Module – Create Custom Processor

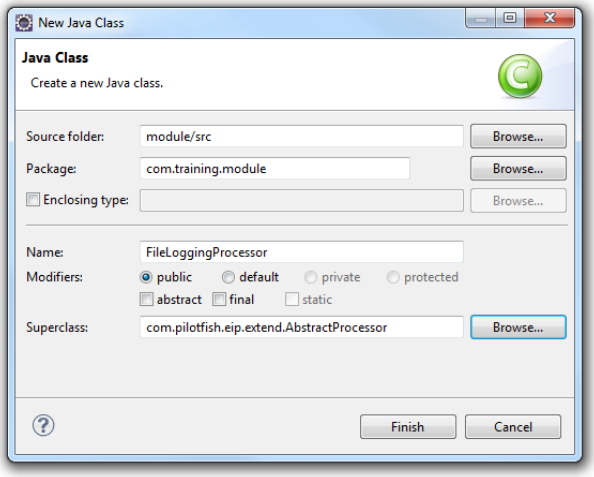
Overview

This document will describe how to create a custom 'Module' that logs the contents of the stream.

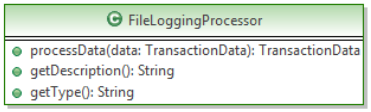
Once setup you will be able to create your own custom modules and deploy them to your PilotFish Console and eventually to Connect5 Platform.

Module

Start by creating a new Class called "com.training.module.FileLogProcessor" that extends "com.pilotfish.eip.extend.AbstractProcessor":



This will generate a skeleton class with the following methods:

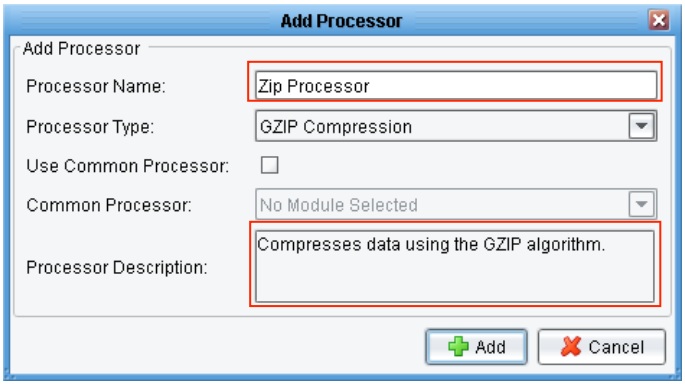


In the processData() method, rename the argument from “arg0” to “data”.

Implement the getDescription() and getType() methods as follows:

```
@Override
public String getDescription() {
     return "Log Transaction Data to file in existing
folder.";
}

@Override
public String getType() {
     return "** File Logger";
}
```

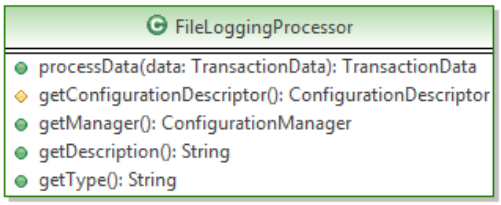
Once we add the processor to PilotFish in the next lab the Type and Description field get used when adding a module:



We will see later that all Connect5 modules start with “* <type>” so they show at the top of the list. All custom modules should start with two-stars; “** <type>”.

Configuration

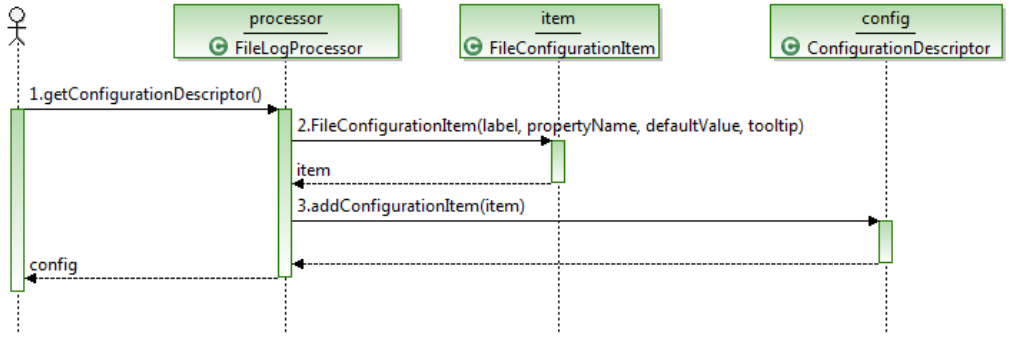
Since we want the user to be able to specify where to log the contents of the file we will need to add a field to the protected com.pilotfish.utils.ccp.ConfigurationDescriptor.



To do this override the super's getConfigurationDescriptor() method.

