GWT Web Toolkit

Formerly known as Google Web Toolkit

Compiler

The heart of GWT is the compiler, which compiles Java into JavaScript

- Only supports a subset of the Java language
- The resulting code is JavaScript, which is single-threaded, hence you cannot compile multithreaded code to JavaScript
- Creates a separate JavaScript implementation for each supported browser
- Optimizes JavaScript for each browser. Takes into account browser specific features, bugs and performance optimizations/issues.

Component framework

- Contains a number of widgets that can be used to create an AJAX application
- Client-side framework, which only provides RPC for Java backend. Typically, the application logic is implemented in Java and compiled to JavaScript
- Backend is not tied to Java, but you have to implement all server-side code yourself, including the communication to the server.
- Similar programming model to Vaadin

Client

```
HorizontalPanel panel = new HorizontalPanel();

Button button = new Button("Caption");
button.addClickHandler(new ClickHandler() {
    public void onClick(ClickEvent event) {
        // TODO
    }
});

panel.add(button);
```

JSNI - JavaScript Native Interface

You can embed native JavaScript code into your Java code using the JavaScript Native Interface (JSNI). JavaScript code is wrapped inside comment-like blocks.
Using JSNI, you can access native JavaScript interfaces and communicate between your GWT Java code and JavaScript.

Client

```
public static native void exampleCode(String msg) /*-{
    // Insert your javascript code here
    $wnd.alert(msg);
}- */;
```

Extending Vaadin

If the core framework does not have the component you need, you should follow these steps:

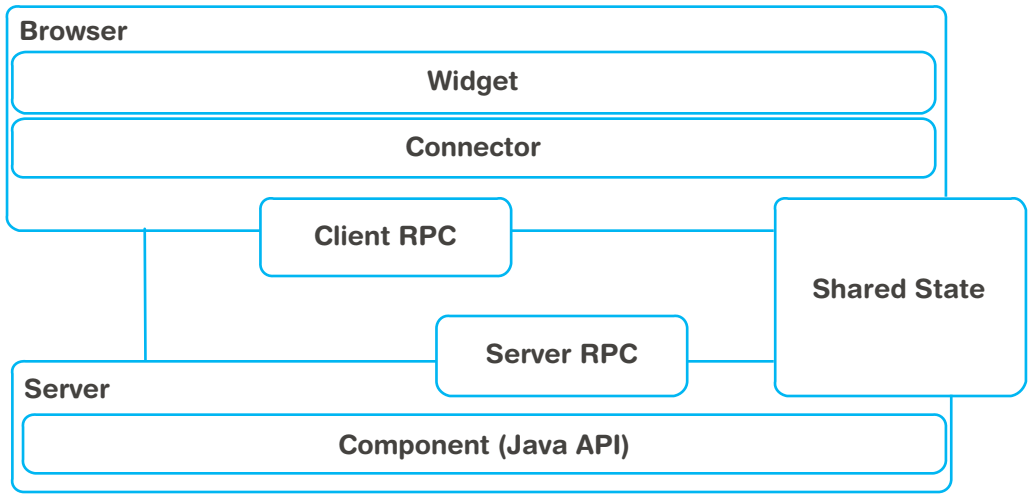
1. Does the **directory** have the component I need?
   - → Go to https://vaadin.com/directory and see, if there is a (third party) add-on that provides you with the functionality you want. No need to reinvent the wheel.
2. Can I create it completely on the server-side using **CustomComponent/ CustomField**?
   - → Try to avoid introducing your own client-side code, if possible. Create a server-side composition, if possible, since then you are less likely to introduce browser specific bugs.
3. If possible, and only if, try **extending a Vaadin widget** and modify its behavior
   - → Most Vaadin widgets are not designed to be extended, hence even a trivial change might end up tedious to do.
4. Is there a **GWT widget** that provides you with the functionality you need?
   - → Create a Vaadin wrapper for the GWT widget.
5. Can you create the component you need by **combining existing GWT widgets**?
   - → You can create a GWT widget composition similar to CustomComponent. Then create a wrapper for the GWT composition widget to use it in a Vaadin application.
6. You can **create a GWT widget from scratch**, so that you define the DOM tree and the DOM interactions.
   - → Do this as a last resort, as creating a widget from scratch can be hard and error prone to browser specific bugs.

Even though most of the add-ons in the directory are created by enthusiastic Vaadin users as hobby projects and not backed up by corporations, they are still often of good quality and maintained. See add-on ratings for user feedback.

