ElectricFlow SDK Plugin Developer Guide

Version 6.0

Electric Cloud, Inc.
35 South Market Street, Suite 100
San Jose, CA
www.electric-cloud.com

Contents

Chapter 1: Introductions, Types, and Definitions 1-1
What is a Plugin? 1-1
What can a plugin do? 1-1
Plugin Types 1-1
Plugin contents 1-4
JAR File Elements 1-4
urlLoader component 1-9
JAR File Layout 1-10
Plugin Versioning 1-11

Chapter 2: Plugin References 2-1
Reference Contexts 2-1
Plugin Visibility in the UI 2-1
Plugin Projects 2-2
Plugin Procedures 2-2
Referencing Plugin Scripts on an Agent 2-3
Plugin Property Path Syntax 2-3

Chapter 3: Installing, Uninstalling, or Promoting a Plugin 3-1
Installing a Plugin 3-1
What Happens Behind the Scenes? 3-1
Promoting or Demoting a Plugin 3-2
To Install and Promote Your Plugin 3-2
Uninstalling a Plugin 3-2
Plugin Setup Scripts 3-2
What’s in ec_setup? 3-3

Chapter 4: Using the ElectricFlow SDK 4-1
System Requirements 4-1
Functionality 4-1
Download and Unzip the File 4-1
Creating a Plugin Project 4-2
Other Plugin Development Tools 4-2
Plugin Source Layout 4-3
Plugin Build System 4-5
Building Your Plugin 4-6
Building and Deploying Your Plugin 4-6

Chapter 5: Eclipse Integration 5-1
Importing SDK Examples into Eclipse 5-1

Installing Launchers 5-1
Finding and Setting the Eclipse Workspace 5-1
Finding and Setting the ElectricFlow Installation Directory 5-2
Setting Favorite Launchers 5-2
Using the launchers 5-2
Avoiding a typical Eclipse error message 5-2
Ant Launchers 5-2
Program Launchers 5-3
Handling Multiple ElectricFlow Servers 5-3
Refreshing Corrupt Launchers 5-4

Chapter 6: Examples and Tutorials 6-1
Codeless and CGI Examples 6-1
Codeless Example Overview 6-1
HelloWorldCGIExample Overview 6-2
JobsLite Example 6-3
Some GWT Basic Information 6-3
GWT Plugin Examples 6-5
HelloWorldExample overview 6-5
ConfigureExample Overview 6-6
RequestResponseExample Overview 6-7
RunProcedureExample Overview 6-8
GWT Parameter Panels 6-9
How to Add a GWT Parameter Panel to your Plugin 6-9
ExampleParameterPanel Overview 6-10

Chapter 7: Debugging your GWT Plugin 7-1
Debugging Google Web Toolkit (GWT) Plugins 7-1
Step 1 - Download and Install the Google for Eclipse Plugin 7-2
Step 2 - Setting Google-Related Properties for Your Project 7-3
Step 3 - Configure Your Project’s Build Path 7-6
Step 4 - Creating the Debug Configuration in Eclipse 7-8

Chapter 8: Troubleshooting 8-1
Common Problems and Resolutions 8-1

Appendix A: Converting SDK Versions A-1
Moving to ElectricFlow SDK 6.0 A-1
Moving to CommanderSDK 5.0 A-1
Moving from CommanderSDK 1.2 to 2.0 A-2
Moving from CommanderSDK 1.1 to 1.2 A-3
Moving from CommanderSDK 1.0 to 1.1 A-3
Migrating from CommanderSDK 0.9.1 to 1.0 A-4
Summary A-4
Making ElectricFlow Requests and Parsing the Result A-5
Changes to Factory Classes that Extend ComponentBaseFactory A-9
Transitioning from Using MultiRequestLoader A-10
CommanderSDK 1.0 File Changes A-11
Updating “Plugin-in-a box” to the CommanderSDK A-11

Loading SDK 1.0 Examples in Eclipse A-11
Updating an existing plugin to Use CommanderSDK 1.0 .... A-11
Updating External Tool Launchers A-11

Appendix B: plugin.xml Schema B-1
<plugin> Element Schema Definition B-1

Appendix C: ElectricCommander::View API C-1

Chapter 1: Introductions, Types, and Definitions

What is a Plugin?

A plugin is a collection of one or more features that can be added to ElectricFlow. A plugin is delivered as a JAR file containing the feature(s) implementation. When a plugin is installed, the ElectricFlow server extracts the JAR contents to disk into a configurable plugins directory. A plugin has an associated ElectricFlow project that can contain procedures and properties required by the implementation.

What can a plugin do?

You can add any number of plugins to ElectricFlow, depending on the additional functionality you would like to add to the product. A plugin can add:

- New tabs for the ElectricFlow UI
- New product pages - for example, to integrate another software product you want to use in conjunction with ElectricFlow
- A configuration page, specific to plugin implementation requirements
- A plugin can contain its own help topic to provide specific information to users about the plugin
- A re-write of existing tabs or pages to incorporate one or more special functions you need
- A place where scripts “live” on the web server so other plugins can use them

Also, it is possible to replace virtually every page in the ElectricFlow UI, with the exception of the ElectricFlow login screen.

Plugin Types

There are three different ways to develop plugins for ElectricFlow. These three types are named based on the language of their source code.

- Codeless
- CGI
- GWT/Java

Chapter 6 contains descriptions of examples for each of these plugin types. The actual code for each of these examples is in the ElectricFlow SDK.

Codeless

A codeless plugin is the simplest of the plugin types. It may be just a library of logic (in the form of ElectricFlow procedures defined in `project.xml`) for other projects or plugins to use. This plugin type could consist of a single page that leverages the `urlLoader` component to embed one or more pages in a section of an existing ElectricFlow platform page. For more information on urlLoader, see urlLoader component.

Another example of a codeless plugin is one that defines a page to be the target of a tab. For this case, you can avoid creating the plugin altogether and simply create the tab in ElectricFlow with the page contents "inlined". A tab is typically specified in a view definition like this:

```
<tab>
  <label>MyTab</label>
  <url>pages/MyPlugin/mypage</url>
</tab>
```

For an inline page, it would look like this:

```
<tab>
  <label>MyTab</label>
  <page>
    <componentContainer>
    …
    </componentContainer>
  </page>
</tab>
```

See ElectricFlow online Help topic, “Customizing the ElectricFlow UI”, for more information on tabs. See ElectricFlow Platform Pages for details on creating pages.

Pros: Rapid development

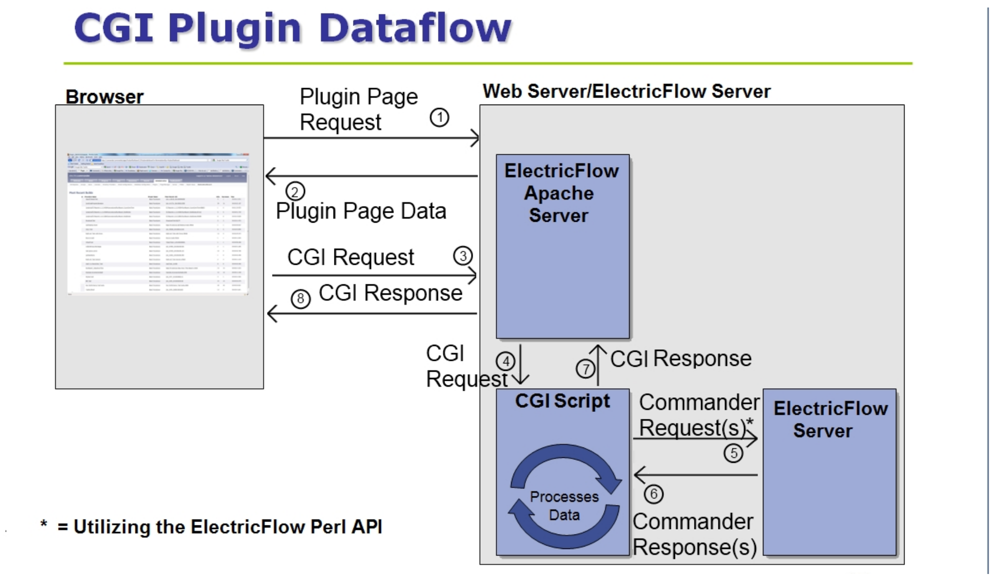
Cons: Limited flexibility

CGI

This plugin style contains at least three files: `plugin.xml`, a page definition file in the pages directory, and a CGI script in the `cgi-bin` directory. The page definition uses `urlLoader` to embed the output of the CGI script in a ElectricFlow platform page.

The CGI script runs as the same OS user as Apache. This script runs with the same ElectricFlow session the browser uses when issuing ElectricFlow server requests.

The following diag



ram illustrates the data flow in a CGI plugin.

The plugin page request is for a URL in the form `/commander/pages/<plugin-key>-<version>/mypage`. The CGI request is in the form `/commander/plugins/<plugin-key>-<version>/cgi-bin/myscript.cgi`. Any relative URLs in the CGI script result HTML is interpreted relative to the page URL. Thus, to refer to auxiliary files in the `cgi-bin` directory, specify an absolute path or a relative URL like "`../../cgi-bin/foo.jpg`".

Pros:

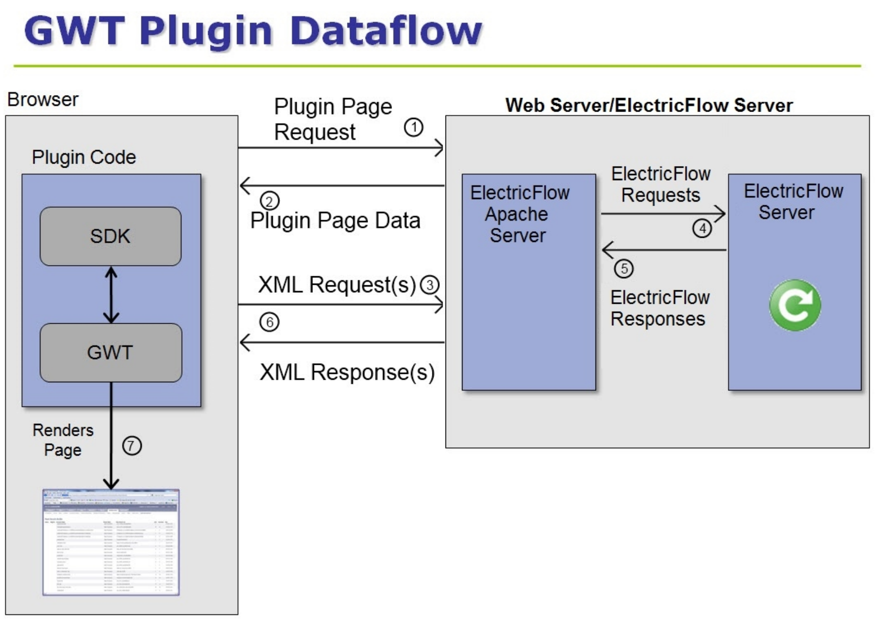
- Simple
    - ElectricFlow Perl API is available, if the CGI script is written in Perl.
- A common basic scripting language
- Fast
    - CGI script runs on the web server machine.
    - Usually co-located with the ElectricFlow Server.
    - Works well for building UI pages that primarily display data (such as Dashboards).

Cons:

- Difficult to implement dynamic UI elements.
- Developer is responsible for things like matching ElectricFlow “look-and-feel” (CSS).

GWT/Java

With this plugin type, ElectricFlow requests are issued with the server session associated with the ElectricFlow web browser session, just as with the core web UI.



Pros:

- Rapid development of rich, dynamic UI's with easy connection to ElectricFlow
  - SuggestBox/Oracle and other AJAX UI features.
- Tools and IDE's ease development
  - Full-featured debugger available.
- Easy to create content with the “look-and-feel” of ElectricFlow
  - For example, correct style sheets (CSS) are used automatically.

Cons:

- Must be familiar or willing to learn Java and GWT.
- By default, GWT/Java performs "processing logic" in the web browser.
  For data intensive tasks, a combination of GWT/Java + CGI/perl may be required to offload work to the web server.

Plugin contents

The plugin JAR file may contain the definition of the project, web UI pages, scripts, and components (JavaScript units of logic).

JAR File Elements

A plugin generally contains the following elements in the JAR file:

Scripts

Some plugins use auxiliary scripts to help perform their functions. Typically, these are CGI scripts, or scripts that run in a step on an agent. GWT components are implemented in JavaScript and that logic is run in the browser. It is also possible for a plugin developer to create JavaScript files by hand and have a plugin page reference those files. Depending on a script's execution context, the script could be located in different parts of the plugin jar.

Metadata

plugin.xml

This file defines the plugin and declares what it contains. Some of the metadata is used by the Plugin Manager when the plugin is listed in the plugin list. Some of this information is critical to the operation of the plugin. For example, if a plugin contains GWT components, the components must be defined in `plugin.xml` for component references in pages to resolve to the appropriate Javascript file containing that component's logic.

Note: The `key` and `version` element names are required.

XML element name	Description
`author`	The name of the plugin author.
`authorUrl`	The author's web site URL.
`category`	The name of the category for this plugin. Plugins that do not specify a category are placed in the "Other" category. Categories are used for grouping similar plugins in the UI—you can create your own category types. Some standard categories include: `System` - components that are part of the core system `Source Control` - software configuration management drivers `Defect Tracking` - for integrations of various defect tracking tools `Reporting` - contains custom report types `Resource Management` - to manage VMs or other resources
`commander-version`	The range of ElectricFlow server versions the plugin is compatible with. The install will fail if the server's version is outside the specified range. If min (or max) is omitted, the range is considered to include all earlier (or later) versions. The `commander-version` element can be omitted if no version checks are required.
`components`	Components are interfaces exposed by the plugin—used to define page definitions. See Components for more information about components.
`configure`	The location of the plugin configuration page, interpreted as a relative path based at plugin's pages directory, and should contain a `componentContainer`. Setting this field results in displaying a Configure link for this plugin's entry on the **Administration** > **Plugin** page.

XML element name	Description
`customTypes`	A list of custom types defined by this plugin, as `customType` elements. A custom type declares a plugin page that replaces a regular ElectricFlow platform page for performing a particular action. In ElectricCommander 4.0, the most common use case is to define a parameter panel or replace a full ElectricFlow platform page. For a full-page replacement, a `customType` element has a name attribute and the following child elements: `displayName` - The name of the type to display on an ElectricFlow UI page. Currently, neither the Plugin Manager nor any other ElectricFlow web page uses this element. `description` - Description of the custom type. `page` - Declares a plugin page to use for a particular action. This element contains the following attributes: `pageName` - the name of an ElectricFlow page being replaced. For example, "`runProcedure`". `definition` - plugin page to load for this action. For example, "`pages/MyComponent_run.xml`". A `customType` may contain multiple `<page>` elements, re-mapping multiple ElectricFlow actions to custom pages. For redefining parameter panels, a `customType` element contains a name attribute and a `parameterPanel` child element. The `parameterPanel` element contains a javascript child element. The `<javascript>` element declares which Javascript file contains the component logic. See the `RunProcedureExample` (in the ElectricFlow SDK) as an example for creating a custom Run Procedure page. See the `ExampleParameterPanel` as an example for redefining a parameter panel.
`depends`	A list of plugin keys this plugin depends on. The install will fail if the listed plugins are not already installed and promoted. If the optional `min` attribute is specified, the server will match plugins at least as recent as the specified version only.
`description`	A short text description of the plugin displayed on the **Administration** > **Plugin** page.
`executables`	A list of pattern elements that contain glob patterns relative to the JAR file that identifies any files that need to be marked executable on Linux. By default all files that match `'cgi-bin/*', '*.sh', or '*.pl'` will be marked with mode 0755 while others will be marked 0644. Additional patterns can be specified. If the optional attribute `useDefaults="0"` is specified, only listed patterns will be used.

XML element name	Description
`help`	Location of the Help page for the plugin, interpreted as a relative path based at the plugin's pages directory. Setting this field results in displaying a Help link for this plugin's entry on the **Administration** > **Plugin** page.
`key`	Unique plugin identifier that must not change between plugin versions. Because this value is used to construct directory names, limit the number of characters to what is safe to appear in filenames. This identifier must be unique across all plugins.
`label`	The name of the plugin you want to be displayed in the UI. If not specified, it will be the same as the `<key>`. Displayed on the **Administration** > **Plugin** page.
`version`	Plugin version number in the form `major.minor.patch.buildNumber`. Versions will be sorted lexicographically on the **Administration** > **Plugin** page.

See plugin.xml Schema for the `plugin.xml` schema definition.

project.xml

A plugin is associated with a project in ElectricFlow. By default, this project is empty and can be used for storing and retrieving plugin state (for example, configuration information). However, some plugins require procedures and other initial state set at install-time.

This file, if present, contains that state, in the form of a relocatable project export from ElectricFlow. You should create the project and state in ElectricFlow, obtain a relocatable export of the project, then store it as `project.xml` in the plugin `META-INF` directory. When the plugin is installed, the server will create the plugin project from this XML file instead of creating a default project.

To create a relocatable export, issue a command like the following:

```
ectool export c:/project.xml --path /projects/<project-name> --relocatable 1
```

This command creates the export file in the root directory of the ElectricFlow server (for a Windows server). Move it from the root directory to the `META-INF` directory in your plugin source tree.

Components

A plugin may contain one or more components. A component is a self-contained unit of logic (JavaScript) that performs one or more tasks. Multiple instances of the same component may be used to create a page in your plugin. Also, you can reference or embed your component in other, future plugins. Components allow you to build modular plugins to share with other plugin developers in your group.

Include components in the `htdocs/war` section of the JAR file.

A component is defined in `plugin.xml` with the `<component>` element. At a minimum, a component has a "name" attribute declaring the name of the component and a `<javascript>` child element declaring which Javascript file contains the component logic. In addition, it could have `<style>` elements that refer to style sheets containing styles the component uses. Also, a component definition can contain arbitrary child elements the component can access as parameters—the component can then alter its behavior based on these XML parameters. A GWT component uses the `BrowserContext.getParameter()` method to retrieve parameter values specified this way.

ElectricFlow Platform Pages

<page>.xml

This page is generally the web browser entry point for your plugin. The `createPlugin.pl` script names this file in the format of `<component>_run.xml`.

A plugin may contain pages that are shown in the ElectricFlow platform UI. Pages are XHTML documents defined in the pages directory and must be named `<pageName>.xml`. A plugin page is referenced with a URL like `https://<webserver>/commander/pages/<plugin key>-<version>/<pageName>`.

Note: The page is accessed without the "`.xml`" suffix.

The top-level element in a page definition may be any legal XML element name. However, to reference components, a page must have a `<componentContainer>` element that is an ancestor of the `<component>` elements. Our recommendation: The top-level page element should be `<componentContainer>`.

A page definition may contain HTML interspersed with component references:

```
<componentContainer>
  <title>My Page</title>
  <table>
    <tr>
      <td><component plugin="foo" version="1.0.0" ref="MyComp" /></td>
    </tr>
  </table>
  <p>My paragraph.</p>
</componentContainer>
```

Each page ***must*** be XHTML and every element must be terminated.

A component reference contains the following attributes:

`plugin`	The name of the plugin that defines the desired component
`version`	(optional) The plugin version to reference. Defaults to the currently promoted version.
`ref`	The name of the component.

A component reference may have child elements that set/override the same child elements that can exist in the component definition in `plugin.xml`.

Help Links

To specify an associated Help page for your plugin page, use the `<helpLink>` element. This link shows up as the target of the EC Help link in the top-right corner of the page. Like other page URLs, this link must not end in "`.xml`". For example, if you are planning to use a `help.xml` page, the link would be "`<helpLink>help</helpLink>`".

Note that this is a help page associated with this plugin page. This mechanism provides the ability to specify a different help page for every plugin page, in addition to having an overall plugin help page declared in `plugin.xml`. This overall help page is linked from the plugin's entry on the Administration > Plugins web page, to distinguish it from other help page links.

Styles

Pages can refer to styles using relative URLs. For example, to refer to a style defined in the core ElectricFlow platform UI, specify a URL like "`../../lib/styles/data.css" or "../../styles/JobStatus.css`".

Note: Referring to styles defined in the core platform UI is risky because future ElectricFlow versions may or may not use the same `.css` files. To refer to styles defined in the plugin `htdocs` directory, a URL like `../../plugins/@PLUGIN_KEY@-@PLUGIN_VERSION@/foo.css` would be appropriate. The plugin build process substitutes the tokens used in this URL with the values appropriate for your plugin.

