Lab 1

SwitchYard

Lab Steps

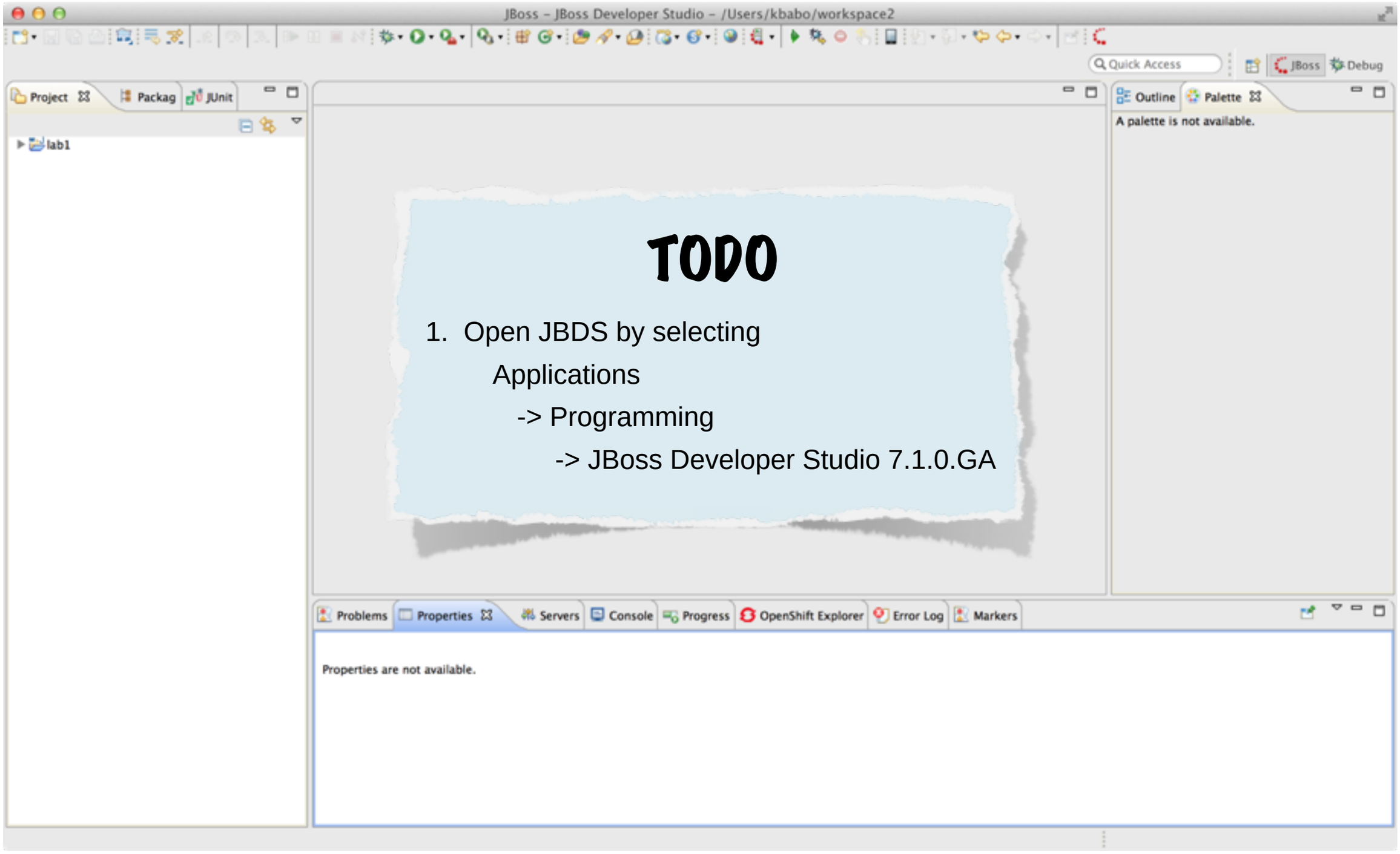
- Step 0 : Getting Started
- Step 1 : Component Service
- Step 2 : Component Reference
- Step 3 : Camel Routing
- Step 4 : Reference Binding
- Step 5 : Transformation
- Step 6 : Service Binding
- Step 7 : Implement RESTful status service

Step 0

Getting Started

Goals

- *Import the lab project*
- *Introduce key editor concepts*


JBoss – JBoss Developer Studio – /Users/kbabo/workspace2
Quick Access
JBoss
Debug
Project
Packag
JUnit
lab1
Outline
Palette
A palette is not available.
TODO
1. Open JBDS by selecting
Applications
-> Programming
-> JBoss Developer Studio 7.1.0.GA
Problems
Properties
Servers
Console
Progress
OpenShift Explorer
Error Log
Markers
Properties are not available.