The directory also contains official Vaadin add-ons, which are maintained and backed up by Vaadin Ltd.

Consider contributing back to the community by publishing your components as open source add-ons in the directory.

Vaadin Widget Architecture



Widget

- GWT's equivalent to Vaadin's components. Doesn't necessarily need to be GWT code, can also be JavaScript.
- Should be implemented so that it is not bound to any specific server technology and can be used independently in a GWT application.

Component

- Server-side API for your component. Used directly in your Vaadin application.

Connector

- Connectors are the glue between the server and the client.
- The Connector's role is to mediate data and events between the widget and the server
- The Connector creates the widget instance in **createWidget()** method
- Connects the server-side APIs to the widget using the **@Connect** annotation

Client

```
@Connect(MyComponent.class)
public class MyComponentConnector extends
                AbstractComponentConnector {

        protected Widget createWidget(){
                return GWT.create(MyWidget.class);
        }

        protected MyWidget getWidget(){
                return (MyWidget) super.getWidget();
        }
}
```

Remote Procedure Calls (RPC)

- Allows you to make direct method calls between client and server code

Server RPC

- An interface that publishes methods on the server side to the client
- Server RPC is always used when the widget/connector wants to relay data/events to the server
- Create your own interface, which extends ServerRPC
- To publish the interface to the client, call **registerRpc(...)** in the component. RPCs that are not registered, cannot be called.

Shared

```
public interface MyServerRPC extends  ServerRPC {
    public void valueChanged(String newValue);
}
```

Server

```
public class MyComponent extends  AbstractComponent {

    private MyServerRPC rpc = new MyServerRPC() {
        public void valueChanged(String newValue) {
            // TODO
        }
    };

    public MyComponent() {
        registerRpc(rpc);
    }
    ....
}
```

To make a call to the Server RPC, you need to get an RPC instance in your Connector class. The RPC instance is received from the getRpcProxy method in the connector.

Client

```
@Connect(MyComponent.class)
public class MyConnector extends  AbstractComponentConnector {
    private MyServerRPC rpc;

    public MyConnector() {
        rpc = getRpcProxy(MyServerRPC.class);
        ...
    }
    ....
    public void someOtherMethod(String msg) {
        rpc.valueChanged(msg);
    }
}
```

Client RPC

- An interface that publishes methods on the client side to the server
- Client RPC should be used when we want to perform an action in the widget that doesn't change the state of the widget
- Create your own interface, which extends ClientRPC
- To publish the interface to the server, call **registerRpc(<class>, <instance>)** in the connector. RPCs that are not registered, cannot be called.

Shared

```
public interface MyClientRPC extends  ClientRPC {

    public void alert(String msg);

}
```

Client

```
@Connect(MyComponent.class)
public class MyConnector extends  AbstractComponentConnector {

    public MyConnector() {
        registerRpc(MyClientRPC.class, new MyClientRPC() {
            public void alert(String msg) {
                Window.alert(msg);
            }
        });
        ...
    }
    ....
}
```

To make a call to the Client RPC, you need to get an RPC instance in your Component class. The RPC instance is received from the getRpcProxy method in the Component.

Server

```
public class MyComponent extends  AbstractComponent {
    ....
    public void showMessage() {
        getRpcProxy(MyClientRPC.class).alert("My error message");
    }
    ....
}
```

SharedState

- A plain old java object that contains the state of the widget/component
- Shouldn't have any logic, only fields
- Extends AbstractComponentState or AbstractFieldState

Shared

```
public class MyComponentState extends  AbstractComponentState {
   public String text;
}
```

The state object is generated automatically, but you have to override the getState() method in the component class so that the return type is changed to your state object type. If you do not have your own state object, then you do not need to override this method.

Server

```
public class MyComponent extends  AbstractComponent {
    ....
    @Override
    public MyCompentState getState() {
        return (MyComponentState) super.getState();
    }
    ....
}
```

- Access the state object in the connector by calling getState()
- If you want to know if a property's value has changed, override **onStateChanged()** method and call **hasPropertyChanged()** from the event object.

Client

```
@Connect(MyComponent.class)
public class MyConnector extends  AbstractComponentConnector {
    ....
    @Override
    public MyCompentState getState() {
        return (MyComponentState) super.getState();
    }

    @Override
    public void onStateChanged(StateChangedEvent event) {
        if(event.hasPropertyChanged("text")) {
            // TODO do something
        }
    }
    ....
}
```

SharedState (continued)

You can use **@OnStateChange** annotation to mark a method in your Connector that it should be called if the property or properties specified in the annotation has changed.