Two ways to refer to styles in a page:

1. Using `<style>` elements in a component reference.
2. Using a `<link>` element in the page definition.

   For example:

   ```
   <link rel="Stylesheet" href="../../plugins/@PLUGIN_KEY@-@PLUGIN_VERSION@/foo.css
   "type="text/css" media="screen" />
   ```

Note: This section is applicable for developing CGI plugins, not GWT plugins. For information on including a CSS file for GWT-based plugins, see GWT Plugin Examples.

urlLoader component

ElectricCommander 3.6 and later ships with plugins that provide core UI functionality (for example, EC-Homepage), integrations with third-party systems (for example, EC-Ant), or utility functions that build procedures or other plugins may use (EC-Core).

The `urlLoader` **component** of EC-Core is *at the heart of any non-trivial, non-GWT plugin*. Its purpose is to embed the results of a web request into a plugin page. The web request may be to an arbitrary URL or a relative URL under a plugin. You can provide ElectricFlow requests to run prior to invoking a web request. The ElectricFlow response is passed to the specified URL or plugin page as POST data.

The `urlLoader` component takes the following arguments:

`url`	URL to invoke, may be absolute or relative. For a plugin CGI URL, this is typically in the form "`cgi-bin/myscript.pl`"
`plugin`	Plugin name containing the page to load. Required if specifying a relative URL.
`version`	Plugin version containing the page to load. Required if specifying a a relative URL.
`enableGetPassthru`	For a plugin URL, this passes `get parameters` on the current page to the target URL. Defaults to `true`.
`evalScripts`	If the target URL response contains any `<script>` elements, evaluate those scripts in the Browser. This requires the URL response be proper XHTML. Defaults to `false`.
`requests`	ElectricFlow “requests” block in XML for issuing one or more inline requests. Responses are fed to the target URL as post data. See the “Inline requests” section below for more details.

Note: `urlLoader` does ***not*** modify page results shown to the user. In particular, URLs for images or links are evaluated relative to the plugin page. Absolute URLs that do not include the server name will resolve to the ElectricFlow web server. In these cases, you will have broken links.

Inline Requests

If your CGI script requires ElectricFlow data to perform its function, you can use one or more inline requests (specified in the `<requests>` element) in the call to `urlLoader`. This element introduces an extra round trip of communication for ElectricFlow data between the browser and web server, after retrieving the plugin page, but before calling the CGI script. This approach is useful if it is inconvenient or impossible for the CGI script to talk to the ElectricFlow server. For example, if the CGI script is written in a language other than Perl, it may be difficult for the script to communicate with ElectricFlow.

Limitations of using inline requests:

- You can make one batch only of inline requests prior to invoking your CGI script. There is no way to conditionally issue the request or parameterize the request based on a runtime state.
- If the ElectricFlow response is large and it makes it all the way to the browser, the response is sent back in the request to the CGI script as post data. A large amount of response data may encumber browser scalability limits or cause slow performance.

Benefits of using inline requests:

The batch request is issued through ElectricFlow's web framework, which caches connections to the server. Thus, unlike a CGI script that would need to open a new connection to the server to issue requests, inline requests can leverage connections already opened. Using an existing open connection can save 0.5-2 seconds in overall page-load time (approximately the time it takes to establish an SSL connection to the ElectricFlow server).

An example for retrieving all projects and resources using inline requests:

```
<requests>
  <request requestId="projects">
    <getProjects/>
  </request>
  <request requestId="resources">
    <getResources/>
  </request>
</requests>
```

Note that each request contains a `requestId` attribute. This attribute is included in responses so you can distinguish one response from another response in the CGI script.

JAR File Layout

The plugin JAR file is laid out as follows:

`plugin.jar/`

`META-INF/`

`plugin.xml` - plugin configuration file. Defines the plugin and includes the plugin name, key, version, and so on.

`project.xml` - (optional) project definition file containing procedures and properties to be set automatically during plugin installation. If project.xml does not exist, the installation creates an empty ElectricFlow project for the plugin.

`agent/` - (optional) contains files used by steps running on agents. This directory may be structured however you desire. Typically, this directory contains "bin" and "lib" sub-directories.

`htdocs/` - (optional) Apache makes the contents of this directory accessible by using `/commander/plugins/<plugin key>-<version>/...` If this is a GWT plugin, the GWT compiler outputs generated JavaScript code to the war subdirectory. Put images and CSS files in this directory if you want to refer to them in your source code.

`cgi-bin/` - Apache invokes CGI scripts in this directory using `/commander/plugins/<plugin key>-<version>/cgi-bin/mycgi.cgi` `pages/` - Apache renders plugin pages located in this directory using `/commander/pages/<plugin key or name>/pages/...`

For more information regarding individual files, see Plugin Source Layout.

Plugin Versioning

Multiple versions (as specified in the plugin.xml file) of the same plugin (for example, those plugins that have the same plugin key) can co-exist in ElectricFlow. This feature allows you to simultaneously deploy a production and development plugin version. By default, users are directed to the production version—advanced users or plugin developers could work with the development version while refining it.

A production plugin version is marked as "promoted" to indicate it is the default plugin version to invoke when a plugin URL is specified without a version number:

```
https://<webserver>/commander/pages/MyPlugin/mypage
```

For example, the ElectricFlow Project Details page contains a link for creating a procedure using the Procedure Wizard. This link refers to the promoted version of the EC-CreateProcedureWizard plugin. If you install a new version of this plugin, users will not be aware of the new version until it is promoted. However, you can access the new version by using a URL containing the version.

The plugin version is formatted as follows: `Major.Minor.Patch.BuildNumber.`

At a minimum, the plugin must specify a `Major` version number. ElectricFlow allows one plugin with a given `Major.Minor` to be installed at a time. Installing a plugin with a version that differs only in the Patch or Build numbers will replace the old version.

For example:

HelloWorld plugin version 1.2.3.4 is installed. A user attempts to install a HelloWorld plugin version 1.2.4. The HelloWorld plugin version 1.2.3.4 will be removed and replaced with version 1.2.4.

At a later date if the user installs version 1.3, version 1.2.4 will also remain installed.

Chapter 2: Plugin References

Reference Contexts

Plugins can be referenced in several ways as follows:

- You can view a plugin page by clicking a link or typing in the URL directly.
- A page definition refers to a component in its own plugin, or a component in another plugin.
- You or a component retrieves or sets a property on a plugin.

In all of these cases, the requester has the choice of accessing the currently promoted plugin version or a specific plugin version. For different contexts, one method is better than another, and you need to decide how your plugin will be referenced when you create tabs, links, or property references for your plugin.

Some guidelines for making the decision:

- URLs set in a page or component for content in its own plugin must include the plugin version in the URL. For example, a page may have a link to another page in the plugin. This link must include the plugin version. If not, if a different plugin version is promoted, the link will lead to the promoted plugin's page, not the one intended for use with *this* plugin. This is called an "internal reference".
- URLs set in a page or component for content in a different plugin should refer to the promoted plugin. If this plugin is upgraded, this plugin's URL reference will still be valid (assuming the new plugin version still contains the page in question). This is called an "external reference".
- Tab definitions typically refer to the promoted plugin because tabs are generally created as part of promoting a plugin. See Plugin Setup Scripts for information on how to manipulate tabs when promoting or demoting a plugin.
- Users who bookmark plugin pages should save a non-versioned URL so as newer plugin versions are deployed, the bookmark will not break.

Note: Use the same guidelines for property paths. For external references, do not include the version, but for internal references, include a version.

Plugin Visibility in the UI

Plugin projects and procedures belonging to those projects can be hidden or exposed in various places throughout the ElectricFlow UI, including plugin list “pickers”, procedure list “pickers”, and the Projects page.

A note about list "pickers":

In the ElectricFlow UI, when you are presented with a popup dialog box to supply a project, plugin project, or procedure name, a "picker" list is displayed as you begin to type the object name. This list is provided as a shortcut to selecting the object name you need. The picker list is limited to 20 names in alphabetical order from which to choose. As you type, the list is continually filtered, displaying 20 names closest in spelling to the one you are typing. When you see the name you need in the list picker, select it to populate the text box.

Note: Prior to ElectricCommander 4.0, list pickers were not limited to 20 names only. With ElectricCommander 4.0, the list's scroll bar was eliminated.

Plugin Projects

Plugins (plugin projects) are referenced in various picker lists available in the ElectricFlow UI and can be added to the Projects page. If you actively use one or more plugins, it could be more convenient to see and access those plugins from the Projects page, rather than the Plugin Manager page. You can control where a plugin project is visible by setting a custom property called `ec_visibility` on the plugin project.

Legal ***values*** for the `ec_visibility` property on a plugin project are:

- `all` - use this value to see the plugin project displayed in all contexts—in any project list and in any procedure or credential picker list.

  Note: Set appropriate ACLs on "exposed" plugin projects to avoid inadvertent modifications by casual users. Generally, users should interact with a plugin only by using its published pages and scripts.

- `pickListOnly` - hides the plugin project from any project list, but allow it to show in any procedure or credential picker list.
- `hidden` - hides the plugin project from any project list and all procedure or credential picker lists.

  Note: `hidden` is the default value if the `ec_visibility` property is not set.

The `ec_visibility` property can be set in the plugin's `project.xml` file or from the ElectricFlow UI.

Plugin Procedures

A plugin project can have any number of procedures just like any other project. However, a plugin project may contain procedures you do not want to expose to users (for example, utility procedures). In this case, you could hide those procedures. You can control which procedures are visible or hidden in a list picker by setting a custom property called `ec_visibility` on the procedure.

Legal values for the ec_visibility property on a plugin procedure are:

- `all` - use this value to display a procedure in all contexts.

  Note: `all` is the default value if the `ec_visibility` property is not set on a plugin's procedure.

- `stepOnly` - shows the procedure for subprocedure calls, but hides it for job configurations or schedules.
- `private` - shows the procedure in procedure pickers only when "Current" is selected.

The `ec_visibility` property can be set on a procedure in the plugin's `project.xml` file or from the ElectricFlow UI.

How to Expose a Plugin's Procedure in the Choose Step Dialog Menu

Any procedure can be registered to be available in the Choose Step dialog box menu. The contents for this dialog box come from the server property sheet `ec_customEditors/pickerStep`. To add a procedure to the dialog box , create a property in this property sheet, using the following format:

- Name - Displayed under the name column.
- Description - Displayed under the description column.
- Value - An XML block that specifies the project/procedure to call, as well as the categories where the step should appear within the popup. For example:

```
<step>
<project>Default</project>
<procedure>HelloWorld</procedure>
<category>Test</category>
</step>
```

Selecting this option creates a new step that calls the “HelloWorld” procedure in the project “Default”. If the procedure is in a plugin project, set the project to `/plugins/pluginName/project`.

The option is displayed in the “Test” category, creating it if it does not already exist. Also, the option is displayed in the “All” category, along with all other special purpose steps. The category is optional, in which case the option is displayed only in “All”. A step can be registered for any number of categories.

Referencing Plugin Scripts on an Agent

A plugin project can contain procedures that perform various operations on behalf of the plugin. These procedures can have steps that refer to scripts to run on an agent. These scripts are typically placed in a plugin "agent" subdirectory. To make plugin scripts available to an agent, you must either copy the plugin to the Agent machine or store the plugin on a file server the Agent can access.

Just as the ElectricFlow server has a setting that declares the location of all installed plugins, the Agent has a "pluginsPath" setting in `agent.conf`. For an agent to access plugin data or logic, the plugin must be installed in the directory indicated by this setting. When a step runs, a COMMANDER_PLUGINS environment variable is set with the location, so a step can refer to the variable to find the correct script. For example, if a plugin has an `agent/runme.pl` script, the step command could be

```
ec-perl $COMMANDER_PLUGINS/@PLUGIN_NAME@/agent/runme.pl
```

The @PLUGIN_NAME@ token will be expanded to the `<plugin key>-<version>` when building the plugin, so actual plugin values are set in the final jar.

If your plugin contains Perl modules for the step to use, you can set this up as follows:

- Make the step shell be ec-perl and make the step command be the step logic in Perl.
- Load the Perl module with a line like this:

```
use lib $ENV{COMMANDER_PLUGINS}.'/@PLUGIN_NAME@/agent/lib';
use MyPackage;
```

This assumes the Perl modules are in the `agent/lib` directory in the plugin.

Plugin Property Path Syntax

To make it possible to reference properties of the currently promoted plugin version, the server supports the following property path syntax:

```
/plugins/<plugin key>[-<ver>]
```

If no version is specified, the server will choose the current promoted plugin version. If only one plugin version is installed, the server will use that one.

The plugin path provides access to any property on the plugin. The project associated with the plugin is reachable using the project property, which means the following two paths are equivalent:

```
/plugins/<plugin key>-<ver>/project/projectName
```

```
/projects/<plugin key>-<ver>/projectName
```

The preferred method for an external reference is:

```
/plugins/<key>/project/...
```

because it defers the version lookup as late as possible.

In addition, a new relative path syntax is available:

```
/myPlugin
```

Using this property resolves to the plugin associated with the project that would be returned by `/myProject`.

Chapter 3: Installing, Uninstalling, or Promoting a Plugin

Installing a Plugin

A plugin can be installed using one of four methods:

- Catalog Install–Click on **Install** (next to a plugin listed in the catalog), which downloads and installs the plugin.
- File Install–Use this method to upload and install a local file from the machine running your web browser.
- URL Install–Install a plugin from a location specified by a URL. This location can be an external web server (`using http://`), or a file on the ElectricFlow server host (`using file://`)
- ectoo–ectool, the ElectricFlow command-line tool, contains a full set of commands to perform plugin tasks.

  For more information on available ectool plugin commands, see the “Plugin Management” section in the API commands online help topic.

Note: The ElectricFlow server has a configurable maximum file upload size that defaults to 50MB. Refer to the server "Maximum upload size" setting to adjust the default.

What Happens Behind the Scenes?

The server provides an API to install an available plugin:

```
installPlugin <local file or url for plugin jar> [--force <0|1|true|false>]
```

The `installPlugin` command uploads a file from the client's file system or downloads a file via URL. The server discovers the plugin key and version from the `plugin.xml` file. The server creates a project named *`<plugin key>`*`-`*`<version>`* and grants the importing user full access to the project. The operation fails if the same plugin version is already installed or if dependencies specified in plugin.xml are not satisfied.

The `--force` option replaces an existing plugin and project. The user must have modify privileges on the "Plugins" system object. If replacing an existing plugin with `--force`, the user must have modify permission on the plugin and project being replaced also.

The server looks in the jar for a `META-INF/project.xml` file and imports the contents (if present) into the plugin project. The file should contain the result of a prior relocatable project export. This mechanism allows a plugin to automatically create a set of procedures, steps, properties, and so on that can be referenced by the plugin's logic or potentially other plugins and projects.

The server extracts the `.jar` contents into a `<plugin key>-<version>` directory under a globally configurable directory where the server must have write permission. The default location is `<dataDir>/plugins`. You can specify a new directory by changing the `/server/settings/pluginsDirectory` property.

Note: Back up this directory to preserve plugins during upgrades and installs.

After ElectricFlow is installed, files in the plugins directory are owned by the ElectricFlow server. Any plugins installed afterwards are owned also by the ElectricFlow server. Default permissions are set so files are marked read-only for all users, and files executable by the owner are executable by all users. No one has modify privilege on an installed plugin in use.

A newly installed plugin will not be promoted automatically.

Promoting or Demoting a Plugin

You can specify which plugin to promote or demote by calling this API:

```
promotePlugin <plugin key>-<version> [--promoted <0|1|true|false>]
```

This API marks the plugin as the promoted version. You can re-promote any older version to revert to an older default. Specifying `false` for the promoted flag demotes the current version without promoting a different version.

To promote or demote a plugin, you must have execute privileges on the currently promoted version (if there is one) as well as the new version.

For more information, see Plugin Setup Scripts.

Note: Promoting or demoting a plugin refers to an action performed by you, the user. “Upgrading” or “downgrading” a plugin is the result of promoting or demoting a plugin.

To Install and Promote Your Plugin

To install and promote your plugin, type the following into your plugin source directory:

```
<SDK installation directory>/tools/ant/bin/ant deploy
```

Ant calls the deploy target. For this call to succeed, ectool must have an active session.

For more information about promoting your plugin, see Plugin Versioning.

Uninstalling a Plugin

For most plugins, you can uninstall the plugin by simply deleting the plugin project and the plugin directory from the server. The ElectricFlow server provides an API to perform the uninstall operation:

```
uninstallPlugin <plugin key>-<version>
```

If no version is specified, the server uninstalls the currently promoted version. If the currently promoted version is uninstalled, there is no promoted version. You must promote a different version before doing an uninstall to ensure an atomic version change.

The server deletes the plugin project and then removes the `<plugin key>-<version>` directory from the server “plugins directory”.

To delete a plugin, you must have modify permission on that plugin.

Plugin Setup Scripts

A plugin setup script allows you to programmatically perform actions to the ElectricFlow environment. For example, a setup script can copy properties and insert tabs or subtabs in the ElectricFlow UI. This functionality is exposed with a property called `ec_setup` on the plugin project.

What's in ec_setup?

The setup script is responsible for performing actions when the plugin is promoted, demoted, upgraded, or downgraded. The script is invoked prior to the transition taking place. For example, when the script is invoked as part of promoting a currently unpromoted plugin, the plugin is still in the unpromoted state.

If a plugin does not change state, the setup script is not invoked. For example, promoting a plugin that is already promoted does not invoke the setup script.

Note: References to the plugin "state" generally refer to the promoted or demoted state of the plugin.

Which Variables and Objects "live" in ec_setup?

A plugin setup script is a Perl script that is invoked by the ElectricFlow Perl module when a plugin changes state. The script is evaluated by the API client (for example, ectool) requesting the state change. The script is invoked in a context that contains several variables that tell the script what needs to happen:

- `$commander` - a handle to an ElectricFlow object that can be used for immediate read operations.
- `$batch` - a batch request object to use for all write operations.
- `$view` - an instance of ElectricFlow::View that refers to the default view. Modifications are committed when the batch request is submitted. For more information, see ElectricCommander::View API.
- `$pluginName` - the name of the plugin being promoted/demoted.
- `$promoteAction` - one of 'promote', 'demote', or ''
- `$upgradeAction` - one of 'upgrade', 'downgrade', or ''
- `$otherPluginName` - the name of the other plugin involved in an upgrade or downgrade action.

Examples:

- Plugin version 1.0 installed, promoting 1.0

  `ec_setup` for plugin version 1.0 is executed with `$promoteAction = "promote"`
- Plugin version 1.1.0 installed and promoted. Installing plugin version 1.1.1

  `ec_setup` for plugin version 1.1.1 is executed with `$promoteAction = "promote"` and `$upgradeAction = "upgrade"` and `$otherPluginName = "plugin-1.1.0"`
- Plugin version 1.0 installed and promoted. Promoting plugin version 2.0

  `ec_setup` for plugin version 2.0 is executed with `$promoteAction = "promote"` and `$upgradeAction = "upgrade"` and `$otherPluginName = "plugin-1.0"`

The following sample ec_setup script creates a "myTab" tab during promotion and removes the tab during demotion:

```
if ($promoteAction eq 'promote') {
    $view->add(["myTab"],
               { url => 'pages/somePlugin/fileYouWantToPointTo' });
} elsif ($promoteAction eq 'demote') {
    $view->remove(["myTab"]);
}
```

Typically, the setup script is stored in a file in the plugin source tree and integrated into the project.xml file as part of the build process. See Plugin Source Layout for more details.

Chapter 4: Using the ElectricFlow SDK

System Requirements

- Windows or Linux operating system
- 32- or 64-bit Java Runtime Environment (JRE) v 1.6 or later, installed, and JAVA_HOME set in your environment.
  - JRE 1.6 or later is required to use the Ant-based build system. Your ElectricFlow installation may contain a JRE that fulfills this requirement. If so, find it in the JRE directory within the ElectricFlow installation directory.
    - When Ant is invoked with a JRE, it may emit an innocuous warning because it could not locate tools.jar before performing the desired task (for example: build, deploy). To avoid the warning, install a JDK v1.6 and run Ant against it. Download the JDK from this URL: http://www.oracle.com/technetwork/java/javase/downloads/index.html
    - The easiest way to configure Ant to use a JRE or JDK is to set the PATH environment variable with an entry to the JRE/JDK bin directory, where the Java executable can be found.
- ElectricCommander 3.6 or later server accessible to you for installing and accessing plugins. This does not have to be the machine where the ElectricFlow SDK is installed.
- ElectricCommander 3.6 or later “tools” install (or more) on your development machine.
- If planning to use Eclipse, you must know where it is installed. See Eclipse Integration for more information.

Functionality

- Ability to create a base plugin from a template as a starting point for CGI or GWT plugin development
- Ability to build CGI or GWT ElectricFlow plugins
- Examples, including source code for CGI and GWT ElectricFlow plugins
- A JAR file containing the ElectricFlow SDK Java binding and associated JavaDocs

Download and Unzip the File

To download the SDK, go to https://electric-cloud.sharefile.com and browse to **Folders** > **products** > **commander** > **integrations** > **SDK**. The direct link to this folder is https://electric-cloud.sharefile.com/app/#/home/shared/fo29731f-c7fb-446f-bd6b-bcac0933c4f3.