Always remember to call the super's method then add your own ConfigurationItem. Be aware that the call to super may return null, so you need to code defensively for that. A null simply means that the base class did not indicate that there are any necessary user configuration elements for the module.

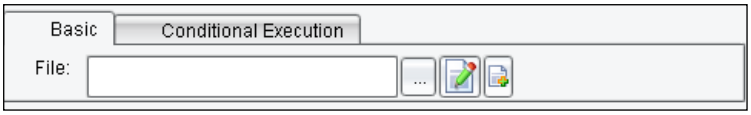
In this case we are going to instantiate a new com.pilotfish.utils.ccp.FileConfigurationItem, then add it to the ConfigurationDescriptor.



The code will look something like this:

```
@Override
protected ConfigurationDescriptor getConfigurationDescriptor() {

      ConfigurationDescriptor config = super.getConfigurationDescriptor();

      // if super does not allocate a CD that's okay and we can do it.
      if(config == null) {
      config = new ConfigurationDescriptor(false);
      }

      // parms
      //   - label
      //   - PF variable which will hold the user-entered data
      //   - default value
      //   - tool tip
      //
      // there are other constructors which allow you to specify the tab name
      //   and required-ness of the field
      PanelItem item = new FileConfigurationItem("File", "FilePath", "",
"Specify the path for the log file. Processor assumes the folder hierarchy
has already been created.");
      config.addConfigurationItem(item);

      return config;
}
```

Once we add our new module into the PilotFish Console (again, later lab) we would see the following result:



Once we configured the the module the route.xml would look like this:

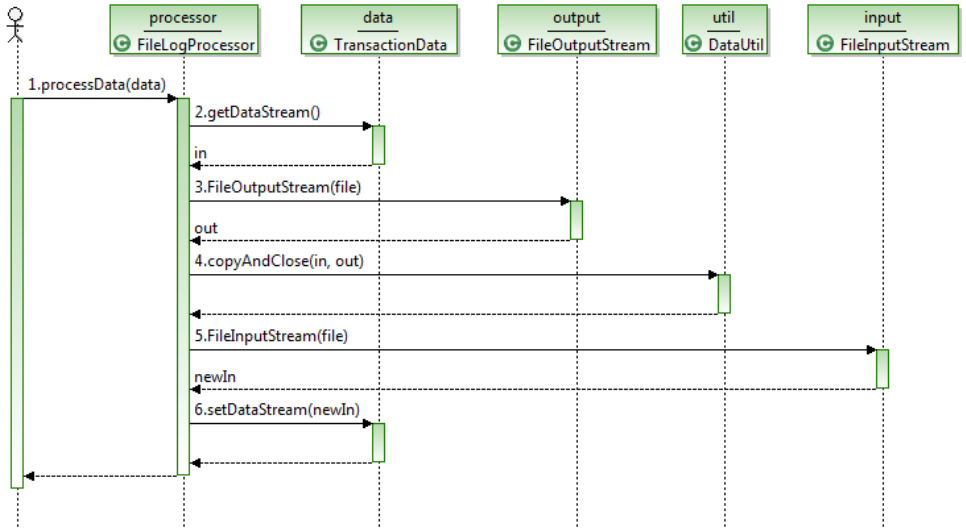
```
<Processor class="com.training.module.FileLogProcessor" name="File Processor">
  <ModuleConfig>
    <ExecuteProcessor>true</ExecuteProcessor>
    <FilePath>../log/output.txt</FilePath>
  </ModuleConfig>
</Processor>
```

Execution

When a Processor gets executed its processData() method is called.

The number one rule for any Module is to hand a new InputStream to the next Module in line.

In our module we are going to write to disk and then read from it.



The first iteration of the processData() code should now look like this (though it won't compile):

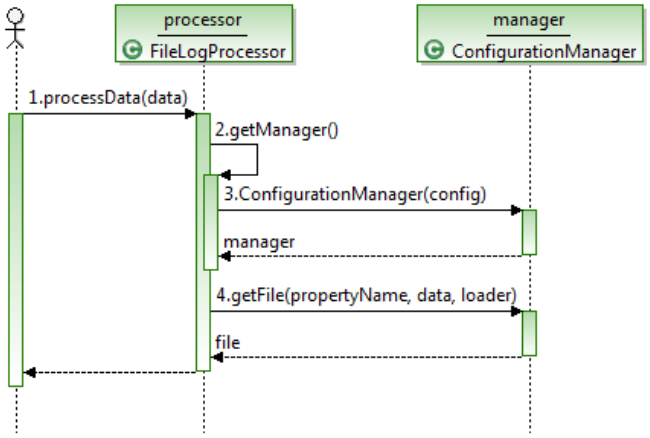
```
// get the handle to the output file
File file = new File();
FileOutputStream out = new FileOutputStream(file);

// write the current data to the file. we must copy the stream because
stream
//   processing is destructive.
DataUtil.copyAndClose(data.getDataStream(), out);

// finally, get the data back and use that as the response. we must
remember
//   we're obligated to pass a usable stream to the next stage.
FileInputStream in = new FileInputStream(file);
data.setDataStream(in);

return data;
```

The last piece of the puzzle is to wire up the configuration with execution.

