Creating Client Plugins

This page is loosely based on a great manual written by Patrick Talbot (as found on the Servoy forum), and is done so with the author's approval.

In This Chapter

Client plugins are Java classes contained in jar files stored under `/{servoyInstall}/application_server/plugins/` folder, and by which new features can be added to Servoy Developer.

Plugins can be developed in any Java development environment, like Eclipse for example.

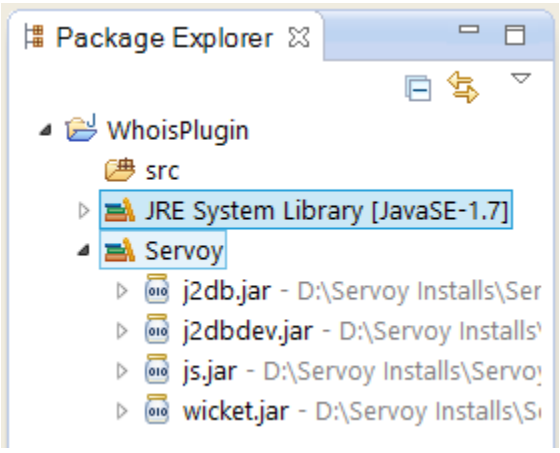
Preliminary Settings

- Create a new Java Project.
- Add a set of needed Servoy libraries.

  The minimal set of libraries may be found in `/{servoyInstall}/application_server/lib/` folder, and is the following:
  - j2db.jar
  - j2dbdev.jar
  - js.jar
  - wicket.jar

  A 'User Library' can be defined in **Window** > **Preferences** which will be available for all future relevant projects. Name it 'Servoy', for example.

Example This is an example of a Java project called WhoisPlugin, and the Servoy libraries added to it.



Implementing the Plugin

Create a New Package under the Java Project

A package is a neat way in Java to organize libraries coming from arbitrary sources and make them work together without problems of 'Name collision'.

The developer can name the package whatever they want, but, by convention, the package hierarchy should reflect the domain name (if there is any) in reverse (from domain to sub-domains). This will avoid name collision with any other libraries, and will spare the developer from having to type the 'fully qualified name' in their code; Eclipse will automatically do that.

Example

This is an example of naming a package: `com.servoy.plugins.whois` - where 'com.servoy' is a prefix which follows the rule listed above, and 'whois' is the plugin name.

Implement the Plugin Interface

In order to implement a client plugin, create a class that implements one or more of the three interfaces: IServerPlugin, ISmartClientPlugin, and IClientPlugin.

Before starting coding, do study these interfaces on api docs.

Example

This example shows the implementation of a component which will query a whois server and retrieve whois information about a domain.

```
public class WhoisPlugin implements IClientPlugin {

    public static final String PLUGIN_NAME = "whois";
    private WhoisPluginProvider provider;

    @Override
    public Properties getProperties() {
        Properties props = new Properties();
        props.put(DISPLAY_NAME, getName());
        return props;
    }

    @Override
    public void load() throws PluginException {
        // ignore
    }

    @Override
    public void unload() throws PluginException {
        provider = null;
    }

    @Override
    public void propertyChange(PropertyChangeEvent arg0) {
        // ignore
    }

    @Override
    public IScriptable getScriptObject() {
        if (provider == null) {
            provider = new WhoisPluginProvider();
        }
        return provider;
    }

    @Override
    public Icon getImage() {
        URL iconUrl = getClass().getResource("images/whois.png"); //the image is added under a package 'com.
servoy.plugins.whois.images' added to the WhoisPlugin project.
        if (iconUrl != null) {
            return new ImageIcon(iconUrl);
        } else {
            return null;
        }
    }

    @Override
    public String getName() {
        return PLUGIN_NAME;
    }

    @Override
    public void initialize(IClientPluginAccess arg0) throws PluginException {
        // ignore
    }
}
```

Implement the Script Object

The method `getStriptObject` inherited by IClientPlugin from IScriptableProvider interface, returns the object that will provide the plugin with scripting properties and methods. So, by convention, it is called a Provider.

