly implements one of these interfaces or because it derives from a superclass that does
2. all non-`final`, non-`transient` instance fields are themselves serializable, and
3. it has a `public` default (zero argument) constructor

The objects you create must hence fulfill the above requirements. Here is an example object that we could use by our authenticateUser method instead of a Boolean return value:

User.java:

```
import com.google.gwt.user.client.rpc.IsSerializable;
public class User implements IsSerializable {
    private String username;
    private String password;
    @SuppressWarnings("unused")
    private User() {
        //just here because GWT wants it.
    }
    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }
}
```

Note, how the zero parameter constructor is present, but private. This is just to illustrate that the constructor MUST be there, but not necessarily be used. You can hide it by making it private. I used the @SuppressWarnings("unused") Java annotation to make the Java compiler shut up about unused private method elements in the class... apparently javac doesn't know about GWT (surprise...). You don't have to do this, though, but it is good code.

6.8. Create the Servlet.

Finally, you get to create what's actually important for the DB queries. Up to now, all we have done was some "pluming" around GWT, Java, and your Tomcat to make what comes now as simple and smooth as possible. On your server package, i.e. `com.yourdomain.projectname.server`, create a class called `MySQLConnection`. Class name and package **must** be identical to what you have specified in the class argument in your `LoginScreen.gwt.xml` configuration file. Remember that one? Double-check, that class name and package name is correct.

This `MySQLConnection` class will do all the DB work for you. As I said, it is a Servlet, hence it must extend the `com.google.gwt.user.server.rpc.RemoteServiceServlet` class. Additionally, it **absolutely must** implement the synchronous interface from your client side. Hence, you will write your queries for everything you want to do with your database right in this class. Here is a full and good implementation of the class:

MySQLConnection.java:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Vector;
import com.google.gwt.user.server.rpc.RemoteServiceServlet;
import com.yourdomain.projectname.client.User;
public class MySQLConnection extends RemoteServiceServlet implements DBConnection {
        private Connection conn = null;
        private String status;
        private String url  = "jdbc:mysql://yourDBserver/yourDBname";
        private String user = "DBuser";
        private String pass = "DBpass";
        public MySQLConnection() {
                try {
                        Class.forName("com.mysql.jdbc.Driver").newInstance();
                        conn = DriverManager.getConnection(url, user, pass);
                } catch (Exception e) {
                        //NEVER catch exceptions like this
                }
```

```
	}
	public int authenticateUser(String user, String pass) {
		User user;
		try {
			PreparedStatement ps = conn.prepareStatement(
					"select readonly * from users where
username = \"" + user + "\" AND " +
					"password = \"" + pass + "\""
			);
			ResultSet result = ps.executeQuery();
			while (result.next()) {
				user = new User(result.getString(1),
result.getString(2));
			}
			result.close();
			ps.close();
		} catch (SQLException sqle) {
			//do stuff on fail
		}
		return user;
	}
}
```

Note a couple of things here. Firstly, **NEVER, EVER,** catch exceptions the way I do in here... by catching a generic Exception object. That's bad code and not only the Ninjas, but also the Pirates will get you. Secondly, I included a detailed list of import statements so that you can see what is actually made use of. Note the User object - this is the serializable object you created on client side. The URL to the database can be anything, but must be the URL or your MySQL server. Could be localhost or your ISP. Who knows. The username and password here are NOT what you are trying to authenticate in your GWT application. In this case, the username and password are those that your ISP has given you to connect to your database. I assume that you will have some table called user in your DB with the columns "user" and "password" and we hence query the database to `SELECT * FROM user WHERE user = 'your authentication user name' AND password = 'your authentication password'.`

That's it! After that, just do stuff with the result, as in create a User object (the seriablizable one from

the client package of your GWT app) and return it.

Let's see what we have done so far. We have:

- added MySQL Connector/J to Eclipse and Tomcat
- Configured the GWT project XML file to be aware of a DB connector Servlet
- Created the synchronous and asynchronous interface
- Created the serializable database objects needed for callback
- Created the Servlet that will query the actual database.

Not bad at all. What's next is making your client side GWT GUI issue the RPC and do stuff with the result.

6.9. Prepare an RPC Provider in your GWT App.

Of course, all this pluming and coding is worth nothing if you have nothing using it. And since you want to feed your database output into the GUI and do stuff with it accordingly, you will need to make your GUI component call the database. For that, you need to create an RPC provider. I typically do this once when the application gets loaded, as the RPC call provider may be shared across all database accessing GUI components. In your `EntryPoint` class - `LoginScreen.java` - add a variable of type `DBConnectionAsync`. Since `DBConnectionAsync` is an interface, you will not be able to instantiate it directly, but that doesn't matter - you will just ask GWT to factor a method for you. This is how it may look like:

LoginScreen.java:

