White Paper: Enterprise GWT

Combining Google Web Toolkit, Spring and Other Features to Build Enterprise Applications

By Dr Ant Kutschera & Barbara Spillmann
January 2010

Google Web Toolkit (GWT) provides developers with a powerful means of developing AJAX front ends without the worry of having to maintain complex Java script libraries to support multiple browsers and browser versions.

GWT also provides support for Remote Procedure Calls (RPC) to the server. Since April 2009 the Google App Engine has existed, which allows developers to deploy their GWT applications and also provides support for Java Data Objects (JDO) and the Java Persistence API (JPA).

However what is missing for GWT to be deployed to a modern Enterprise environment is a service framework providing dependency injection and inversion of control (IoC), transaction demarcation and security, such as that provided by Spring or Enterprise Java Beans (EJB) 3.0. Furthermore GWT does not define any patterns for User Interface designs, or composite widgets.

This paper describes how to successfully integrate Spring into a GWT Application with the aim of creating a fully scalable development framework for deployment in the Enterprise and beyond (including simple and small applications), with very little start up time being required, because you can download the demo application. It includes UI Patterns and composite widgets to improve the development of the front end. This GWT Demo Application is live at http://gwtdemo.maxant.co.uk and is available for download at http://www.maxant.co.uk/whitepapers.jsp.

General Architecture

Figure 1 shows the architecture of a GWT application which uses Spring and Hibernate in the back end. A GWT Application runs as Javascript in a Browser (1) and makes requests to the GWT RPC infrastructure running in a Web Application (2). RPC calls to and from the server, can use basic Java types such as String, Integer, etc. as well as collections out of the `java.lang` package. A full list of java packages and classes that GWT can translate automatically is provided in the JRE Emulation Reference (http://code.google.com/webtoolkit/doc/1.6/RefJreEmulation.html). However in order to keep interfaces efficient, Transfer Objects (TO) are used to pass information to and from the server (see discussion later).

Once inside the web container, an RPC call is delegated to the Spring framework (3) via a Façade which has four main functions. Firstly, since it is a Spring service it provides transaction demarcation. Secondly its job is to map from Transfer Objects into Persistence Objects (PO, which are used by Hibernate when mapping Java Objects to the Database and back) or a Business Object Model (BOM). The decision whether to map to a BOM first, or straight to POs, depends on the design of the back end. Thirdly, this Façade ensures that security is checked and the caller is authorised to make the call. Finally this Façade orchestrates the call by delegating to lower level services which implement the business logic or persistence logic.

Persistence is taken care of by Hibernate (4) and the database. Persistence Objects (PO) and their mappings are generated out of the database schema, which is used as the "master" in this generation process.

As such, the server side hierarchy (including layer responsibilities) is shown in Figure 2.

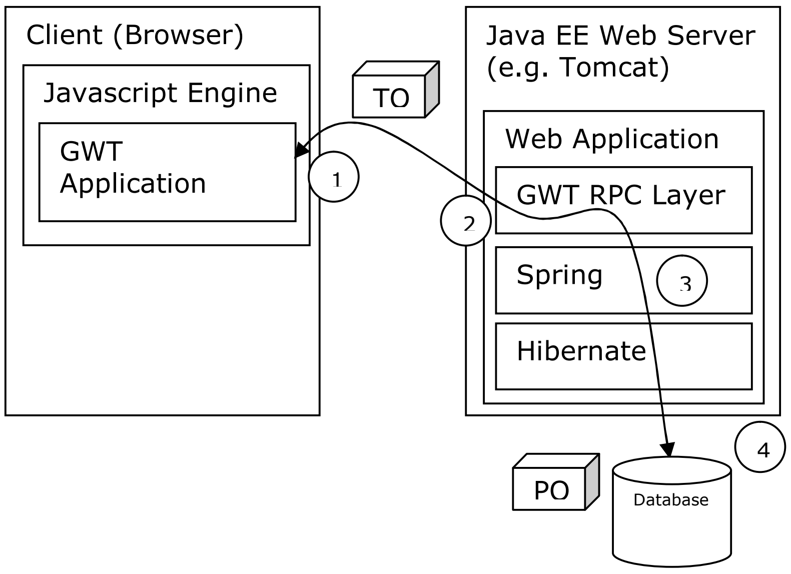
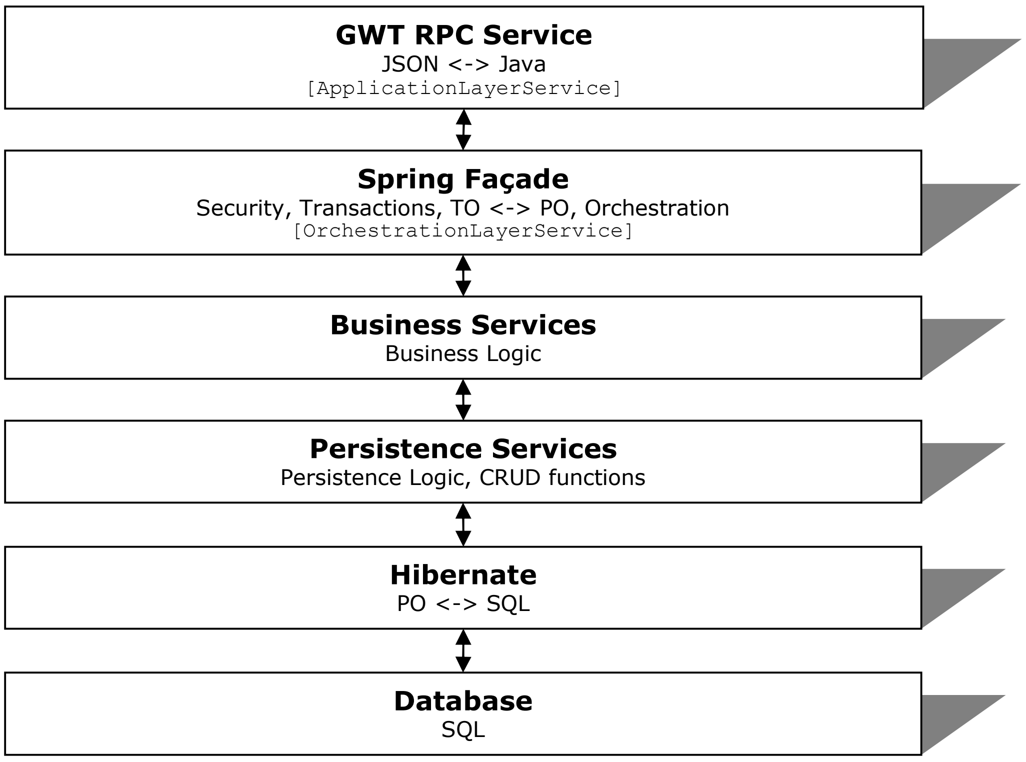


Figure 1: General Architecture



GWT RPC Service
JSON <-> Java
[ApplicationLayerService]
Spring Façade
Security, Transactions, TO <-> PO, Orchestration
[OrchestrationLayerService]
Business Services
Business Logic
Persistence Services
Persistence Logic, CRUD functions
Hibernate
PO <-> SQL
Database
SQL

Figure 2: Server Hierarchy

The User Interface (UI) also has some noteworthy points which are discussed in this paper. As such, this paper covers the following points:

- Demo application – the downloadable application which serves as a basis for the points discussed here,
- UI wizard pattern,
- Model-View-Controller design pattern for the UI,
- Security throughout the application,
- Transactions, exceptions and dependency resolution,
- File Upload pattern for GWT,
- Composite Widgets and a UI factory,
- Captcha technology and GWT,
- Multiple GWT modules in a single application,
- Google Analytics integration,
- Deployment for development and testing,
- GWT maturity and its future.

Demo Application

In order to demonstrate the various themes discussed in this paper, a demo application was written, which is live at http://gwtdemo.maxant.co.uk. This application is a simple shop which sells artist's designs, which are printed out and shipped to the customers. Artists may also register and upload their designs and manage them using admin screen. Back office staff who pick and ship orders have screens to empower them to do their job. Administrators may also manipulate designs, artists and customer profiles directly, using admin screens. Various users have been created for the site, each with different roles. The login screen provides the details of these users so that you can play with all facets of the application. You can download the complete application from http://www.maxant.co.uk/whitepapers.jsp.

Figure 3 shows how the various actors/roles interact with the site. It also shows how the application is made up of two GWT modules, namely a "shop" and an "admin" site.

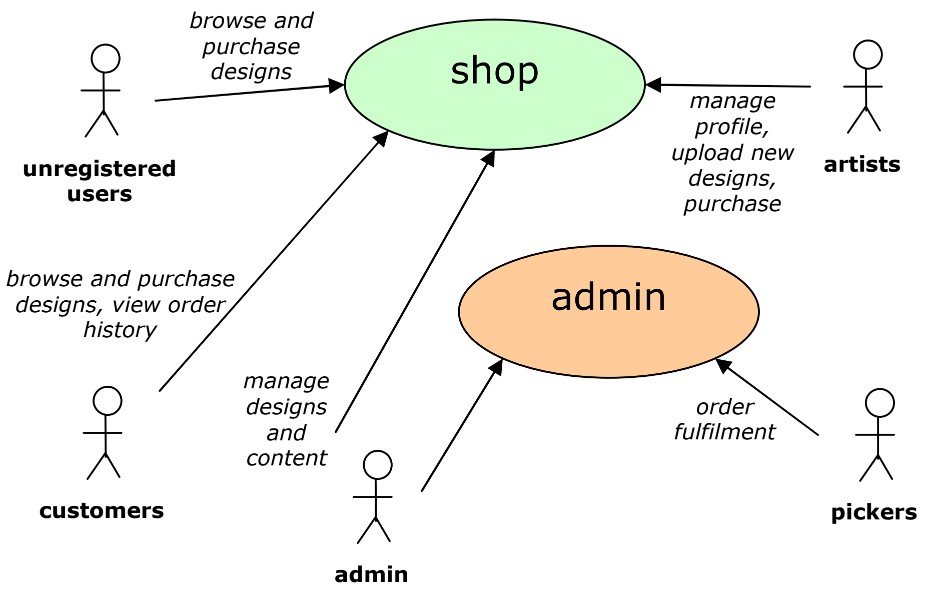

browse and purchase designs
shop
manage profile, upload new designs, purchase
artists
unregistered users
browse and purchase designs, view order history
admin
manage designs and content
order fulfilment
customers
pickers
admin

Figure 3: Actors Involvment

Screen flow is shown in Figure 4. Note that you can get to any screen using the menu, which is built using the users roles, so for example a customer cannot view sales orders which need to be picked, rather they can only view their own sales order history.

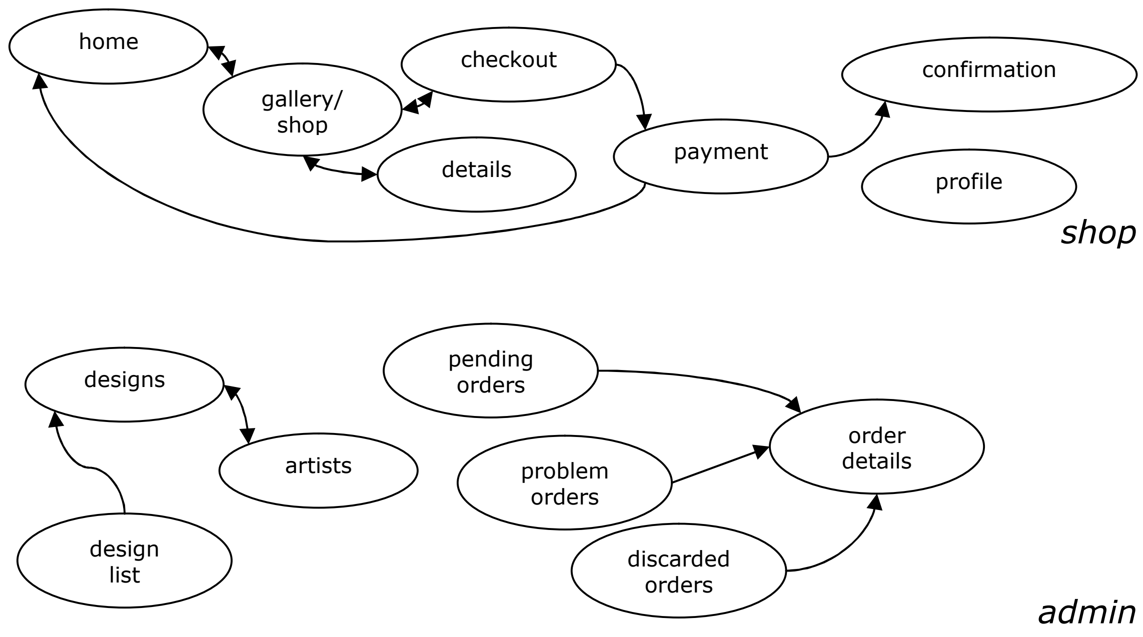


Figure 4: Screen Flow

User Interface Patterns

Most applications consist of a number of "pages". Sometimes these pages can be selected using a tabbed pane, other times these pages are selected using a menu or wizard. The demo application consists of 11 pages for the shop and 8 pages for admin. For the shop, the screen flow in Figure 4 shows clearly that a process of purchasing designs can be modelled using a wizard. As such, GWT does not provide such a class and so one was designed and implemented, as shown in Figure 5.