A provider implements interfaces IScriptable and IReturnedTypesProvider. Do study them on api docs.

Create a class which implements the two interfaces. This class will provide methods representing the plugin behavior.

Code the Plugin Main Behavior

In order to specify which methods are what, use the JavaDoc annotations system which identifies getter/setter methods for plugin properties, as well as function methods for the plugin functions. The JavaDoc annotation system is also used for documenting the plugin.

For a proper understanding of how to use JavaDoc and how to build the documentation of a plugin, see Documenting the Plugin Api.

Example

This example shows the implementation of the WhoisPluginProvider - the scriptable object which provides the behavior for the 'whois' plugin.

The plugin will expose an overloaded `query` method, as well as three other properties: `port`, `server`, and `timeout`.

The main `query` method contains JavaDoc which provides a description of the function and a sample. The other overloaded methods will display the same description and sample, having them copied via `@clonedesc` and `@sampleas` annotations from the main method.

```
@ServoyDocumented(publicName = WhoisPlugin.PLUGIN_NAME, scriptingName = "plugins." + WhoisPlugin.PLUGIN_NAME)
public class WhoisPluginProvider implements IScriptable, IReturnedTypesProvider {

    @Override
    public Class<?>[] getAllReturnedTypes() {
        return null;
    }

    private String server = "whois.networksolutions.com";
    private int port = 43;
    private int timeout = 30 * 1000; // unit is milliseconds

    @JSGetter
    public String getServer() {
        return server;
    }

    @JSSetter
    public void setServer(String server) {
        this.server = server;
    }

    @JSGetter
    public int getPort() {
        return port;
    }

    @JSSetter
    public void setPort(int port) {
        this.port = port;
    }

    @JSGetter
    public int getTimeout() {
        return timeout;
    }

    @JSSetter
    public void setTimeout(int timeout) {
        this.timeout = timeout;
    }

    /**
     * @clonedesc query(String, String, int, int)
     * @sampleas query(String, String, int, int)
     * @param domainName
     */
    @JSFunction
    public String query(String domainName) {
```

```
        return query(domainName, this.server, this.port, this.timeout);
    }

    /**
     * @clonedesc query(String, String, int, int)
     * @sampleas query(String, String, int, int)
     * @param domainName
     * @param server
     */
    @JSFunction
    public String query(String domainName, String server) {
        return query(domainName, server, this.port, this.timeout);
    }

    /**
     * @clonedesc query(String, String, int, int)
     * @sampleas query(String, String, int, int)
     * @param domaninName
     * @param server
     * @param port
     */
    @JSFunction
    public String query(String domainName, String server, int port) {
        return query(domainName, server, port, this.timeout);
    }

    /**
     * Calls a whois server to retrieve information about the provided domain name
     *
     * @sample
     * // call a whois server by providing a domain name and get info in return
     * var result = plugins.whois.query('servoy.com');
     * // alternatively, provide an alternate server (default is networksolutions.com)
     * var result = plugins.whois.query('servoy.com', 'whois.internic.net');
     * // provide a port, if not standard (43 by deafault)
     * var result = plugins.whois.query('servoy.com', 'whois.internic.net', 43);
     * // provide a timeout length (unit is milliseconds, default is 30 seconds)
     * var result = plugins.whois.query('servoy.com', 'whois.internic.net', 43, 50000);
     *
     * @param domainName
     * @param server
     * @param port
     * @param timeout
     * @return
     */
    @JSFunction
    public String query(String domainName, String server, int port, int timeout) {
        try {
            // create the socket
            Socket socket = new Socket(server, port);
            socket.setSoTimeout(timeout);
            // create a reader to get the response from the server
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            // create an output stream to send our query to the server
            DataOutputStream out = new DataOutputStream(socket.getOutputStream());
            // call the service with the domainName supplied
            // and terminate with carriage return)
            out.writeBytes(domainName + " \r\n");
            // read the response from the server
            String str1 = null;
            StringBuffer buffer = new StringBuffer();
            while ((str1 = in.readLine()) != null) {
                buffer.append(str1);
                buffer.append("\r\n");
            }
            // close our stream and reader
            out.close();
            in.close();
            // close the socket
            socket.close();
            // return the result as String
            return buffer.toString();
        } catch (IOException ioEx) {
```

```
            return ioEx.getLocalizedMessage();
        } catch (Exception ex) {
            return ex.getLocalizedMessage();
        }
    }
}
```


Make sure to have selected the correct target against which the project is compiled. It needs to be consistent with the Java version the Servoy install is built against. For this, do check **Project** > **Properties** > **Java Compiler Node** > **JDK Compliance Panel**.