The SDK is packaged in a zip file named "`commander-sdk-<version number>.zip`". Unzip the contents of this zip file to a directory of your choice.

After unzipping the SDK zip file, run the `configureSDK.pl` script in the `tools/scripts` directory:

```
ec-perl <SDK installation directory>/tools/scripts/configureSDK.pl
```

This script updates portions of the SDK that need to be aware of the “installation directory” with the actual installation directory path. If you move or rename the SDK directory in the future, you must re-run this script.

Unzipped, the directory structure should look similar to the following:

`CommanderSDK/`

`build/`

This directory contains scripts required to build ElectricFlow plugins.

`docs/`

This directory contains JavaDocs for the ElectricFlow SDK.

`examples/`

This directory contains source code for example plugins.

`lib/`

This directory contains GWT 2.5.0 jars and the ElectricFlow SDK Java binding (`ec-gwt.jar`).

`tools/`

This directory contains the following subdirectories you will interact with.

`ant` - this directory contains Ant 1.7.1

`scripts` - Scripts for manipulating plugins (creating, adding GWT components to plugins, setting the browser type for GWT component builds, and so on)

Creating a Plugin Project

The createPlugin.pl script in the tools/scripts directory allows you to create a new CGI or GWT plugin from a template. The usage of createPlugin.pl is:

```
createPlugin.pl <templateName> <pluginPath> <pluginAuthor> <category> <componentNam
e>
```

`<templateName>` is the name of a template in the tools/templates directory (for example, GWTTemplate or CGITemplate)

To create a CGIPlugin:

```
> ec-perl createPlugin.pl CGITemplate c:/FancyCGIPlugin JaneProgrammer Other FancyC
omponent
```

To create a GWTPlugin:

```
> ec-perl createPlugin.pl GWTTemplate c:/FancyGWTPlugin JaneProgrammer Other FancyC
omponent
```

These plugins can be built and deployed immediately.

Other Plugin Development Tools

In addition to the `createPlugin.pl` file, the ElectricFlow SDK contains several other helpful tools in the `tools/scripts` directory.

`addComponent.pl`

If you created a plugin using createPlugin.pl and a GWTTemplate, use this script to add a GWT component to that plugin.

`setBrowsers.pl`

This script configures all GWT components to build a plugin for Firefox, IE, or all browsers. Restricting browser types reduces build time and is useful during development because you will have many "code, build, test" cycles—developing and testing a plugin on one browser saves time. When your plugin functions the way you want, enable all browsers, perform a final build, then test it on all browsers.

`configureSDK.pl`

After unzipping ElectricFlow SDK, run this script to configure the examples to point to the SDK build scripts. Re-run this script to reset the examples if the SDK is moved to a different directory. See Download and Unzip the Filefor more informationDownload and Unzip the File

`configureEclipse.pl`

This script installs several Eclipse external tool launchers into an Eclipse workspace. These launchers allow you to build or deploy your plugin from Eclipse. Also, these launchers allow you to create new plugins, add components to existing plugins, and set browser types for building plugin projects. See Eclipse Integration for more details.

`login.pl`

This script uses the ElectricFlow Perl API to log into the ElectricFlow server, and it is designed to be the back-end of the Eclipse "CommanderLogin" launcher.

`util.pl`

This script supplies utility functions used by some of the other tools. Rarely would you interact with this file directly.

Plugin Source Layout

The name of your plugin project becomes the top-level directory container for your plugin files, scripts, and any other information your plugin requires. Some or all of the directories listed in the plugin source code will be part of YourPlugin directory. For example, if you are creating a CGI plugin, you do not need the GWT directories.

You can use token placeholders in your source code to denote plugin attributes that will be replaced with actual values when you build the plugin. The following list of tokens is supported:

- @PLUGIN_KEY@ - the key is the unique identifier for this plugin
- @PLUGIN_VERSION@ - the current plugin version number
- @PLUGIN_NAME@ - the plugin name is "key-version"

Source Layout

The following layout is modeled after the JAR file layout to simplify the build process. The ElectricFlow plugin build system uses the source layout to build the JAR file to create your plugin.

`YourPlugin/`

`build.xml`

Ant build script that declares the plugin key, version, and any optional, custom logic required to build the plugin.

`META-INF/`

`plugin.xml`

`project.xml` (optional) - Reminder: If `project.xml` does not exist, the installation will create an empty ElectricFlow project for the plugin.

`pages/`

All files in this directory must end in `.xml`

For example: <page>.xml

`htdocs/`

Contains supporting content for CGI scripts and plugin pages, including images and CSS files.

`cgi-bin/`

Plugin CGI scripts reside here.

`src/ecplugins/YourPlugin/client/`

For GWT based plugins, Java files reside here.

`src/ecplugins/YourPlugin/public/`

For GWT components, supporting web content such as images and CSS files reside here.

`project/`

Directory containing code snippets to embed in project.xml. See the “project directory” section below for more information.

`agent/` (optional)

Contains files used by steps running on agents. This directory may be structured however you desire. Typically, this directory contains "bin" and "lib" sub-directories.

Project Directory

In addition to creating a project.xml file in the META-INF directory, the ElectricFlow SDK build system provides an alternative mechanism that allows the project.xml to be generated from auxiliary files in the plugin source tree. This alternative may be an appealing approach in the following cases:

- You have a large amount of code or data in properties, and it would be easier to manage those in files you can manipulate in the editor of your choice. For example, you may prefer storing the Perl setup script in a file, and use this mechanism to set the plugin project ec_setup property (in the generated project.xml) with the file contents.
- You want to check in these pieces of data into your SCM system as independently versioned entities.

The project directory is an optional directory in the YourPlugin source tree that houses your auxiliary files as well as the following:

`project.xml.in`

A template for project.xml. This should be a relocatable project export, just as if you are generating a project.xml, but named project.xml.in and placed in the project directory.

This file contains all properties (with at least empty values), which will be loaded from files. The build process will populate values only for existing properties in the project—it will not add new properties.

`manifest.pl`

A Perl script containing a mapping of XPath expressions denoting an XML node in project.xml.in to file names in the project directory.

For example, if you store your setup script in project/ec_setup.pl, your manifest.pl would contain the following code:

```
@files = (['//property[propertyName="ec_setup"]/value', 'ec_setup.pl']);
```

Note that @files is an array of array-refs. Thus, if you also wanted to map data from mydata.xml to the criticalData property, the manifest.pl would look like this:

```
@files = (
    ['//property[propertyName="ec_setup"]/value', 'ec_setup.pl'],
    ['//property[propertyName="criticalData"]/value', 'mydata.xml']
);
```

Note that the XPath expressions above are fairly loose. The ec_setup expression, for example, says, "Find any XML node in the document of type 'property' that has a 'propertyName' child-node whose value is 'ec_setup'". If multiple nodes match, the behavior is undefined, so be sure the expression is specific enough to meet your needs. For example, for a top-level project property, the expression could be:

```
//project/propertySheet/property[propertyName="specificProp"]/value
```

A note of caution: A plugin source tree can have a META-INF/project.xml file or a project directory. If both exist, the build system's behavior is undefined.

Plugin Build System

Using your plugin source files, the ElectricFlow SDK provides a build system (using Ant) that produces a plugin JAR to upload to ElectricFlow.

You must use Ant to build an ElectricFlow plugin. The buildTargets.xml file used to build the plugin can be found in the build directory.

The five primary Ant build targets:

`clean` - Cleans the build output directory.

`buildJavascript` - Builds the plugin source code into JavaScript (for GWT components)

`buildStaging` - Copies generated JavaScript and various parts of the plugin source to a staging area.

`buildProject` - If you have a project directory in your source code, this target will create the final `project.xml` in the staging area.

`package` - Builds a plugin JAR file from the contents of the staging area. This target lists earlier targets as dependencies, so invoking this target actually does a clean plugin build.

Other available Ant targets:

`uninstallPlugin` - Uninstalls the plugin.

`installPlugin` - Installs the built plugin JAR to the ElectricFlow server—uninstalls the plugin first.

`promotePlugin` - Promotes the plugin.

`build` - an alias for the 'package' target

`deploy` - a convenience target that effectively invokes the 'installPlugin' and 'promotePlugin' targets

In addition, to customize the build, each of these targets has corresponding "pre" and "post" targets that can be defined in a plugin's build.xml to inject logic between build phases. For example, defining `buildStaging.post` causes its logic to run ***after*** generated JavaScript code and various parts of the source tree are copied to the staging area, but ***before*** building the JAR.

Also, using “pre” or “post” is useful when running one high-level target. For example, if some out-of-band data should be cleaned up for a particular plugin when the `clean` target is invoked, the build.xml can specify a `clean.pre` or `clean.post`. Now whenever you invoke the `clean` target, it will do the general clean as well as a plugin-specific clean.

Building Your Plugin

To build any plugin, the COMMANDER_HOME environment variable must be set to the ElectricFlow install directory. The default install directory:

- on Windows - `C:\Program Files\Electric Cloud\ElectricCommander`
- on Linux - `/opt/electriccloud/electriccommander`

To build your plugin, type the following into your plugin source directory:

```
<SDK installation directory>/tools/ant/bin/ant build
```

Ant calls the build target. As part of the build process, Ant creates an "out" directory in your plugin source directory. In the out directory, Ant creates the staging directory and the plugin JAR file.

- If the plugin source directory contains a project directory, the `buildProject.pl` script is called. The script generates a `project.xml` file in the staging area from the contents of the project directory in the source tree.
- The GWT compiler is called if there is a `src` directory in the plugin source directory. The JavaScript output is put in an `htdocs` directory in the staging directory.
- The final build phase takes the staging directory contents and puts it in a JAR file. This is the plugin JAR file you install in ElectricFlow.

Building and Deploying Your Plugin

These Ant commands can be chained, so you can type the following in your plugin source directory:

```
<SDK installation directory>/tools/ant/bin/ant build deploy
```

In addition, the build environment contains Ant properties you can set on the command-line (using the Ant -D option) in a plugin's build.xml (using a <property> element) to alter build behavior. The most common are:

Property Name	Description	Default Value
`commander.server`	ElectricFlow server host name, for uninstalling and installing plugins	The value of the COMMANDER_SERVER environment variable. If this environment variable does not exist, default is "localhost"
`ectool`	Path to ectool executable	"ectool" - location is obtained by path lookup
`perl`	Path to ectool executable	"ec-perl" - location is obtained by path lookup
`gwt.localWorkers`	The number of local workers to use when compiling permutations	1
`gwt.style`	(explained below)	"OBF", meaning obfuscated

By default, GWT obfuscates the JavaScript it produces, which means the generated output will be compressed. By overriding the `gwt.style` Ant property, we are telling the GWT compiler to provide detailed JavaScript.

This (`gwt.style`) flag has one of three possible values:

- OBF (for obfuscated), the default, results in the smallest amount of JavaScript code for faster page loads—we recommend using this setting for plugins deployed in production.

- PRETTY, makes the output humanly readable
- DETAILED, improves on PRETTY with more detail (such as very verbose variable names)

For example, you may override `gwt.style` on the command-line using:

```
<SDK installation directory>/tools/ant/bin/ant -Dgwt.style="DETAILED" build
```

For more build configuration options, see buildTargets.xml.

Chapter 5: Eclipse Integration

The ElectricFlow SDK includes an integration with the Eclipse IDE in the form of External Tool launchers. These launchers were tested on Eclipse Galileo and Helios. If you are a Plugin-in-a-box user, see Converting SDK Versions for details on converting the Plugin-in-a-box workspace to work with CommanderSDK 1.0 and later.

Note: Using Eclipse is not required, however, if you plan to use Eclipse, download it from the Eclipse web site and install it now.

Importing SDK Examples into Eclipse

To import all GWT plugin examples from the ElectricFlow SDK into Eclipse as projects, use the following process:

- Go to **File** > **Import**.
- Select **General / Existing Projects into Workspace** and then click **Next**.
- For "root directory", select **Browse** and choose the examples directory, click **OK**.
- You should see a “selected” projects list—click **Finish** to import them.

  Note: If projects are not listed, it probably means you did not run `configureSDK.pl` after unzipping the SDK. Run `configureSDK.pl`, then try importing the projects again.

Installing Launchers

Install the launchers by invoking the `configureEclipse.pl` script from the tools/scripts directory:

```
ec-perl <SDK installation directory>/tools/scripts/configureEclipse.pl <eclipseWork
spaceDir> [commanderInstallDir]
```

Finding and Setting the Eclipse Workspace

The Eclipse workspace is the directory containing the .metadata and typically various project directories. For a clean Eclipse installation, the default is typically to a location in the user's home directory (on Linux) or user profile directory (on Windows). The workspace directory can be set from the command-line when launching Eclipse with the `-data` option:

```
eclipse -data c:\workspace
```

Finding and Setting the ElectricFlow Installation Directory

`commanderInstallDir` is an optional argument to `configureEclipse.pl`. This script finds the ElectricFlow installation if it is installed in your platform's default location. If the script cannot find the ElectricFlow installation, it produces an error and you will need to provide the location.

Setting Favorite Launchers

After `configureEclipse.pl` completes, restart Eclipse. The launchers are available for use under Run > External Tools > External Tools Configurations... To make the launchers more conveniently available under the external tool launcher button (the green arrow button with a suit-case), set these launchers as Favorites:

1. Go to Run > External Tools > Organize Favorites...
2. Click Add... to add the installed launchers to the Favorites list.
3. Move the launchers up or down to set the order as desired. Typically, users put BuildAndDeployPlugin at the top of the list because it is the most common operation.

Using the launchers

Two types of external tool launchers are defined in Eclipse:

- Ant launchers - Invoke Ant targets on the build.xml file in a plugin project.
- Program launchers - Invoke scripts in the SDK tools/scripts directory to manipulate plugin projects.

All launcher output is displayed in a Console tab.

Avoiding a typical Eclipse error message

Many launchers rely on a project being selected in Package Explorer. If no project is selected, or the Package Explorer does not have focus, Eclipse emits an error message with the text "Variable references empty selection." If this occurs, select the desired project in Package Explorer and retry.

Ant Launchers

BuildPlugin

This launcher cleans and builds the selected plugin project. Ant output is shown in a console window.

DeployPlugin

This launcher deploys a previously built plugin to an ElectricFlow server. "Deploying" involves uninstalling the currently installed plugin if present, then installing the new plugin and promoting that plugin.

By default, this launcher deploys the plugin to the server specified in the COMMANDER_SERVER environment variable, if present. If this environment variable is not present, Ant deploys the plugin to localhost. To override this default, either set the COMMANDER_SERVER environment variable and restart Eclipse, or update the launcher definition as follows:

1. Select a plugin project in Package Explorer.

   Note: It is critical to select a plugin project ***before*** going to External Tools Configurations. If a plugin project is not selected, you may corrupt the launcher.

2. Go to Run > External Tools > External Tools Configurations…
3. Select the relevant Ant launcher.
4. Select the Properties tab.

5. Edit the value of the `commander.server` property.
6. Click Apply to save the change.

BuildAndDeployPlugin

This launcher cleans and builds the selected project, then installs and promotes the plugin to an ElectricFlow server. By default, this launcher deploys the plugin to the server specified in the COMMANDER_SERVER environment variable if present. If this environment variable is not present, Ant deploys the plugin to localhost. To override this default, either set the COMMANDER_SERVER environment variable and restart Eclipse or update the launcher definition. See the instructions in the “DeployPlugin” section above for more details.

CleanPlugin

This launcher cleans the selected project's output directory.

Program Launchers

CreatePlugin

The launcher invokes `createPlugin.pl` to create a new plugin, with the template argument set to "GWTTemplate". The launcher prompts for the plugin name and sets the plugin path to the Eclipse workspace, appended with the plugin name. After the script completes, you can import the new plugin project:

- Go to File > Import…
- Select General / Existing Projects into Workspace, click **Next**.
- For "root directory", click **Browse**.
- Choose the Eclipse workspace directory and click **OK**.
- Make sure your new plugin project is selected and click **Finish**.

AddComponent

This launcher invokes the `addComponent.pl` for the selected project so you can add a new GWT component. After it completes, you can see the generated source in `src/ecplugins.<`*`YourPluginName`*`>.client` in Package Explorer. If not, select the project in Package Explorer and press the F5 key to refresh Package Explorer.

SetBrowsers

This launcher invokes `setBrowsers.pl` for the selected project so you can set browser types for GWT component builds.

CommanderLogin

This launcher invokes `login.pl`, using the COMMANDER_SERVER environment variable value as the server to log into. If the environment variable is not set, the script will try to log into localhost. To modify this behavior, edit the “Arguments” section of the CommanderLogin external tool launcher.

Handling Multiple ElectricFlow Servers

If you need to switch between multiple ElectricFlow servers when deploying plugins, the techniques outlined earlier for setting the ElectricFlow server are impractical. A better solution for this case is to leverage Eclipse's `string_prompt` macro in relevant launcher definitions. This macro causes the launcher to prompt you for the ElectricFlow server name and IP address at launch time. For the BuildAndDeployPlugin and DeployPlugin launchers, set the `commander.server` Ant property to have this value:

```
${string_prompt:Commander server:localhost}
```

This value causes Eclipse to prompt you with "Commander server", a text entry box pre-populated with "localhost". Update the default value in the `string_prompt` for your requirements.

For the CommanderLogin launcher, update the Arguments section with the following:

```
login.pl "${string_prompt:Commander server:localhost}" "${string_prompt:Commander u
ser-name}"
```

Refreshing Corrupt Launchers

Occasionally, Eclipse launchers for building and deploying plugins become corrupted. The primary corrupt symptom is that launchers simply do not perform the operation you are requesting. For example, the BuildAndDeployPlugin gets into a state where it builds, but does not deploy.

To refresh the launchers, run the `configureEclipse.pl` script again, and restart Eclipse. If, however, you have made changes to launchers in the past and would like to preserve those changes across a refresh, back up your launcher changes to the SDK `tools/launchers` directory. The `configureEclipse.pl` file refreshes launchers from files stored in this directory.

Chapter 6: Examples and Tutorials

ElectricFlow plugins are either codeless or written in CGI and/or GWT. The source code for these examples is available in the examples directory in the ElectricFlow SDK.

Note: Make sure you run the "CommanderLogin" launcher ***before*** building any of the examples.

Codeless and CGI Examples

Codeless Example Overview

This plugin example provides a minimal plugin without using CGI or GWT, which illustrates the idea that plugins do not need to contain source code.

Relevant Files

Developing a Codeless plugin can be as simple as “jar'ing up” two files.

`pages/Codeless.xml`

```
<componentContainer>
  <table width="100%">
    <tr>
      <td valign="top" width="50%">
        <component plugin="EC-Core" ref="urlLoader">
          <url>http://www.electric-cloud.com</url>
        </component>
      </td>
    </tr>
  </table>
</componentContainer>
```

`META-INF/plugin.xml`

```
<?xml version="1.0"?>

<plugin>
  <key>Codeless</key>
  <version>1.0</version>
  <label>CodelessExample</label>
  <description>Displays the Default Project</description>
  <vendor>Electric Cloud</vendor>
  <depends>EC-Core</depends>
</plugin>
```

This simple plugin pulls information from a 3rd party website (in this case, http://www.electric-cloud.com) and displays the information in ElectricFlow.

Use the following URL to access the output for this plugin:
https://<webServer>/commander/pages/CodelessExample/codeless

HelloWorldCGIExample Overview

This plugin example provides:

- A simple plugin
- A new tab in ElectricFlow and linking the tab to this plugin
- Calling a CGI script and displaying the output on screen
- Bundling and showing an image on screen

Relevant Files

`cgi-bin/helloworld.cgi`

This CGI script shows how to obtain a reference to the ElectricFlow server. The script then simply prints out "Hello World".

`META-INF/plugin.xml`

Defines the plugin, including the name, version, and so on.

`pages/helloworld_run.xml`

Defines a page that references the `helloworld.cgi` script.

`pages/help.xml`

Defines the plugin Help page. This must be proper XHTML, although like any other page, it can contain references to components. This page is denoted as the plugin's help page in the `plugin.xml <help>` element.

`htdocs/electric-cloud_logo-sm.gif`

This image is shown in the helloworld_run page. The image is the Electric Cloud logo.

What Should I See?

After building and deploying the plugin using Ant, select the Administration tab in the ElectricFlow UI, then select the Plugins tab. You will see a row with "Hello World" that has a Help link. The Help link points to `help.xml`. On this page you can promote the plugin also - click the promote link.