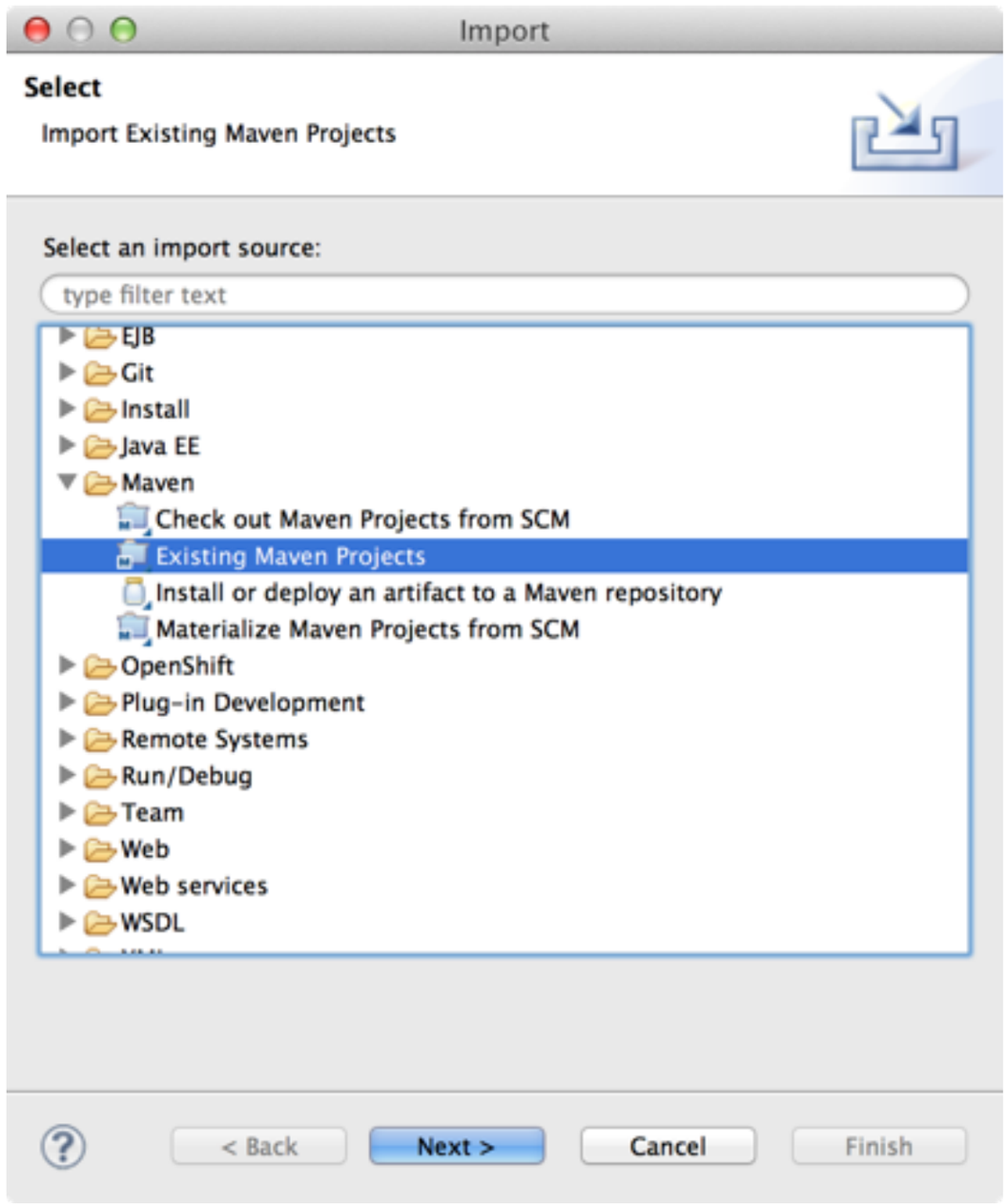
Importing Lab 1

TODO

1. File -> Import ... from the JBDS menu.

2. Select Maven -> Existing Maven Projects

3. Click Next



Importing Lab 1

TODO

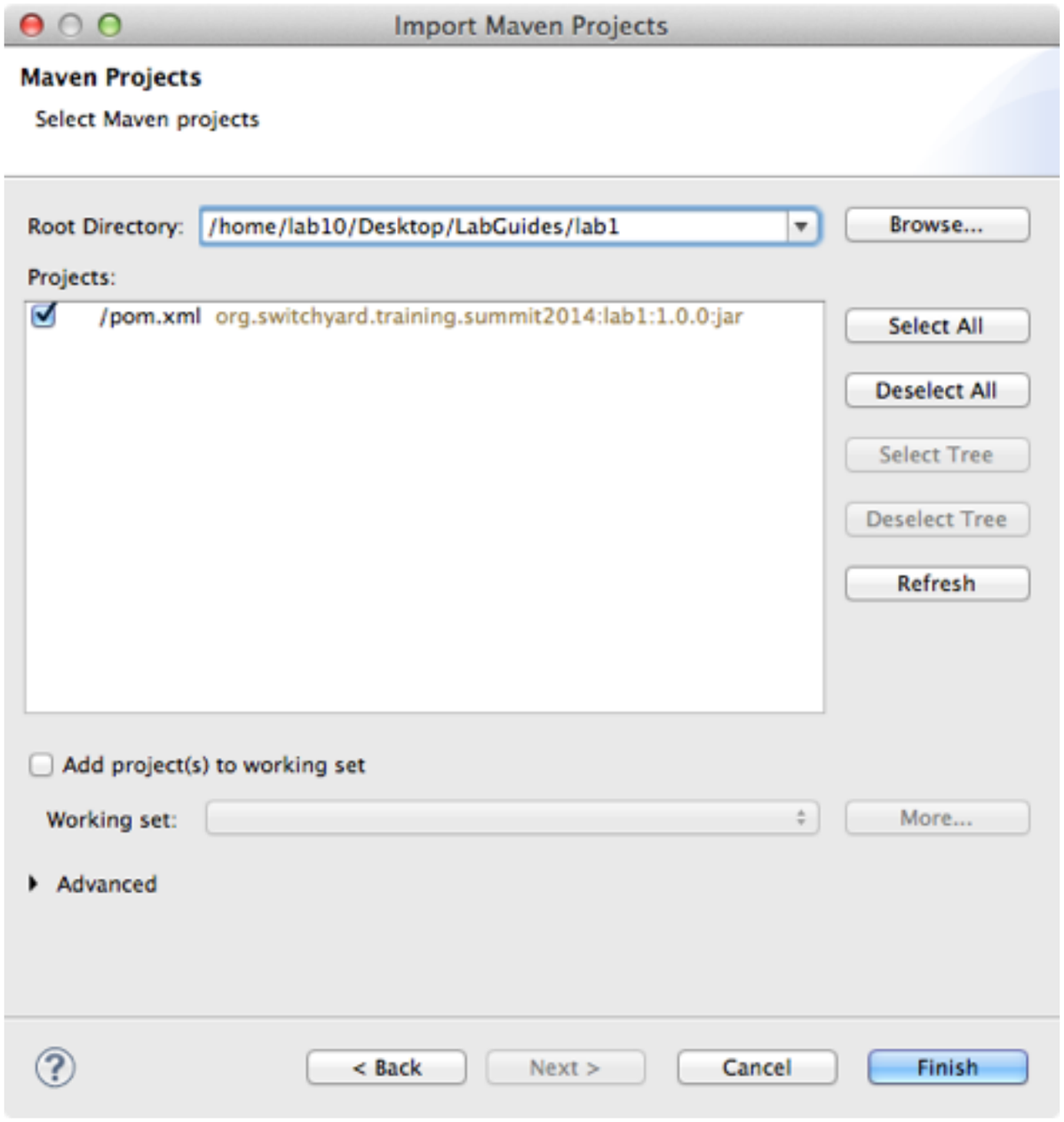
1. Click Browse ... and navigate to the location of lab1. For example:

```
/home/lab10/Desktop/LabGuides/lab1
```

2. Make sure the pom.xml is checked for:

```
org.switchyard.training.summit2014:lab1
```

3. Click Finish



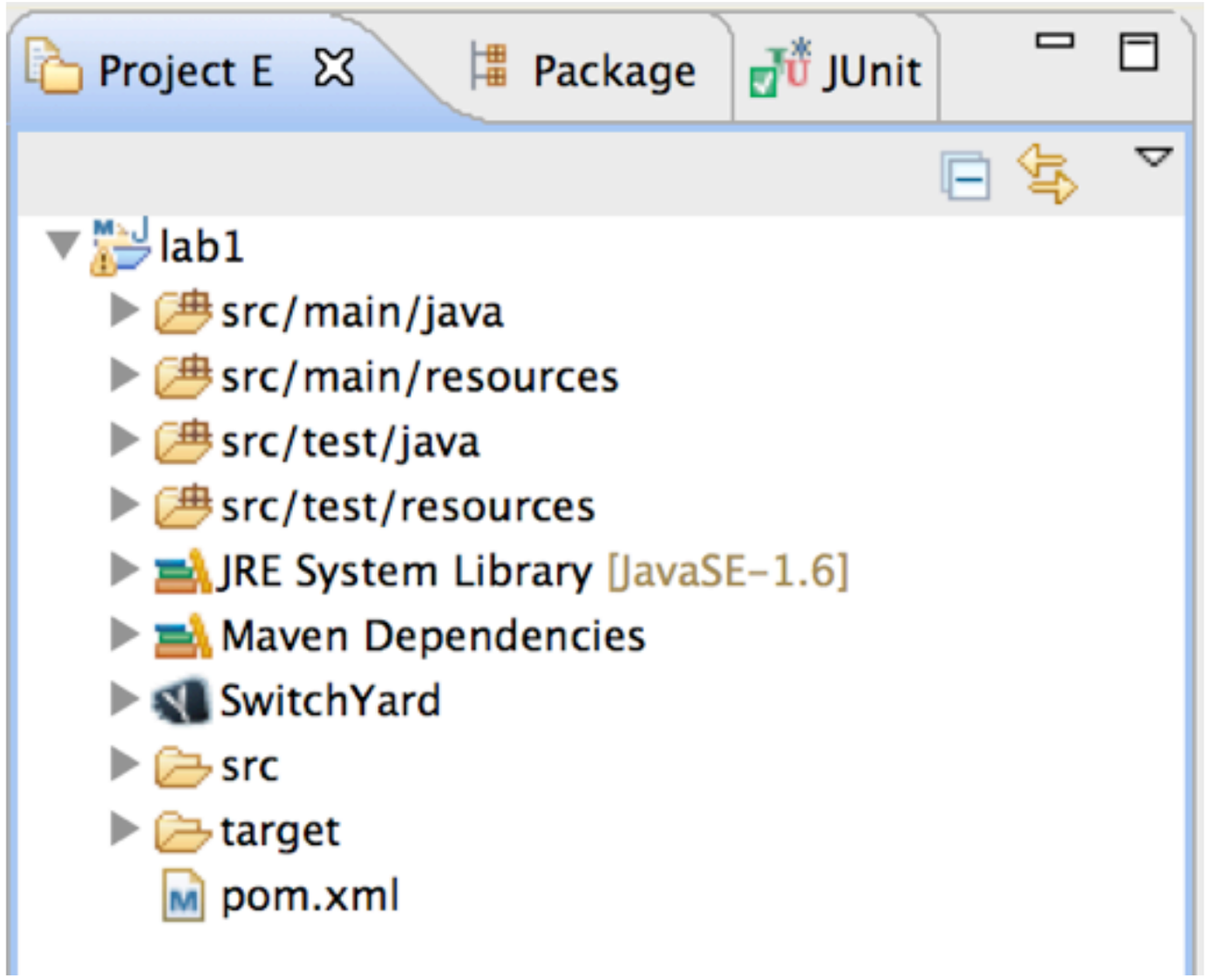
Application Structure

TODO

1. Go to the Project Explorer in the left frame and expand the lab1 project node.

FYI

The next set of slides will examine each section of the project in detail.