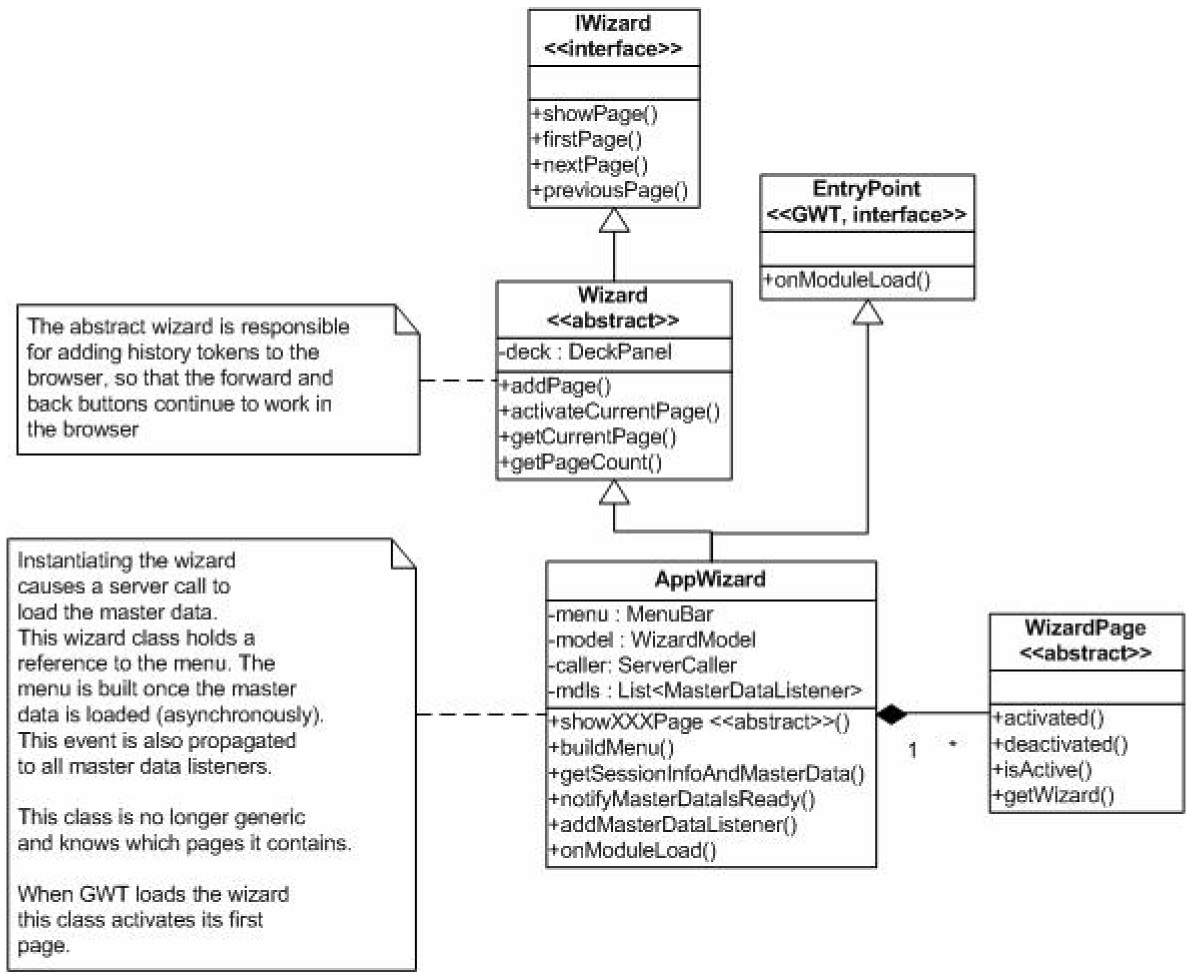
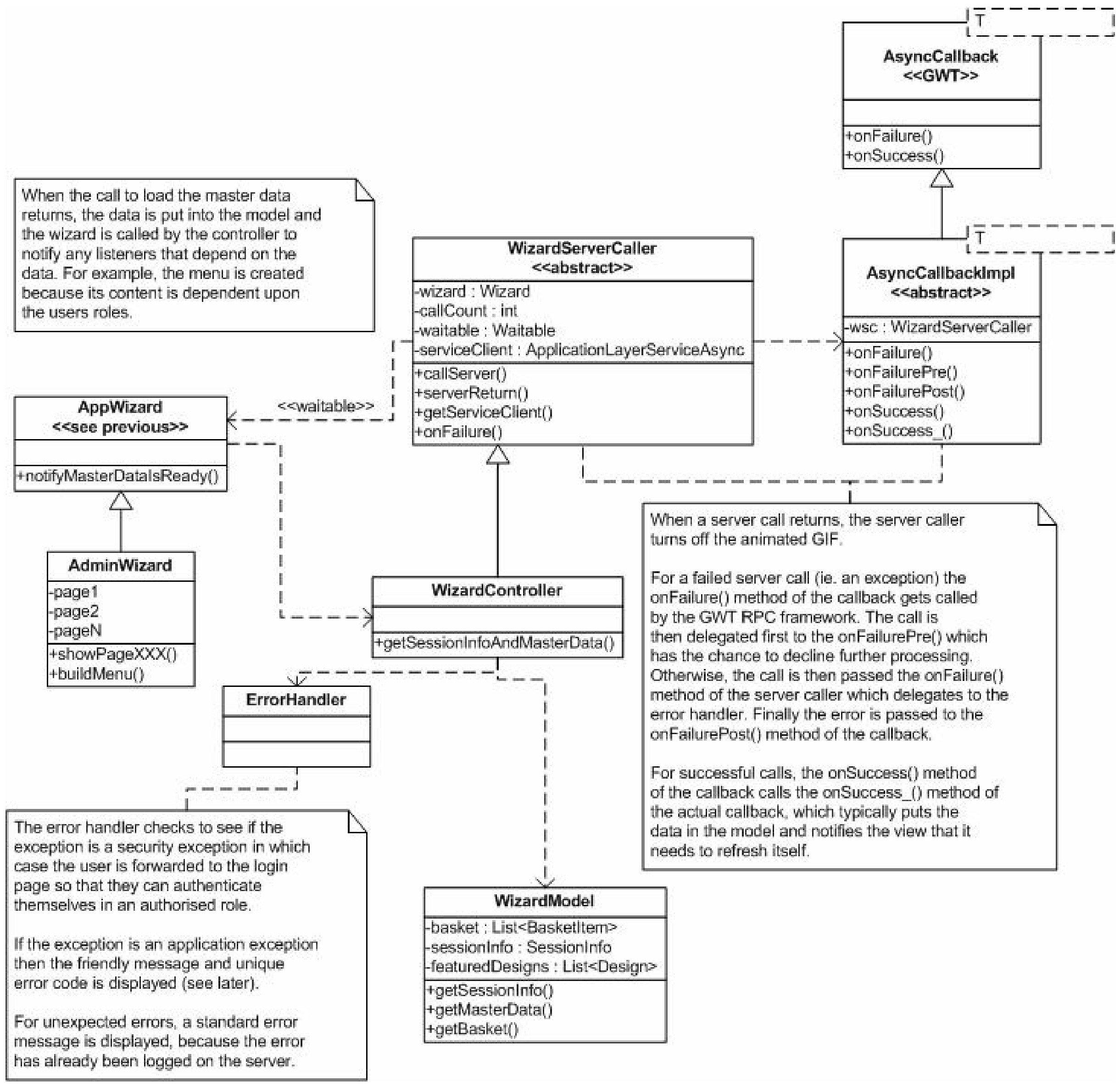

IWizard
<<interface>>
+showPage()
+firstPage()
+nextPage()
+previousPage()
EntryPoint
<<GWT, interface>>
+onModuleLoad()
Wizard
<<abstract>>
-deck : DeckPanel
+addPage()
+activateCurrentPage()
+getCurrentPage()
+getPageCount()
The abstract wizard is responsible for adding history tokens to the browser, so that the forward and back buttons continue to work in the browser
AppWizard
-menu : MenuBar
-model : WizardModel
-caller: ServerCaller
-mdls : List<MasterDataListener>
+showXXXPage <<abstract>>()
+buildMenu()
+getSessionInfoAndMasterData()
+notifyMasterDataIsReady()
+addMasterDataListener()
+onModuleLoad()
Instantiating the wizard causes a server call to load the master data. This wizard class holds a reference to the menu. The menu is built once the master data is loaded (asynchronously). This event is also propagated to all master data listeners.
This class is no longer generic and knows which pages it contains.
When GWT loads the wizard this class activates its first page.
WizardPage
<<abstract>>
+activated()
+deactivated()
+isActive()
+getWizard()
1
*

Figure 5: Wizard Classes

The main entry point for each module is a wizard which subclasses the AppWizard shown at the bottom of Figure 5. The wizard then has all the functionality it needs to display multiple pages. Each time a page is changed, a history token is added to the browser, so that the forward and backward buttons work, as provided for by standard GWT.

The very first RPC call is made when the module loads and is done to get the session information and master data. The session information includes user data such as their name, unique ID and a list of their roles. Even for unauthenticated users, it is also useful to know that they should not see any extra menus, for example updating their (non-existent) profile. Master data such as a list of all artists is also loaded and cached, client side, for performance reasons. The demo application does not contain that much functionality, but in the real world the customer might need to also choose which media the design should be printed on. In that case, media types would be ideal candidates for master data which is cached on the client.

When this data is loaded, the Wizard caches it in its model, as shown in Figure 6, and notifies all master data listeners, such as the specific Wizard implementation itself which uses the user’s roles, to build the menu dynamically.



When the call to load the master data returns, the data is put into the model and the wizard is called by the controller to notify any listeners that depend on the data. For example, the menu is created because its content is dependent upon the users roles.

When a server call returns, the server caller turns off the animated GIF.

For a failed server call (ie. an exception) the onFailure() method of the callback gets called by the GWT RPC framework. The call is then delegated first to the onFailurePre() which has the chance to decline further processing. Otherwise, the call is then passed the onFailure() method of the server caller which delegates to the error handler. Finally the error is passed to the onFailurePost() method of the callback.

For successful calls, the onSuccess() method of the callback calls the onSuccess_() method of the actual callback, which typically puts the data in the model and notifies the view that it needs to refresh itself.

The error handler checks to see if the exception is a security exception in which case the user is forwarded to the login page so that they can authenticate themselves in an authorised role.

If the exception is an application exception then the friendly message and unique error code is displayed (see later).

For unexpected errors, a standard error message is displayed, because the error has already been logged on the server.

Figure 6: More Wizard Classes

Figure 6 gives more details about how the server is called. Importantly, GWT RPC is an asynchronous technology because of the nature of browsers threading models: all Javascript runs in one thread and that thread is also in charge of rendering the UI. So if it is blocked waiting for a server response, the UI may freeze.

As such, the mechanism for calling the server in GWT is somewhat different in comparison to a traditional rich client written in Java Swing or Eclipse Rich Client Platform (RCP). Traditionally a server call is started on a separate thread from the UI thread so that updates to the UI are not blocked and when the server call returns an event is fired to the UI thread so that it can react and refresh itself at the next opportunity.

With GWT the mechanism is a little different because Javascript cannot start new threads. Instead, the browser does it in the background. In order for the browser to notify Javascript that a response has arrived, you must pass it an instance of a callback interface which it uses to call the single (UI) thread once the server returns. Figure 7 show the details.