Adding the Hello World Tab

1. Select the Administration / Server tabs
2. Under Custom Server Properties, select 'ec_ui', then select availableViews.
   Click the Create Property link and enter the following XML snippet:

```
<view>
  <base>Default</base>
    <tab>
      <label>Hello World</label>
      <url>pages/HelloWorldCGIExample/helloworld_run</url>
    </tab>
</view>
```

3. Logout, then login again to see your changes.
4. After logging in, click on your name in the upper-right portion of the screen.
   Click Edit Settings.
5. Under the Tab View, choose your Hello World View.
   Your new tab will be displayed now.

   Click the Hello World tab to see “Hello World” displayed.

For more information, see the ElectricFlow online help topic, Customizing the ElectricFlow UI, “Custom Tabs” section. This section includes a view definition syntax, and information for adding, changing, or deleting tabs.

JobsLite Example

This plugin example provides the following:

- How to construct a page that displays job-related information
- How to create a new tab in the ElectricFlow UI and link the tab to this plugin
- How a CGI script issues ElectricFlow requests, processes the results, and returns result XHTML to display in a plugin page
- How a CGI script can use styles defined in CSS files for the ElectricFlow UI

Relevant Files

`cgi-bin/jobslitedashboard.cgi`

This CGI script issues an ElectricFlow findObjects request, processes the response, and formats and displays the results.

`META-INF/plugin.xml`

Defines the plugin, including the name, version, and so on.

`pages/jobslitedashboard_run.xml`

Defines a page that references the `jobslite.cgi` script and uses CSS files.

`project/`

`ec_setup.pl`

Contains the logic that creates a "JobsLite" tab during plugin promotion and removes the tab during plugin demotion.

`manifest.pl`

Maps the "ec_setup" property to the contents of ec_setup.pl.

`project.xml.in`

Defines the "ec_setup" property under the current plugin project.

What Should I See?

After building and deploying the plugin, using Ant, logout of any existing ElectricFlow web browsing sessions and then login to see the new tab—select the tab to see the new product information page.

Some GWT Basic Information

Review this basic GWT information section before previewing the GWT examples in the section that follows.

GWT provides an infrastructure and set of widgets that allows application code running inside a browser to make requests to web servers and process the responses. Developers write Java code, importing GWT packages. But the compiled Java byte code is used only when running in development mode for debugging—it does not run in the browser.

GWT provides its own compiler to translate Java code into JavaScript (that can run in various browsers). GWT also supports internationalization, so for each browser GWT can generate JavaScript for every language the application supports. Rather than generating one large JavaScript file containing conditional logic based on which browser and language the user has for every component, the GWT compiler builds a unique JavaScript file for every permutation of browser and language. This means compilation can take quite some time, but the end result is code that works anywhere.

Another side-effect of GWT's architecture is that not all Java code can be translated to JavaScript. Specifically, most of the language itself can be translated, but arbitrary classes cannot be translated to JavaScript. Details can be found at http://code.google.com/webtoolkit/doc/latest/DevGuideCodingBasicsCompatibility.html. Thus, when you try to build the plugin, the GWT compiler may display errors. In many cases, alternative classes/methods can be used to get translatable code.

Finally, GWT has an event model. For the purpose of page processing, browsers are single-threaded but event-driven. This means that as a page loads, it loads artifacts from multiple URLs simultaneously, but one thread cycles through all in-progress requests: for each in-progress request, check if response data is available. If so, process the data. If not, go to the next in-progress request. When the end is reached, return to the beginning. The main point is that when the browser issues a request to the web server, it does not halt all other activities until it receives a response.

When writing a GWT-based ElectricFlow plugin, you are exposed to this: whenever you make a request, set a callback to handle the response. In handling the response, you can alter the UI or make other requests (with more response handlers to handle *those* responses).

An additional consequence of the event model is that your code needs to relinquish control so the event loop can continue processing events. If you have a long-running loop in your code, essentially “blocking” until some condition is satisfied, the event loop will be stalled. Write your code in a way that you do not block—register a callback to invoke when the desired condition is satisfied.

What is the “Same Origin Policy”?

Generally, browsers implement a security model known as the Same Origin Policy (SOP). Conceptually, it is a simple model, but the limitations it applies to JavaScript applications can be quite subtle.

Simply stated, the SOP states that JavaScript code running on a web page may not interact with any resource not originating from the same web site. The reason this security policy exists is to prevent malicious web coders from creating pages that steal web users' information or compromise their privacy. While very necessary, this policy can have the side effect of making a web developer’s life difficult.

It is important to note that SOP issues described here are not specific to GWT—these issues are true for any AJAX application or framework. So what does this mean if you are using GWT to develop ElectricFlow plugins? If you want to request data from another server, you need to use CGI.

Speeding Up Builds

As was mentioned earlier, the GWT compiler can take quite a long time to build components. One optimization encoded in the `buildTargets.xml` file is concurrent invocations of the GWT compiler for all components. So if it would take 2 minutes to build four components serially, that time is reduced to about 30+ seconds, but even 30 seconds is a long time for the edit-build-deploy-test cycle.

One way to speed up an individual component compile is to limit browser support. This reduces the number of permutations tremendously. The `setBrowsers.pl` script makes it easy to modify the `*.gwt.xml` files to limit browser support.

For more detailed information, see the componentName.gwt.xml file for any component of any example plugin project, and Other Plugin Development Tools.

GWT Plugin Examples

The next five plugin examples described are GWT-based.

Note: Review, try, or become familiar with each of the next five plugins in the order they are presented. The second example assumes you are already familiar with the first example, and the third example assumes you are familiar with the previous two examples, and so on.

HelloWorldExample overview

This example provides a minimal plugin and GWT component that contains fundamental mechanics for developing a plugin. This example also demonstrates where to define a cascading style sheet (CSS) and how to use it during plugin development.

Relevant Files

`build.xml`

Ant build file with information about this plugin. All real work is done by the `buildTargets.xml` ant file.

`META-INF/plugin.xml`

Defines the plugin, including name, version, and so on. The critical pieces of information are the plugin component list and the JavaScript file implementing each component. Whenever a component is added to the plugin, this file must be updated. The `addComponent.pl` script handles this update.

`pages/help.xml`

Defines the plugin help page, which must be proper XHTML, although like any other page, it can contain references to components. This page is denoted as the plugin's help page in the `<help>` element in `plugin.xml`.

`pages/HelloWorldComp_run.xml`

Defines a page that references the `HelloWorldComp`. In a “real-world” plugin, this file can have multiple component references, complex HTML layout, and so on.

`src/ecplugins/HelloWorldExample/client/custom.css`

Defines a style to be used in the GWT java code. All CSS files intended for use by your GWT-based plugin need to reside in the same public directory.

`src/ecplugins/HelloWorldExample/client/HelloWorldResources.java`

This interface extends `ClientBundle` and houses the `CssResource`. The link between Java/GWT and the actual CSS file is defined in this file.

`src/ecplugins/HelloWorldExample/client/HelloWorldStyles.java`

This interface extends `CssResource` and contains functions that correspond to CSS classes defined in the actual CSS file. In this example, `custom.css` is the CSS file.

`src/ecplugins/HelloWorldExample/client/HelloWorldComp.java`

Defines a component class. A component is typically the core of a UI page, with logic for performing all sorts of actions based on user-input. Such logic includes dynamically altering the UI based on input in other parts of the UI and issuing background requests to the server and processing the results. In this trivial example, the component just creates a caption-panel UI widget and applies a CSS style to text.

`src/ecplugins/HelloWorldExample/client/HelloWorldCompFactory.java`

Defines the factory class that provides component class instances. This is a boiler-plate file and rarely requires modification.

`src/ecplugins/HelloWorldExample/HelloWorldComp.gwt.xml`

GWT build spec for this component. The GWT compiler uses this file to generate component JavaScript files that ultimately run in the browser. All CSS files used by a GWT component must be explicitly referenced here. This file is typically modified to specify internationalization information or to limit browser support to gain faster compiles during plugin development.

What Should I See?

After building and deploying the plugin using ant, in the ElectricFlow UI go to the Administration tab > Plugins. You should see a row for "HelloWorldExample" along with a Help link. That link points to the `help.xml` page, as specified in the `<help>` element in `plugin.xml`.

The help page contains text and a link to the HelloWorldComp Run page. Whenever components and pages are added to the plugin using the `addComponent.pl` tool, a link to the newly generated page is added to the plugin help.

The HelloWorldComp Run page should have a page title "HelloWorldExample 1.0 HelloWorldComp Title" (see the browser title bar) and some text with the plugin name, version, and component name. The page should also display, in red text, "This text has a CSS style applied". The ElectricFlow Help link at the top-right part of the window should take you to the plugin help page, as specified in the `<helpLink>` element in `HelloWorldComp_run.xml`.

ConfigureExample Overview

This plugin example shows how a plugin configuration page/component could be set up. For simplicity, the help page was not updated to reflect what the plugin and component do.

Relevant Files

See the previous “HelloWorldExample” for a list of relevant files for this plugin. Basically, relevant files are the same for this GWT example, but note that file names are different to correspond to the example you are viewing.

How This Plugin was Created

This plugin was developed using the tools provided in the CommanderSDK as follows:

1. Used the `createPlugin.pl` script to create the ConfigureExample plugin with the ConfigComponent component and ConfigComponent_run.xml page.
2. Edited `plugin.xml` to specify that `ConfigComponent_run.xml` is the configure page for the plugin.
3. Edited `ConfigComponent.java` to create two text boxes linked to properties in the plugin project.
4. Edited `ConfigComponent_run.xml` to update the page title.

If you were actually going to use this plugin, you would probably have more widgets and logic in the component, and help pages would be filled with relevant content.

All other files in the plugin are either boiler-plate or already set up properly by the `createPlugin.pl` script and do not need to be modified.

The point of this example: A “configure” page is like any other page except that it is declared as the configure page in `plugin.xml`.

What Should I See?

After building and deploying the plugin using Ant, go to the **Administration** > **Plugins** tab in the ElectricFlow platform UI. You should see a row for "ConfigureExample" along with Configure and Help links. The Configure link points to the `ConfigComponent_run.xml` page, as specified in the `<configure>` element in `plugin.xml`. The Help link points to `help.xml`, like the HelloWorldExample.

The help page contains some text and a link to the `ConfigComponent_run` page, like the HelloWorldExample.

The `ConfigComponent_run` page should have a page title "ConfigureExample 1.0 ConfigComponent Title" and a form with two inputs: “Build Counter” and “Very Long Something Or Other”. Vastly different label sizes were chosen to show how a FormTable widget lays out its contents in this situation. The Build Counter field is restricted to positive integers. If invalid data is entered, pressing the OK button produces an error message and the keyboard focus is set in the Build Counter field. After successful input, the OK button is disabled and its label changes to "Saving...". When the form values are saved in the project, the page redirects back to the ElectricFlow Administration > Plugins web page.

Verify your values were saved by clicking the Configure link again. Previously saved values are loaded into the form before displaying it. Also, you can click on the plugin name in the Plugins list to go to the plugin project and see custom properties that were saved there. Incidentally, this form could have manipulated intrinsic properties just as easily as custom properties, and barring access control restrictions, it could have manipulated any object in the system. Thus, you can construct a form backed by any number of ElectricFlow objects.

RequestResponseExample Overview

This plugin example shows how components can interact with the ElectricFlow server and plugin CGI scripts.

The RequestResponseComponent includes several examples showing how to issue requests to the server for various scenarios. We strongly recommend that you walk through the examples in the order they are presented. Comments in later functions assume familiarity with earlier functions.

The CGIRequestComponent shows how to hit CGI scripts that are part of our plugin in both GET and POST modes and process the responses. This is particularly useful because creating the UI often involves talking to legacy systems and processing a large amount of data. Instead of doing the work in the browser, it may be faster to use a CGI script that does all the “heavy lifting” and simply presents relevant data to the browser. The GWT component can then render data as appropriate.

How This Plugin was Created

This plugin was developed using the tools provided in the ElectricFlow SDK as follows:

1. Used the `createPlugin.pl` script to create the RequestResponseExample plugin with RequestResponseComponent component and RequestResponseComponent_run.xml page.
2. Edited `RequestResponseComponent.java` to create several panels, each responsible for running a method that issues requests.
3. Used the `addComponent.pl` external tool to create the CGIRequestComponent component, and then modified the component.
4. Created a `cgi-bin` directory to contain the printenv and copycat CGI scripts. These scripts are packaged automatically when building the plugin.

All other files in the plugin are either boiler-plate or already set up properly by the `createPlugin.pl` script and `addComponent.pl` external tools and do not need to be modified.

What Should I See?

After building and deploying the plugin using Ant, go to the **Administration** > **Plugins** tab in the ElectricFlow platform UI. You should see a row for "RequestResponseExample" along with a Help link.

The help page contains some text and links to the RequestReponseComponent_run and CGIRequestComponent_run pages.

The RequestReponseComponent_run page shows several sections, each with buttons for running a particular type of request and clearing results. Follow along with the code and run queries to gain a deeper understanding of the new request mechanism.

The CGIRequestComponent_run page shows the result of hitting the printenv CGI with a GET request and the copycat CGI with a POST request.

RunProcedureExample Overview

Note: The following example remains relevant for ElectricCommander versions prior to ElectricCommander 4.0. For ElectricCommander 4.0 and later, see ExampleParameterPanel Overview.

This plugin shows how a custom Run Procedure page/component could be set up. For simplicity, the help page was not been updated to describe actions for the plugin and component.

Note: This plugin does not handle "Run Again" or running a job configuration defined in a homepage property. The plugin page always renders as if it is a "Run..." request.

How This Plugin was Created

This plugin was developed using the tools provided in the ElectricFlow SDK as follows:

1. 1. Used the `createPlugin.pl` script to create the RunProcedureExample plugin with BuildProcedure component and BuildProcedure_run.xml page.
2. Edited `plugin.xml` to define a CustomRun custom-type to cause the Run Procedure link to redirect to the BuildProcedure_run page for any procedure that has a `customType` property with a value of `$[/plugins/RunProcedureExample]/CustomRun`.
3. Edited `BuildProcedure.java` to create the custom page.

In addition, a sample project export file containing a procedure with the `customType` property was created, and a new target was added to `build.xml` to import the export file into ElectricFlow.

All other files in the plugin are either boiler-plate or are set up properly by the CreatePlugin external tool and do not need to be modified.

Note: A Run Procedure page is like any other page except that it is declared in a custom-type in `plugin.xml`.

What Should I See?

Build and deploy the plugin using Ant. Install the BuildSystem project using the “setup” target, then go to the **Administration** > **Plugins** tab in the ElectricFlow platform UI. You should see a row for "RunProcedureExample".

Go to the BuildSystem project and run the BuildProcedure procedure. The custom Run Procedure page should pop up.

Clicking the Run button results in the component putting together the run-procedure request, submitting it, getting the result job ID, and redirecting to the Job Details page for that job ID.

This Component Demonstrates a Few Important Concepts

- Debug logging

  If you add a `debug=1 get-arg` to the page request (add "`&debug=1`" to the end of the URL), you will see extra logging in a panel near the bottom of the page. This technique allows the plugin developer to collect debug info about the plugin on demand if problems arise, but in a way that does not get in the end user's way normally.

- Using ECFormPanel

  The SDK contains some complex UI objects provide certain UI functionality out-of-the-box. Thus, you write less code to get a functional plugin.

- Structuring a component whose UI elements depend on one another

  In this example, specifying a “branch name” results in calling a CGI script that returns a list of valid values for “release name”.

What's Next?

To see more examples of GWT widgets, see the GWT Showcase. For more sophisticated examples, see the source code for plugins bundled with ElectricFlow or any recently developed plugin from an ElectricFlow Sales Engineer.

GWT Parameter Panels

Note: ElectricCommander v4.0 is required to use this feature.

Two ways to customize parameter presentation for the ElectricFlow web interface:

1. Create an XML document to reorder parameters or set display labels that are different from the parameter names. This process is documented in the "Customizing the ElectricFlow UI" help topic in ElectricFlow online help.
2. Replace the default parameter panel with a customized GWT parameter panel. The parameter panel interface in the ElectricFlow SDK allows you to render different types of form elements, add custom validation code, make multiple additional ElectricFlow requests, create interdependent form fields, and so on. The following section contains instructions for creating a GWT parameter panel. For an example, see the ExampleParameterPanel overview.

Using one of these methods, you will see parameters anytime a user interacts with a customized procedure or state definition in the ElectricFlow web interface. For example, a customized parameter panel for a procedure is displayed when you create a step that calls the procedure as a subprocedure, create a schedule that calls the procedure, run the procedure directly, and so on.

How to Add a GWT Parameter Panel to your Plugin

Note: You need to create a procedure or state definition if it does not already exist in your plugin.

Create a New Component

Create a class that extends ComponentBase and implements ParameterPanel, ParameterPanelProvider, and HasInitCallback. This class includes all of the parameter panel functionality. Refer to the Java Doc on the respective interfaces for details about each of the methods you need to implement.

If you use a standard plugin structure, this class is located in:

```
src/ecplugins/YOUR_PLUGIN_NAME/client
```

Create a New Factory to Return This Component

Create a class that extends ComponentBaseFactory. You need to implement `createComponent`, which will return an instance of your new component.

If you use a standard plugin structure, this class is located in:

```
src/ecplugins/YOUR_PLUGIN_NAME/client
```

Create a GWT XML File

Note: When creating a parameter panel, the GWT XML filename must end with “ParameterPanel”. For example, the filename might be `newXParameterPanel.xml`.

Create a GWT XML file to specify the entry point into the factory. Here is an example of the contents for this file if you are using a standard plugin structure:

```
<module>
  <entry-point class="ecplugins.YOUR_PLUGIN_NAME.client.YOUR_FACTORY_NAME"/>
  <inherits name="com.electriccloud.commander.gwt.ComponentBase"/>
</module>
```

If you use a standard plugin structure, this class is located in:

```
src/ecplugins/YOUR_PLUGIN_NAME
```

Create a Custom Type in Your Plugin XML File

Note: The `customType` name concatenated with ParameterPanel must equal the GWT module name. For example, if your `customType` name is “`foo`”, your GWT module name must be “`fooParameterPanel`”.

In the <customTypes> section of your plugin.xml file, add a new entry to reference the parameter panel. A `<customType>` element example if you are using a standard plugin structure:

```
<customType name="YOUR_CUSTOM_TYPE_NAME">
  <parameterPanel>
    <javascript>war/ecplugins.YOUR_PLUGIN_NAME.YOUR_COMPONENT_CLASS_NAME/
    ecplugins.YOUR_PLUGIN_NAME.YOUR_COMPONENT_CLASS_NAME.nocache.js
    </javascript>
  </parameterPanel>
</customType>
```

Link the Parameter Panel to the Procedure or State Definition

The ElectricFlow web interface loads a parameter panel if the procedure or state definition references the panel. To do this, create a top-level property called `customType` whose value is:

```
YOUR_PLUGIN_NAME/YOUR_CUSTOM_TYPE_NAME
```

Remember: This procedure or state definition needs to have the same formal parameters the parameter panel is expecting. You will encounter an error if the panel returns values that do not match the formal parameters.

Register Your Procedure for the Step Creation Dialog Box

Note: Only register your procedure if you want it displayed as an option when a user clicks the Create Step link.

If you do not already have an `ec_setup.pl` script for your plugin, create one (see the "Plugin source layout" section for details).

In your `ec_setup.pl` script, populate the `@::createStepPickerSteps` array with information for all procedures in the plugin that you would like to show up in the step creation dialog box.

The following example creates two entries in the "Custom" category:

```
# Data that drives the step creation dialog registration.
my %build = (
    label       => "Build ABC",
    procedure   => "Build",
    description => "Check out and build product ABC.",
    category    => "Custom"
);
my %test = (
    label       => "Test ABC",
    procedure   => "Test",
    description => "Test a built version of product ABC.",
    category    => "Custom"
);
@::createStepPickerSteps = (\%build, \%test);
```

ExampleParameterPanel Overview

Note: ElectricCommander v4.0 is required to use this feature.

This plugin shows a GWT component in a parameter section. The GWT component in this example performs an ElectricFlow request, validates user input, and dynamically disables input fields based on previous input fields.

How This Plugin was Created

This plugin was developed using the tools provided in the ElectricFlow SDK as follows:

1. Used the `createPlugin.pl` script to create the ExampleParameterPanel plugin with ExampleParameterPanel component and ExampleParameterPanel_run.xml page.
2. Edited `ExampleParameterPanel.java` to implement ParameterPanel, ParameterPanelProvider, and HasInitCallback.
3. Added a `customType` property to the `plugin.xml` file.
4. Added a `hash to ec_setup.pl` that allows the procedure defined in this plugin to be displayed in the Create Step picker dialog box.
5. Added a reference to `ec_setup` in a manifest.pl file.
6. Added a procedure with formal parameters to `project.xml.in`.