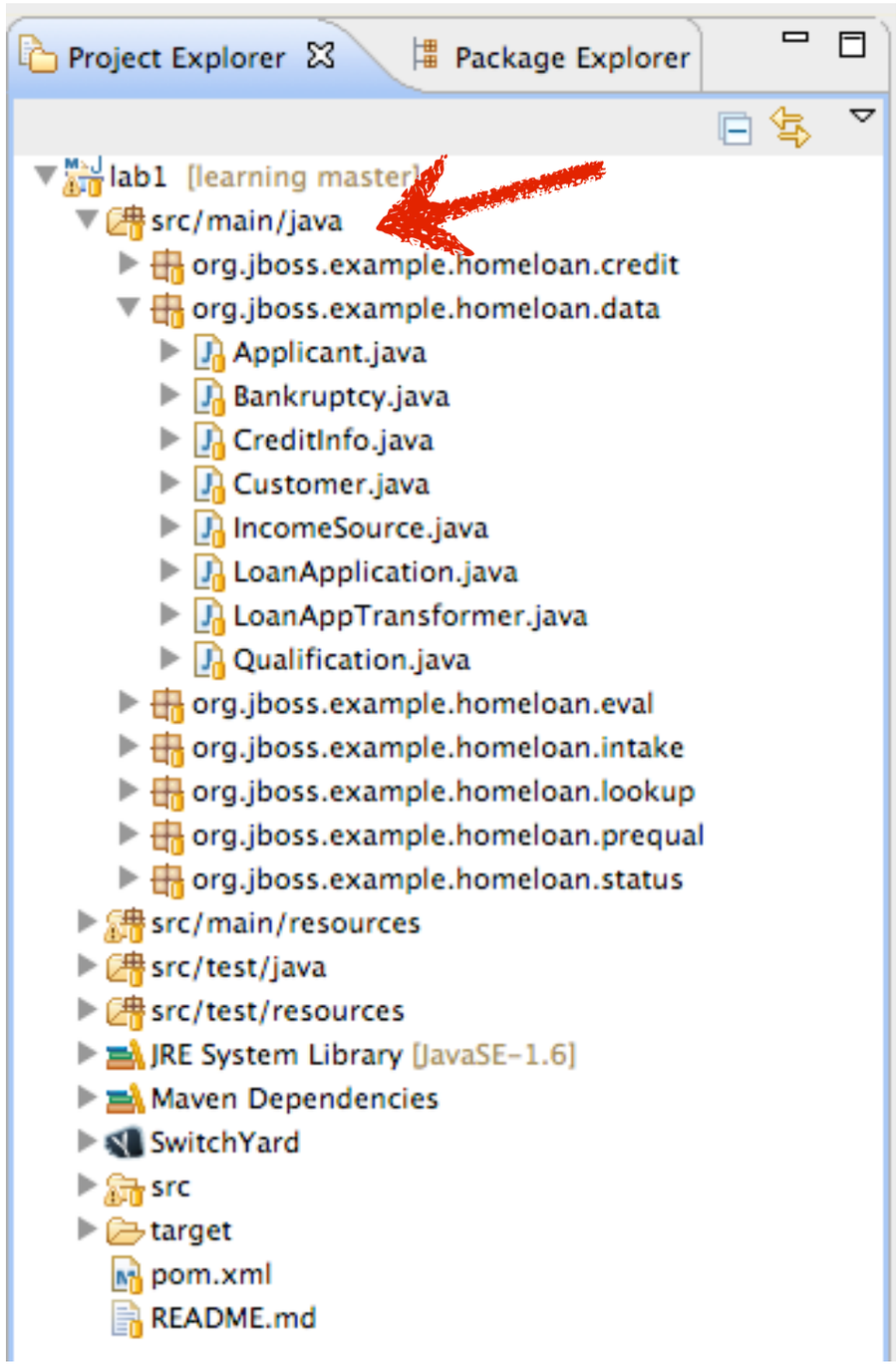
src/main/java

FYI

This is the home of Java files which provide your application's logic. These files are packaged with your application for deployment.

Examples of files you will find here:

- *Java service interfaces*
- *Camel Java DSL routes*
- *CDI Beans*
- *Java Transformers*



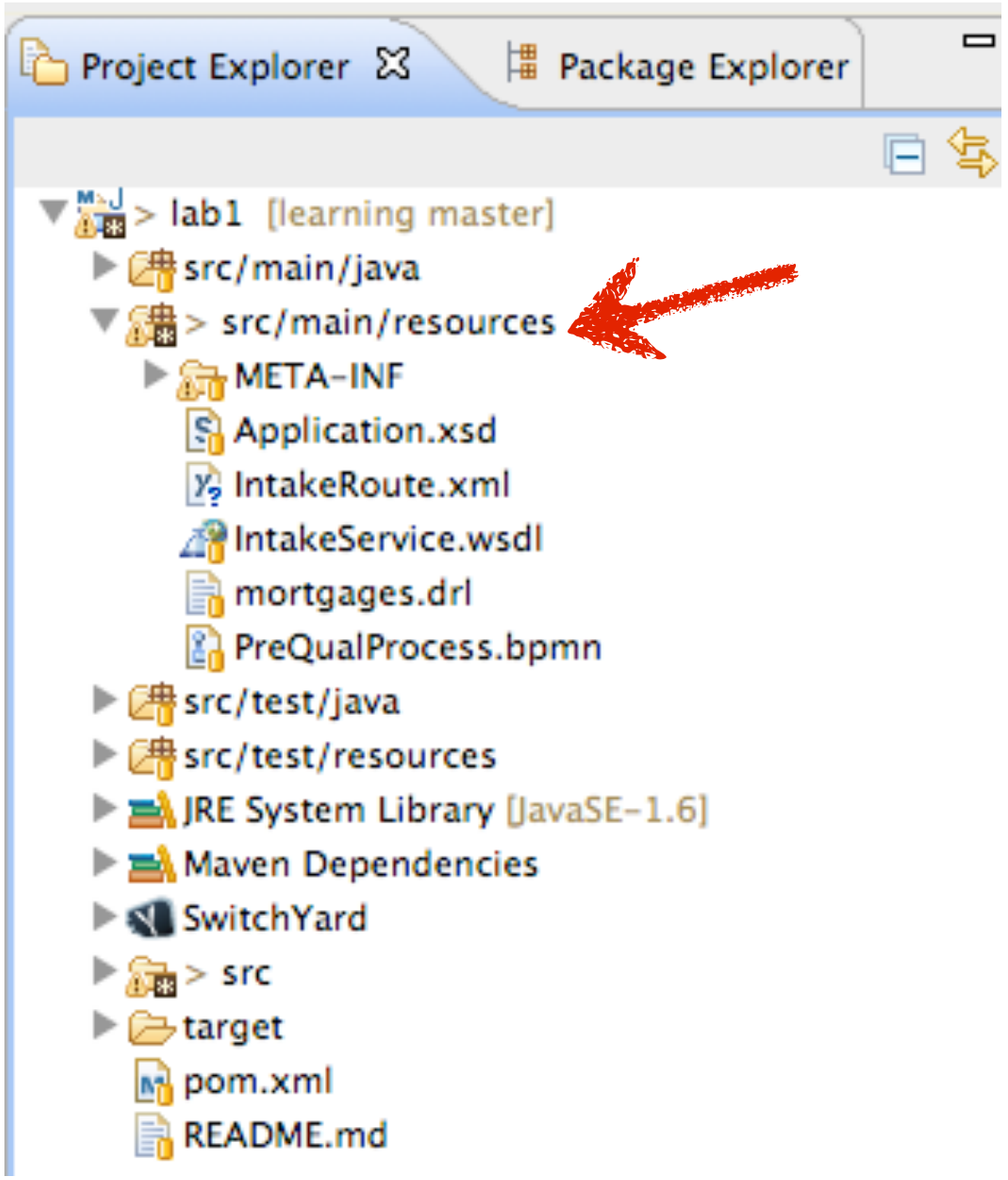
src/main/resources

FYI

Non-Java resources that will be packaged with your application go here.

Examples of files you will find here:

- *SwitchYard application descriptor (switchyard.xml)*
- *Camel XML route definitions*
- *Transformers (XSLT, Smooks)*
- *Drools rule definitions*
- *Process definitions (BPMN 2, BPEL)*
- *WSDL*
- *Schema*



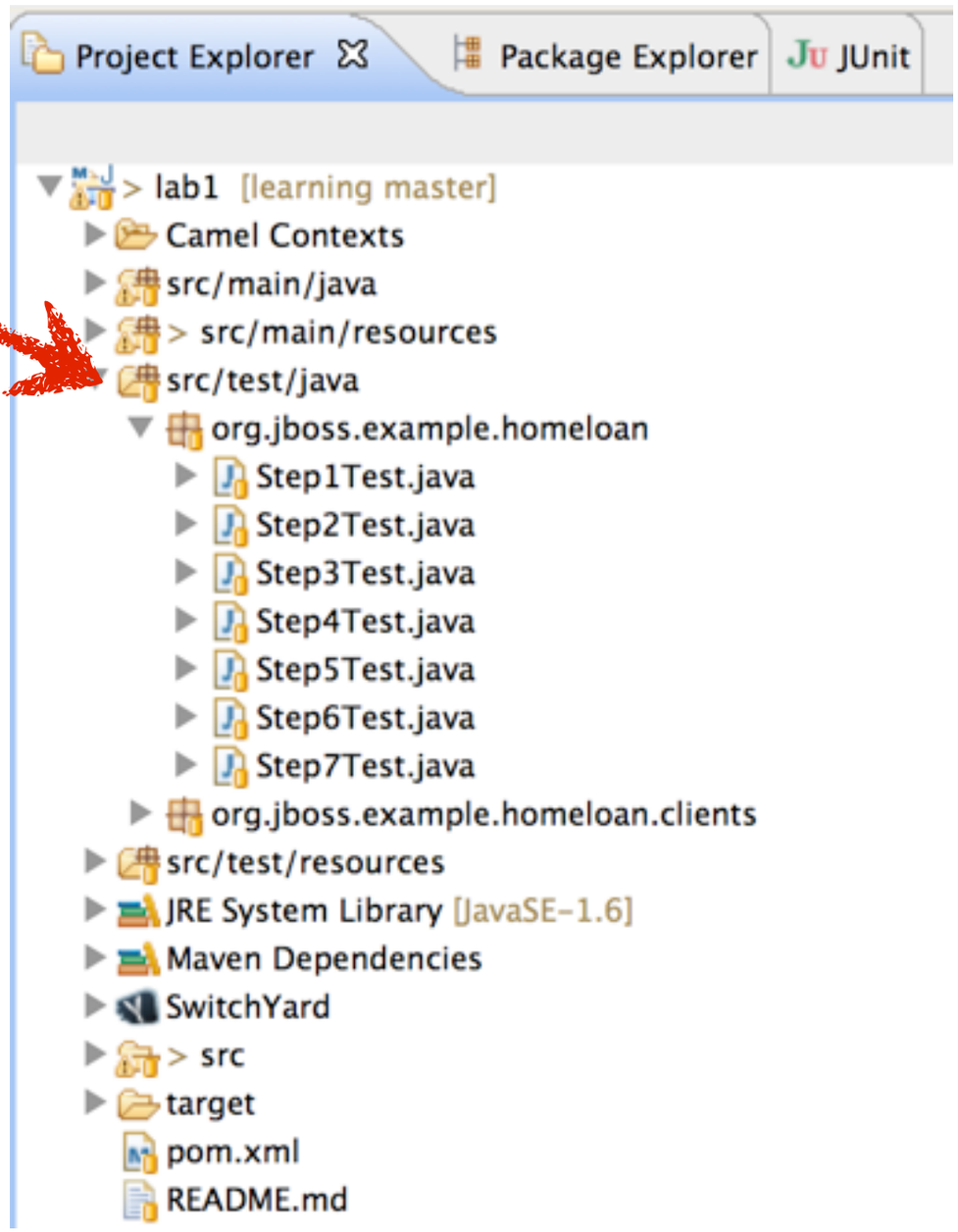
src/test/java

FYI

All Java classes related to testing your application go in src/test/java.