Entry Points

Since the class which implements the IServerPlugin, ISmartClientPlugin or IClientPlugin is one file among many inside the jar, it's advised indicate which file is the plugin entry point.

The plugin jar can use Java Service Provider to expose Servoy Plugin classes. There should be a file inside the plugin jar at the path: `META-INF/services/com.servoy.j2db.plugins.IPlugin` which contains a line for each class in the jar that implements IPlugin). The plugin should also have a default constructor (with no parameters). If file `com.servoy.j2db.plugins.IPlugin` is missing or contains invalid entries Servoy will automatically scan the jar for all classes that implement interface IPlugin. An example of file content (for whois plugin) is:

META-INF/services/com.servoy.j2db.plugins.IPlugin

```
com.servoy.plugins.whois.WhoisPlugin
```

Package the Plugin

- Right click on the project and choose **Export** > **Java** > **JAR file**.
- Click Next

  Optionally, deselect the `.classpath` and `.project` files to avoid polluting the jar with unwanted files only used by Eclipse.

  Select the export destination. One may choose to export the jar directly into the `/{servoyInstall}/application_server/plugins` directory.
- Click **Next**

  Leave the 2 **Export class files…** checked, and check the **Save the description of this JAR in the workspace**. Use the **browse** button to navigate to the project, and give a name to the definition. Eclipse automatically adds the 'jardesc' extension.

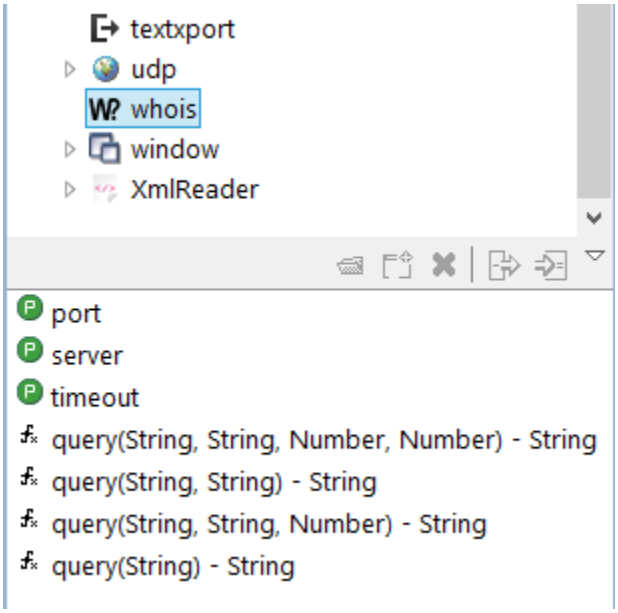
  

  What is nice about this option is that the next time the developer wants to deploy the jar (with modified sources for example), all they will need to do is right-click on the file `xxx.jardesc` in the Package explorer and choose **Create JAR** in the menu, with no need to go through all the Export dialogs each time a change occurs in the plugin that's being developed.
- Click **Next** once more. Leave it as is, or choose other options.
- **Finish**.

Testing the Plugin

When opening the Servoy Developer, it should be visible under **Plugins** node in **Solution Explorer**.