```
import com.google.gwt.core.client.EntryPoint;
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.RootPanel;
public class LoginScreen implements EntryPoint {

        private DBConnectionAsync rpc;
```

```
        public LoginScreen() {
          rpc = (DBConnectionAsync) GWT.create(DBConnection.class);
        ServiceDefTarget target = (ServiceDefTarget) rpc;

        // The path 'MySQLConnection' is determined in ./public/LoginScreen.gwt.xml
        // This path directs Tomcat to listen for this context on the server side,
        // thus intercepting the rpc requests.
        String moduleRelativeURL = GWT.getModuleBaseURL() + "MySQLConnection";
        target.setServiceEntryPoint(moduleRelativeURL);
        }
        public void onModuleLoad() {
            RootPanel.get().add(this);
            }
        }
}
```

The factoring mechanism is similar to what you do in the DB Driver instantiation: You ask the static method create in the GWT class to return an implementation of your `DBConnection` interface. Conveniently, the only known implementation is your `MySQLConnection` Servlet, which is hence stuffed into your `DBConnectionAsync` variable called "rpc" and will hence act as your Remote Procedure Call Provider. Next, you need to set the service entry point to the path of your connection provider. Remember in your `LoginScreen.gwt.xml` file? You have specified a path argument in your `servlet` tag. The path URL that you are providing here **must** be the one you have provided in the XML file, otherwise things will fail.

6.10. Make Remote Procedure Calls from your GWT application.

Now, all you have left to do is create your actual GUI in this `EntryPoint` class, preferably with some buttons, and upon clicking on of these buttons, do something. This "something" is in fact issuing the RPC call with the call provider. Of course, you will need an callback handler that takes the result and does something with it. What I always do to accomplish that is create a nested, private class that implements the `com.google.gwt.user.client.rpc.AsyncCallback` interface, with appropriate Generic type. This interface specifies an `onSuccess` and `onFailure` method, depending on whether or not the RPC call failed. Here is how it could look like:

nested, private AuthenticationHandler class:

```
private class AuthenticationHandler<T> implements AsyncCallback<User> {

     public void onFailure(Throwable ex) {

          RootPanel.get().add(new HTML("RPC call failed. :-(");
     }

     public void onSuccess(User result) {

          //do stuff on success with GUI, like load the next GUI element

     }

}
```

Note the use of Generics to retrieve the correct result type. The User object that is used here is the same, serializable object that you have created much earlier.
Now, all you have to do is call the RPC provider. Let's assume you have a button called OK, you will need a `ClickListener` and issue the RPC call when OK is clicked. The RPC call needs an instantiation of the `AsyncCallback` class you have just written. Like this:
OK button ClickListener:

```
public void onClick(Widget sender) {
     if (sender.equals(OK)) {
          AsynchCallback<User> callback = new AuthenticationHandler<User>();
          rpc.authenticateUser(username, password, callback);
     }
}
```

That's it! Essentially, you are done at this point! Below is an example implementation of how the entire, RPC enabled, callback handling, DB query issuing `LoginScreen` class may look like, with GUI components and all:

complete LoginScreen.java:

```
import com.google.gwt.core.client.EntryPoint;
```

```
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.*;
public class LoginScreen implements EntryPoint, ClickListener {

        private DBConnectionAsync rpc;
          private TextBox usernameBox;
          private TextBox passwordBox;
          private Button OK;

        public LoginScreen() {
          rpc = (DBConnectionAsync) GWT.create(DBConnection.class);
        ServiceDefTarget target = (ServiceDefTarget) rpc;