Examples of files you will find here:

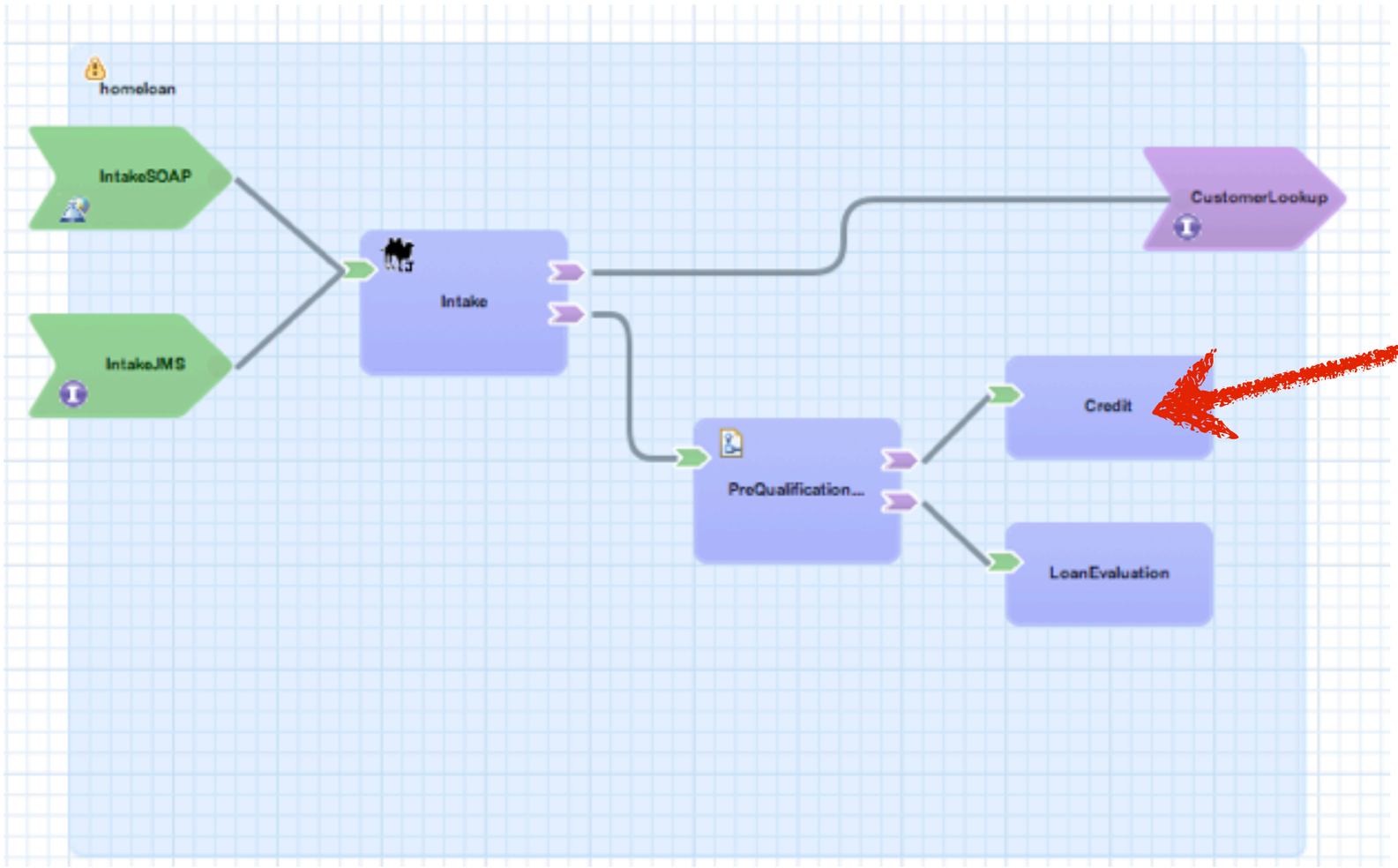
- *Service unit tests*
- *Remote test clients*
- *Ancillary test classes (e.g. mock web services)*

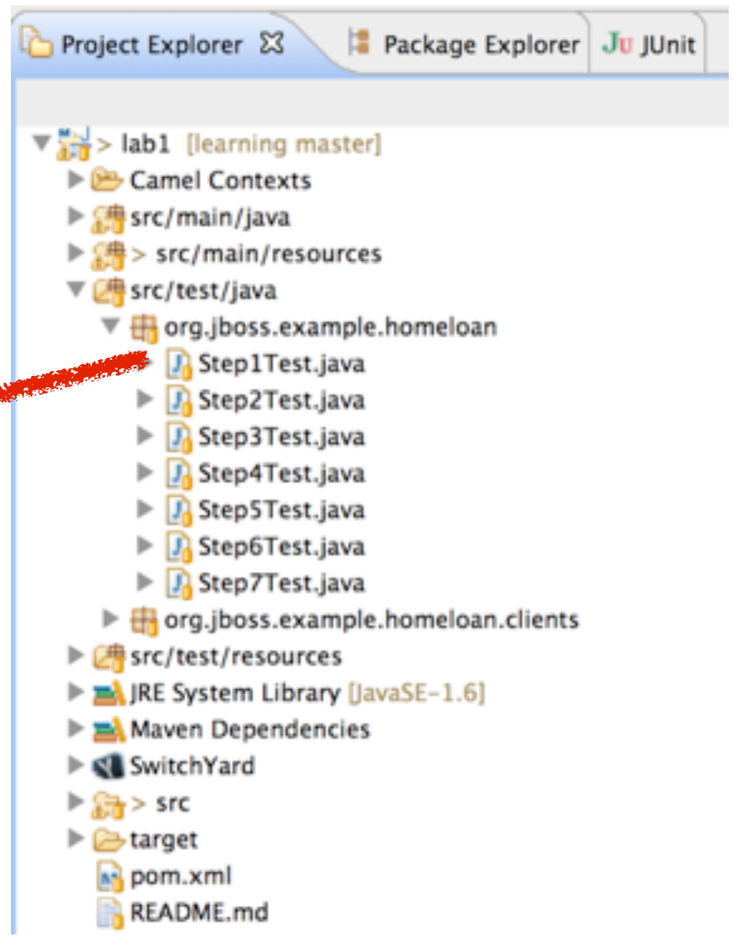


Unit Testing Services

FYI

SwitchYard allows you to unit test individual services in your application as you build it. For example, Step1Test contains test methods which invoke CreditService directly to verify that it's behavior matches requirements.