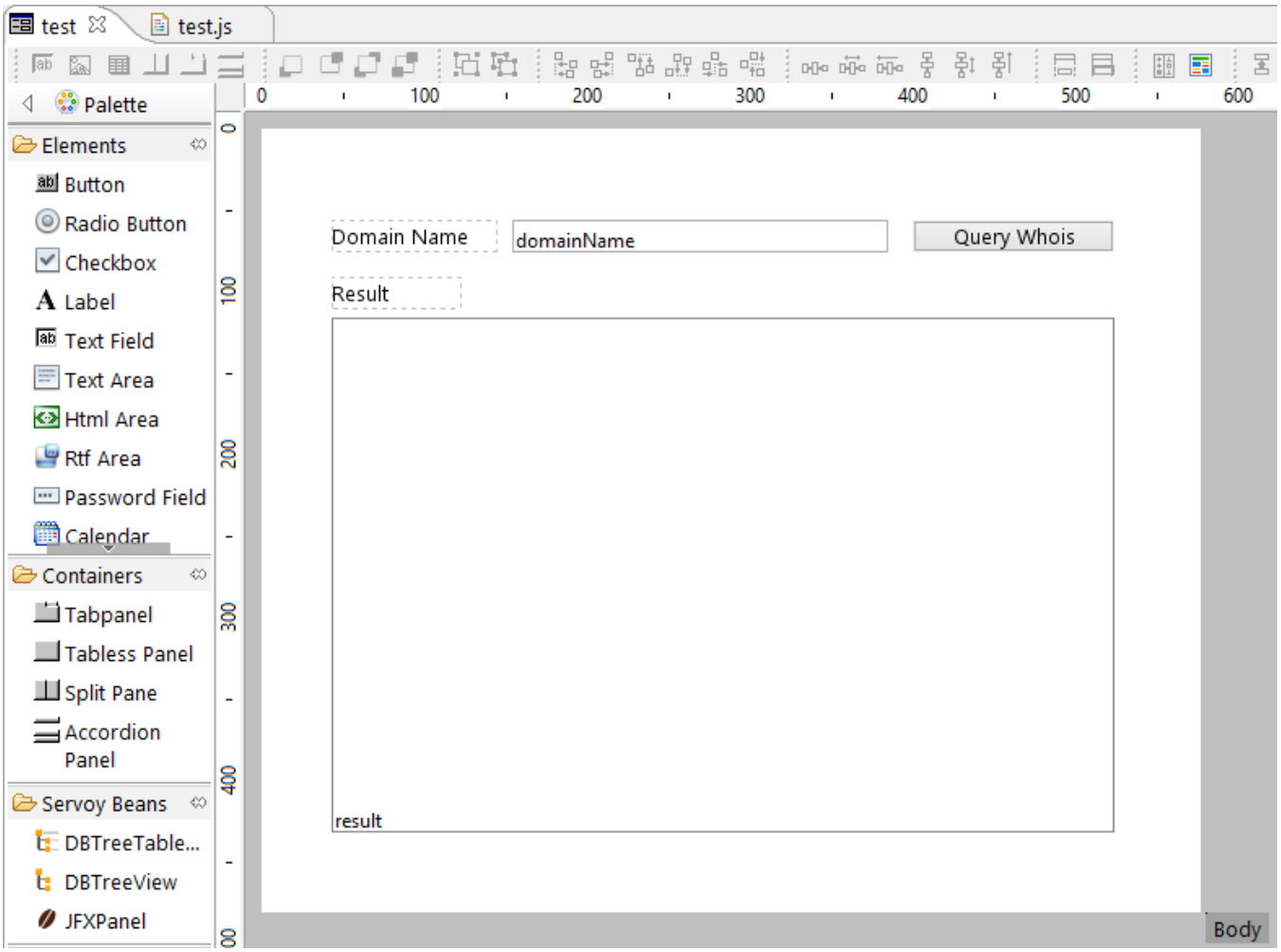
Example #1 This example shows how the 'whois' plugin is displayed in the **Solution Explorer**. Notice the overloaded function `query` and the three properties.



Example #2 This example shows a small sample solution 'WhoisTest' which tests the 'whois' plugin.

The solution has a simple test form with two fields based on two form variables `domainName` and `result`, and a button `Query whois` whose action will call the `query` function.

The Form Editor



The Script Editor

```
var domainName = "";
var result = "";

function onQueryWhois()
{
        if (domainName != null && domainName.length > 0) {
                result = plugins.whois.query(domainName, "whois.internic.net");
        }
}
```

Running Smart Client