        // The path 'MySQLConnection' is determined in ./public/LoginScreen.gwt.xml
        // This path directs Tomcat to listen for this context on the server side,
        // thus intercepting the rpc requests.
        String moduleRelativeURL = GWT.getModuleBaseURL() + "MySQLConnection";
        target.setServiceEntryPoint(moduleRelativeURL);
           initGUI();
        }
        public void onModuleLoad() {
            RootPanel.get().add(this);
            }
        }
          private void initGUI() {
                Grid g = new Grid(3, 2);
            usernameBox = new TextBox();
                passwordBox = new TextBox();
                OK = new Button(„OK");
                g.setWidget(0, 0, new Label(„Username: „));
                g.setWidget(0, 1, usernameBox);
                g.setWidget(1, 0, new Label(„Password: „));
                g.setWidget(1, 1, passwordBox);
                g.setWidget(2, 1, OK);
          }
          public void onClick(Widget sender) {
```

```
                if (sender.equals(OK)) {
                  AsyncCallback<User> callback = new AuthenticationHandler<User>();
                        rpc.authenticateUser(usernameBox.getText(),
passwordBox.getText(),
                                                        callback);
                }
        }
        private class AuthenticationHandler<T> implements AsyncCallback<User> {
             public void onFailure(Throwable ex) {
            RootPanel.get().add(new HTML("RPC call failed. :-(");
                }
             public void onSuccess(User result) {
            //do stuff on success with GUI, like load the next GUI element
             }
        }
}
```

7. Summary.

OK, that's it! This completes the tutorial. If we did everything right, it should work just fine. It took me forever to write this stuff up, so, I'd appreciate any comments you may have. And it took me even longer to figure out how to do all this... I didn't have any help like you do :-p If you have questions, don't hesitate to post, or send me a mail.

Hope you like this little tutorial and find it helpful in your next GWT programming venture. Let me know of any successes, failures, and whatnot.

8. Acknowledgements.

This document was made possible, in part, by many kind souls in various forums across the Internet that post answers to questions of others. Those people are way to many to mention them all, but all of them shall be thanked much. Also thanks to Doug, Gil, and Wayne for their kind support in all my work – your help will never be forgotten.

9. References.

[1] Dewsbury, R. (2007). *Google Web Toolkit Applications*. Boston, MA: Addison-Wesley Professional.

[2] Geary, D. with Gordon, R. (2008). *Google Web Toolkit Solutions*. Boston, MA: Prentice Hall PTR.

[3] Google's GWT MySQL Example Project, accessed October, 24th, 2008

http://code.google.com/p/gwt-examples/wiki/project_MySQLConn

[4] Google Search Engine, accessed October, 24th, 2008

http://www.google.com/

[5] GWT – Google Web Toolkit, accessed October, 24th, 2008

http://code.google.com/webtoolkit/

[6] GWT Download Section, accessed October, 24th, 2008

http://code.google.com/webtoolkit/download.html

[7] The Eclipse Project, accessed October, 24th, 2008

http://www.eclipse.org/

[8] Eclipse IDE v3.4 "Ganymede", accessed October, 24th, 2008

http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/ganymede/SR1/eclipse-jee-ganymede-SR1-win32.zip

[9] Cypal Studio, GWT Plug-in for Eclipse Ganymede, accessed October, 24th, 2008

http://www.cypal.in/studio

[10] Cypal Studio Documentation, accessed October, 24th, 2008

http://www.cypal.in/studiodocs

[11] WAMPServer – Apache, MySQL, PHP on Windows, accessed October, 24th, 2008

http://www.wampserver.com/en/

[12] MySQL Connector/J, accessed October, 24th, 2008

http://www.mysql.com/products/connector/j/

[13] GWT Serializable Types, accessed October, 24th, 2008 http://www.gwtapps.com/doc/html/com.google.gwt.doc.DeveloperGuide.RemoteProcedureCalls.SerializableTypes.html

[14] Niederst, J. (1999). *Web Design in a Nutshell, 1st Edition.* O'Reilly.

[15] Silberschatz, A., Korth, H. F., Sudarshan, S. (2006). *Database System Concepts, 5th Edition.* McGraw-Hill Higher Education.