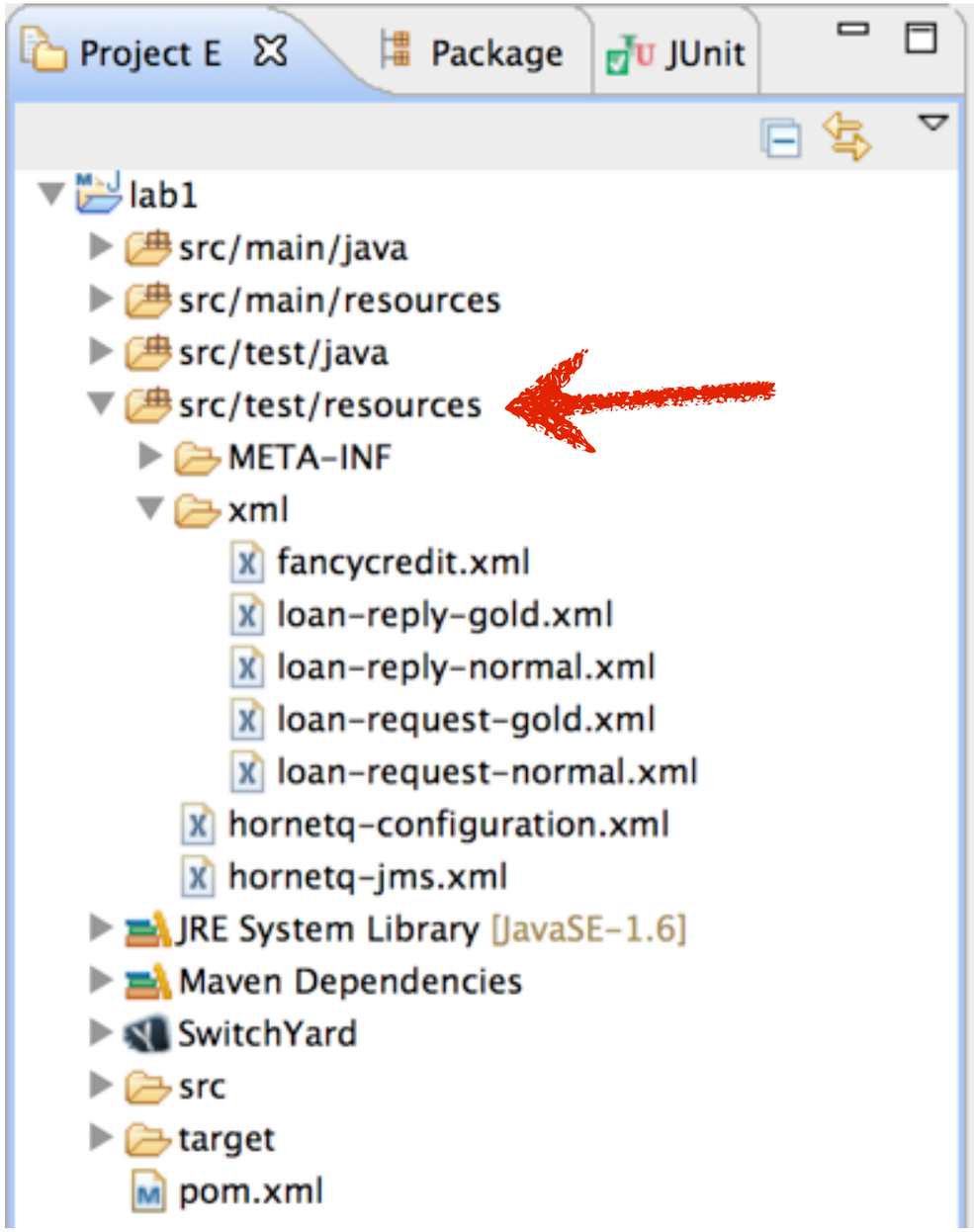
src/test/resources

FYI

Any resources that are used during test execution, but are not included in the application, belong in src/test/resources.

Examples of files you will find here:

- *Test payloads*
- *Expected results for test assertions*
- *JMS destinations used for testing*
- *log settings used for test execution*



SwitchYard!!!!

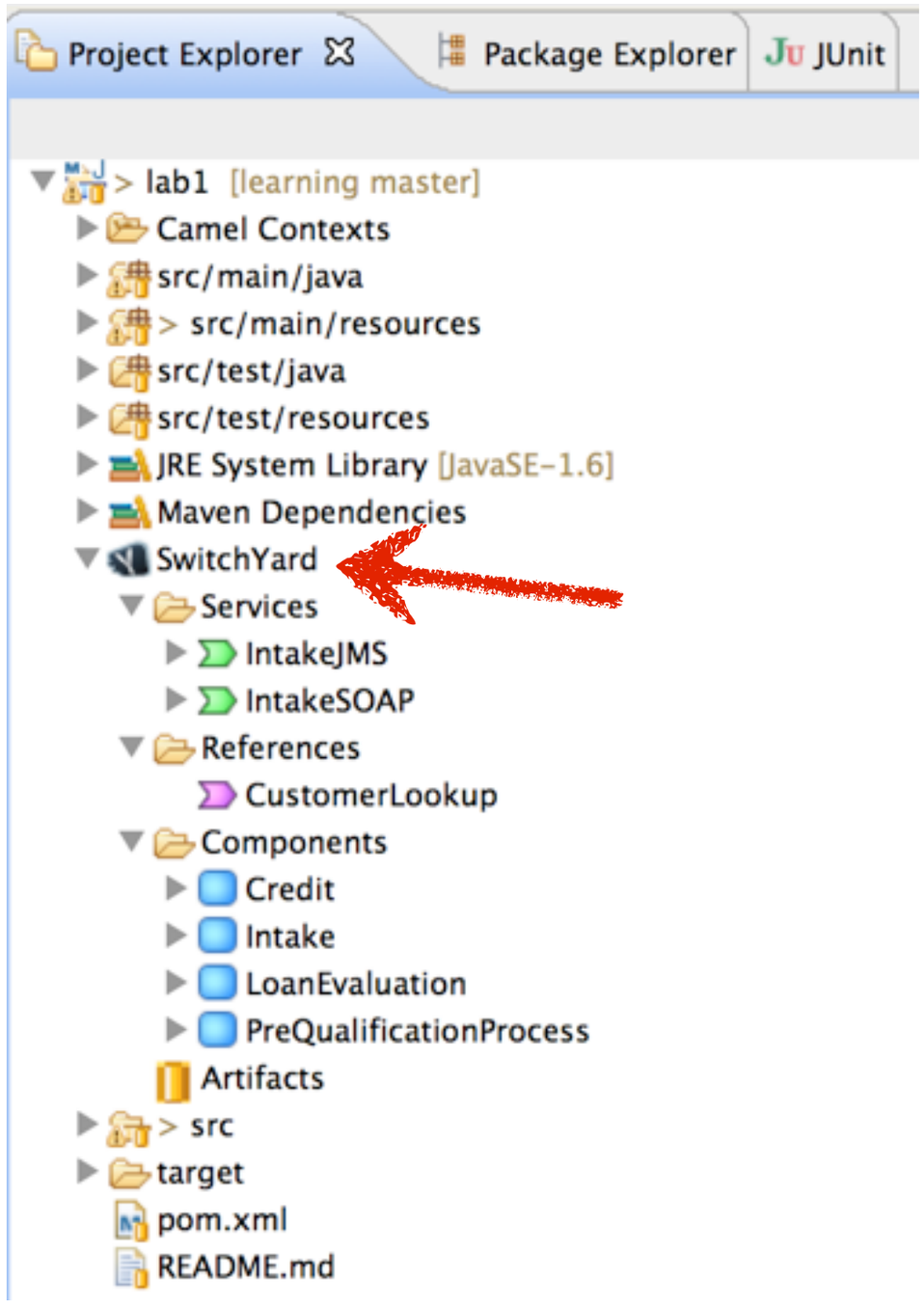
FYI

The SwitchYard node provides two functions in the Project Explorer:

- *An 'outline' style view of the application configuration (services, references, components, etc.)*
- *Shortcut to the SwitchYard visual application editor via a double-click.*

TODO

1. Double-click the SwitchYard node to begin your journey down the rabbit hole!



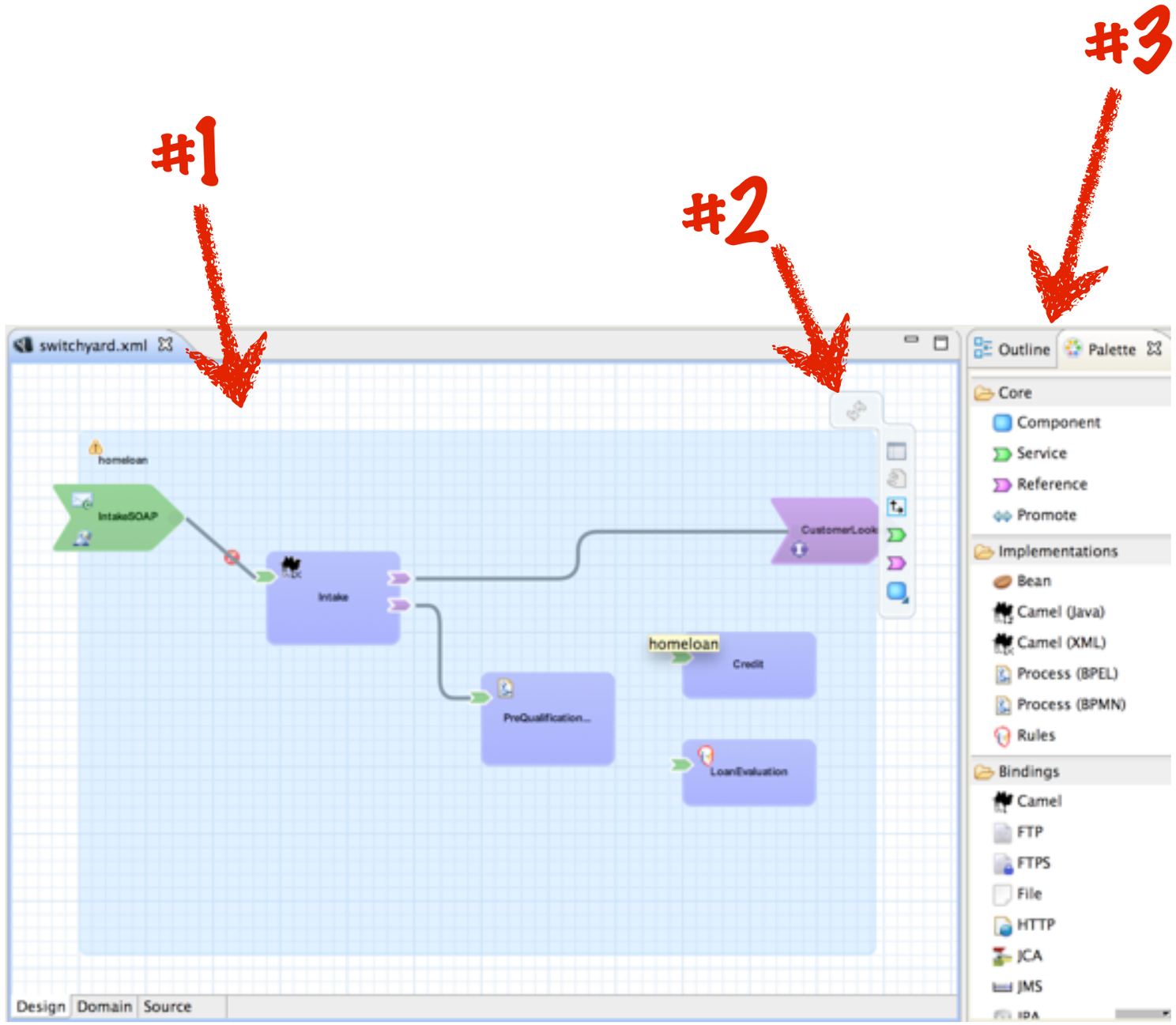
Visual Editor

FYI

*#1 : The **canvas** provides a visual representation of the structure and relationships in a SwitchYard application.*