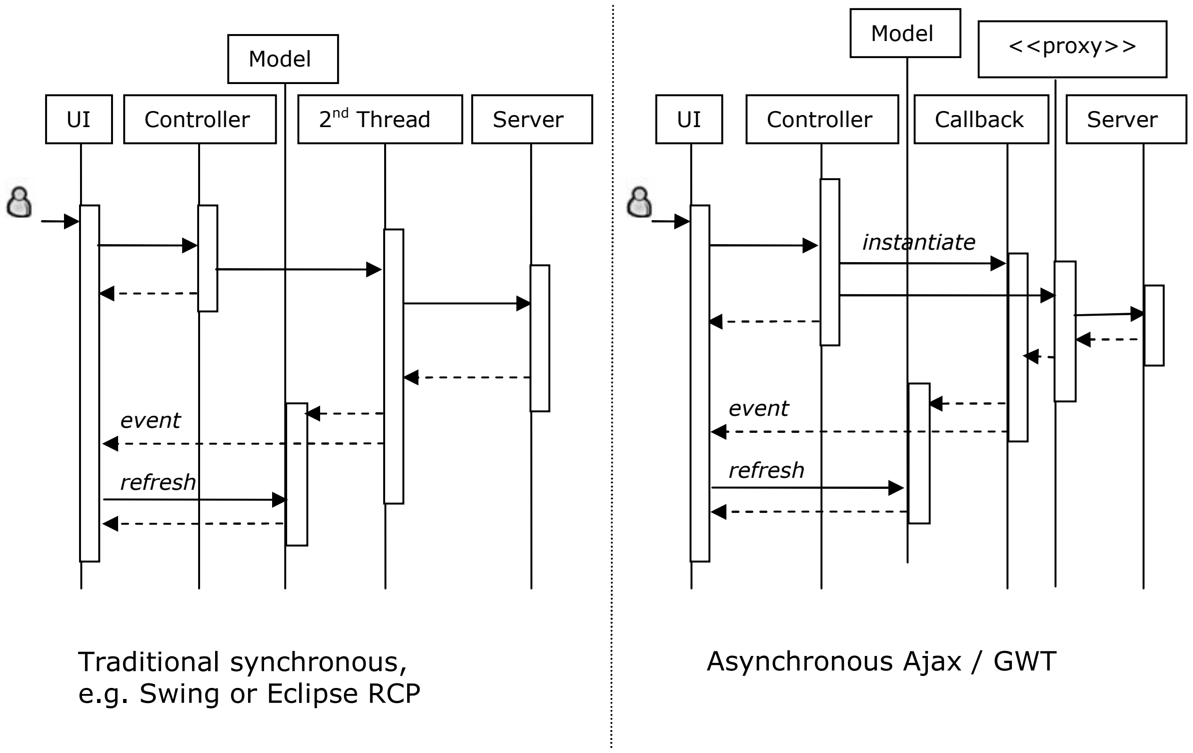


Figure 7: Traditional vs Asynchronous Server Mechanisms

This asynchronous callback mechanism is an area where the same coding pattern is used repeatedly. So in the demo application, a mini-framework was written to make provide an abstract implementation which handled several things:

- A “wait” icon, indicating that a server call was in process: ,
- Setting of the hour-glass cursor when server calls were in progress,
- Error handling.

This functionality is handled in the `WizardServerCaller` shown in Figure 6. A counter is maintained which is incremented with each outgoing request and decremented with every incoming response. This way the client knows exactly how many outstanding responses there are and can ensure that the cursor shows the hour-glass and if the page is `Waitable`, then for example its animated “wait” GIF can be set visible. This way the user knows that a call to the server is still outstanding, because in the traditional world of browsers, the user sees that a server call is still underway in that the status bar contains a progress bar. With AJAX this progress bar is not always used by the browser, because the call is asynchronous and occurs in the background. The error handling mentioned in the list above is described in more detail in Figure 6.

Figure 8 shows a standard Model-View-Controller (MVC) pattern being used in the demo application. There are a number of important concepts shown.

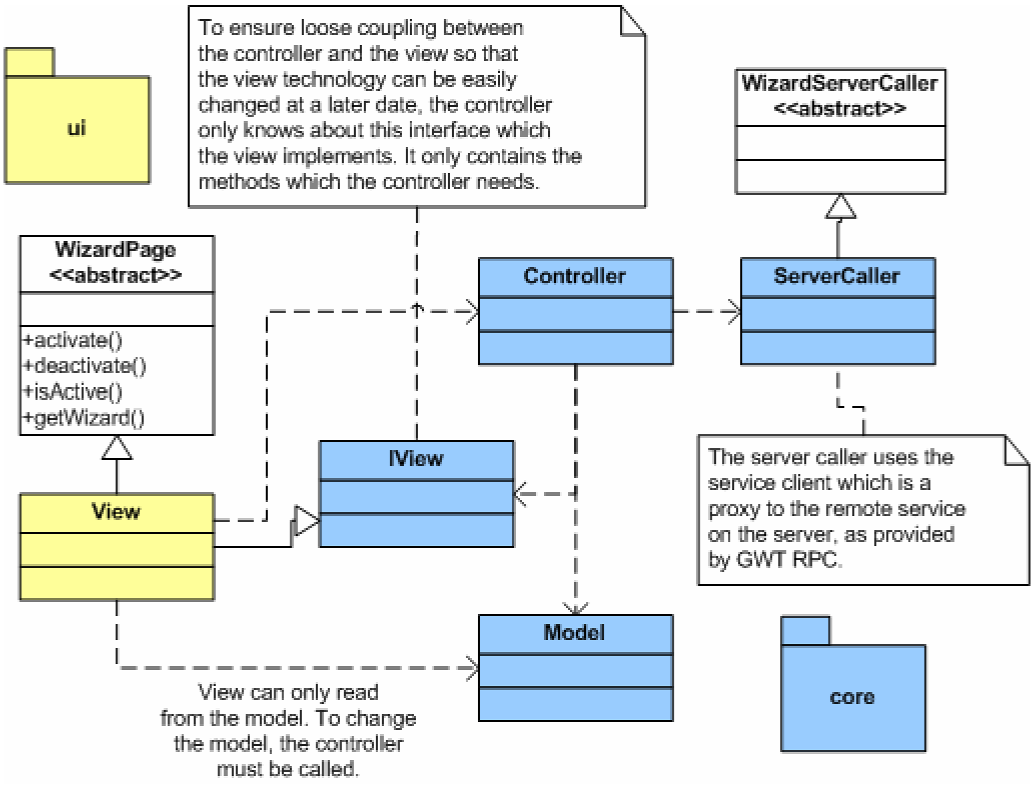

ui
To ensure loose coupling between the controller and the view so that the view technology can be easily changed at a later date, the controller only knows about this interface which the view implements. It only contains the methods which the controller needs.
WizardServerCaller <<abstract>>
WizardPage <<abstract>>
+activate()
+deactivate()
+isActive()
+getWizard()
Controller
ServerCaller
IView
View
The server caller uses the service client which is a proxy to the remote service on the server, as provided by GWT RPC.
Model
core
View can only read from the model. To change the model, the controller must be called.

Figure 8: MVC and Packages

Firstly, the model, the controller and the server caller (which is probably part of the controller in the academic world) are in a different package than the view. As such, the controller does not even need to know the view, but only its interface. This is nice because the view is now totally decoupled from the rest of it, meaning that changing the view is simple, and that is a major idea of MVC. Moreover, because the model is in a different package than the view, one can enforce the idea that the view only knows the model in a read only sense, yet the controller which is in the same package can write to the model. This is also important to MVC and allows business rules which are not implemented in the server to be centrally located in the controller, rather than randomly scattered throughout the view, which is typically large and made up of many classes. This pattern has been copied from Eclipse IDE implementation where bundles (components) are typically split into two deployable bundles, namely a "ui" bundle and a "core" bundle.

Security

For any application which provides multiple roles, security is used to allow authorised users to call certain functionality. For example, an artist may edit their own profile information, but a customer may not edit an artists profile information.

The users roles are loaded as part of the session information in the first RPC call which is made when the application loads. Based on this information the client shows links to various pages for example in the menu, or as icons on the screen, for example to edit a profile, as shown below in the red circle:

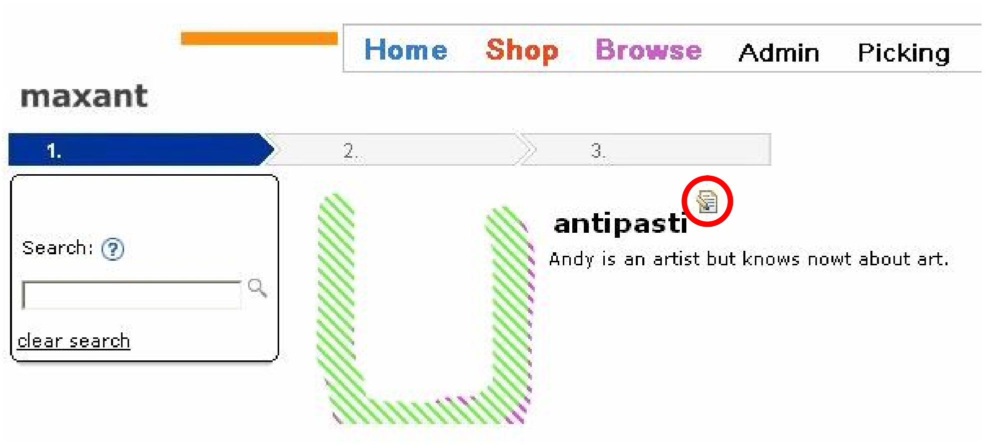


Figure 9: Example of links related to security roles

While the user interface knows about the users roles, it is important to take the security a step further to ensure hackers cannot break in where they are not welcome. In the example shown, a hacker could enter a URL into the browser to cause the screen to be shown where the artists profile can be modified. Alternatively, the hacker could just call the remote RPC service, since its interface uses standard JSON notation. But the call to the server which then fetches private information about the artist should check the users authorisation to see if they are allowed to view such information.

Figure 2 shows that it is the job of the Spring "Orchestration Service" to check security. While Spring provides security using ACEGI modules, a simpler implementation based on standard Java EE security was implemented for this demo. There is the maxant Spring Security extension available under http://www.maxant.co.uk/tools.jsp which is also discussed in detail on the maxant blog (http://blog.maxant.co.uk/pebble/2009/06/02/1243967400000.html). Basically, using a web filter, every server request is intercepted and the security principal is cached. Further along the call stack, a Spring AOP aspect intercepts the call just before it enters the service and security annotations on the service method can be read and used to check if the cached principal has the required role to call the service method. Once inside the orchestration service, security can be checked in more detail because services requiring it have the principal passed to them as a parameter.

An example of such a security annotation is shown below:

```
@RolesAllowed({Roles.BACKOFFICE, Roles.PICKING})
public List<Design> getDesigns(String filter) throws ApplicationException
```

Once this is set, there is no need to configure any security constraints in the `web.xml` for the application, because the maxant Spring Security extension automatically checks the incoming requests principal, and in the event of insufficient authorisation, it throws a security exception.

For GWT to handle this security exception correctly, it is important that the GWT RPC Service (in the demo application case see the `ApplicationLayerServiceImpl` class) is extended to handle the security violation in a manner which the client will receive notification properly so it can handle it by asking the user to log in with appropriate authorisation.