All other files in the plugin are either boiler-plate or already set up.

What Should I See?

After building and deploying the plugin using Ant, go to a project—you can use the Default project.

1. Create a procedure named "test" and click OK.
2. Click the Create Step link and choose the Example tab.
3. Double-click on the line where the name is "Example Procedure using Parameter Panels."

The Parameters section is the GWT plugin component rendered on screen.

You should see an Enable Windows checkbox, a Windows Configuration text area, and a Resource dropdown menu.

The Enable Windows checkbox determines if the Windows Configuration text area is enabled or disabled. The Resource dropdown menu should be populated with previously defined resources for ElectricFlow.

Running the step outputs text entered in the Windows Configuration text area.

Chapter 7: Debugging your GWT Plugin

Debugging Google Web Toolkit (GWT) Plugins

If you have developed a plugin for ElectricFlow, using the ec-gwt.jar file, and run into a problem, you can debug your code. This chapter is a step by step guide to debugging your ElectricFlow plugin, using Eclipse. This process takes you to the point where you can set and hit breakpoints in your Java GWT code.

Notes:
1. The screen captures in this chapter are specific to GWT 2.5.0. If you are using a different GWT version, your screens will look slightly different.
2. How you debug plugins developed with the ElectricFlow SDK Java binding in GWT Development Mode (debugger) is dependent on your ElectricFlow server version andElectricFlow SDK version. To debug a plugin developed with CommanderSDK 2.0, deploy the plugin to an ElectricCommander 4.2 server. If you are using a newer ElectricFlow SDK version and plan to use the debugger, you may need to upgrade your ElectricFlow server.

This document assumes you have:

- ElectricCommander 3.6.x or later, installed

  Note: If you plan on using the debugger for plugins developed with CommanderSDK 2.0, you must be using Commander 4.2.

- CommanderSDK 1.0 or later

Your development machine must include:

- CommanderSDK 2.0 or later downloaded and installed on your machine
- Internet access and the ability to install add-ons to your web browser (IE 7.0 or later, Firefox 10.0 or later)
- Java Runtime Environment (JRE) installed
- An ElectricFlow Tools installation
- Access to an ElectricFlow server
- The ability to build and deploy plugins to your ElectricFlow server

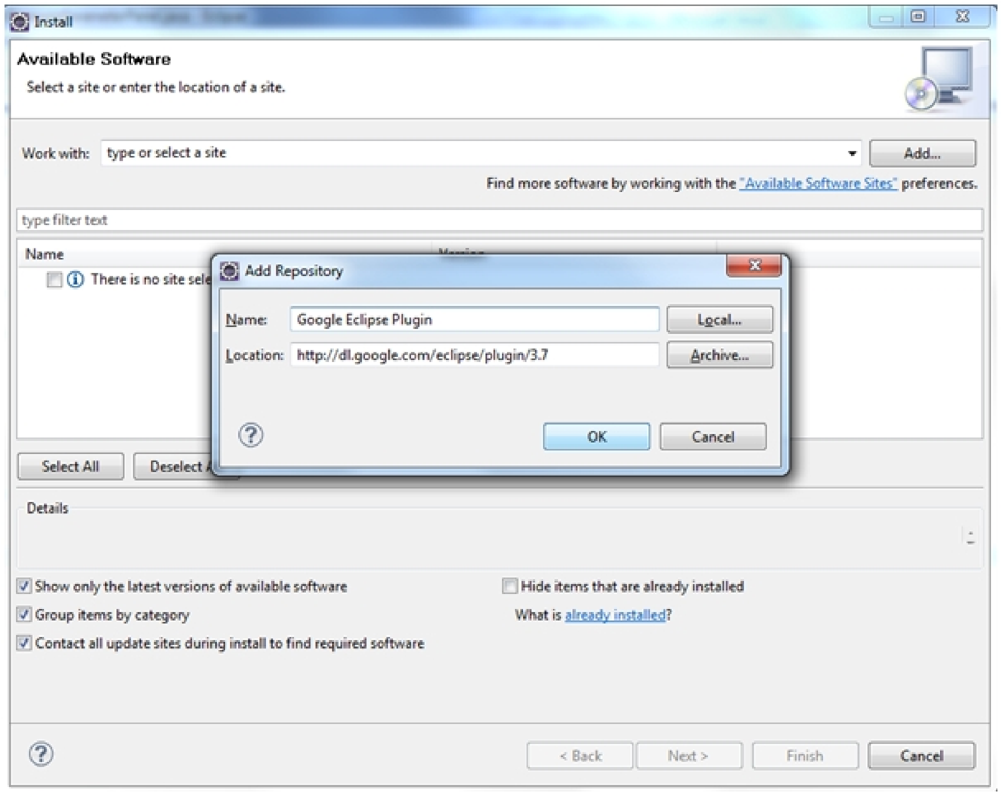
Instructions for the next two items are in this chapter:

- The Google for Eclipse Plugin, downloaded from the Internet and installed
- GWT Developer Plugin for your web browser (developed by Google), downloaded from the Internet and installed

Note: Also, you need some familiarity with Eclipse and Java, including how to set breakpoints and toggle Eclipse perspectives.

Step 1 - Download and Install the Google for Eclipse Plugin

1. Double-click the Eclipse executable file to launch Eclipse.
2. In Eclipse, click **Help** > **Install New Software** to see the next screen.


Install

Available Software

Select a site or enter the location of a site.

Work with: type or select a site

Find more software by working with the "Available Software Sites" preferences.

type filter text

There is no site sele

Add Repository

Name: Google Eclipse Plugin

Location: http://dl.google.com/eclipse/plugin/3.7

Details

☑ Show only the latest versions of available software

☑ Group items by category

☑ Contact all update sites during install to find required software

☐ Hide items that are already installed

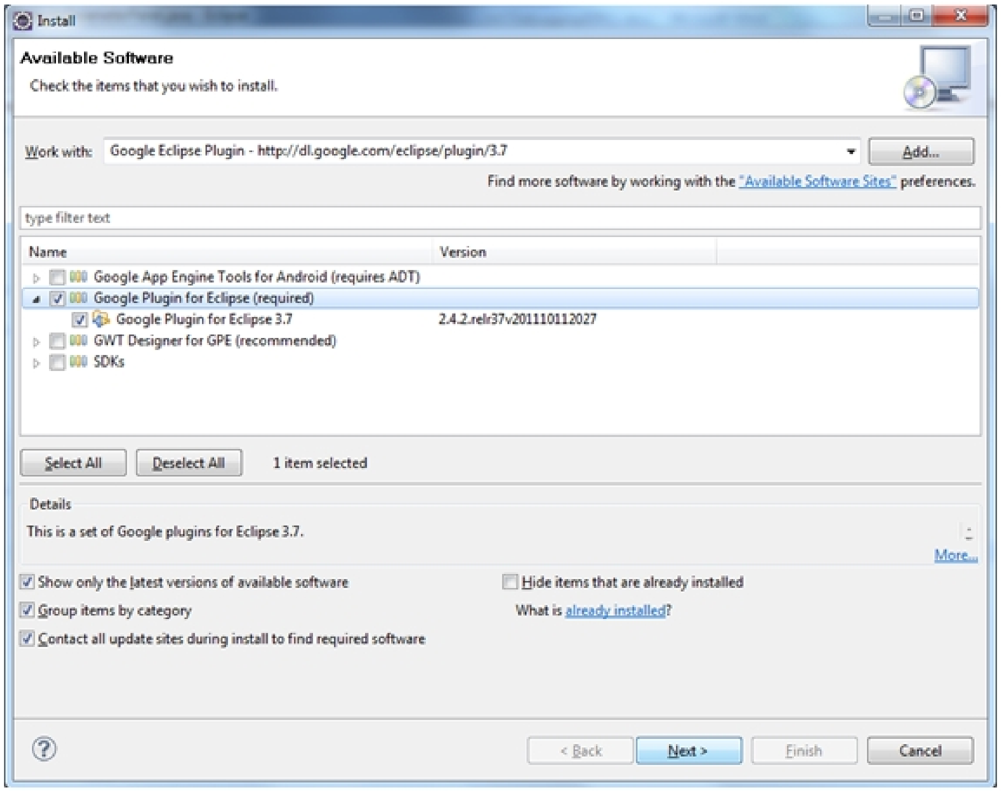
What is already installed?

3. Click the **Add** button to open the **Add Repository** popup box.
4. In the **Name** field, enter "`Google Eclipse Plugin`".
5. In the **Location** field, enter "`http://dl.google.com/eclipse/plugin/3.7`".

   Note: “`3.7`” refers to the Eclipse version. Change this number if your Eclipse version is different.

   Click **OK**.

   When the next screen appears, select **Google Plugin for Eclipse**.


Install

Available Software

Check the items that you wish to install.

Work with: Google Eclipse Plugin - http://dl.google.com/eclipse/plugin/3.7

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
☐ Google App Engine Tools for Android (requires ADT)	
☑ Google Plugin for Eclipse (required)	
☑ Google Plugin for Eclipse 3.7	2.4.2.relr37v201110112027
☐ GWT Designer for GPE (recommended)	
☐ SDKs	

1 item selected

Details

This is a set of Google plugins for Eclipse 3.7.

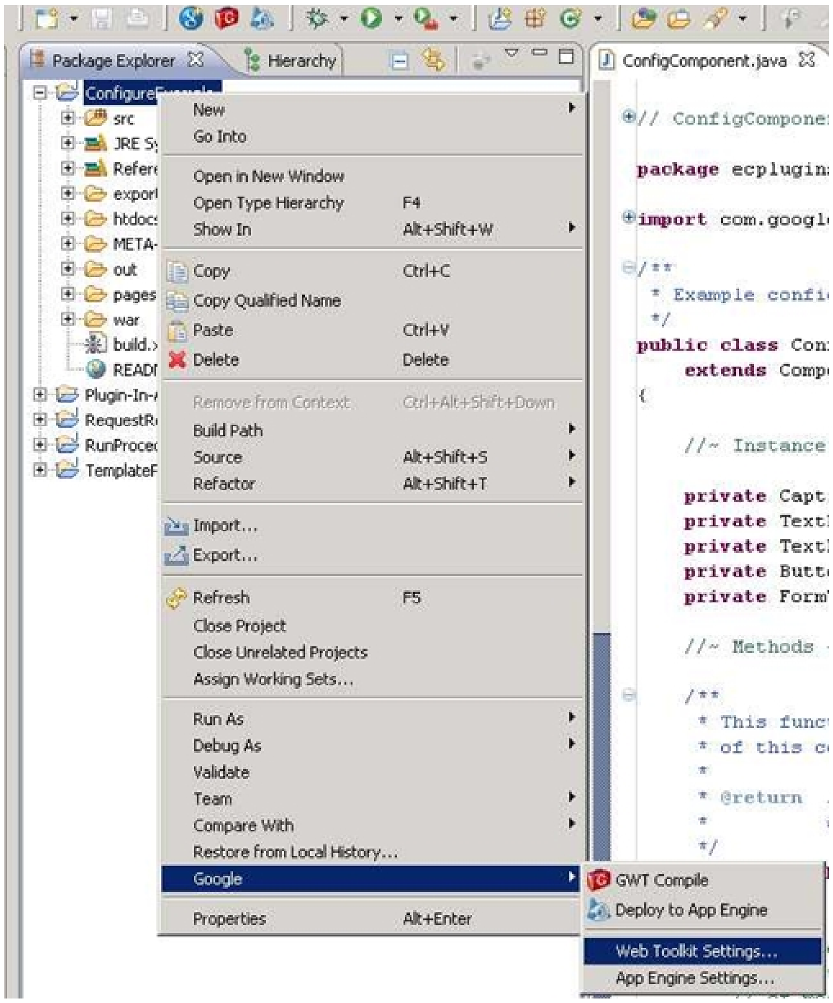
☑ Show only the latest versions of available software
☐ Hide items that are already installed
☑ Group items by category
What is already installed?
☑ Contact all update sites during install to find required software

6. Click **Next** and then click **Next** again.
7. Review and accept the license agreement.
8. Click **Finish** to complete the download and installation.

   Note: You may see security warnings requiring you to “trust” certificates. Trust these certificates and restart Eclipse if prompted. Failure to trust the certificate can cause your installation to fail.

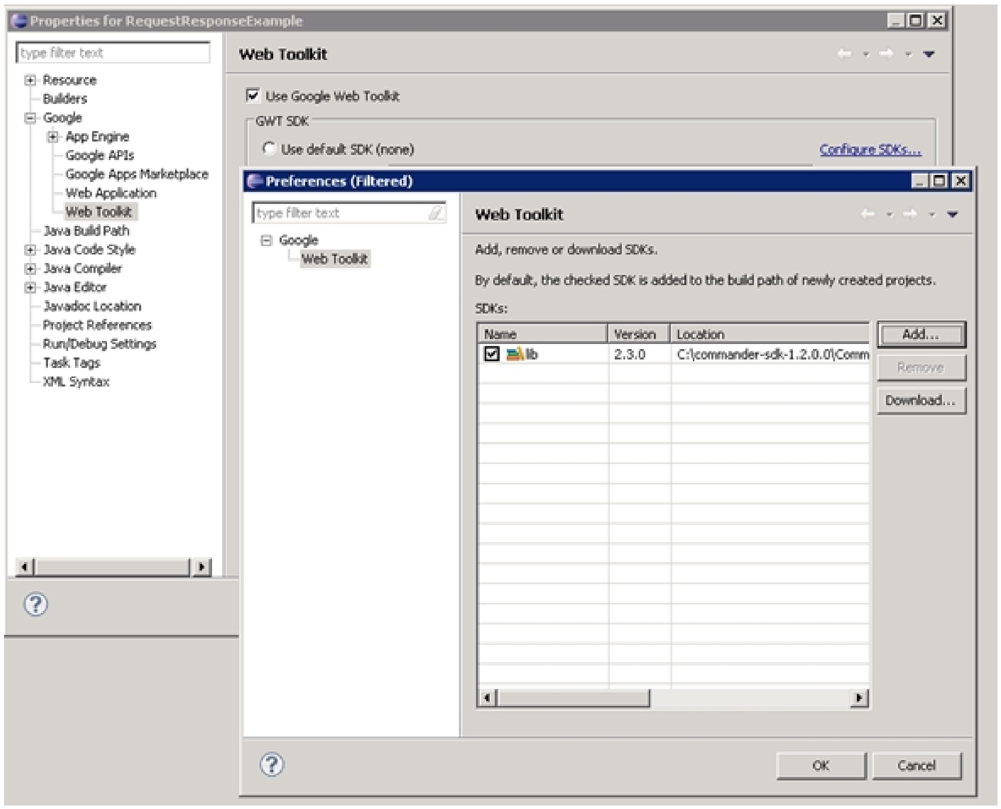
Step 2 - Setting Google-Related Properties for Your Project

1. Right-click to select the project you want to debug, then click **Google** > **Web Toolkit Settings**.



On the next screen:

2. Select the **Use Google Web Toolkit** checkbox.
3. Click **Configure SDK** and then click **Add**.
4. Browse to your ElectricFlow SDK directory, select the `lib` directory, and click **OK**.
5. Click **OK** again and make sure the small checkbox in the Name column is selected.



☑ Use Google Web Toolkit

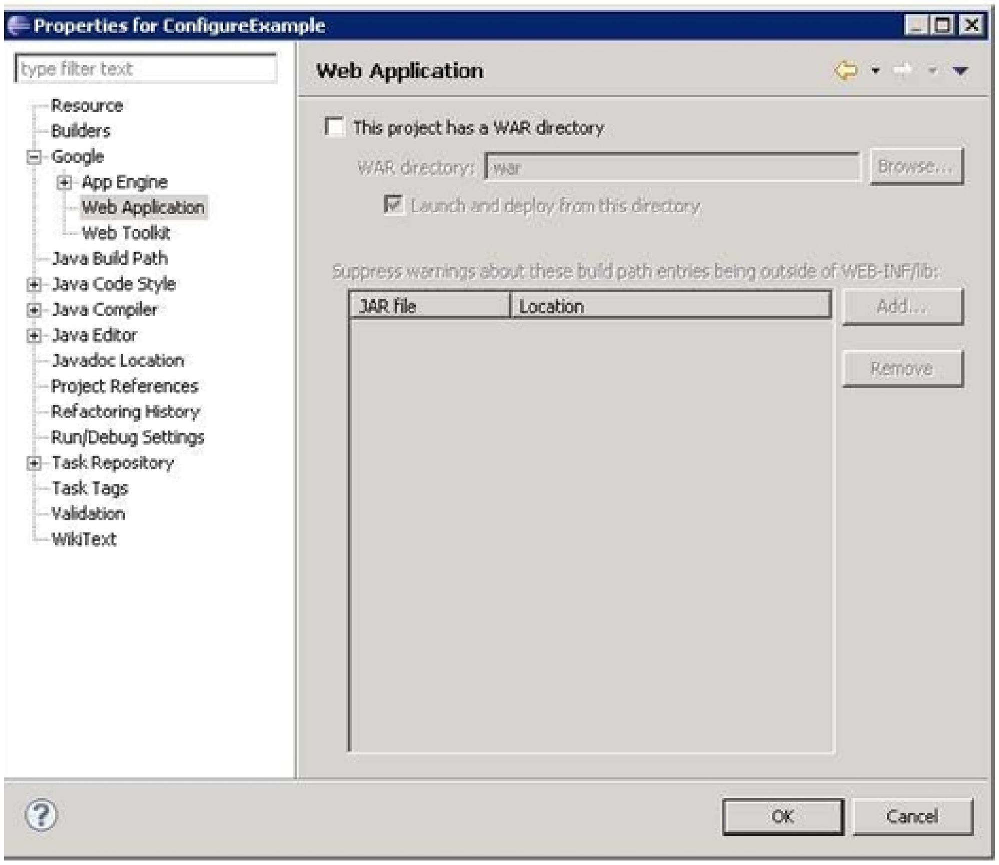
○ Use default SDK (none)

Add, remove or download SDKs.

By default, the checked SDK is added to the build path of newly created projects.