*#2 : The **button bar** appears when you hover over items on the canvas and provides context-sensitive options for each item.*

*#3 : The **palette** provides a set of application building blocks which can be dragged onto the canvas to create an application.*



Wiring

FYI

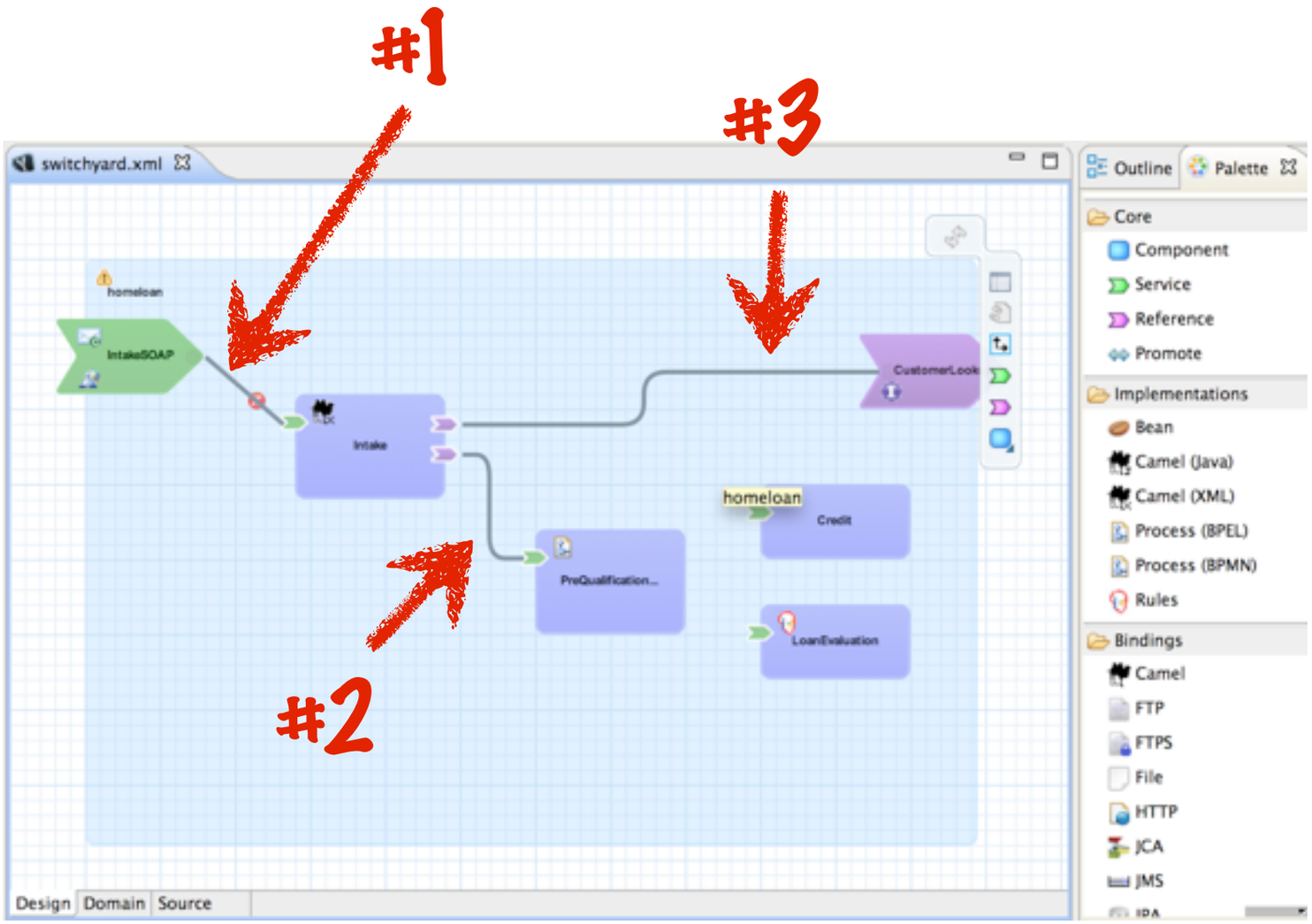
The black connector lines depict wiring which defines the relationship between services and references.

#1 : A component service is wired to a composite service to make it available outside the application through a binding.

#2 : A component reference is wired to a service provided in the local application.

#3 : A component reference is wired to a composite reference to consume an external service through a binding.

Each wire represents a message path where runtime interceptors provide functionality such as transformation, validation, policy enforcement, etc.



Living Large with the Button Bar

FYI

The button bar is really convenient when interacting with the visual editor. Just hover over the portion of the diagram you want to interact with and choose an option from the button bar.

This page provides a key to the more frequently used buttons on the button bar. The properties button is used the most by far, so definitely make note of that one. If the other buttons don't make sense now, don't worry they will soon.

NOTE: the focus for the button bar can be kinda touchy at times when there are multiple areas of focus close together. Do or do not. There is no try.

Opens properties view

Edit the interface for a service or reference.

Promotes a component service.

Generate WSDL from a Java interface.

Adds a reference to a component or composite.

Generate Java interface from a WSDL.

Adds a service to a component or composite.

Create a unit test class for a service.

Adds a binding to a composite service or reference.

Step 1

Component Service

Goals

- *Implement a component service using a CDI Bean.*
- *Run unit tests to verify changes.*

Component Service

Contract

FYI

All services in a SwitchYard application have a contract. You can view the contract for any service by double-clicking on the green service icon.

TODO

1. Double-click on the green component service icon *(see red arrow)* for CreditService and view the contract.

QualificationService

interface.java

CreditService.java

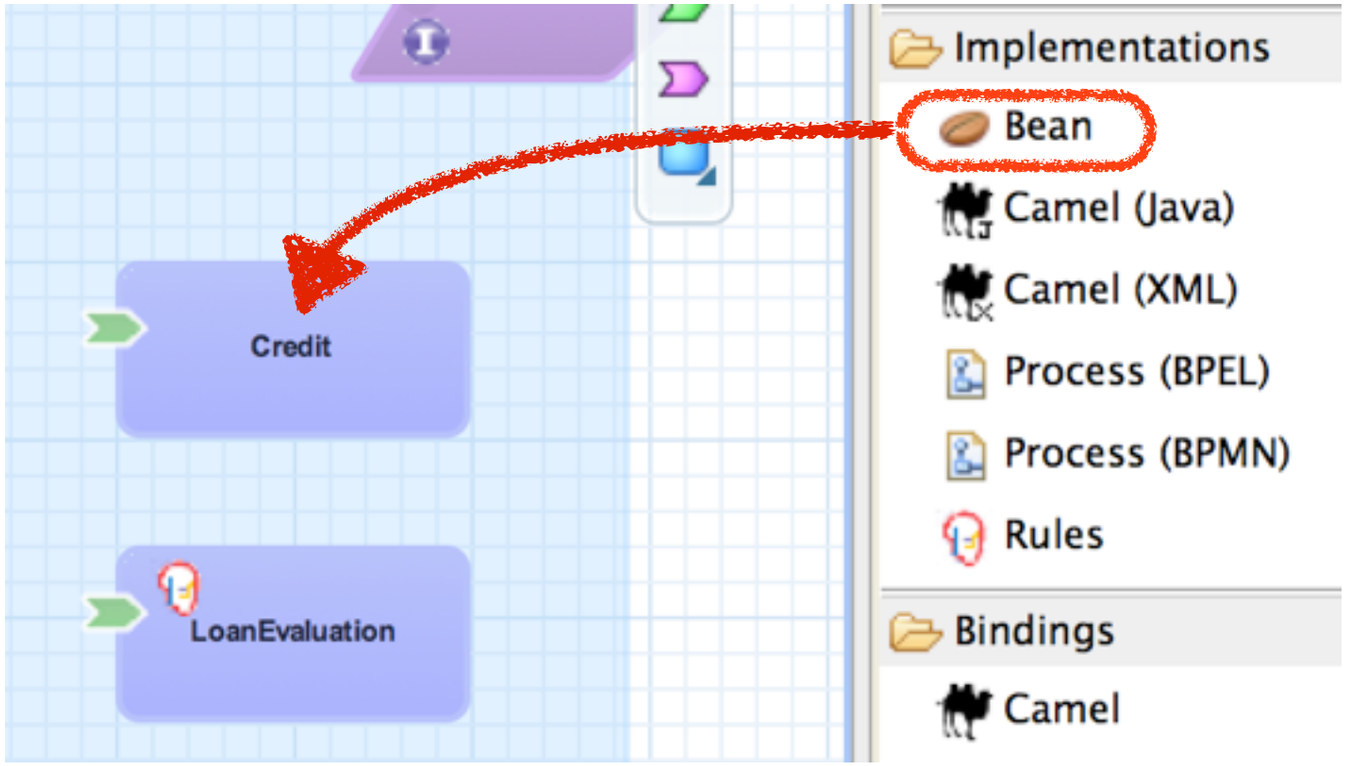
```
package org.jboss.example.homeloan.credit;

import org.jboss.example.homeloan.data.Applicant;


public interface CreditService {

    CreditInfo creditCheck(Applicant applicant);
}
```

Step 1

CreditService Implementation

FYI

Service logic is provided in SwitchYard using an 'implementation'. Implementation types include: Java/CDI, Camel, BPEL, BPMN 2, and Drools.