Any RPC service must extend the GWT `RemoteServiceServlet`. To handle security properly, the following method is overridden:

```
/** ***********************************************************
 * override GWT in case necessary...
 *
 * check for security issues, and handle them appropriately,
 * by returning a 401 error. the client can then
 * catch this and redirect to the login page.
 *
 * @param  e  the exception to handle
 */
@Override
protected void doUnexpectedFailure(Throwable e) {

    if ((e.getCause() != null) &&
            (e.getCause() instanceof SecurityException)) {

        try {
            getThreadLocalResponse().sendError(
                    HttpServletResponse.SC_UNAUTHORIZED);
        } catch (IOException ioe) {

            // shouldnt ever happen... log and handle
            //original exception normally.
            log.warn("failed to send 401 response " +
                    "to a SecurityException", ioe);
            super.doUnexpectedFailure(e);
        }
    } else {
        super.doUnexpectedFailure(e);
    }
}
```

By sending an error in the HTTP Servlet Response, the GWT RPC framework on the client side calls the `onFailure()` method of the asynchronous callback, where the status code can be examined. In the demo application this code is in the ErrorHandler class, which is delegated to by the WizardServerCaller and the callback, as described in Figure 6. The following code block shows the `ErrorHandler` doing this.

```java
public static void handleError(Throwable t){
    boolean handled = false;
    if(t instanceof StatusCodeException){
        StatusCodeException sce = (StatusCodeException)t;
        if(sce.getStatusCode() == 401){
            //force login. under normal circumstances, this
            //feature is not used to force a login for
            //changing the users role. if a user wants to
            //do something which their role does not allow,
            //the GUI doesnt even give them the option. so
            //this redirect helps hackers find the login
            //page, and more importantly, if the session
            //has timed out, it lets the user login again.
            handled = true;
            UIFactory.createErrorDialog(
                    "No Authorisation",
                    "Please log in with sufficient " +
                    "rights to perform this action.",
                    new Dialog.Callback() {
                        public void onClose() {
                            Window.Location.assign("/secure");
                        }
            }).center();
        }
    }else if(t instanceof ApplicationException){
        ApplicationException ae = (ApplicationException)t;
        UIFactory.createErrorDialog("Error",
                ae.getUserDetails() + "<br><br>" +
                ae.getCode() + "/" + ae.getTimestamp()).center();
        handled = true;
    }
    if(!handled){
        UIFactory.createErrorDialog("Error", "Error: " +
                t.getMessage()).center();
    }
}
```

An alternative to this security concept would be to provide security constraints on URL mappings in `web.xml` and providing multiple RPC services, each with their own security checks depending on which roles can call them. The following is the GWT annotation added to any RPC service:

```java
@RemoteServiceRelativePath("services")
public interface ApplicationLayerService extends RemoteService {
```

In this example it maps to a servlet running under the same URL pattern:

```xml
<servlet>
  <servlet-name>services</servlet-name>
  <servlet-class>uk.co.maxant.gwtdemo.server.ApplicationLayerServiceImpl</servlet-class>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>services</servlet-name>
  <url-pattern>/gwtdemoshop/services</url-pattern>
</servlet-mapping>
```

In the above mapping "gwtdemoshop" refers to the GWT module name in which this service is mapped. Since the RPC service is running as a servlet, one could add a security constraint to that servlet as follows, although this concept was not used in the demo:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>SecuredRPC</web-resource-name>
        <url-pattern>/gwtdemoshop/services</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>
```

The authors prefer the maxant Spring Security extension because it does not force service interfaces to be split up according to the roles which can call them. Security and deployment are two separate issues which should not affect one another so strongly.

BOM vs TO vs PO

Software architecture has recently been changing in the way that domain objects are directly passed to the client. The advantage of such an approach is that business objects do not have to be mapped to transfer objects. This is very practical for GWT applications. However, the following points need to be considered.

The domain objects and their fields that come over Hibernate from the database are sometimes wrapped in a dynamic proxy and are not standard Java classes. For Hibernate specific collections, such as `org.hibernate.collection.PersistenSet`, `org.hibernate.collection.PersistentList` and `org.hibernate.collection.PersistentBag`, but also for the `java.sql.Date` class no GWT emulation exists. As a consequence, these objects have to be unwrapped and converted in standard Java classes in order to make them understandable to the GWT compiler.

Such a filtering is done using reflection for all fields of a domain object, by applying the following code:

```
public static Object filterValue(final Object value) throws Exception {
    if (value == null) {
        return value;
    } else if (value instanceof Collection) {
        return filterCollection((Collection) value);
    } else if (value instanceof Date) {
        //casting java.sql.Date or any timestamp to java.util.Date
        //works since they are all subclasses. But the following does
        //look strange...
        return new Date(((Date) value).getTime());
    } else if (
        value.getClass().isPrimitive() ||
        value instanceof String ||
        value instanceof Number ||
        value instanceof Boolean ||
        value instanceof Enum) {
        return value;
    } else {
        //throw adequate exception
        throw new Exception();
    }
}

private static Collection filterCollection(Collection instance) {
    if (instance == null) {
        throw new NullPointerException();
    } else if (instance instanceof PersistentSet) {
        Set<Object> hashSet = new HashSet<Object>();
        PersistentSet persSet = (PersistentSet) instance;
        if (persSet.wasInitialized()) {
            hashSet.addAll(persSet);
        }
        return hashSet;
    } else if (instance instanceof PersistentList) {
        // do the same
        ...
    } else if (instance instanceof PersistentBag) {
        // do the same for PersistentBag
        ...
    } else if (instance.getClass().getName().contains(CGLIB)) {
        throw new UnsupportedClassVersionError("To implement");
    } else {
        // plain collection, do not filter
        return instance;
    }
}
```

Furthermore, an AOP aspect that invokes filter(Object value) can be put around each service call before the domain objects are passed to the client.

This approach is a possible alternative to using TOs. However, in the demo application the option of mapping from BOs to TOs has been followed.

Other Service Considerations

In the demo, all services are configured using Spring Autowiring. For this to work you simply annotate each Service with a name and optional transactional declarations:

```
@Service("designService")
@Transactional(
        propagation=Propagation.REQUIRED,
        isolation=Isolation.DEFAULT,
        rollbackFor=ApplicationException.class)
public class DesignService {
```

The Spring application context (configuration) can then be configured to load all autowired services:

```
<context:annotation-config />
<context:component-scan
    base-package="uk.co.maxant.gwtdemo.server" />
```

This step also automatically resolves all dependencies between these services. The Orchestration Service is dependent upon the Design Service, and this is configured through another Spring annotation:

```
public class OrchestrationService extends Mapper {

    @Autowired
    private DesignService designService;
```

This dependency is then automatically injected when the application context is loaded.

This mechanism is great for Spring Services, however the RPC Service which is the entry point to a server call is running as a servlet outside of the Spring Context (see Figure 2). So a `ServiceLocator` class was written which caches a reference to the Spring Application Context and fetches beans (services) out of it. This was also extended to include a `DependencyResolver` which uses reflection to dynamically inject the target class instance with beans from Spring. Figure 10 as well as the code below shows how this works.

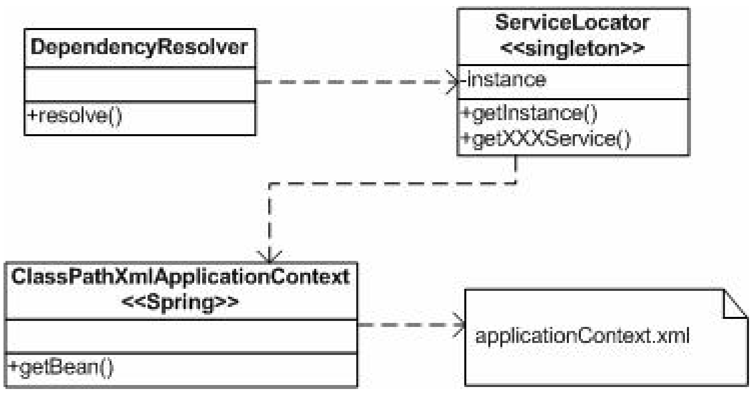


Figure 10: Dependency Resolver and Service Locator

```java
/**
 * searches all fields and looks in spring to see if a bean with the
 * same name is registered. if so, it sets them, so long as the
 * attribute on that field is @Autowired
 *
 * @param objToSetInstancesOn object on which to set the spring beans
 * @throws IllegalAccessException if setting is not possible
 * @throws IllegalArgumentException if setting is not possible
 */
public static void resolve(Object objToSetInstancesOn)
                throws IllegalArgumentException, IllegalAccessException{

    Field[] fields = objToSetInstancesOn.getClass().getDeclaredFields();

    for(Field f : fields){
        Annotation[] annotations = f.getAnnotations();
        boolean relevant = false;
        for(Annotation a : annotations){
            if(Autowired.class.isAssignableFrom(a.getClass())){
                relevant = true;
                break;
            }
        }
        if(relevant){
            Object o = ServiceLocator.getInstance().getBean(f.getName());
            if(o != null){
                f.setAccessible(true);
                f.set(objToSetInstancesOn, o);
            }
        }
    }
}
```

The code shown previously for the Design Service and repeated below, has an interesting attribute on the transaction annotation, namely the `rollbackFor` attribute:

```java
@Service("designService")
@Transactional(
        propagation=Propagation.REQUIRED,
        isolation=Isolation.DEFAULT,
        rollbackFor=ApplicationException.class)
public class DesignService {
```

This attribute signals the framework that in the event of such an exception that it should also perform a rollback. Normally, Spring only performs a rollback on Runtime Exceptions, or if the "setRollbackOnly" method is called on the transaction context. By adding this attribute to the annotation, we save ourselves having to tell the transaction to rollback explicitly. In the demo application this makes sense too. An `ApplicationException` is thrown in the event that any business logic or technical error makes it impossible to successfully complete the server call. We basically want to rollback in all such cases, as though the call had never been made.

In special cases where a database entry does indeed need to be kept, this would be handled by an inner transaction, which can be started by adding the `@Transactional` annotation to the method, and specifying a `Propogation` of `NEW`.

The `ApplicationException` is something used widely throughout applications written by maxant as a means of communicating a unique error code and friendly message to the user, while simultaneously causing a rollback of the call to tidy up. The nice thing about it is that wherever it is thrown, the unique code provides second level support with a good starting point for analysing the problem, since it points to exactly the line of code where it occurred. This class also contains a timestamp for when it is instantiated, to further assist in finding the problem in the logs. This exception also contains technical details which can be logged for technical support, as well as a user friendly exception, which can even be enriched further along the call stack. Basically the technical message contains details of exactly what went wrong, whereby the user details contain contextual information describing which user function failed. Together these two contexts allow technical support to see what the user was trying to do, where the error occurred, when it occurred and what it exactly was. Recreating the error should then be fairly easy, and its resolution should be quick.

The following code block shows the `handleException()` method of the Orchestration Service, which is used to handle all exceptions which get as far as that service. Basically it logs the exception as it exits the server (meaning you do not need to log it further down the call hierarchy, avoiding it getting logged many times as is typical of enterprise applications). This method also ensures that only `ApplicationException`s are thrown.

```java
/**
 * logs exception as it passes out of the spring context
 * (ie this service) and transforms it into an application
 * exception to ensure a rollback. regardless of what happens
 * we want a rollback. spring only automatically performs a
 * rollback if the exception is runtime. but regardless, this
 * application requires that any exception leads to a rollback.
 */
private void handleException(String code, String msg,
                    Exception e) throws ApplicationException {

    log.error(msg, e);

    if(e instanceof ApplicationException){
        throw (ApplicationException)e;
    }

    throw new ApplicationException(
            code,
            msg + "\r\n" + e.getMessage(),
            "An error occurred. Please try again.");
}
```

The code block below shows a typical method in the orchestration service which catches all exceptions and ensures they are handled with a unique error code (`ORCH002` in this example).

```
/**
 * @see  ApplicationLayerService#addToBasket(BasketItem)
 */
public List<BasketItem> addToBasket(BasketItem basketItem,
        HttpServletRequest request) throws ApplicationException {

    try{
        List<BasketItem> basket = getBasket(request);
        basket.add(basketItem);