Client

```
@Connect(MyComponent.class)
public class MyConnector extends  AbstractComponentConnector {
    ....
    @OnStateChange("text")
    public textFieldChanged() {
        // TODO do something
    }
    ...
}
```

If you upon a state change simply call setters in the widget, you can automate the process by annotating the property with **@DelegateToWidget**. For example, for the property "text", the connector would automatically call **setText(getState().text)** for the widget.

Shared

```
public class MyComponentState extends  AbstractComponentState {

    @DelegateToWidget
    public String text;
}
```

NOTE

Client-side code (connector, widget) should **always** treat the state object as **read-only**. Only server-side code may make changes to the state object. This is because state changes made on the client-side are not synchronized to the server.

If you need to make state changes from the client-side, you should make a server RPC call and have the component do the state change.

Extensions

Extensions are "plugable features" that you can apply to components or for the whole UI. For example, if you want to add client-side validation for a TextField, you can implement it as an extension that just "adds" the feature to the TextField in runtime, rather than extending the TextField and implementing your own version of it.

NOTE

You can create both **UI extensions** and **Component extensions**. UI extensions allow you to add functionality on the UI level, while component extensions only apply on a single given component.

Server

```
// Extension bound to a button. Note that it doesn't contain a separate ClickListener
Button button = new Button("Open web page");
BrowserWindowOpener browserWindowOpener = new BrowserWindowOpener("https://vaadin.com");
browserWindowOpener.extend(button);
```

An extension consists of

- Server-side API
- Connector (extends AbstractExtensionConnector)
- State object (optional, extends SharedState)
- RPC (optional, client RPC or server RPC)

Server

```
public class MyExtension extends AbstractExtension {
    // You could pass it in the constructor
    public MyExtension(AbstractComponent componentToExtend) {
        super.extend(componentToExtend);
    }

    // Or in an extend() method
    public void extend(AbstractComponent componentToExtend) {
        super.extend(componentToExtend);
    }

}
```

Client

```
@Connect(MyExtension.class)
public class MyExtensionConnector extends AbstractExtensionConnector {

    @Override
    protected void extend(ServerConnector target) {
        // Get the extended widget
        final Widget targetWidget = ((ComponentConnector) target).getWidget();

        // TODO: Add any client-side logic you wish
    }
}
```

JavaScript Widgets

A widget's client-side code doesn't necessarily need to be written using GWT - it can also be written using plain JavaScript. This means that you can take your favorite JavaScript widget, create a Vaadin wrapper for it and use it in your Vaadin project just like any other component.

Since no GWT code is involved in creating a JavaScript widget, there is also no reason to recompile your widgetset. Just make your changes, deploy and test!

An extension consists of

- Server-side API (extends AbstractJavaScriptComponent)
- State object (optional, extends JavaScriptComponentState)
- Connector (optional, implemented in JavaScript)

In its simplest form, it is enough to have a server-side class which extends AbstractJavaScriptComponent and the JavaScript file to be wrapped is given through the @JavaScript annotation.

Server

```
@JavaScript({"yourJavaScriptWidget.js"})
public class MyJavaScriptComponent extends AbstractJavaScriptComponent{
    ...
}
```

Calling a JavaScript function from your server-side component

Server

```
@JavaScript({"yourJavaScriptWidget.js"})
public class MyJavaScriptComponent extends AbstractJavaScriptComponent{
    ...
    public void reset(){
        this.callFunction("someJavaScriptFunction");
    }
    ...
}
```

You can publish server-side methods to your JavaScript component through JavaScriptFunctions.

Server

```
@JavaScript({"yourJavaScriptWidget.js"})
public class MyJavaScriptComponent extends AbstractJavaScriptComponent{
    ...
    this.addFunction("myMethod", new JavaScriptFunction(){
            @Override
            public void call(JSONArray arguments) throws JSONException {
                // TODO: implement server-side actions here
            }
        });
    ...
}
```

JavaScript Widgets (continued)

You can publish server-side methods to your JavaScript component through normal RPC.

Server

```
@JavaScript({"yourJavaScriptWidget.js"})
public class MyJavaScriptComponent extends AbstractJavaScriptComponent{
    ...
    private MyServerRPC rpc = new MyServerRPC() {
        public void valueChanged(String newValue) {
            // TODO
        }
    };

    public MyJavaScriptComponent() {
        registerRpc(rpc);
    }
...
}
```

Your JavaScript widget can have a shared state object, just like normal widgets. Your state object should extend the JavaScriptComponentState.