Name	Version	Location
☑ lib	2.3.0	C:\commander-sdk-1.2.0.0\Comm

6. Click **OK** again.
7. Select **Web Application**.
8. Deselect the **This project has a WAR directory** option.



This project has a WAR directory

WAR directory: war

Launch and deploy from this directory

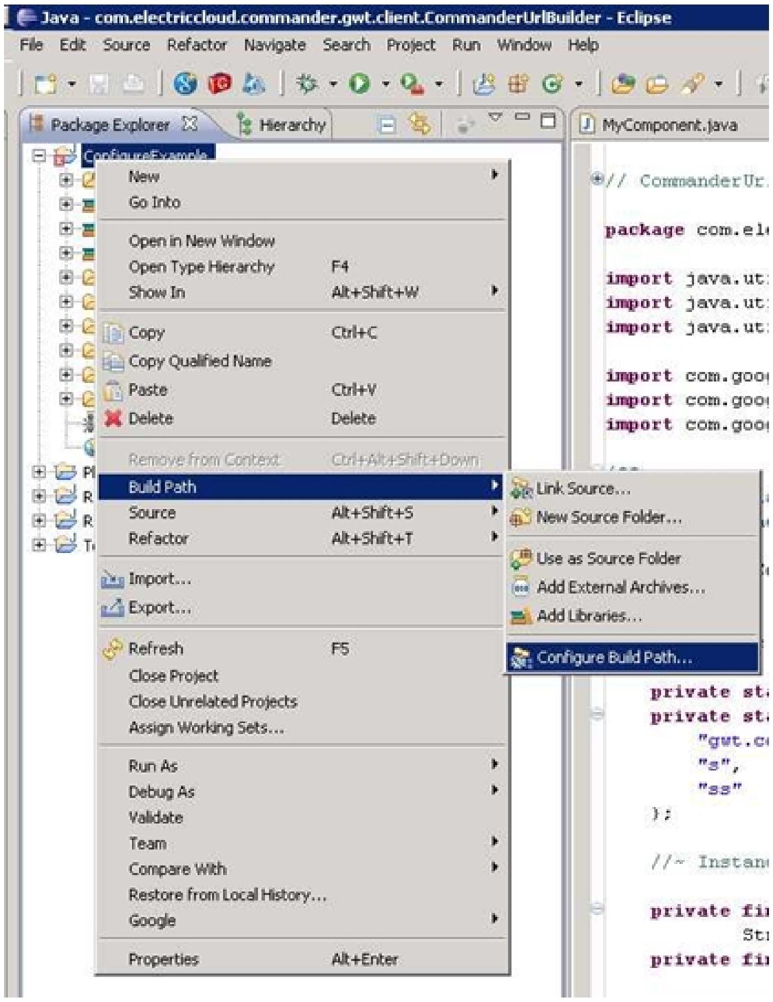
Suppress warnings about these build path entries being outside of WEB-INF/lib:

JAR file	Location

9. Click **OK**.

Step 3 - Configure Your Project’s Build Path

1. Right-click the project you want to debug and navigate to **Build Path** > **Configure Build Path**.



```
// CommanderUr

package com.el

import java.ut
import java.ut
import java.ut

import com.goo
import com.goo
import com.goo

    private st
    private st
        "gwt.c
        "s",
        "ss"
    };

    //~ Instan

    private fi
            St
    private fi
```

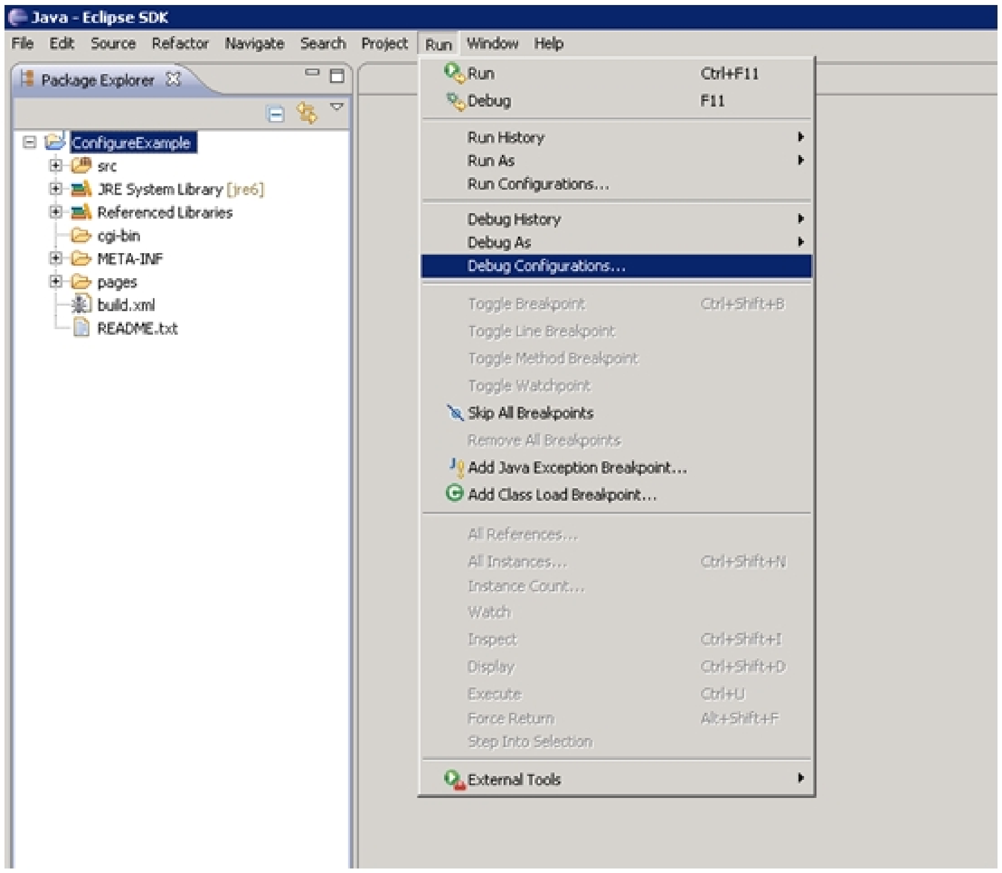
2. Select the Libraries tab. Click **Add External JARS...** and navigate to the ElectricFlow SDK `lib` directory. Then select all of the following jar files:
   - **annotations.jar**
   - **ant-contrib-1.0b3.jar**
   - **commander-client.jar**
   - **guava.jar**
   - **guava-gwt.jar**
   - **gwt-dev.jar**
   - **gwt-servlet.jar**
   - **jcip-annotations.jar**
   - **validation-api-1.0.0.GA.jar**
   - **validation-api-1.0.0.GA-sources.jar**
3. Click **Open**.

4. Click **OK**.

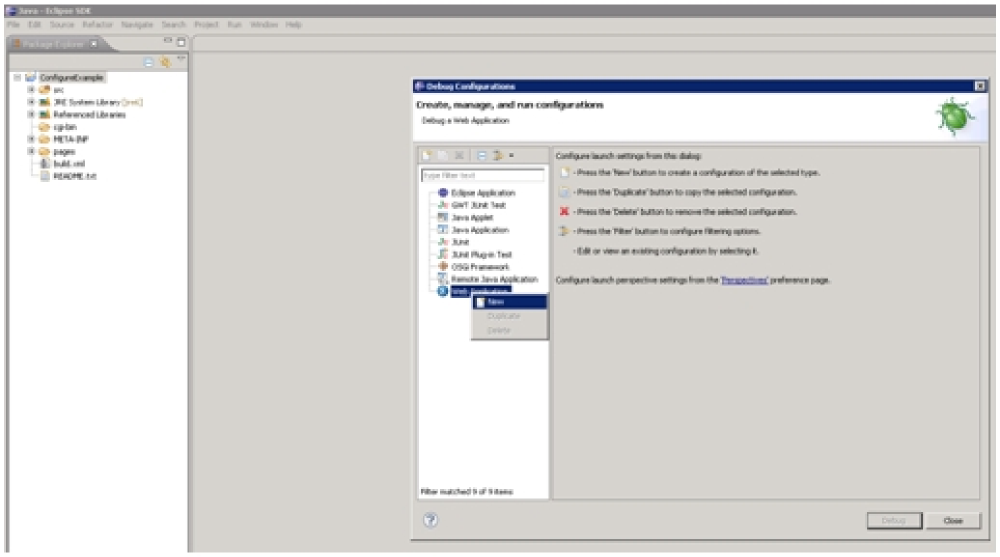
Step 4 - Creating the Debug Configuration in Eclipse

This step downloads and installs the GWT Developer Plugin for your web browser.

1. In the Package Explorer window, select the project you want to debug.
2. Click the **Run** tab, then click **Debug Configurations**.



3. Right-click **Web Application**, then click **New**.


Debug Configurations

Create, manage, and run configurations

Debug a Web Application

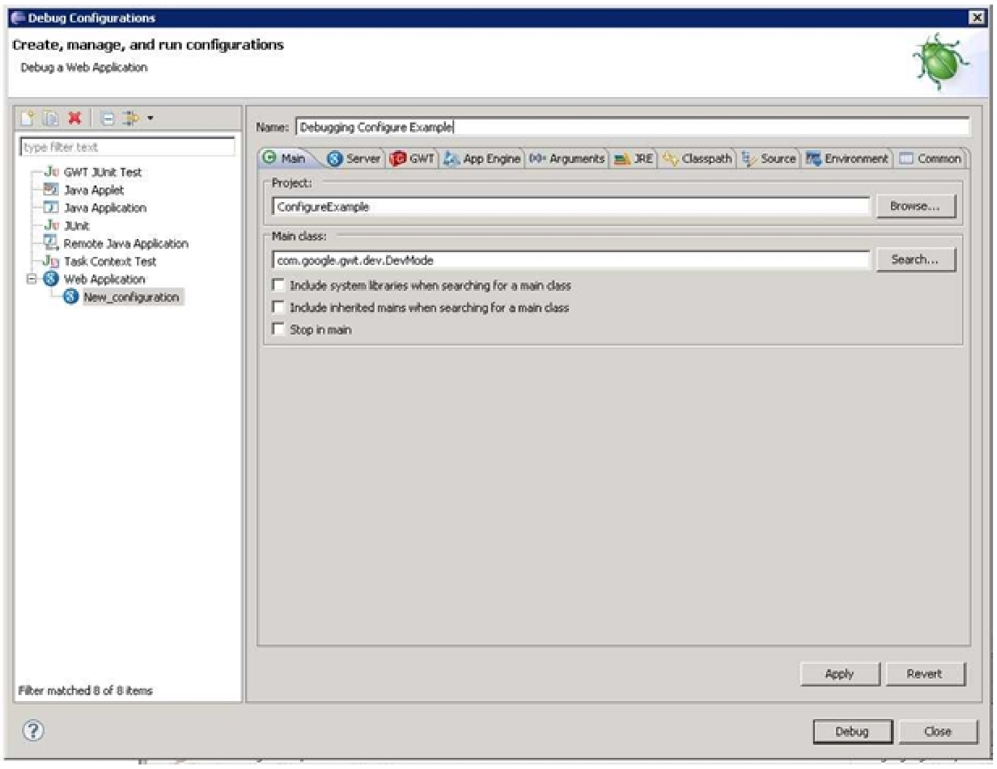
Configure launch settings from this dialog:

- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the 'Perspectives' preference page.

Filter matched 9 of 9 items

4. Type any name you choose in the **Name** field.
5. Enter "`com.google.gwt.dev.DevMode`" in the **Main** class field.
   (The Project field is pre-populated if you selected your project in the Package Explorer panel.)


Debug Configurations

Create, manage, and run configurations

Debug a Web Application

Name: Debugging Configure Example

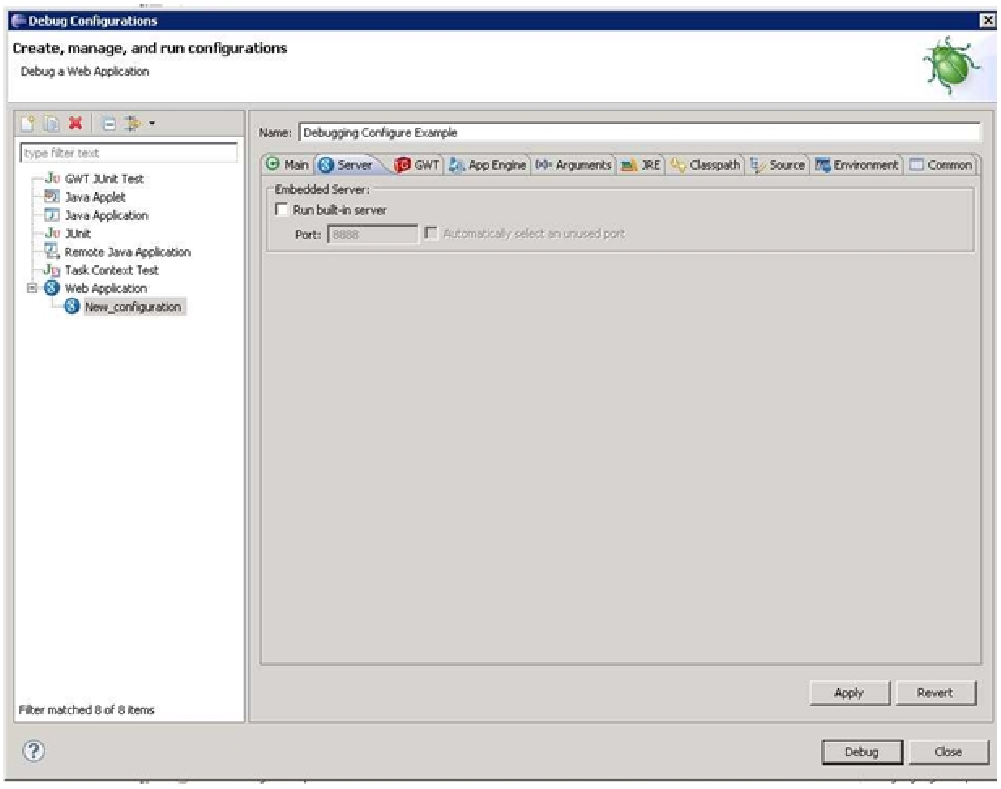
Project: ConfigureExample

Main class: com.google.gwt.dev.DevMode

- [ ] Include system libraries when searching for a main class
- [ ] Include inherited mains when searching for a main class
- [ ] Stop in main

Filter matched 8 of 8 items

6. Click the **Server** tab.
7. Deselect **Run built-in server**.



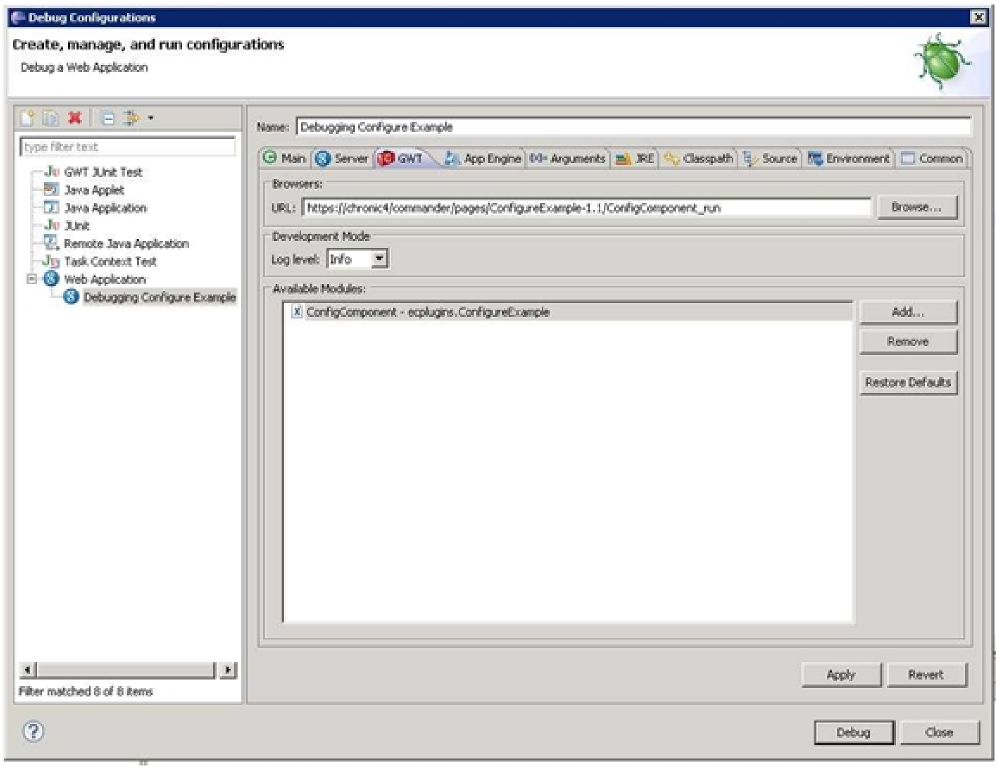
Name: Debugging Configure Example

Embedded Server:

Run built-in server

Port: 8888 Automatically select an unused port

8. Click the **GWT** tab.
9. In the URL field, enter the web URL for your plugin.



Name: Debugging Configure Example

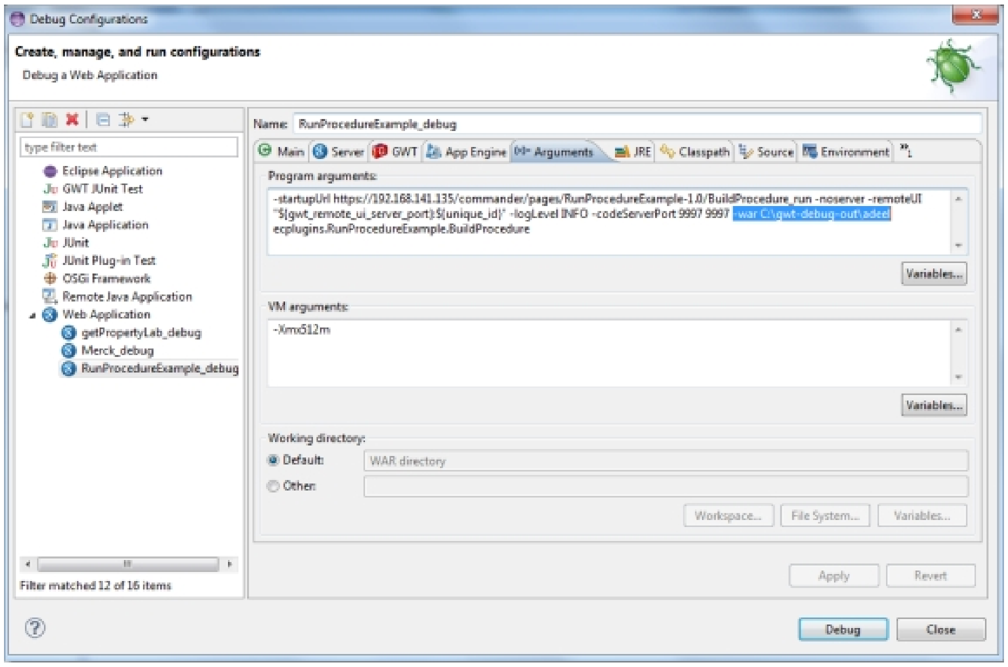
URL: https://chronic4/commander/pages/ConfigureExample-1.1/ConfigComponent_run

Log level: Info

ConfigComponent - ecplugins.ConfigureExample

10. Click the **Arguments** tab.

    - Insert a “-war” argument followed by a directory location, for example, C:\debug\war. Note that the specified directory should be at least two directories deep.
    - Also note that this cannot be the last argument. As shown in the screen shot below, it can be placed before the GWT module string.



Name: RunProcedureExample_debug

Program arguments:

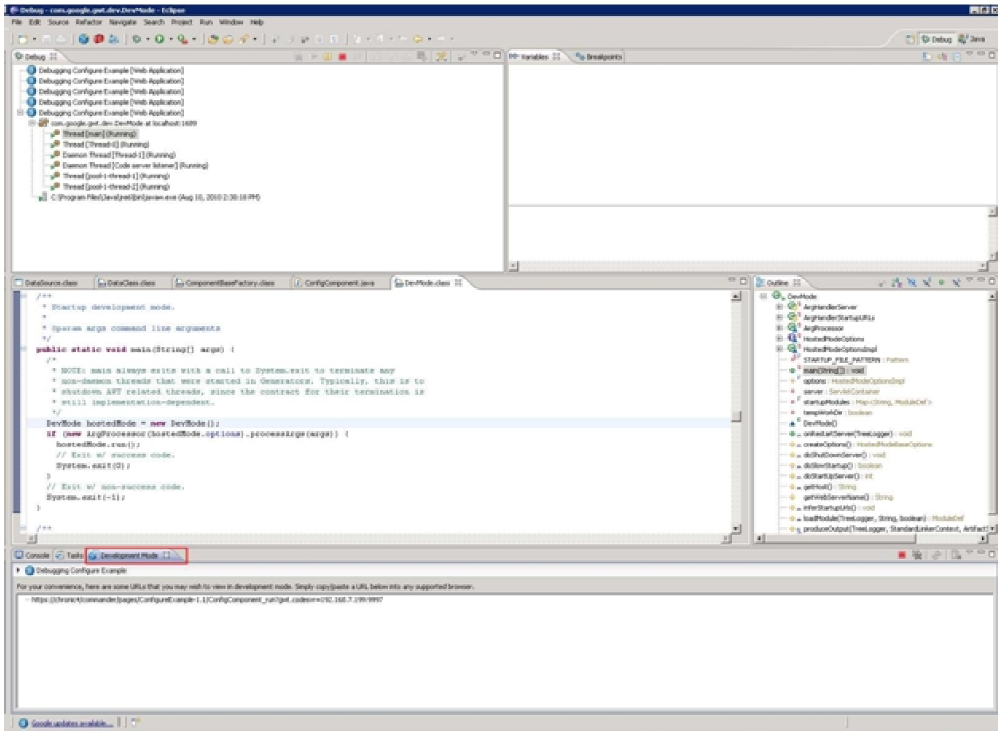
```
-startupUrl https://192.168.141.135/commander/pages/RunProcedureExample-1.0/BuildProcedure_run -noserver -remoteUI
"${gwt_remote_ui_server_port}:${unique_id}" -logLevel INFO -codeServerPort 9997 9997 -war C:\gwt-debug-out\adeel
ecplugins.RunProcedureExample.BuildProcedure
```

VM arguments:

```
-Xmx512m
```

Working directory:
Default: WAR directory
Other:

11. Click **Debug**.
12. Switch to the “Debug” perspective when Eclipse prompts you to do so.
13. Select the **Development Mode** tab (highlighted in “red”).
14. Copy and paste the displayed URL into your web browser.


Debug - com.google.gwt.dev.DevMode - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Java
Variables
Breakpoints
Debugging Configure Example [Web Application]
Debugging Configure Example [Web Application]
Debugging Configure Example [Web Application]
Debugging Configure Example [Web Application]
Debugging Configure Example [Web Application]
Thread [main] (Running)
DataSource.class
DataClass.class
ComponentBaseFactory.class
ConfigComponent.java
DevMode.class
Outline
Console
Tasks
Development Mode
Google updates available...