        return basket;
    }catch(Exception e){
        handleException(
                "ORCH002",
                "failed to get session info",
                e);
    }
    return null; //never happens since #handleException() ALWAYS throws one
}
```

As all exceptions which come back to the client are `ApplicationExceptions`, it can easily display them in a generic fashion, showing the error code, timestamp and user friendly details. An error handler can handle the exceptions serverside before they exit and log or email them to support, allowing a proactive handling of problems.

Another important aspect of the Orchestration Service is that it maps Transfer Objects (TOs) to Persistence Objects (POs). In a larger application it may well be worth also having a Business Object Model (BOM) between the TOs and POs which the business services use. The mappings in the demo application were partially done using reflection as all POs and TOs are essentially standard Java Beans with standard accessor methods (`setX()` / `getX()`) (see section on BOM vs TO). The mapping could not be completely done automatically, because for example GWT does not know the `java.math.BigDecimal`, and `double` had to be used instead (although a GWT emulation for `BigDecimal` can be found at http://code.google.com/p/gwt-math/). However, the Orchestration Serivce extends the `Mapper` class so that it can quickly map as required. Because designs and artists do not necessarily have an image associated with them (at least not until they are approved), the Orchestration Service overrides the relevant mapping and sets the `imageExists` field of the TO, so that the client doesn't try to load a non-existent image and can show a generic image instead.

File Upload in a Form

One area of GWT which leads to confusion is file uploads. File uploads are not handled over RPC but rather in the traditional sense whereby a servlet is called and the request contains Mime entries for the uploaded data. If however the page which the user is filling out need to contain other information which should also be uploaded, there are two ways to handle it.

As an example, take the dialog which an artist uses to upload a new design, shown in Figure 11.

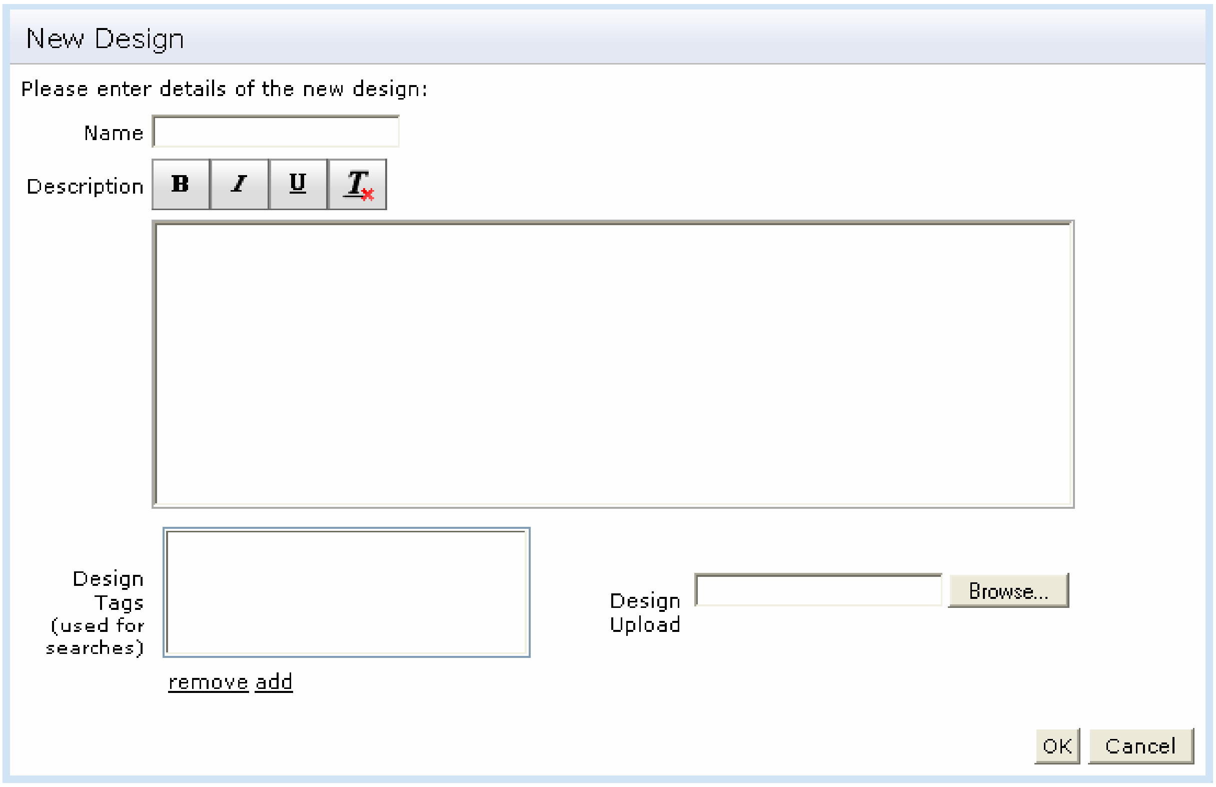

New Design
Please enter details of the new design:
Name
Description
B
I
U
Design Tags (used for searches)
remove add
Design Upload
Browse...
OK
Cancel

Figure 11: Submitting a New Design

This screen allows the user to enter details about the design, as well as an image file.

GWT File uploads occur by adding a `FormPanel`, which results in calling a servlet, but *not* an RPC service. On the other hand, if the file upload were not part of this screen, hitting the OK button would result in a simple RPC call using a transfer object to hold the input data.

So that the user does not need to submit the data first, and then click again to submit their image, there are two solutions. Firstly when the user hits the OK button, the data could be collected from the screen and set into the GWT `FormPanel` as `Hidden` objects, which are then submitted with the form, in order to load the file and create the database record at the same time. The problem with this is that the servlet needs to then locate the service in order to create a design record in the database and then save the uploaded file.

The alternative is that a normal RPC call is made with all the data except the file, and when that returns, a second call to the file upload servlet is made. This is the approach used in the demo, but it has several noteworthy points. The `FormPanel` on which the `submit()` method is called may not be disposed before it is called (ie. before the RPC call which creates the database record returns), meaning that the dialog which contains it may not be closed until after it is submitted. This process is also more complex, because you need to maintain a reference to the `FormPanel` in a place which the asynchronous callback from the RPC call can access it to call `submit()` on it.

Using the `ServiceLocator` described in Figure 10, it would have been simpler to take the information that the user input and set it on `Hidden` fields in the `FormPanel` and simply submit that. Serverside, the servlet could have used the `ServiceLocator` to find the required service and call it.

For serverside file upload, the Apache Commons FileUpload library was used.

Composite Widgets and a UI Factory

User Interface widgets in the demo application are instantiated using the `UIFactory` class, whose primary role is to ensure that all widgets have the correct CSS styles applied to them. This way, it is possible to easily change the style applied to say all buttons in the application, because their style name is defined centrally.

GWT provides a number of very useful standard widgets for building the user interface. However a number of other useful widgets are missing. Many common widgets not found in GWT are provided by third-party libraries, such as Smart GWT, GWT-Ext or Ext GWT. Smart GWT is based on the SmartClient Javascript library, where the other two use the Ext-JS Javascript library. Unfortunately, the licence for Ext-JS has been recently restricted from LGPL to GPL with the consequence that GWT-Ext is no longer under active development. Thus, it is recommended to use Smart GWT. It is not only available under LGPL but is also currently being rapidly evolved and is well supported. Of course it is possible to forego any GWT extension libraries since GWT allows a programmer to quite simply create their own new widgets by plugging standard ones together by extending the `Composite` class.

One such composite widget has already been shown in Figure 11, namely for tag input. Tags are simple key words which are associated with a design, so that when the user searches for a design, they are more likely to find what they are looking for. To capture a list of such tags, the `TagListBox` composite widget was created, as shown in Figure 12.

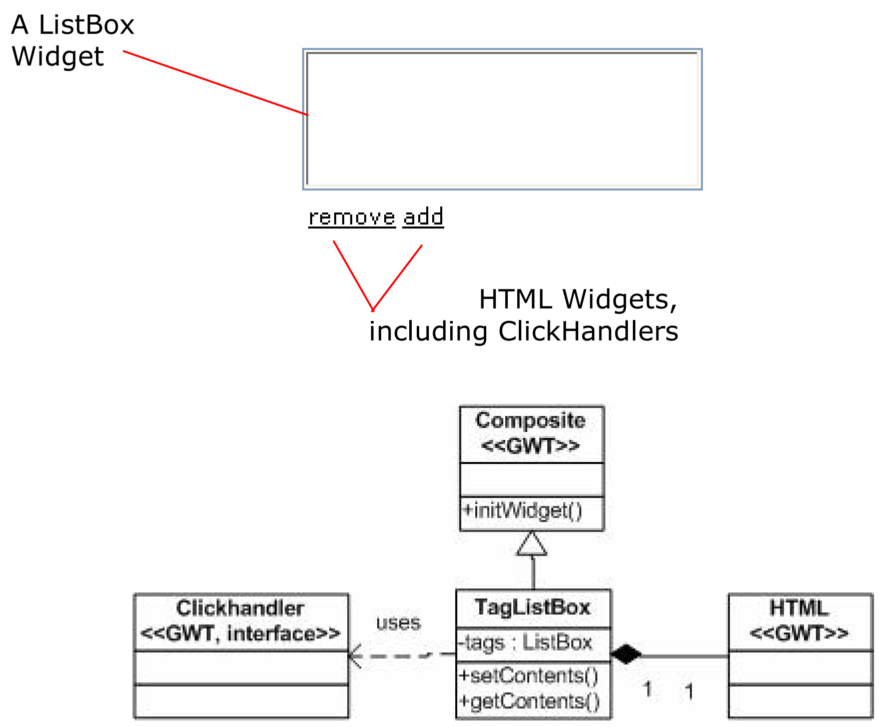
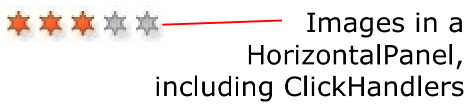


Figure 12: TagListBox

The `TagListBox` is based on a `ListBox` with two clickable `HTML` widgets for adding and removing items from the list box. To add an item, an input dialog (see below) is opened to prompt the user for the input. The items can be read from the widget by calling the `getContents()` method. Alternatively, to set the contents, the corresponding setter method can be called.

A “rater” widget for rating designs was also built:



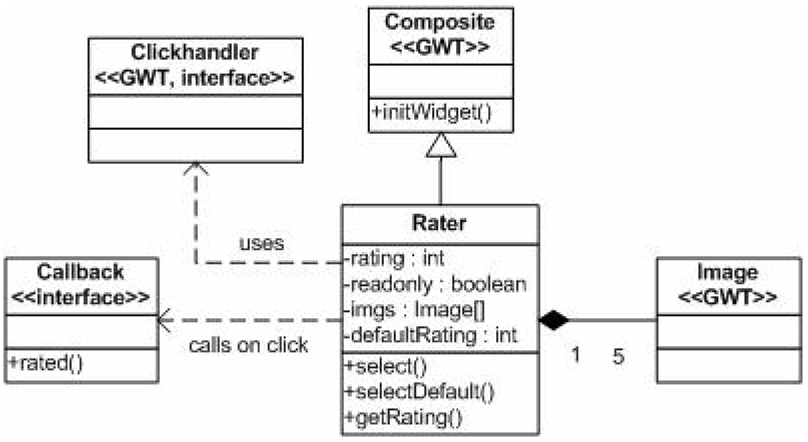
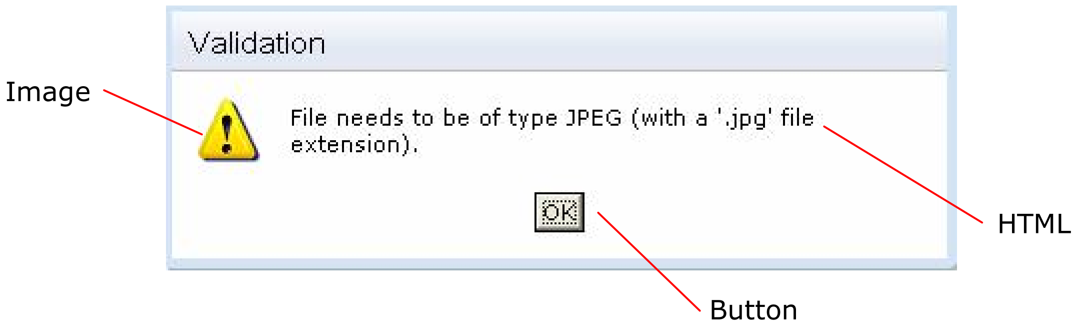


Figure 13: Rater

The Rater widget is read only until a user logs in. Once logged in, it will highlight the image over which the mouse hovers. If you click on a star, then that rating is set in the widgets internal model, and a callback is notified, so that the client can react to the event, for example by calling a remote service to save the rating to the database.

Standard dialogs for showing errors, warnings or information, as well as prompting questions or for input are not provided by standard GWT. A small dialog framework was built for the demo application:



File needs to be of type JPEG (with a '.jpg' file extension).

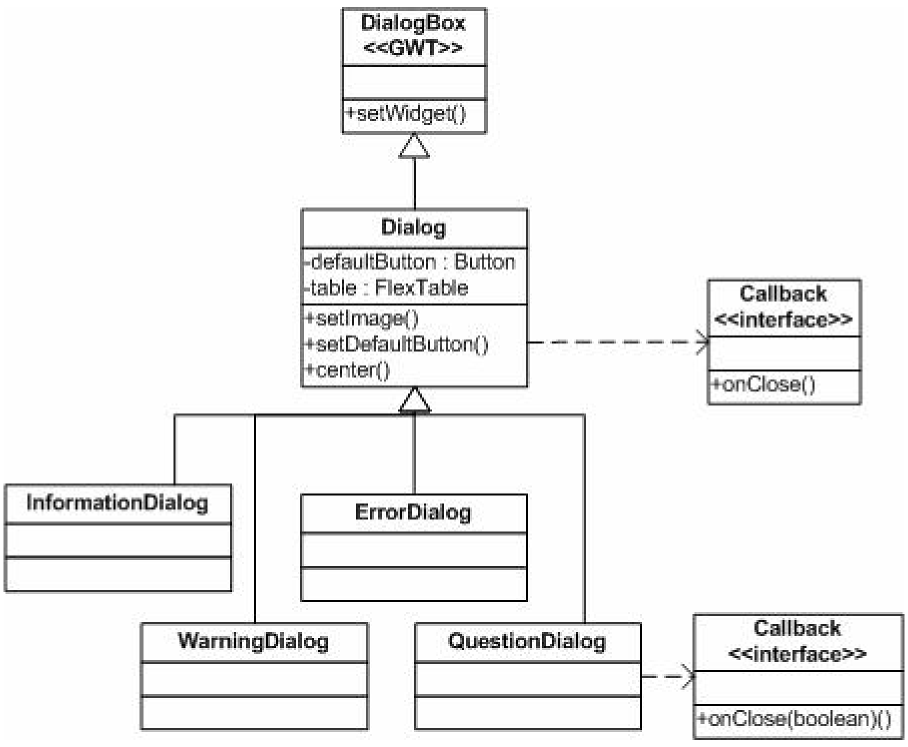
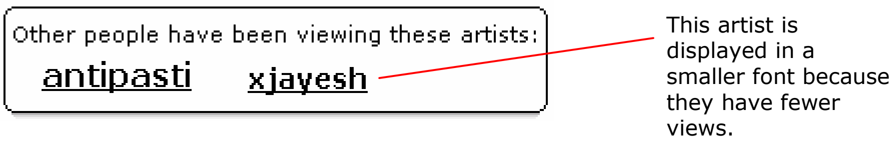


Figure 14: Dialogs

Dialogs all extend the `Dialog` class that was written as part of the demo application. The GWT `DialogBox` can be modal, but it does not block code execution from the moment it is visible until it is closed. For this reason call-backs were added so that code execution could be continued once the user interacted. Each subclass simply supplies the relevant image from an `ImageBundle`. All other implementation is in the `Dialog` super class. Since the `QuestionDialog` has two buttons ("yes" and "no"), its implementation is a little larger as it replaces the single "OK" button of the super class. Additionally, the QuestionDialog needs to pass its state to the callback so the client knows which button was pushed, so it has a different callback interface.

In all cases the `center()` method of the `DialogBox`, which opens the dialog in the center of the screen, is overridden so that it first sets focus on the default button.

Although not a widget as such, the "tag cloud" (http://en.wikipedia.org/wiki/Tag_cloud) which shows most viewed artists on the front page, is worth noting.

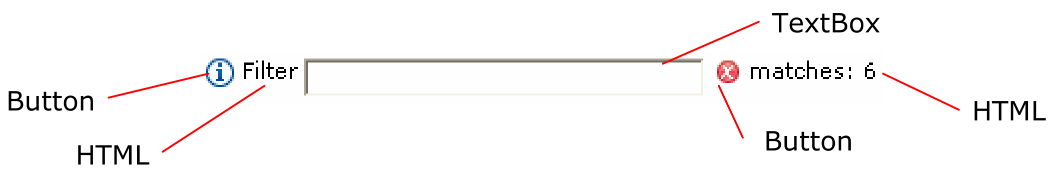

Other people have been viewing these artists:
antipasti
xjayesh
This artist is displayed in a smaller font because they have fewer views.

The tag cloud is created by showing links to the shop (which show the artist and all their designs) in a `FlexTable`. The links are created by retrieving a list of artists views, ie. which artists have been viewed the most. If for example the artists were listed in order of most views as follows, then their position in the table indicates the font size and weight used to create the link:

Artist	Number of Views	Font
Ed Nortson	455	Size 16, Weight 100
John McLean	450	Size 14, Weight 85
Fred McIntyre	360	Size 12, Weight 70
Jason Stove	325	Size 10, Weight 55
Jennifer Stoats	227	Size 8, Weight 40

The links are arranged either randomly, or in a spiral with the highest weighted artist shown in the center of the table.

The final composite widget described here is the filter box. A filter box can filter the contents of a table (locally in the client) or pass the filter to a server call so that the content returned from the server is filtered.



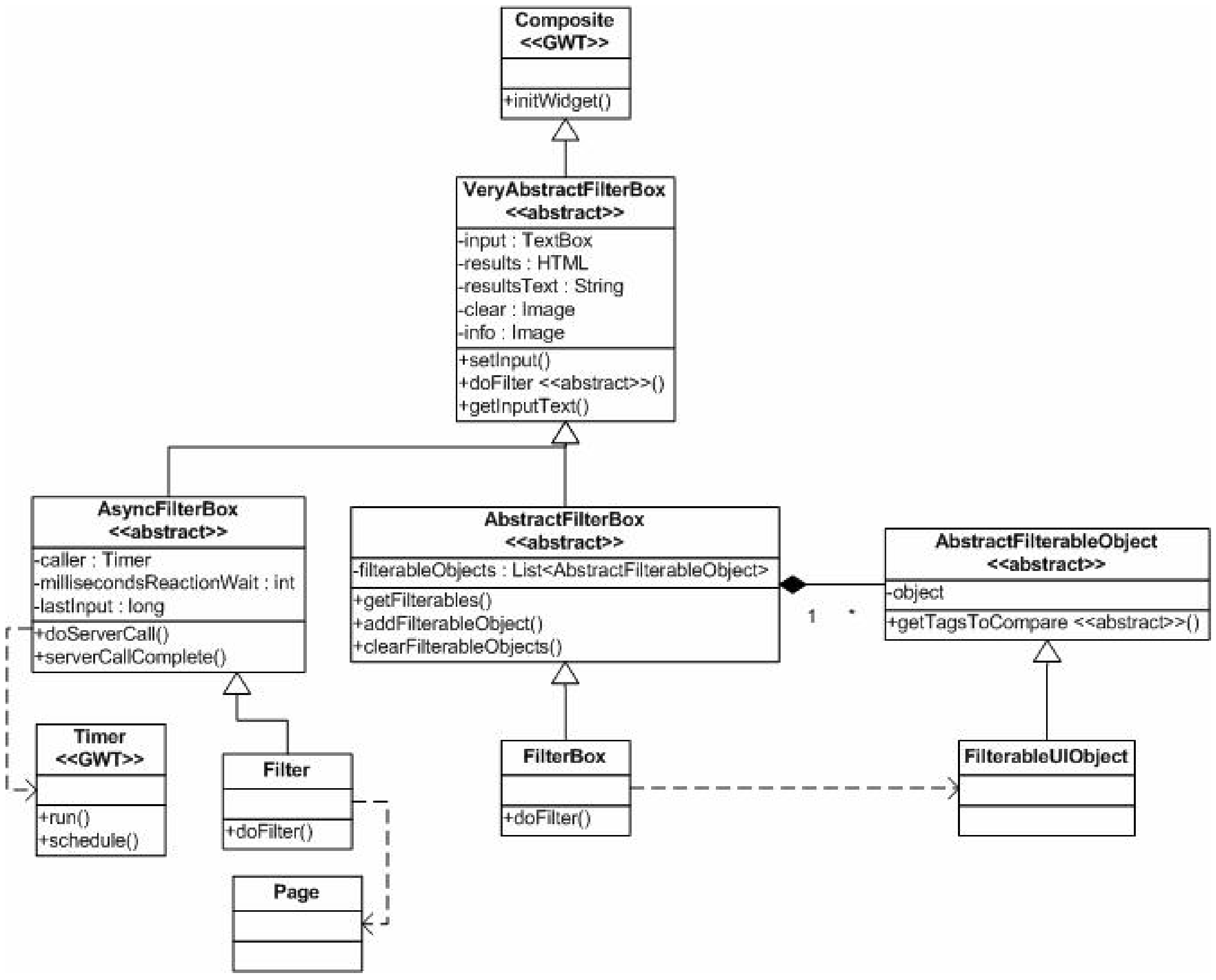


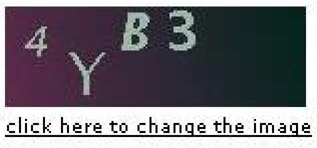
Figure 15: FilterBox

The `FilterBox` composite widget is by far the most complex of those created for the demo application. In fact there are two types. The first filters data in say a table. As text is entered into its input box, it calls the `doFilter()` method. This in turn calls the `getFilterables()` method which returns a list of `FilterableUIObject`s. These objects contain a reference to a `UIObject` and a method for getting relevant strings – meta-data to be compared to the filter input, which can be a regular expression. Any matching strings mean that the `UIObject` is set visible. Any non-matching strings, means that the related `UIObject` is set *not* visible. The DHTML engine then takes care of hiding the required widgets.

The second type of filter is an asynchronous one which sends the filter text to the server. It does this automatically, but only after the input text has changed, and remains unchanged for 500 milliseconds. It waits for this small period because otherwise the server would be flooded with every single input change (ie. every typed letter), most of which does not interest the user – they want to filter based on an entire word or phrase. It also reacts to the user hitting return, in which case it sends the request to the server immediately. The server call is typically done as usual, over the controller, from the `Filter` which is contained in the `Page`. Importantly, there is a method here called `serverCallComplete()` which the page calls during its refresh at the end, passing it the number of results so that the filter can display it in its right-hand `HTML` widget.

Jcaptcha and GWT

Jcaptcha is a Java implementation of captcha technology (http://jcaptcha.sourceforge.net/). This technology helps to guarantee that the inputs are coming from a human source as opposed to a robot. An example is shown below, from the "register" page of the demo application:



Upon registering, the client calls the `jcaptcha` servlet which returns an image and sets its text in the server-side session. When the user hits the register button, the data is sent to the Application Layer Service (the GWT RPC remote service) which checks the provided text against that stored in the session.