TODO

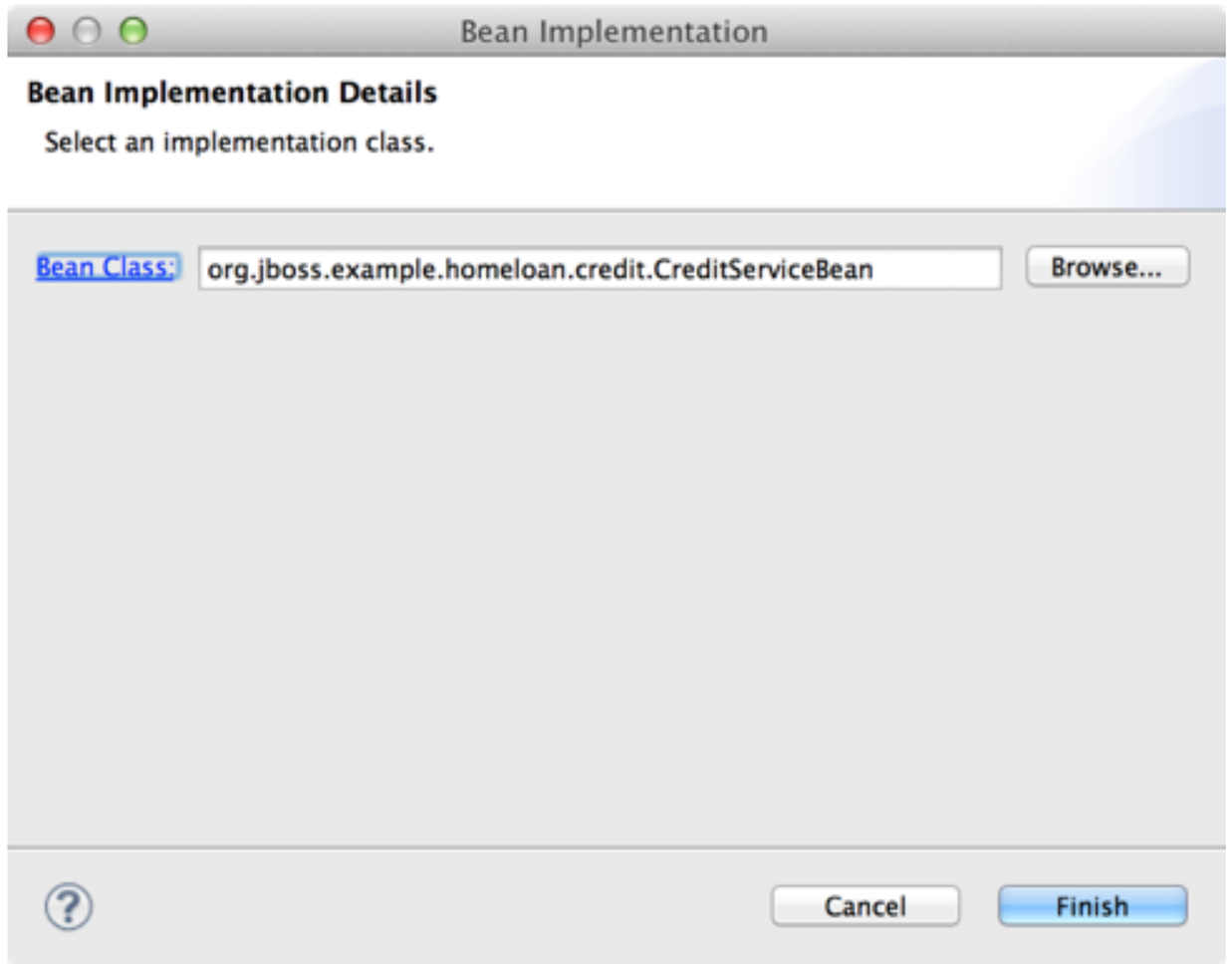
1. Click and hold on the Bean implementation in the palette and drag it onto the Credit component.

Step 1

CreditService Implementation

TODO

1. Click on the Browse ... button to select the bean implementation to use.
2. Enter CreditServiceBean in the input field and select it from the list.
3. Click Finish.



Step 1

CreditServiceBean

FYI

You can reveal the implementation for any component in your application by double-clicking on the component in the editor.

TODO

1. Double-click on the Credit component to see the implementation we just added.

CreditServiceBean.java

```
package org.jboss.example.homeloan.credit;

import org.jboss.example.homeloan.data.Applicant;

@Service(CreditService.class)
public class CreditServiceBean implements CreditService {

    @Override
    public CreditInfo creditCheck(Applicant applicant) {
        String creditScore = "000";
        if (applicant != null && applicant.getSsn() != null) {
            creditScore = applicant.getSsn().substring(0, 3);
        }

        CreditInfo credit = new CreditInfo();
        credit.setApplicant(applicant);
        credit.setScore(Integer.parseInt(creditScore));
        return credit;
    }

}
```

Step 1

Validate Changes

FYI

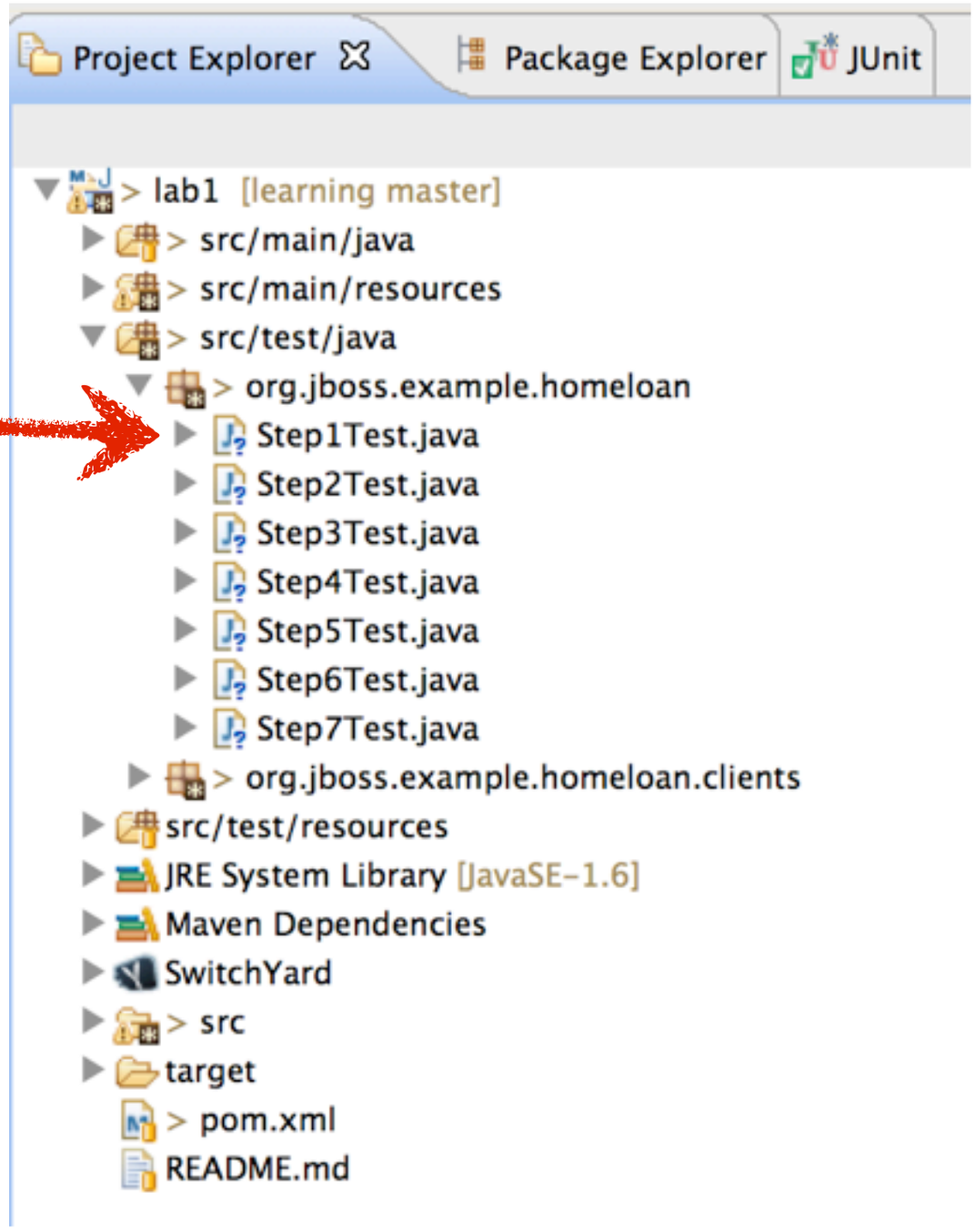
You have completed the changes required for step 1. Let's validate the changes using a service unit test.