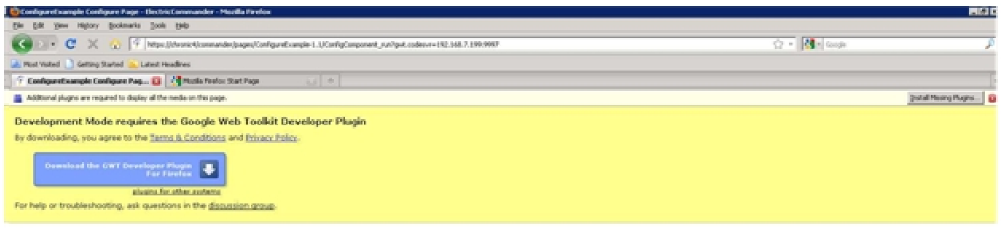
```
/**
 * Startup development mode.
 *
 * @param args command line arguments
 */
public static void main(String[] args) {
  /*
   * NOTE: main always exits with a call to System.exit to terminate any
   * non-daemon threads that were started in Generators. Typically, this is to
   * shutdown AWT related threads, since the contract for their termination is
   * still implementation-dependent.
   */
  DevMode hostedMode = new DevMode();
  if (new ArgProcessor(hostedMode.options).processArgs(args)) {
    hostedMode.run();
    // Exit w/ success code.
    System.exit(0);
  }
  // Exit w/ non-success code.
  System.exit(-1);
}
```

Debugging Configure Example

For your convenience, here are some URLs that you may wish to view in development mode. Simply copy/paste a URL below into any supported browser.

```
https://chronic4/commander/pages/ConfigureExample-1.1/ConfigComponent_run?gwt.codesvr=192.168.7.199:9997
```

15. When you open the URL in the web browser, you are prompted to download and install the GWT Developer Plugin.
16. Install the plugin and restart your browser if necessary.



Additional plugins are required to display all the media on this page.

Development Mode requires the Google Web Toolkit Developer Plugin

By downloading, you agree to the Terms & Conditions and Privacy Policy.

For help or troubleshooting, ask questions in the discussion group.

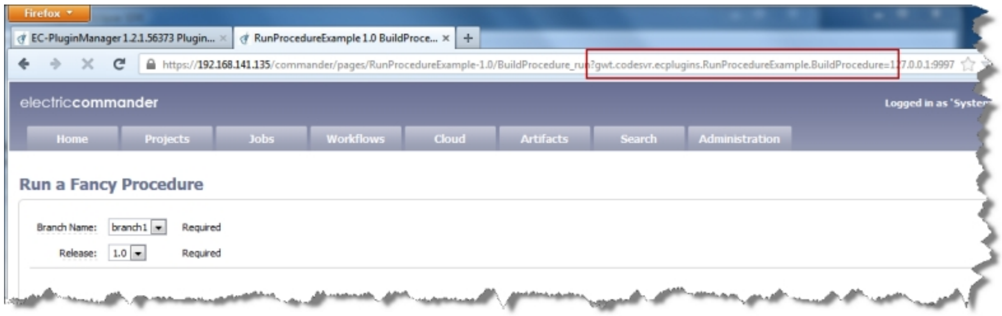
Download the GWT Developer Plugin For Firefox
plugins for other systems
Install Missing Plugins...

17. Reload the page but modify the URL from the previous page.

    For example, the URL for the RunProcedure looks like this (as generated by the Eclipse plugin):
    `https://192.168.141.135/commander/pages/RunProcedureExample-1.0/BuildProcedure_run?gwt.codesvr=127.0.0.1:9997`

    You will need to add `.ecplugins.RunProcedureExample.BuildProcedure` to `gwt.codesvr` in the URL like this:
    `https://192.168.141.135/commander/pages/RunProcedureExample-1.0/BuildProcedure_run?gwt.codesvr.ecplugins.RunProcedureExample.BuildProcedure=127.0.0.1:9997`

    The package path of the `.gwt.xml` file for your plugin is what you will need to add after `gwt.codesvr`.



https://192.168.141.135/commander/pages/RunProcedureExample-1.0/BuildProcedure_run?gwt.codesvr.ecplugins.RunProcedureExample.BuildProcedure=127.0.0.1:9997

Run a Fancy Procedure

Branch Name: branch1 Required

Release: 1.0 Required

You are now ready to set and hit breakpoints.

Chapter 8: Troubleshooting

Common Problems and Resolutions

Problem

After installing a plugin, I go to a page and see nothing.

Solution

- Check Apache logs for errors.
- Simplify the page. Perhaps bad logic is corrupting the page.

Problem

When I open `plugin.xml` in Eclipse, I get a “funky” UI screen.

Solution

plugin.xml is the reserved name of Eclipse plugin config files. You could be in the "Overview" tab of Eclipse's XML editor. Click the `plugin.xml` tab to see/edit the raw XML.

Problem

Plugin build fails because an output directory was deleted as part of the 'clean' target.

Solution

You probably have a file opened in the directory or a shell sitting in that directory (or subdirectory).

Problem

I am developing a GWT plugin and receiving difficult to understand JavaScript errors during run time.

Solution

By default, GWT obfuscates the JavaScript it produces.

If you prefer not to have GWT obfuscate its output, you can specify the "`gwt.style`" argument to ant.

```
<SDK installation directory>/tools/ant/bin/ant -Dgwt.style="DETAILED" build
```

This (`gwt.style`) flag has one of three possible values:

- OBF (for obfuscated), the default
- PRETTY, makes the output humanly readable, but much larger
- DETAILED, improves on PRETTY with more detail (such as very verbose variable names)

Problem

I try to build an example from the command-line, and I get an error like the following:

```
Buildfile: build.xml does not exist!
Build failed
```

Solution

Run `configureSDK.pl` to configure SDK examples.

Problem

In Eclipse, I invoke an external tool launcher and see an error with text "Variable references empty selection".

Solution

Be sure to select a project in Package Explorer and Package Explorer should have the focus.

Problem

Developing a custom plugin and encountering problems with CGI scripts or promote/install logic in `ec_setup.pl`. Where are the log files so I can get more information?

Solution

- Check the commander-log for errors.
- Enable logging for CGI scripts. On a development web server (NOT production), add a ScriptLog directive to the web server apache configuration, `httpd.conf`. Restart the web server and try to reproduce the problem. More information about the ScriptLog directive is available at: http://httpd.apache.org/docs/current/mod/mod_cgi.html

Appendix A: Converting SDK Versions

Moving to ElectricFlow SDK 6.0

ElectricFlow 6.0 includes new rebranded UI, which changes the CSS and the look and feel of the UI elements.

Moving to CommanderSDK 5.0

- In ElectricFlow 5.0, all IDs are converted to universally unique identifiers (UUIDs), including `jobId` and `workflowId`, the most widely public referenced IDs. To use plugins that were developed in previous ElectricFlow Plugin versions in the current Plugin version, you must recompile the plugins.

  For more information about converting to UUIDs, see the"Upgrading from ElectricCommander 4.2.x to ElectricCommander 5.x" chapter in the ElectricFlow 5.x Installation Guide.

- When you migrate from an earlier version to CommanderSDK 5.0, you need to modify any Java class that returns jobId or JobStepId due to the UUID change:

  Before:

  ```
  @Override public void handleResponse(RunProcedureResponse response)
     {
       Long jobId = response.getJobId();
       ....
       ....
   }
  ```

  After:

  ```
  @Override public void handleResponse(RunProcedureResponse response)
     {
       String jobId = response.getJobId();
       ....
       ....
   }
  ```

Moving from CommanderSDK 1.2 to 2.0

- The CommanderSDK now uses GWT 2.5.0, so you may need to make additional changes if you were using a previous version of GWT.
- .gwt.xml file must now include the following line:

  ```
  <inherits name="com.electriccloud.commander.CommanderClient"/>
  ```

- Any class that extends ComponentBaseFactory must perform the following transformation.

  ```
  @Override public Component createComponent(JavaScriptObject jso)
      {
          return new YOUR_COMPONENT();
      }
  ```

  to

  ```
  @Override
      protected Component createComponent(ComponentContext jso) {
          return new YOUR_COMPONENT();
      }
  ```

- Parameter Panel changes:
  - getRequests() has been removed from ParameterPanel.java; move all requests out of the function and into doInit(). You must perform the doRequest(ChainedCallback c, CommanderRequests<?> requests) yourself.
  - The logic contained in onInitComplete should now be relocated to the ChainedCallback specified at doRequest time.
  - Classes implementing ParameterPanel and ParameterPanelProvider no longer need to implement the HasInitCallback interface.
- All plugins that compiled with SDK versions 1.2 and earlier must be changed. If you try to upgrade from SDK 1.2 to SDK 2.0 you will get compilation errors.

  ComponentBaseFactory.java's function protected abstract Component createComponent changed from

  ```
  protected abstract Component createComponent(JavaScriptObject jso);
  ```

  to

  ```
  protected abstract Component createComponent(ComponentContext jso);
  ```

  That means any java class extending ComponentBaseFactory must be changed from

  ```
  @Override public Component createComponent(JavaScriptObject jso)
      {
          return new HelloWorldComp();
      }
  ```

  to

  ```
  @Override public Component createComponent(ComponentContext jso)
      {
          return new HelloWorldComp();
      }
  ```

  The following line must be added to the top of the java file

  ```
  import com.electriccloud.commander.gwt.client.ComponentContext;
  ```

- To build any plugin, the COMMANDER_HOME environment variable must be set to the ElectricFlow install directory. The default install directory:
  - on Windows - `C:\Program Files\Electric Cloud\ElectricCommander`
  - on Linux - `/opt/electriccloud/electriccommander`
- Some packages were renamed and no longer include '.gwt'. Import directives in Java classes must be modified in order to compile the code. For example: RunProcedureRequest used to exist in the com.electriccloud.commander.gwt.client.requests package but now exists in com.electriccloud.commander.client.requests package.

  Renamed packages:
  - com.electriccloud.commander.client.domain
  - com.electriccloud.commander.client.domain.impl
  - com.electriccloud.commander.client.protocol
  - com.electriccloud.commander.client.requests
  - com.electriccloud.commander.client.requests.impl
  - com.electriccloud.commander.client.responses
  - com.electriccloud.commander.client.responses.impl
  - com.electriccloud.commander.client.util

  Requests and responses are now in the file commander-client.jar. As of version 4.2.1, ElectricFlow ships with the commander-client.jar file in the `<`*`installDir`*`>/utils` directory. Contact Electric Cloud Technical Support if you cannot locate the commander-client.jar file.

Moving from CommanderSDK 1.1 to 1.2

CommanderSDK uses GWT 2.3.0, which requires additional library files (jars) in the build path. These libraries are included in the CommanderSDK's build system. However, for GWT Development you need to add the libraries to your Eclipse project manually. Find the following libraries you need in the lib directory:

- `validation-api-1.0.0.GA.jar`
- `validation-api-1.0.0.GA-sources.jar`

Moving from CommanderSDK 1.0 to 1.1

Assuming you have already migrated to CommanderSDK 1.0, CommanderSDK 1.1 is backwards compatible except for the following differences.

Review the following list to become familiar with CommanderSDK 1.1 changes.

- When using GWT 2.2.0, the "gecko" value in *.gwt.xml files will not compile. All instances of "gecko" must be replaced with "gecko1_8".
  For example: `<set-property name="user.agent" value="gecko" />` must be replaced with:
  `<set-property name="user.agent" value="gecko1_8" />`
- Debugging plugins developed with the CommanderSDK Java binding in GWT Development Mode is now dependent on your ElectricFlow Server version and CommanderSDK version number.
  - To debug a plugin developed with CommanderSDK 1.1, deploy the plugin to an ElectricCommander 3.9 server.

    - You must use the CommanderSDK version that coincides with your ElectricFlow released version. If you want to use a newer SDK (and use the debugger), you may need to upgrade to an ElectricFlow version that supports that SDK version.

- Some interface method return types in the domain package have changed from "long" to "Integer". This is a backwards ***incompatible*** change. The following is a list of changes:

Interface	**Methods**
`Resource.java`	`getPort, getProxyPort, getStepCount, getStepLimit`
`ProcedureStep.java`	`getRetries`
`License.java`	`getGracePeriod`
`JobStep.java`	`getExitCode, getPostExitcode, getRetries`
`Emailconfiguration.java`	`getMailPort`
`AgentStatus.java`	`ggetProtocolVersion`

- Some interface methods have changed return types or the type supplied as an argument from a Java primitive has changed to a Java Object.
  For example:
    - `public long getABC()` has changed to `public Long getABC()`, and `public void setABC (long abc)` has changed to `public void setABC(Long abc)`
    - In practice, this limits null pointer exceptions "thrown" from SDK code and allows you to check for nulls in your code. The conversion from primitive to object allows null values to be sent to the ElectricFlow Server during requests to reset a value.

Migrating from CommanderSDK 0.9.1 to 1.0

Summary

- In SDK 0.9.1, you were expected to parse ElectricFlow XML responses after making ElectricFlow API requests. In SDK 1.0, a Java object is returned as a result of an ElectricFlow API request. This object contains methods to retrieve specific values. Any code making ElectricFlow API requests and parsing the results will need to change as a result.
- Slight code changes to Factory classes, extending ComponentBaseFactory. These Factory classes reside in the `ecplugins.YourPlugin.client` package.
- With SDK 1.0, backwards compatibility will be preserved with respect to code compilation. If a plugin builds against SDK 1.0 it will build against SDK 1.1, 1.5, 2.0 and so on.
- Introduced Workflow objects and requests (ElectricCommander 3.8 feature).
- SDK JAR file was renamed from "`commander-sdk.jar" to "ec-gwt.jar`".
- SDK 1.0 does not include the Eclipse Plugin-In-A-Box environment. However, the SDK does include Eclipse launchers you can install into your Eclipse Galileo or Helios workspace.

- SDK 1.0 supports command-line plugin building. See Updating “Plugin-in-a box” to the CommanderSDK for information on updating an existing Plugin-in-a-box environment to leverage the SDK 1.0 build system.

Motivations for Moving to SDK 1.0

We expect the Workflow functionality to be the most common reason to build your existing plugin against CommanderSDK 1.0. You must migrate if the plugin you are developing issues `getWorkflow`, `getWorkflows` requests or `findObjects` requests for workflows.

Making ElectricFlow Requests and Parsing the Result

In CommanderSDK 1.0, the way you make and parse ElectricFlow API requests has changed. The old-style request classes exist in the "`legacyrequests`" package for backward compatibility. These classes are deprecated, so you are strongly encouraged to switch to the new mechanism at your earliest convenience. Difference highlights between the "`legacyrequests`" and the new requests:

- The result of an ElectricFlow API call will no longer be an XML node but rather a Java object (in the form of a Java interface) encapsulating the response.
  - For example, if you are asking for a Workspace, you will now get a Workspace object instead of

    ```
    <workspace>
      <workspaceName>MyWorkspace</ workspaceName >
      <agentDrivePath>n:\ws</agentDrivePath>
      <agentUncPath>//server/ws</agentUncPath>
      <agentUnixPath>/net/server/ws</agentUnixPath>
      ...
    </workspace>
    ```
  - To retrieve the name of the workspace, call the `getWorkspaceName()` method on the Workspace object. There should be a Java "get" method on the object for every element in the XML response.
- Instead of instantiating a request object directly, you ask a Factory to provide the request object.
  - For example, pre-1.0 you would create a `getProperty` request like this:

    ```
    GetPropertyRequest request = new
    GetPropertyRequestImpl();
    ```

  Using SDK 1.0, you would use:

    ```
    GetPropertyRequest request =
    getRequestFactory().createGetPropertyRequest();
    ```

You create requests using `createFoo(...)` methods in the CommanderRequestFactory. The method, `getRequestFactory`, is defined in `ComponentBase`, one of the parent classes of the GWT component class.

- Instead of registering a callback, you are now required to set a callback on the request object.
  - For example, do not use:

    ```
    registerCallback(requestId, <callback>);
    ```

    Instead, use:

    ```
    request.setCallback(<callback>);
    ```

Set a callback on the request object for handling the response. This action is equivalent to calling registerCallback in the component in previous releases, except the callback type is specific to the request being made rather than being an anonymous class implementation of CommanderRequestCallback.

For example, instead of:

```
        registerCallback(request.getRequestId(),
            new CommanderRequestCallback() {
                @Override public void handleError(Node responseNode)
                {
                    //Error Handling logic here
                }
                @Override public void handleResponse(Node responseNode)
                {
                    //responseNode parsing logic here
                }
            });
```

you can now use the following (for a request that returns a property):

```
        request.setCallback(new PropertyCallback() {
                @Override public void handleResponse(Property response)
                {
                }
                @Override public void handleError(CommanderError error)
                {
                    //Error Handling logic here
                }
            });
```

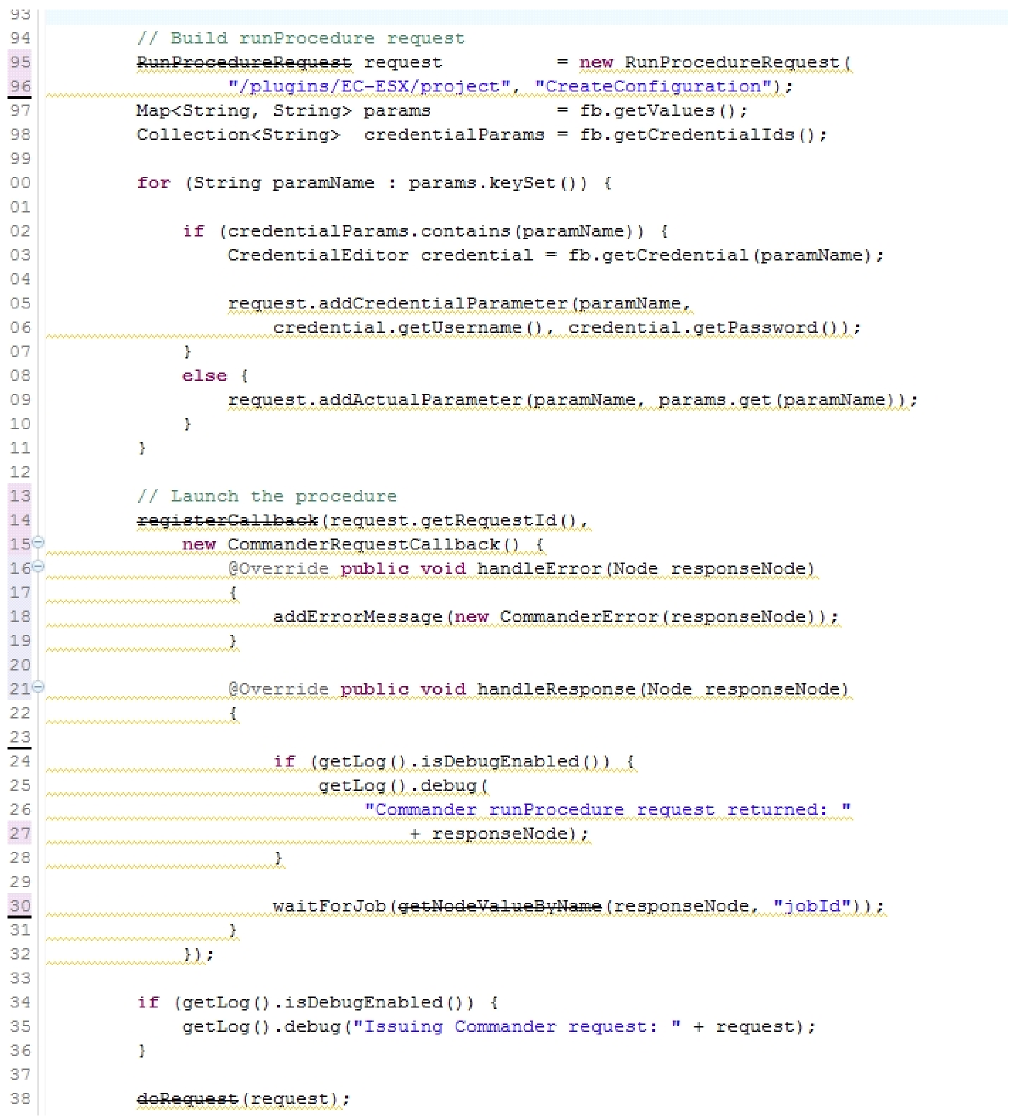
- After a request object is retrieved, you can call various setters on that object to set attributes. This process is the same as in earlier SDK releases. To send a request to ElectricFlow, call `doRequest (req)`, or call `getRequestManager().doRequest(req)`.

Example of ElectricFlow API Request Migration

The following is an example of a RunProcedureRequest migration from SDK 0.9.1 to SDK 1.0.

Before reviewing the following migration example, you may want to review the RequestResponseExample's RequestResponseComponent to acclimate yourself with the new request mechanism.