```
boolean captchaPassed = true; //default, incase of exception
try {
    // validate captcha

    String captchaId = getThreadLocalRequest().getSession().getId();

    //since the generated images ONLY contain big letters and
    //numbers @see CaptchaServiceSingleton.MyImageCaptchaEngine
    captcha = captcha.toUpperCase();

    captchaPassed = CaptchaServiceSingleton.getInstance().
                              validateResponseForID(captchaId, captcha);
} catch (CaptchaServiceException e) {
    //ok, our problem, not customers...
    // should not happen, may be thrown if the id is not valid
    log.warn("failed to validate captcha", e);
}
```

The `CaptchaServiceSingleton` is simply a class which generates captcha images. Jcaptcha only allows each image to be checked once. That means that if the captcha text is entered correctly, but the user gets a validation error because they have for example entered an alias (screen name) which already exists in the database, the captcha image will need to be regenerated and they will need to re-enter the text for the new image. This causes somewhat unfortunate experience for the user, so as much validation needs to be carried out client side as possible. Alternatively, the captcha check could be done as the very last part of the process, although it would need to result in a rollback. Since the code above is outside of the Spring services (it's the RPC service), its too late to do a callback, so in the demo application, the captcha check is the very first thing done in this process.

Multiple GWT Modules in a Single Application

GWT allows multiple modules to be created. Reasons to do so are to:

- create a super module consisting of many smaller logical parts which may eventually also be re-used,
- create a module containing only those parts which the application requires (see below),
- create a page containing multiple small widgets, each being its own module.

Towards the start of this paper, it was discussed that the application was split into two modules, namely an admin module and a shop module. The motivation for doing this was to reduce the size of the shop application, making the startup experience for users quicker. Together, with the modules as one module, the generated JavaScript and HTML resulted in a download of over 500 kilobytes. Once split, the shop modules generated code was only just over 300 kilobytes.

To split the application into two modules the `AppWizard` was split into the `ShopWizard` and an `AdminWizard`. Each contains different pages, and if a page is to be activated which it does not contain, it makes a call to the other module by loading the other modules JSP. The JSP contains a reference to the module it should load:

```
<script type="text/javascript"
        language="javascript"
        src="/gwtdemoshop/gwtdemoshop.nocache.js"></script>
```

The module itself is defined in the `GWTDemoShop.gwt.xml` file, where it is renamed to simply "gwtdemoshop". The entry point is also defined there, as the Java class containing the `onModuleLoad()` method.

```
<module rename-to='gwtdemoshop'>
.
.
.
 <!-- Specify the app entry point class.                         -->
 <entry-point class='uk.co.maxant.gwtdemo.client.widgets.ui.ShopWizard'/>
```

One important aspect of splitting a module into smaller parts is that if two modules call the same RPC service, that service's servlet mapping needs to be defined twice in the `web.xml` deployment descriptor:

```
<servlet>
  <servlet-name>services</servlet-name>
  <servlet-class>uk.co.maxant.gwtdemo.server.ApplicationLayerServiceImpl</servlet-class>
</servlet>
```

```
  <!--
  the services servlet needs to be mapped twice, since the same interface is used by
  both modules. one could use declarative security on URL patterns to ensure only
  authorised user can call specific services, however in this demo service security was
  handled using the maxant Spring Security extension. see the documentation for more
  details.
   -->
<servlet-mapping>
  <servlet-name>services</servlet-name>
  <url-pattern>/gwtdemoshop/services</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>services</servlet-name>
  <url-pattern>/gwtdemoadmin/services</url-pattern>
</servlet-mapping>
```

Note that the composite widgets which have been discussed above could also be turned into their own mini-modules and be published as libraries.

As suggested at the start of this section, there is also a motivation for using a multiple module architecture, to split the project into several logical subprojects, e.g. one containing the model, one for the business logic, one for the services and one for the client parts. This is quite a usual approach in a Java EE application because subprojects are decoupled, logical and easier to maintain. Such an architecture is very rarely seen in GWT demo applications, tutorials or examples in the internet. What can be done here, is to make each subproject which provides GWT specific parts, such as the RPC service interfaces or eventually the domain objects, a separate GWT module (usually without an entry point). The main module containing the entry point then references the sub-modules. Although this works pretty well, one workaround has to be considered to be able to debug the code of all subprojects: the sources output directories of the subprojects have to be set to the client's war projects output directory.

Google Analytics Integration

To include Google Analytics (http://www.google.com/analytics) so that page flow and usage reports can be very easily generated, a couple of things need to be done. Firstly you need to generate an analytics key using the Google site. Once you have this key, it can be added to some standard JavaScript which you can add to the index JSP page.