While the processData() implementation does not need a method to get at the ConfigurationManager, we'll create it now because we will need it later on. Create a new method getManager() that exposes the protected ConfigurationManager object:

```
public ConfigurationManager getManager() {
      return this.manager;
}
```

Next we need to call the above method to retrieve the file parameter.



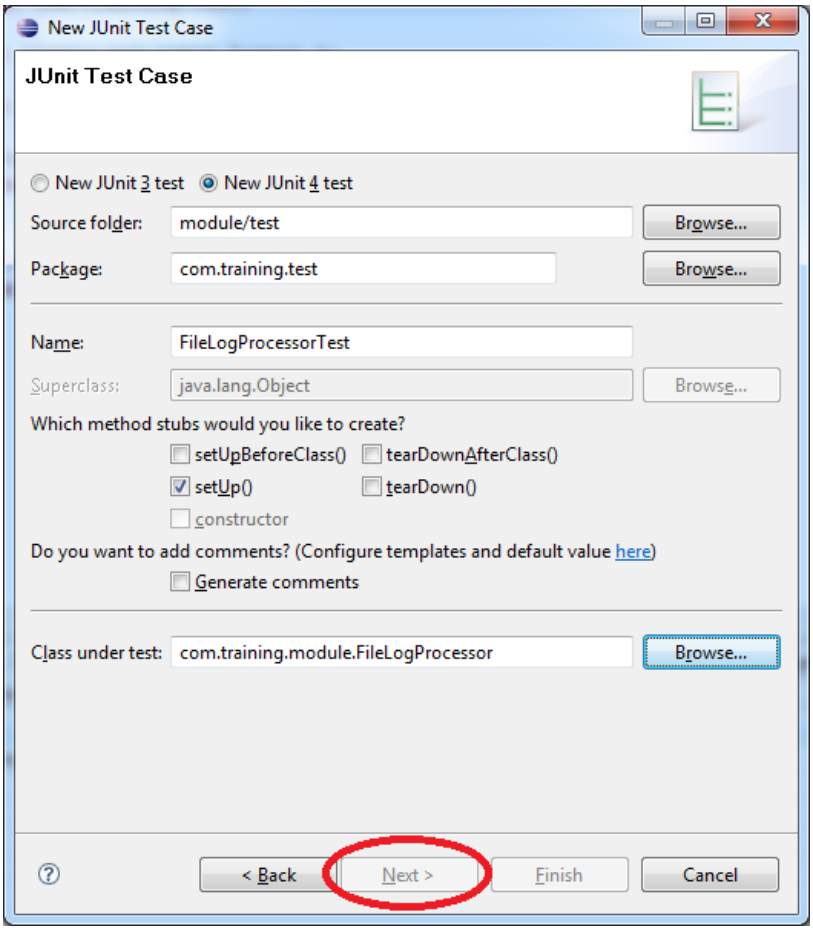
The final code inside processData() will now look like this:

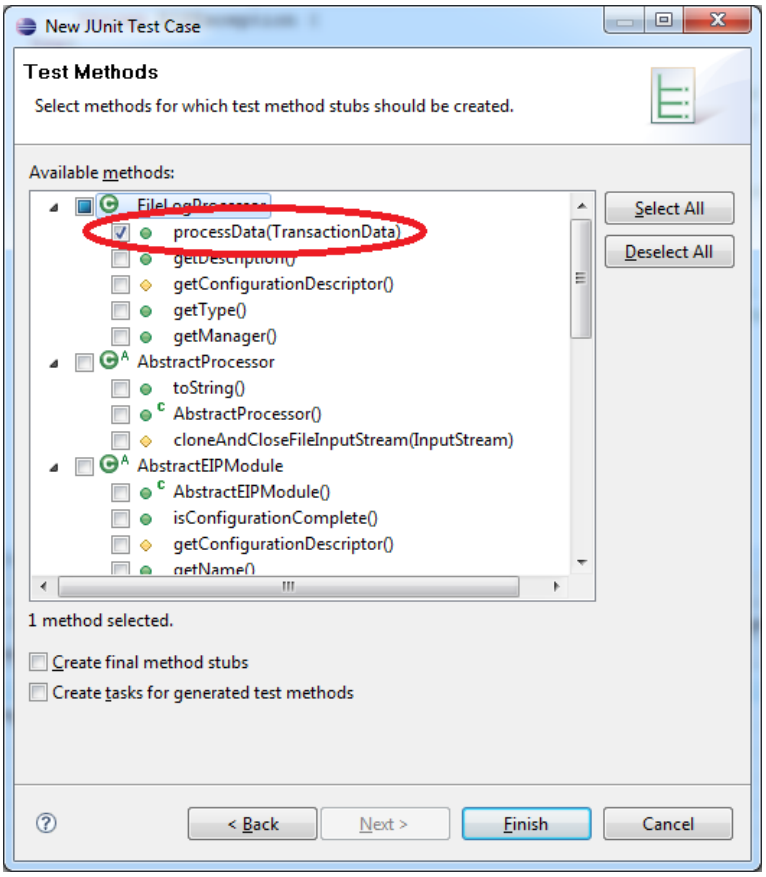
```
try{
	// get a handle to the output file
	// the ResourceLoader is available to us via the base class
	File file = getManager().getFile("FilePath", data, loader);
	FileOutputStream out = new FileOutputStream(file);

	// write the current data to the file. we must copy the stream because
	//   stream processing is destructive.
	DataUtil.copyAndClose(data.getDataStream(), out);

	// finally, get the data back and use that as the response. we must
	//   remember we're obligated to pass a usable stream to the next
	//   stage.
	FileInputStream in = new FileInputStream(file);
	data.setDataStream(in);
}catch (Exception e) {
	throw new EIPException(e.getMessage());
}

	return data;
```