Before:



```
93
94          // Build runProcedure request
95          ~~RunProcedureRequest~~ request          = new RunProcedureRequest(
96                  "/plugins/EC-ESX/project", "CreateConfiguration");
97          Map<String, String> params           = fb.getValues();
98          Collection<String>  credentialParams = fb.getCredentialIds();
99
00          for (String paramName : params.keySet()) {
01
02              if (credentialParams.contains(paramName)) {
03                  CredentialEditor credential = fb.getCredential(paramName);
04
05                  request.addCredentialParameter(paramName,
06                      credential.getUsername(), credential.getPassword());
07              }
08              else {
09                  request.addActualParameter(paramName, params.get(paramName));
10              }
11          }
12
13          // Launch the procedure
14          ~~registerCallback~~(request.getRequestId(),
15              new CommanderRequestCallback() {
16                  @Override public void handleError(Node responseNode)
17                  {
18                      addErrorMessage(new CommanderError(responseNode));
19                  }
20
21                  @Override public void handleResponse(Node responseNode)
22                  {
23
24                      if (getLog().isDebugEnabled()) {
25                          getLog().debug(
26                              "Commander runProcedure request returned: "
27                                  + responseNode);
28                      }
29
30                      waitForJob(~~getNodeValueByName~~(responseNode, "jobId"));
31                  }
32              });
33
34          if (getLog().isDebugEnabled()) {
35              getLog().debug("Issuing Commander request: " + request);
36          }
37
38          ~~doRequest~~(request);
```

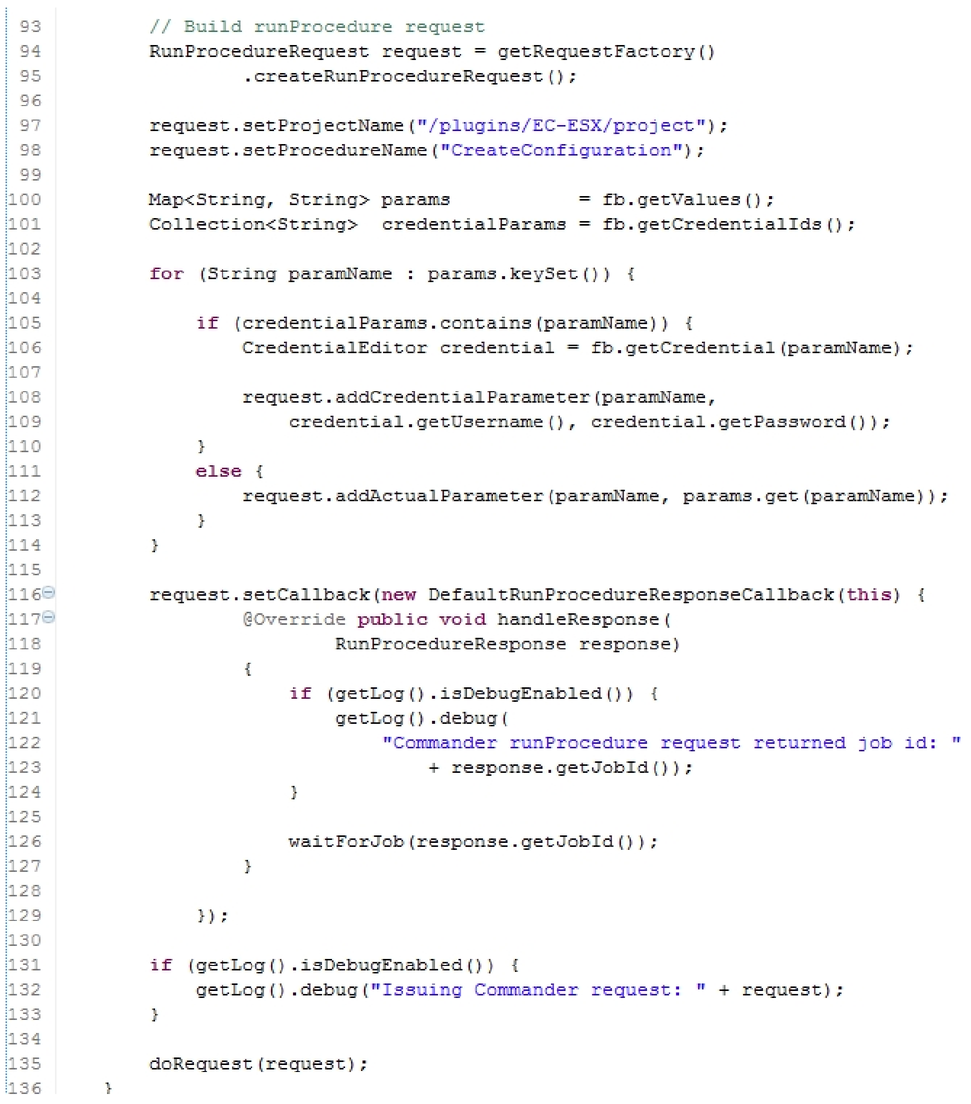
Steps to Migrate

1. Line 95: Use RunProcedureRequest in the `com.electriccloud.commander.gwt.client.requests` package instead of `com.electriccloud.commander.gwt.client.legacyrequests`
   - Do not instantiate the class directly. `getRequestFactory().createRunProcedureRequest()` `getRequestFactory` is defined in `ComponentBase. CreateConfiguration is a ComponentBase`.
2. All methods from the Factory are in the following format: `create + <`*`requestName`*`>`. Most methods do not take arguments. Instead, arguments must be set for the request by setter methods.
3. Two arguments are passed to the old RunProcedureRequest constructor. Looking at the RunProcedureRequest located in the legacy request package, we see the parameter names for the constructor are `projectName` and `procedureName`. Look for setters with these names for the new RunProcedureRequest object.

4. With old request objects, we had to register callbacks. The new request objects require setting a callback on the request object. Pass an anonymous class to the `setCallback` method.
   - In this class, there is no special error handling, so we can use the DefaultRunProcedureResponseCallback.
5. Transfer the functionality from the old callback to the new callback. An XML node was passed to the old callback's `handleResponse` and `handleError` methods. The new callback takes Domain Objects and CommanderError as indicated in the method signatures.
   - "`getNodeValueByName(responseNode, "jobId")`" was changed to "`response.getJobId()`"
   - "`addErrorMessage(new CommanderError(responseNode))`" was changed to "`addErrorMessage(error)`"
   - "`getLog().debug("Commander runProcedure request returned: " + responseNode)`" was changed to "`getLog().debug("Commander runProcedure request returned: " + response.getJobId())`"

The next screen illustrates completed code changes after the migration.

After:



```
        // Build runProcedure request
        RunProcedureRequest request = getRequestFactory()
                .createRunProcedureRequest();

        request.setProjectName("/plugins/EC-ESX/project");
        request.setProcedureName("CreateConfiguration");

        Map<String, String> params           = fb.getValues();
        Collection<String>  credentialParams = fb.getCredentialIds();

        for (String paramName : params.keySet()) {

            if (credentialParams.contains(paramName)) {
                CredentialEditor credential = fb.getCredential(paramName);

                request.addCredentialParameter(paramName,
                    credential.getUsername(), credential.getPassword());
            }
            else {
                request.addActualParameter(paramName, params.get(paramName));
            }
        }

        request.setCallback(new DefaultRunProcedureResponseCallback(this) {
                @Override public void handleResponse(
                        RunProcedureResponse response)
                {
                    if (getLog().isDebugEnabled()) {
                        getLog().debug(
                            "Commander runProcedure request returned job id: "
                                + response.getJobId());
                    }

                    waitForJob(response.getJobId());
                }

            });

        if (getLog().isDebugEnabled()) {
            getLog().debug("Issuing Commander request: " + request);
        }

        doRequest(request);
    }
```

Changes to Factory Classes that Extend ComponentBaseFactory

- The method `onCommanderInit` was replaced with `createComponent`. `createComponent` returns a component.

  So where you may have had the following:

  ```
  public void onCommanderInit(String divId, JavaScriptObject jso)
  {
      renderIntoDiv(divId, jso, new FancyComponent());
  }
  ```

  Now you will use:

  ```
  public Component createComponent(JavaScriptObject jso)
  {
      return new FancyComponent();
  }
  ```

- These Factory classes will no longer implement `EntryPoint`.

Instead of:

```
public class YourComponentFactory
    extends ComponentBaseFactory
    implements EntryPoint
```

You will now use:

```
public class YourComponentFactory
    extends ComponentBaseFactory
```

Transitioning from Using MultiRequestLoader

MultiRequestLoader provided a mechanism for grouping requests together, each with their own response callbacks, but with an additional callback to be invoked when all responses to requests in the group are processed.

For example:

```
MultiRequestLoader loader = new MultiRequestLoader(this, new
MultiRequestLoaderCallback() {
    public void onComplete()
    {
        // Do something after all responses have been processed.
    }
);

// Put together requests r1 and r2...
// ...

// Add the requests to the loader.
loader.addRequest(r1, new CommanderRequestCallback() {...});
loader.addRequest(r2, new CommanderRequestCallback() {...});

// Run the loader.
loader.load();
```

With CommanderSDK 1.0, you would do the same thing as follows (getWorkspace and getProject requests were chosen for this example):

```
GetWorkspaceRequest r1 = m_requestFactory.createGetWorkspaceRequest(...);
r1.setCallback(new WorkspaceCallback() {
    public void handleResponse(Workspace obj) {...}
    public void handleError(CommanderError e) {...}
});

GetProjectRequest r2 = ....;
r2.setCallback(new ProjectCallback() {
    public void handleResponse(Project obj) {...}
    public void handleError(CommanderError e) {...}
});

m_requestManager.doRequest(new ChainedCallback() {
    public void onComplete()
    {
        // Do something after all responses have been processed.
    }
}, r1, r2);
```

Also note: If you want `handleError` to invoke `handleError` in your component, use the `DefaultFooCallback` abstract class. For example, for `r1` above:

```
r1.setCallback(new DefaultWorkspaceCallback(this) {
    public void handleResponse(Workspace obj) {...}
});
```

CommanderSDK 1.0 File Changes

- Many classes from the UI package were removed and will be added back in over time based on demand.
- Many classes were moved from the top-level "client" package to sub-packages.

plugin.xml

- `vendor` tag was subsumed by `author` tag
- `authorUrl` tag was added
- added ElectricFlow `version` tag to indicate minimum and maximum ElectricFlow versions the plugin will work with, which means the Plugin Manager will no longer show incompatible plugin versions as "installable"
- added `category` tag, which allows the Plugin Manager to show plugins grouped by a particular category only

Updating “Plugin-in-a box” to the CommanderSDK

Loading SDK 1.0 Examples in Eclipse

Most examples exist (with the same names) in CommanderSDK. First, delete existing example projects from Eclipse, then import the new examples. You might want to delete the TemplatePlugin project as well. See Importing SDK Examples into Eclipse for details on importing the new examples.

As an alternative, if you want to save previous work, you can create a new Eclipse workspace for CommanderSDK 1.0. See Eclipse Integration for details.

Updating an existing plugin to Use CommanderSDK 1.0

Update the `<import>` line in `build.xml` (in your project) to import the `buildTargets.xml` from the CommanderSDK (for example, `<`*`your SDK installation directory`*`>/build/buildTargets.xml`) when performing plugin builds. This change ensures command-line plugin builds work properly, but not if using the external tool launchers. See the next section below for details.

Update “Referenced Libraries” to point Eclipse to the new CommanderSDK for Java compiles. The easiest way to dothis is to copy the `.classpath` file (from one of the new examples) to your plugin, then refresh your plugin project (select your project in “Package Explorer”, then press F5).

Updating External Tool Launchers

CommanderSDK contains new Eclipse launchers that need to be installed. The SetBuildMode launcher was renamed to SetBrowsers and now provides the opportunity for you to choose which browser you intend the newly built plugin to use. The SetBrowsers launcher is the front-end of the `setBrowsers.pl` script. See Other Plugin Development Tools.

Perform the following steps to set up these launchers:

1. Delete the existing launchers in Eclipse.
    - Go to **Run** > **External Tools** > **External Tools Configurations**.
    - Select each "Ant Build" and "Program" launcher and click the red "X" button to delete it.

2. Follow the instructions provided in Installing Launchers

In addition, the Plugin-in-a-box Eclipse workspace has set some Ant global variables. To unset these variables, go to **Windows** > **Preferences** > **Ant** > **Runtime**, and click the **Properties** tab. Remove the following properties:

- `commanderServer` property
- `gwt.home`
- `ectool`

CommanderSDK 1.0 launchers do not rely on "string substitution" variables defined in Plugin-in-a-box. Remove the following variables from **Workspace** > **Preferences** > **Run/Debug** > **String Substitution**:

- COMMMANDER_HOME
- COMMANDER_SERVER
- EXE_EXT
- GWT_HOME

The DeployPlugin and BuildAndDeployPlugin launchers default to deploying to localhost. See Ant Launchers for configuration details.

Appendix B: plugin.xml Schema

<plugin> Element Schema Definition

The following schema defines the <plugin> element for a plugin.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    commander-plugin.xsd

    This file is a W3C XML Schema that contains the declaration
    for the elements in a ElectricCommander plugin configuration file.

    Copyright (c) 2011 Electric Cloud, Inc.
    All rights reserved.
-->

<xs:schema xmlns="http://electric-cloud.com/commander/3.6/plugins"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified"
            targetNamespace="http://electric-cloud.com/commander/3.6/plugins"
            attributeFormDefault="unqualified">
    <xs:simpleType name="versionType">
        <xs:restriction base="xs:string">
            <xs:pattern value="\d+(\.\d+){0,3}"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:element name="plugin">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="key"/>
                <xs:element ref="version"/>
                <xs:choice minOccurs="0" maxOccurs="unbounded">
                    <xs:element ref="label" minOccurs="0"/>
                    <xs:element ref="description" minOccurs="0"/>
                    <xs:element ref="help" minOccurs="0"/>
                    <xs:element ref="configure" minOccurs="0"/>
                    <xs:element ref="vendor" minOccurs="0"/>
                    <xs:element ref="author" minOccurs="0"/>
                    <xs:element ref="authorUrl" minOccurs="0"/>
                    <xs:element ref="category" minOccurs="0"/>
                    <xs:element ref="commander-version" minOccurs="0"/>
                    <xs:element ref="depends" maxOccurs="unbounded" minOccurs="0"/>
                    <xs:element ref="components" minOccurs="0"/>
                    <xs:element ref="executables" minOccurs="0"/>
                </xs:choice>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="component">
        <xs:sequence>
            <xs:choice maxOccurs="unbounded" minOccurs="0">
                <xs:element name="javascript" type="xs:string"/>
                <xs:element name="request" type="requestType"/>
                <xs:element name="style" type="xs:string"/>
                <xs:element name="parameter" type="parameterType"/>
            </xs:choice>
        </xs:sequence>
        <xs:attribute type="xs:string" name="name" use="optional"/>
    </xs:complexType>
    <xs:complexType name="requestType">
        <xs:sequence>
            <xs:any processContents="skip"/>
        </xs:sequence>
        <xs:attribute type="xs:string" name="requestId" use="optional"/>
    </xs:complexType>
    <xs:complexType name="parameterType">
        <xs:sequence>
            <xs:element type="xs:string" name="key"/>
            <xs:element type="valueType" name="value"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="valueType" mixed="true">
        <xs:sequence>
            <xs:any processContents="skip"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="customType">
        <xs:sequence>
            <xs:element name="displayName" type="xs:string"/>
            <xs:element name="description" type="xs:string"/>
            <xs:element name="page" type="page"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="page">
        <xs:attribute name="pageName" type="xs:string"/>
        <xs:attribute name="definition" type="xs:string"/>
    </xs:complexType>
    <xs:element type="xs:string" name="key">
        <xs:annotation>
            <xs:documentation><![CDATA[
         Unique identifier of the plugin. Must not change between versions. Because
         this value will be used to construct directory names, it should be limited
         to characters that are safe to appear in filenames. It should be unique
         across all plugins.
            ]]>
            </xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element type="versionType" name="version">
        <xs:annotation>
            <xs:documentation>Plugin version in the form major.minor?.patch?.
         Versions will be sorted lexicographically.</xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element type="xs:string" name="label">
        <xs:annotation>
            <xs:documentation>The name of the plugin as it should be displayed in the
         UI. If not specified, assumed to be the same as &lt;key>.</xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element type="xs:string" name="description">
        <xs:annotation>
            <xs:documentation>A short description of the plugin.</xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element type="xs:string" name="help">
        <xs:annotation>
            <xs:documentation>Location of help for the plugin. Interpreted as a
         relative path based at plugin's root. Should contain a
         componentContainer.</xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element type="xs:string" name="configure">
        <xs:annotation>
            <xs:documentation>Location of configuration for the plugin. Interpreted
        as a relative path based at plugin's root. Should contain a
        componentContainer.</xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element type="xs:string" name="vendor">
        <xs:annotation>
            <xs:documentation>The author of the plugin. (deprecated in favor of
         author)</xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element type="xs:string" name="author">
        <xs:annotation>
            <xs:documentation>The vendor of the plugin.</xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element type="xs:string" name="authorUrl">
        <xs:annotation>
            <xs:documentation>The vendor of the plugin.</xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element type="xs:string" name="category">
        <xs:annotation>
            <xs:documentation>A short keyword that specifies the general category of
         plugin.</xs:documentation>
        </xs:annotation>
    </xs:element>
    <xs:element name="commander-version">
```

```
        <xs:annotation>
            <xs:documentation>The range of ElectricCommander server versions a plugin
         works with. The install will fail if the server's version is outside of the
         specified range. If min (or max) is omitted, the range is considered to
         include all earlier (or later) versions. The commander-version element can
         be omitted if no version checks are required.</xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:simpleContent>
                <xs:extension base="xs:string">
                    <xs:attribute type="versionType" name="min" use="optional"/>
                    <xs:attribute type="versionType" name="max" use="optional"/>
                </xs:extension>
            </xs:simpleContent>
        </xs:complexType>
    </xs:element>
    <xs:element name="depends">
        <xs:annotation>
            <xs:documentation>A list of plugin keys that this plugin depends on. The
         install will fail if the listed plugins are not already installed. If the
         optional min attribute is specified, the server will only match plugins at
         least as recent as the specified version.</xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:simpleContent>
                <xs:extension base="xs:string">
                    <xs:attribute type="versionType" name="min" use="optional"/>
                </xs:extension>
            </xs:simpleContent>
        </xs:complexType>
    </xs:element>
    <xs:element name="components">
        <xs:annotation>
            <xs:documentation>.</xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:choice maxOccurs="unbounded" minOccurs="0">
                <xs:element name="component" type="component" maxOccurs="unbounded"
         minOccurs="0"/>
                <xs:element name="customType" type="customType" maxOccurs="unbounded"
         minOccurs="0"/>
            </xs:choice>
        </xs:complexType>
    </xs:element>
    <xs:element name="executables">
        <xs:complexType>
            <xs:sequence maxOccurs="unbounded">
                <xs:element name="pattern" type="xs:string"/>
            </xs:sequence>
            <xs:attribute name="useDefaults" type="xs:boolean"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Appendix C: ElectricCommander::View API

```
NAME
        ElectricCommander::View - manipulate a tabbed view definition document

SYNOPSIS
        require ElectricCommaner::View;

        my $view = new ElectricCommaner::View($xml);

        # Add a tab or subtab
        $view->add("MyTab", { url => 'pages/MyPlugin/aPage' });
        $view->add(["Administration", "SubTab"], {
        url => 'pages/MyPlugin/aPage'
        });

        # Hide a tab
        $view->add("Jobs", { show => 0});

        # Remove a tab
        $view->remove("MyTab");

DESCRIPTION
        A View is used to manipulate a set of tabs contained in an
        ElectricCommander view definition document. In the descriptions below,
        LOC specifies either a scalar top level tab label or a reference to a
        two element array that contains a top level tab label and a subtab
        label. For example "Home" refers to the top level home page, whereas:

        [ "Administration", "Server" ]

        refers to the Server tab under the Administration tab.

METHODS
    new ( XML )
        Parse the specified XML document containing the view to manipulate.

    add( LOC , OPTS )
        Add a tab to the view at the specified LOC. OPTS is a hash containing
        any of the optional attributes of a tab:

        url The url for the tab. This should be a path relative to the
            commander root url (e.g. 'pages/Foo-1.0/aPage').
```

```
position
    The positive integer position for the tab. If not specified and
    the tab has not been added previously, the tab will be appended.

show
    Specifies whether the tab should be visible. Defaults to 1.

accesskey
    A single character access key to use for a top level tab. This
    value is ignored for subtabs.

remove ( LOC )
    Remove the specified tab and all of its contents.

asXML
    Return the XML document that represents the view definition.

base
base ( BASE )
    Returns or sets the base view that this view will inherit.

get ( LOC )
    Return the contents of the given tab as a hash ref or undef if there is
    no such tab.
```

```
perl v5.10.1                        2010-10-28        ElectricCommander::View(3)
```