```
<%--
google analytics - see JSUtil and Wizard
--%>
<script type="text/javascript">
var gaJsHost = (("https:" == document.location.protocol) ? "https://ssl." : "http://www.");
document.write(unescape("%3Cscript src='" + gaJsHost + "google-analytics.com/ga.js' type='te
</script>
<script type="text/javascript">
var pageTracker = _gat._getTracker("<%=JSUtil.GOOGLE_ANALYTICS_UID%>");
pageTracker._initData();
pageTracker._trackPageview();
</script>
```

In the demo application this JavaScript is stored in `header.jsp` which is a common file to all JSP pages.

Notice that it defines a variable called `pageTracker` in the DHTML document. GWT can access this using the following code, contained in the `JSUtil` class:

```
/**
 * increments GoogleAnalytics for the given page
 *
 * @param pageName
 */
public static native void trackPageView(String pageName)
/*-{
try{
    $wnd.pageTracker._trackPageview(pageName);
}catch(err){
    //oh well - faile to track page :-( but no reason to completely fail!
}
}-*/;
```

When the JSP page is loaded, it adds a hit for that page to Google analytics. When the `trackPageView(String)` method above is called, it adds a hit for the given page name. This means that although all pages of the Wizard are contained within the index.`jsp` page, you can still tell Google Analytics that a specific page has been viewed. The method shown above basically calls native JavaScript which takes the `pageTracker` variable which is set using the JavaScript shown above, and calls it with the `pageName` parameter.

This method gets called by the wizard, every time a new page is activated, at the same time as the history token is added to the browser (see previously).

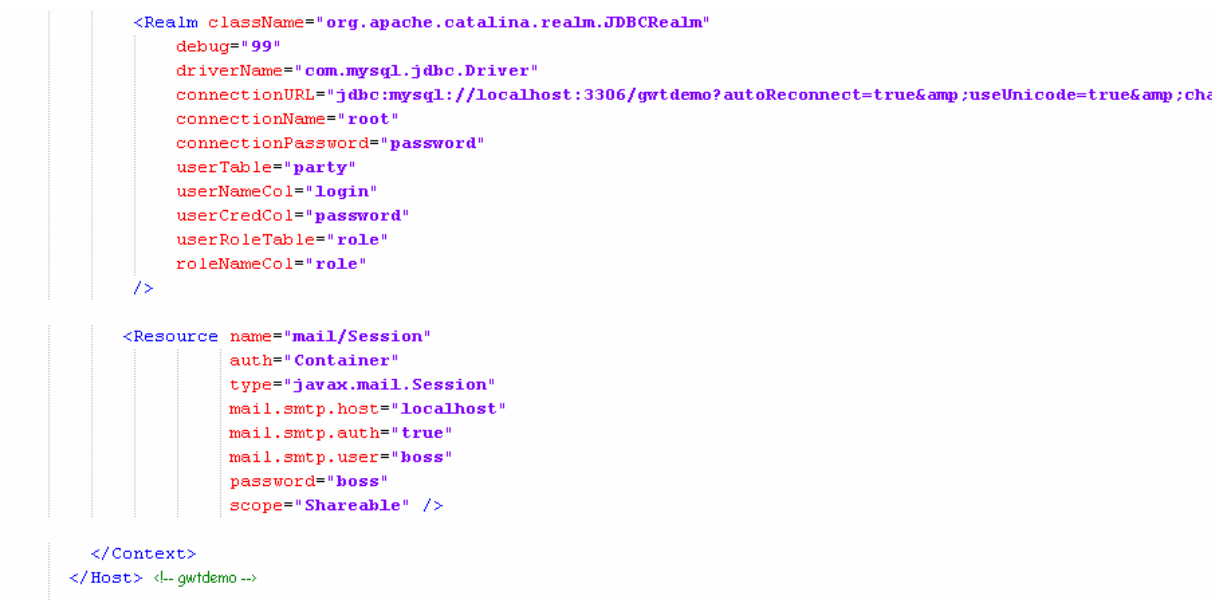
Deployment for Development and Testing

An important aspect to developing for the Enterprise is being able to test against a server running against a database. To do this, the Apache Tomcat server was used. The Eclipse project contains a "war" directory which is the web applications content directory. As such, the Tomcat server configuration is as follows:

```
<Host name="gwtdemomaxantcouk" appBase="r:/maxant/maxantcouk/gwtdemo/ROOT"
 unpackWARs="true" autoDeploy="true"
 xmlValidation="false" xmlNamespaceAware="false">

  <Context docBase="war"
      path=""
      reloadable="true">

      <Resource name="jdbc/db"
          auth="Container"
          type="javax.sql.DataSource"
          maxActive="100"
          maxIdle="30"
          maxWait="10000"
          removeAbandoned="true"
          removeAbandonedTimeout="60"
          logAbandoned="true"
          username="root"
          password="password"
          driverClassName="com.mysql.jdbc.Driver"
          url="jdbc:mysql://localhost:3306/gwtdemo?autoReconnect=true&amp;useUnicode=true&amp;characterEncod
      />
```



```
        <Realm className="org.apache.catalina.realm.JDBCRealm"
            debug="99"
            driverName="com.mysql.jdbc.Driver"
            connectionURL="jdbc:mysql://localhost:3306/gwtdemo?autoReconnect=true&amp;useUnicode=true&amp;cha
            connectionName="root"
            connectionPassword="password"
            userTable="party"
            userNameCol="login"
            userCredCol="password"
            userRoleTable="role"
            roleNameCol="role"
        />

        <Resource name="mail/Session"
                  auth="Container"
                  type="javax.mail.Session"
                  mail.smtp.host="localhost"
                  mail.smtp.auth="true"
                  mail.smtp.user="boss"
                  password="boss"
                  scope="Shareable" />

    </Context>
  </Host> <!-- gwtdemo -->
```

To ensure that the browser can find “gwtdemomaxantcouk” edit the host configuration (in Windows this is found under c:\windows\system32\drivers\etc\hosts). This configuration contains a database resource mapped into the JNDI tree under `jdbc/db` and a security realm which uses the `Party` and `Role` database tables. An email resource is also configured.

Please note that while Tomcat at least provides a server environment to test against, it is not necessarily Enterprise compliant in that for example the default transaction manager is not XA compliant. However testing against a server is important since it ensures that objects are serialised between the server and client, as they would be in production. This is important to continually test and it is recommended to develop this way.

To load the GWT modules in the GWT framework’s hosted mode browser (relevant only to pre GWT 2.0 projects), a launch configuration for Eclipse is provided for each GWT module. These launch configurations contain the following information:

```
Main Class: com.google.gwt.dev.HostedMode
Program Arguments: -logLevel SPAM
                   -noserver
                   -whitelist ".*gwtdemomaxantcouk.*"
                   -startupUrl http://gwtdemomaxantcouk:8089/
                   uk.co.maxant.gwtdemo.GWTDemoShop
```

The `logLevel` is set higher than normal to help with problems. For example if a transfer object does not contain a default constructor the error message logged to the hosted mode browser console is not detailed enough to tell you what is wrong. Turning logging up to the SPAM level ensures these details are output.

The `noserver` option tells GWT that you have your own server and that it should not start one for you. Your own server is the very instance of Tomcat configured above.

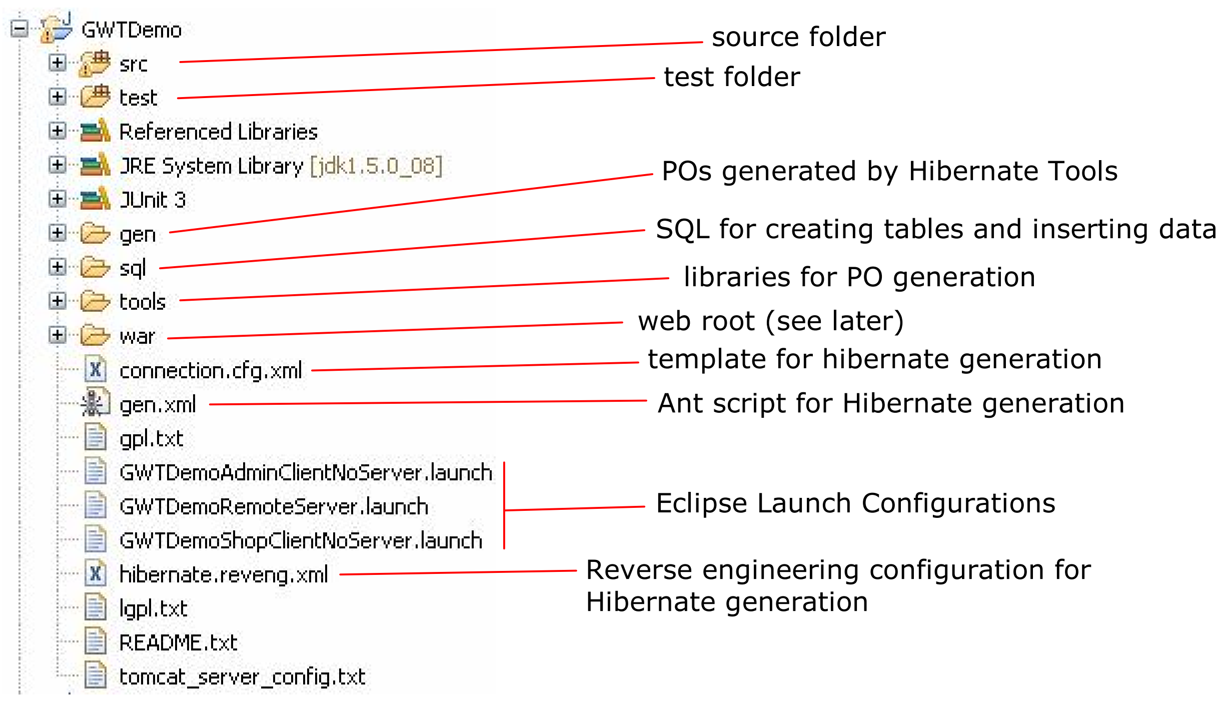
The `whitelist` option tells the hosted mode browser to not complain that you are testing against a domain other than localhost.

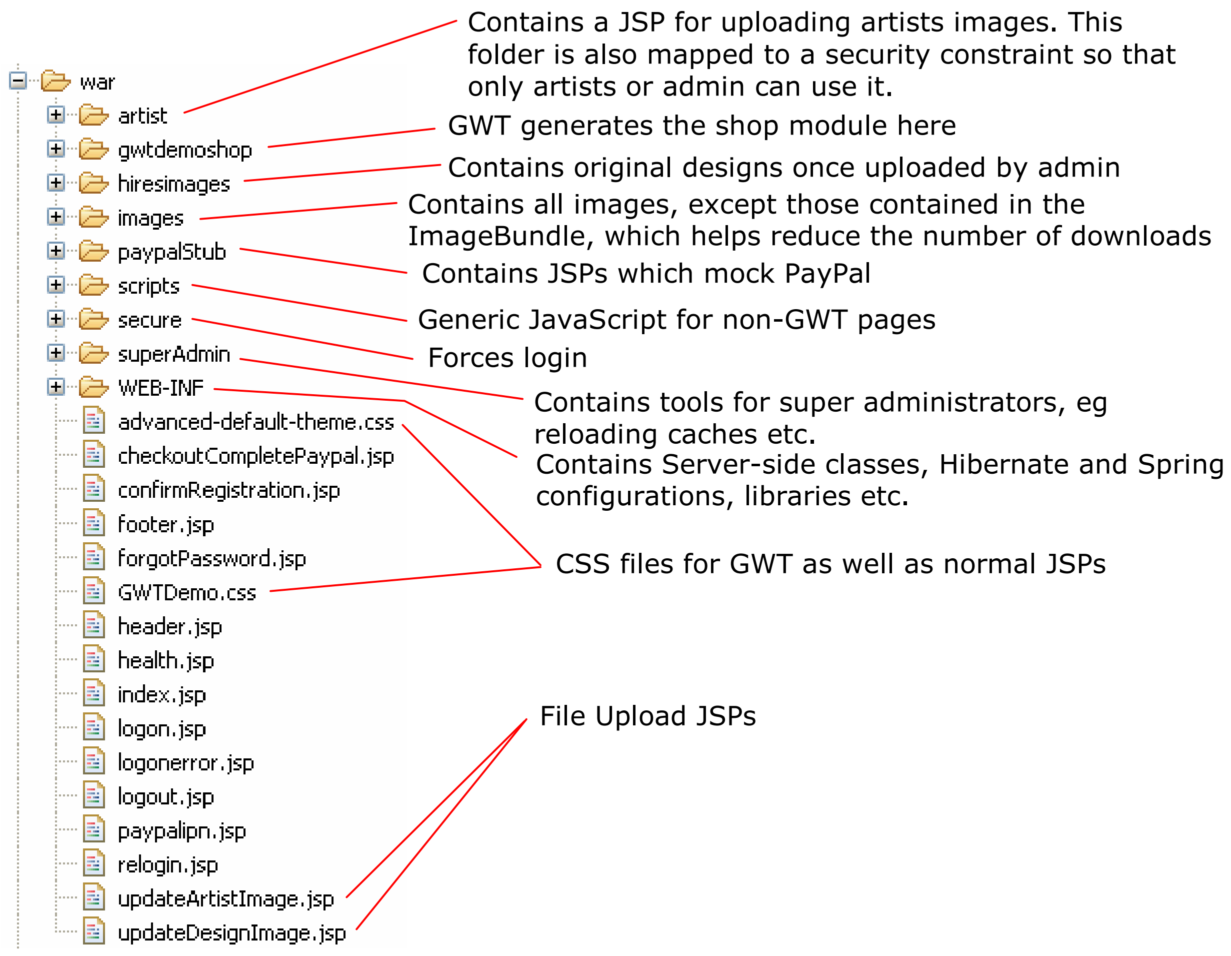
The startupUrl option tells the browser where to find the site.

The last option is the name of the GWT module file including its package path, excluding the file name extension ".gwt.xml".

GWT 2.0 introduces browser specific plugins so that the browser itself interprets the Java code during development time, and so no launch configuration is needed.

The Eclipse Projects folder content is described below:


GWTDemo
src
source folder
test
test folder
Referenced Libraries
JRE System Library [jdk1.5.0_08]
JUnit 3
gen
POs generated by Hibernate Tools
sql
SQL for creating tables and inserting data
tools
libraries for PO generation
war
web root (see later)
connection.cfg.xml
template for hibernate generation
gen.xml
Ant script for Hibernate generation
gpl.txt
GWTDemoAdminClientNoServer.launch
GWTDemoRemoteServer.launch
GWTDemoShopClientNoServer.launch
Eclipse Launch Configurations
hibernate.reveng.xml
Reverse engineering configuration for Hibernate generation
lgpl.txt
README.txt
tomcat_server_config.txt