Server

```
public class MyComponentState extends JavaScriptComponentState {
    ...
    public String text = "example text";
    ...
}
```

Supported data types in RPC and the shared state objects are

- Primitive Java numbers (byte, char, int, long, float, double) and their boxed types (Byte, Character, Integer, Long, Float, Double) are represented by JavaScript numbers
- The primitive Java boolean and the boxed Boolean are represented by JavaScript booleans
- Dates are represented by JavaScript numbers containing the timestamp
- Java Strings are represented by JavaScript strings
- List, Set and all arrays in Java are represented by JavaScript arrays
- Map<String, ?> in Java is represented by a JavaScript object with fields corresponding to the map keys
- Any other Java Map is represented by a JavaScript array containing two arrays, the first contains the keys and the second contains the values in the same order
- A Java Bean is represented by a JavaScript object with fields corresponding to the bean's properties
- A Java Connector is represented by a JavaScript string containing the connector's id

JavaScript Widgets (continued)

When you need to make method calls from your JavaScript widget to the server (for example, when you just want to pass events from you widget to the server side), you need to make your own Connector for your widget. The Connector class is implemented in JavaScript.

Your JavaScript connector should

- Be included as one of the JavaScript files published through @JavaScript
- Have a global initializer function named based on the fully qualified path to the server-side component class, where dots are replaced by underscores
- Handle communication from the widget to the server
- Take care of state changes

myjavascriptcomponent_connector.js

JS

```
window.com_example_MyJavaScriptComponent = function() {
    // Create the widget, the widget will create its DOM inside the given element
    var widget = new MyJavaScriptWidget(this.getElement());

    self=this;

    // An example of doing method calls to the server
    widget.onClick = function() {
        // This one uses the method published through JavaScriptMethod...
        self.myMethod();

        // ...while this one uses normal RPC
        this.getRpcProxy().valueChanged(widget.getValue());
    };

    // An example of handling state changes
    this.onStateChange = function() {
        var text = this.getState().text;
        // TODO: do something with the new value in the state object
    };

};
```

Server

```
@JavaScript({"yourJavaScriptWidget.js", "myjavascriptcomponent_connector.js" })
public class MyJavaScriptComponent extends AbstractJavaScriptComponent{
    ...
}
```

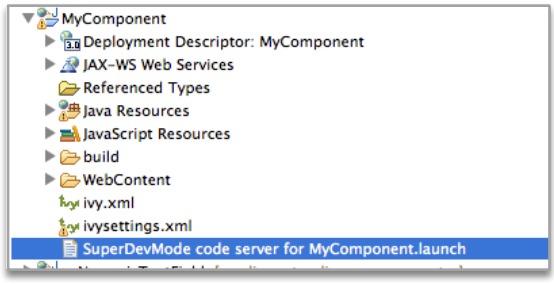
Enabling SuperDevMode In Eclipse

1. Make sure you have already created the widget template
2. Edit your widget set definition file (<widgetname>.gwt.xml)
3. Uncomment devModeRedirectEnabled configuration property

```
<!--
To enable SuperDevMode, uncomment this line.

See https://vaadin.com/wiki/-/wiki/Main/Using%20SuperDevMode for more
information and instructions.
-->
<set-configuration-property name="devModeRedirectEnabled" value="true" />
```

4. Compile the widget set using the compile widget set button in Eclipse
5. Open project properties
6. Choose section Vaadin
7. Click “Create SuperDevMode launch”. This will create a .launch file to your project root
8. Click OK
9. Start your server (Tomcat)
10. Right click on the launch file and select Run As.. (not Debug as…) to launch the SuperDevMode. This will perform an initial compilation of your widget set / module
11. To start using the SuperDevMode, visit your application URL and add ? superdevmode at the end. Alternatively you can launch the super dev mode from the debug window (?debug)
12. After these steps you will see a notification that the widget set is being recompiled, which should only take a few seconds. The page will reload and you are running in SuperDevMode. You can now make changes to your widget code and afterwards press refresh in the browser and immediately start using the new code



NOTE: To be able to debug Java code in Chrome, open the Chrome Inspector (right click > Inspect Element), click the settings icon in the lower corner of the window and check "Scripts > Enable JS source maps". Refresh the page with the inspector open and you will be able to access java code and set breakpoints in the sources tab.