Testing

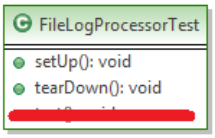
To test this new module we will need to create a new Unit Test and make use of the ConfigurationManager now exposed by the Processor.

Create a new "JUnit Test Case" class called "com.training.test.FileLogProcessorTest" and specify the class under test as "com.training.module.FileLogProcessor".





This will create a new class with the following signature:



Implement testProcessData() method that will instantiate a new instance of our module class, configure it, then execute it by passing in some dummy data, but first we need to set up the data.

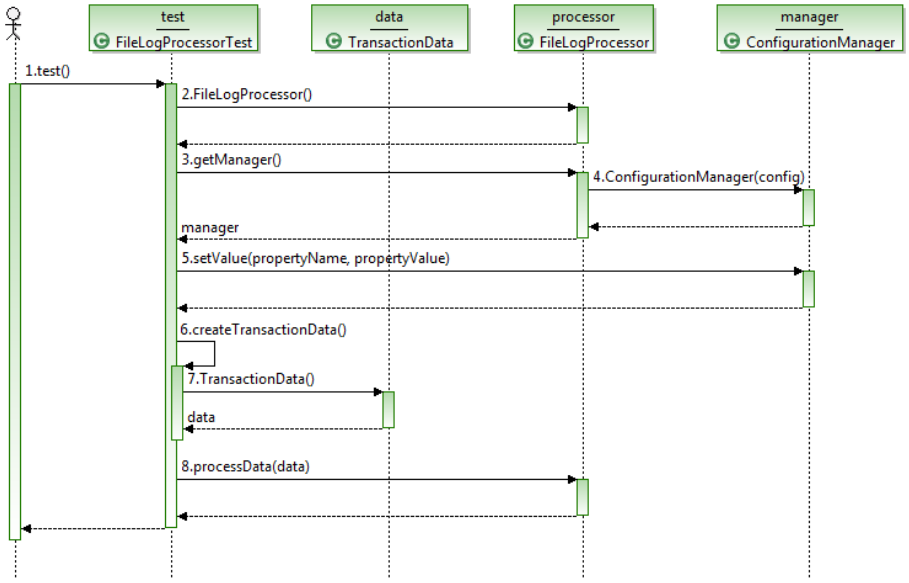
First, create a member variable on the Test class:

```
TransactionData transactionData = null;
```

That's the object we'll inflate in our setUp() method:

```
@Before
public void setUp() throws Exception {

	// the processData() method takes an instance of TransactionData
so we
	//   will fabricate that in the setUp() method
	transactionData = new TransactionData();

	String fakeString = "This is my fake TransactionData stream";
	byte[] byteArray = fakeString.getBytes();
	InputStream inputStream = new ByteArrayInputStream(byteArray);


	transactionData.setDataStream(inputStream);
}
```

Now, we can write our test. All steps together loosely look like the following sequence diagram:



So the code in testProcessData() ends up looking like this:

```
@Test
public void testProcessData() throws EIPException {

      // instantiate our class to be tested
      FileLogProcessor processor = new FileLogProcessor();

      // in the same way the user would have filled in the "File" field we
      //   will fabricate that.
      File cd = new File(".");
      processor.getManager().setValue("FilePath", cd.getAbsolutePath() +
"/test/resources/output.log");

      processor.processData(transactionData);
}
```

Prior to running the test we just need to be sure and create the folder structure up to the file name. To do this just add folder under test named “resources” using Eclipse.

We can now execute our test and validate the output.