```
war
  artist
  gwtdemoshop
  hiresimages
  images
  paypalStub
  scripts
  secure
  superAdmin
  WEB-INF
  advanced-default-theme.css
  checkoutCompletePaypal.jsp
  confirmRegistration.jsp
  footer.jsp
  forgotPassword.jsp
  GWTDemo.css
  header.jsp
  health.jsp
  index.jsp
  logon.jsp
  logonerror.jsp
  logout.jsp
  paypalipn.jsp
  relogin.jsp
  updateArtistImage.jsp
  updateDesignImage.jsp
```

Contains a JSP for uploading artists images. This folder is also mapped to a security constraint so that only artists or admin can use it.

GWT generates the shop module here

Contains original designs once uploaded by admin

Contains all images, except those contained in the ImageBundle, which helps reduce the number of downloads

Contains JSPs which mock PayPal

Generic JavaScript for non-GWT pages

Forces login

Contains tools for super administrators, eg reloading caches etc.

Contains Server-side classes, Hibernate and Spring configurations, libraries etc.

CSS files for GWT as well as normal JSPs

File Upload JSPs

Database

The database schemas were as follows.

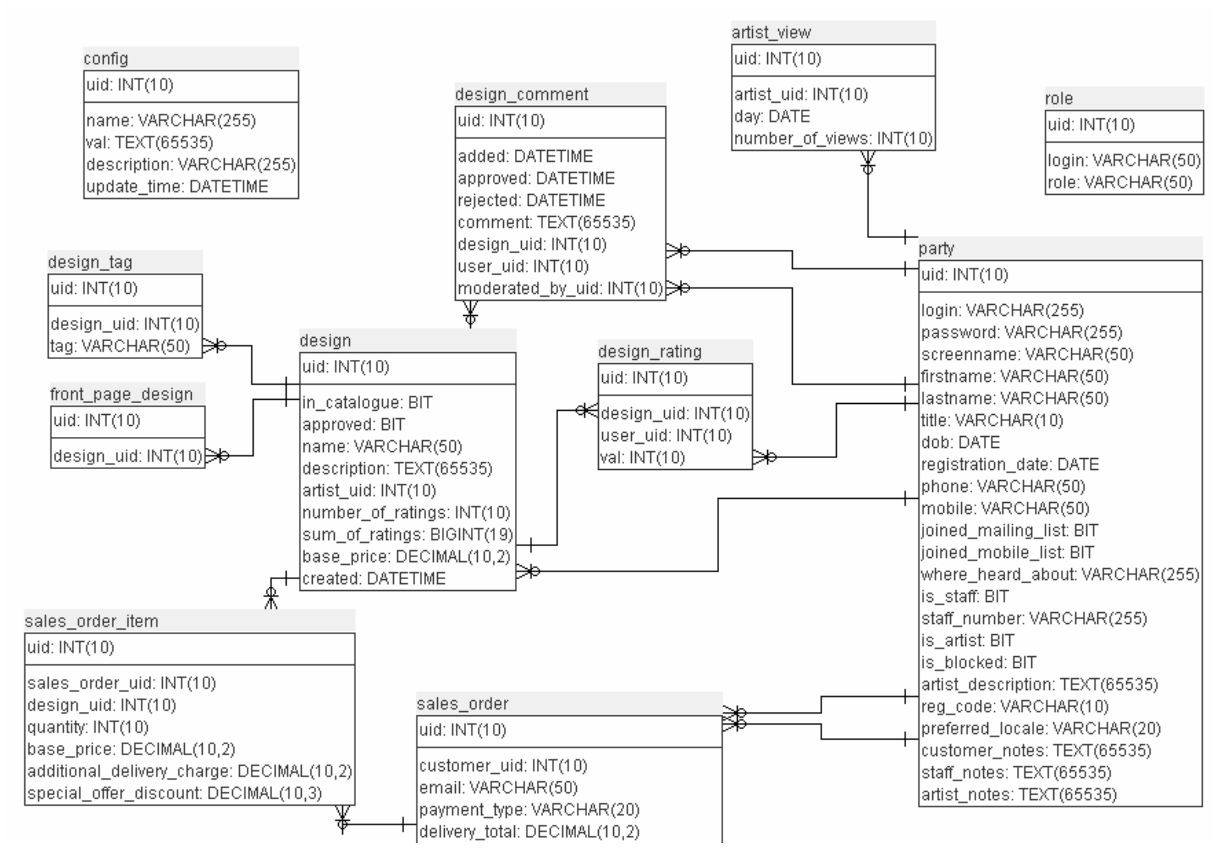


Figure 16: Database Schema

The `party` table represents users (customers, artists and staff). The `artist_view` table contains the number of times each artists profile has been viewed. This information is used to populate the "cloud" on the home page. The `role` table contains the roles of each user, as used by the Tomcat Security Realm. The `design` table contains data about each design that is shown on the site. The `design_comment` table contains customer / artist comments about designs. The `design_rating` contains information about which users have rated which designs, since users may only rate each design once. The `config` table contains the sites configuration, for example email addresses and PayPal configuration. The `design_tag` table contains the tags associated with designs, which are used when searching for designs. The `front_page_design` table contains a list of designs which should be shown on the front page because for example they are top sellers. The `sales_order` table contains an entry for each sales order, sold to a customer and the `sales_order_item` table contains an entry for each design contained in a sales order.

Migration from GWT 1.7 to 2.0

Migration to GWT 2.0 from a previous release is quite easy as it is mostly backward-compatible. The major new features are rather recommendations than obligations. One point to consider is that with the new development mode the URL to the application is expanded by the parameter `gwt.codesvr=localhost:9997`. This parameter lets the browser's developer plugin know that the development mode is on. You have to be careful if there is an HTML redirection in your application. In this case, the `gwt.codesvr` parameter is removed and the application is not launched in the development mode but in the compiled mode. In order to make use of the development mode one should always append the `gwt.codesvr` parameter to the redirected URL.

Example:

The development mode url provided by GWT:

`http://localhost:4671/index.html?gwt.codesvr=localhost:9997`

is redirected to:

`http://localhost:4671/web/redirected.html` (Compiled mode)

add `gwt.codesvr=localhost:9997` again:

`http://localhost:4671/web/redirected.html?gwt.codesvr=localhost:9997`
(Development mode)

Building the application using Maven

If you are building your application with Maven there is the gwt-maven-plugin that compiles the GWT-specific stuff. An example build configuration in the pom.xml looks as follows:

```
<build>
    <outputDirectory>war/WEB-INF/classes</outputDirectory>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>gwt-maven-plugin</artifactId>
            <version>1.2</version>
            <executions>
                <execution>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <output>${basedir}/war</output>
                <runTarget>${basedir}/war/index.html</runTarget>
                <webXml>${basedir}/war/WEB-INF/web.xml</webXml>
                <hostedWebapp>${basedir}/war</hostedWebapp>
            </configuration>
        </plugin>
    </plugins>
</build>
```

GWT Maturity and its Future

Is GWT ready for the Enterprise? In the authors experience it certainly is. The very essence of being able to write once and deploy successfully to every browser is perfect for Enterprise development. With the 1.7 release of GWT and deployment to the `war` folder so that the pre-build environment resembles deployment to production, GWT has reached a level of maturity acceptable for Enterprise development.

An area where GWT is lacking is the amount of widgets it provides. Unfortunately it currently only contains the basics (see http://gwt.google.com/samples/Showcase/Showcase.html). Users of modern web sites expect a very rich experience with sexy widgets, none of which are available with the default GWT download. There are extensions available, but the maturity of those should be investigated separately on a case by case basis.

One area where GWT needs to improve is the development mode performance. With GWT 2.0 the hosted mode browser for developing and debugging has become redundant. Development mode is now supported through the use of a native-code plugin called the Google Web Toolkit Developer Plugin. It exists for many popular browsers and the development mode can be used in the preferred browser. With this change improvements in the performance could be reached, but is still comparatively slow, which can be very frustrating…

With GWT 2.0, released in December 2009, the old style (1.7 and earlier) all-java paradigm was split into a declarative paradigm whereby widget position/presentation on a page are configured via XML and only the logic (handlers, controllers, models, etc.) remain in Java. While this is a welcome development it does mean that anything developed with an older version will likely have to be re-written to keep up to date. A host of other widgets which are currently in the incubator stage should also become available as part of core GWT.

References

Apache Tomcat	http://tomcat.apache.org/
Eclipse	http://www.eclipse.org
Google Web Toolkit	http://code.google.com/webtoolkit/
Hibernate	http://www.hibernate.org
maxant	http://www.maxant.co.uk
maxant Blog	http://blog.maxant.co.uk
MySQL	http://www.mysql.com
Smart GWT	http://code.google.com/p/smartgwt
Spring	http://www.springframework.org
Tag Cloud	http://en.wikipedia.org/wiki/Tag_cloud

Download

The demo application is live at http://gwtdemo.maxant.co.uk and available for download at http://www.maxant.co.uk/whitepapers.jsp

For any questions regarding the example, the platform or indeed anything mentioned in this paper, maxant is available to provide consulting services. Our contact details are below and on http://www.maxant.co.uk.

About the Authors

Dr Ant Kutschera has been working in IT since 2000, implementing software in the enterprise, in all aspects of the software life cycle, including but not limited to requirements gathering, architecture, design, programming, testing, delivery, support and maintenance. He currently works for various clients as an independent consultant, specialising in software architecture, Java EE, Rich Clients and SOA. He can be contacted through:

whitepapers@maxant.co.uk

Barbara Spillmann is software engineer at Swiss Federal Railways, SBB AG. In addition to GWT, she has worked on several enterprise applications based on JEE, JPA, Eclipse RCP, and Spring MVC. She has an MSc in Computer Science from the University of Berne where she wrote her master's thesis in the field of structural and statistical pattern recognition. She can be contacted through:

barbara.spillmann@gmail.com