TODO

1. Make sure the project is completely saved by selecting File -> Save All.

2. Double-click on Step1Test in the explorer to open the unit test.

3. Go to the Run menu in the main menu bar and select 'Run As -> JUnit Test' to run the unit test.

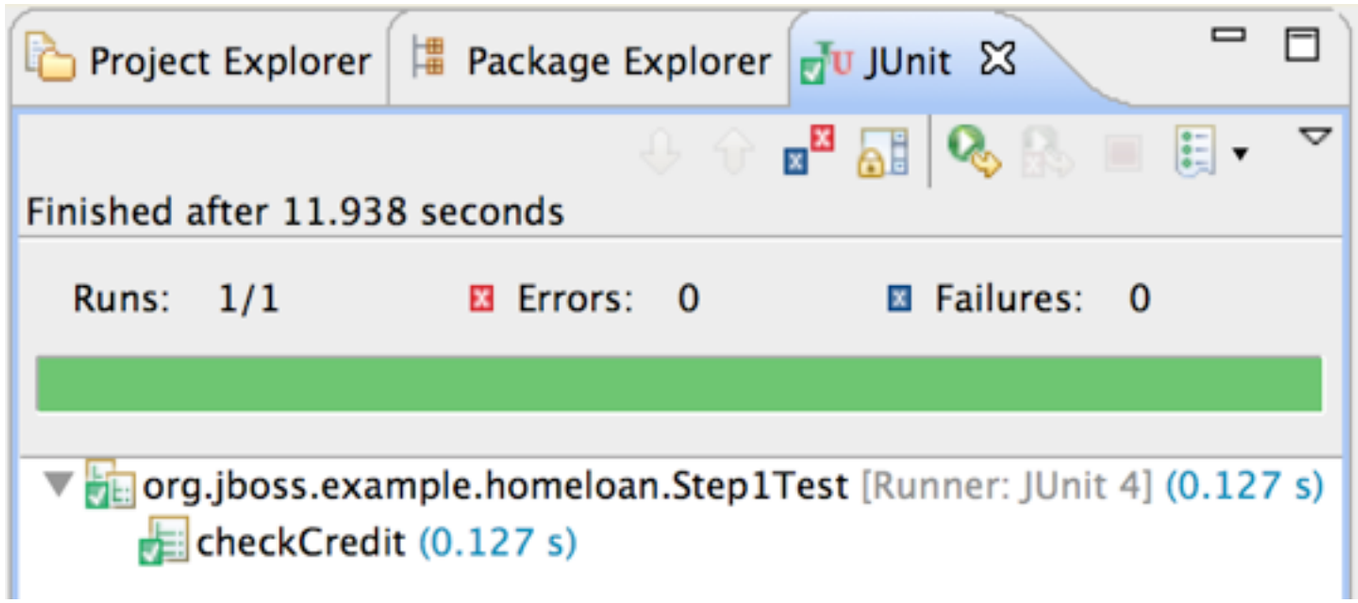


Step 1

Success?

FYI

You should see a nice green bar indicating that the test passed. If you get a red bar indicating test failure, raise your hand and someone will be by to help.



Step 2

Component Reference

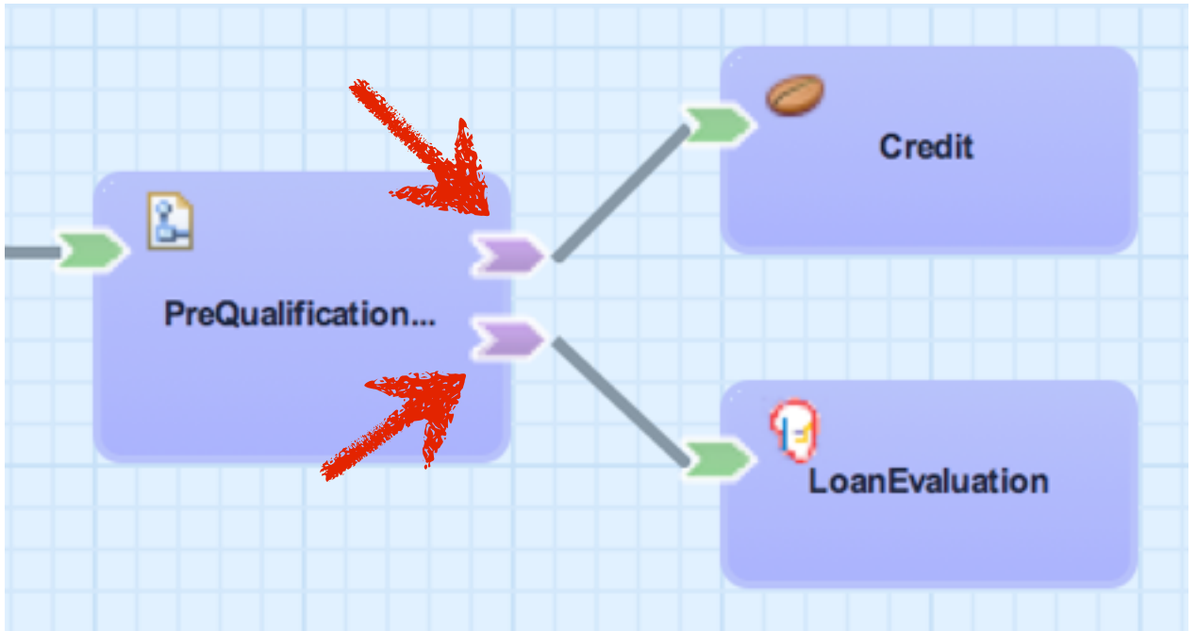
Goals

- *Inspect BPMN 2 workflow used for service orchestration.*
- *Add component reference for each SwitchYard service called from workflow.*
- *Run unit tests to verify changes.*

Component Reference

FYI

*Service implementations can invoke other services through **references**. All references have a contract and can be wired to services inside or outside the application.*

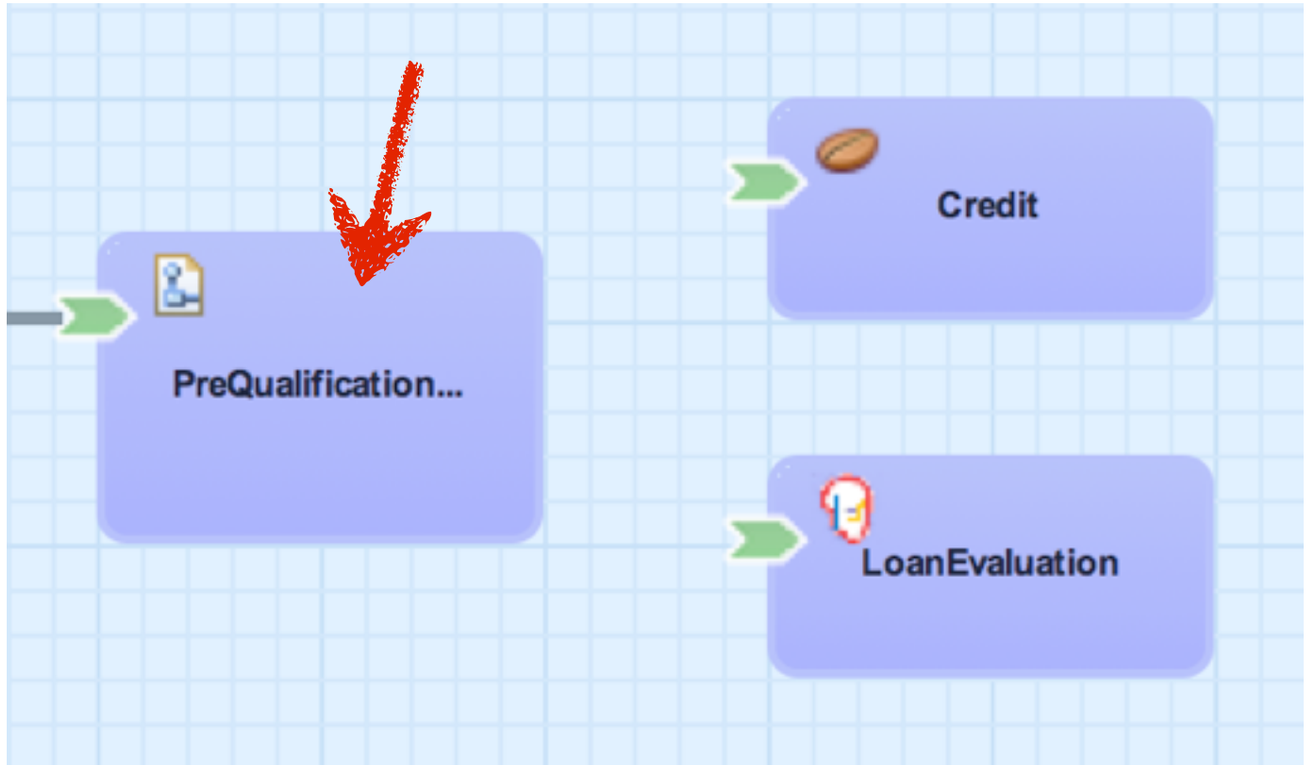


Step 2

Open BPMN2 Process

TODO

1. Double-click on the PreQualification component on the canvas.

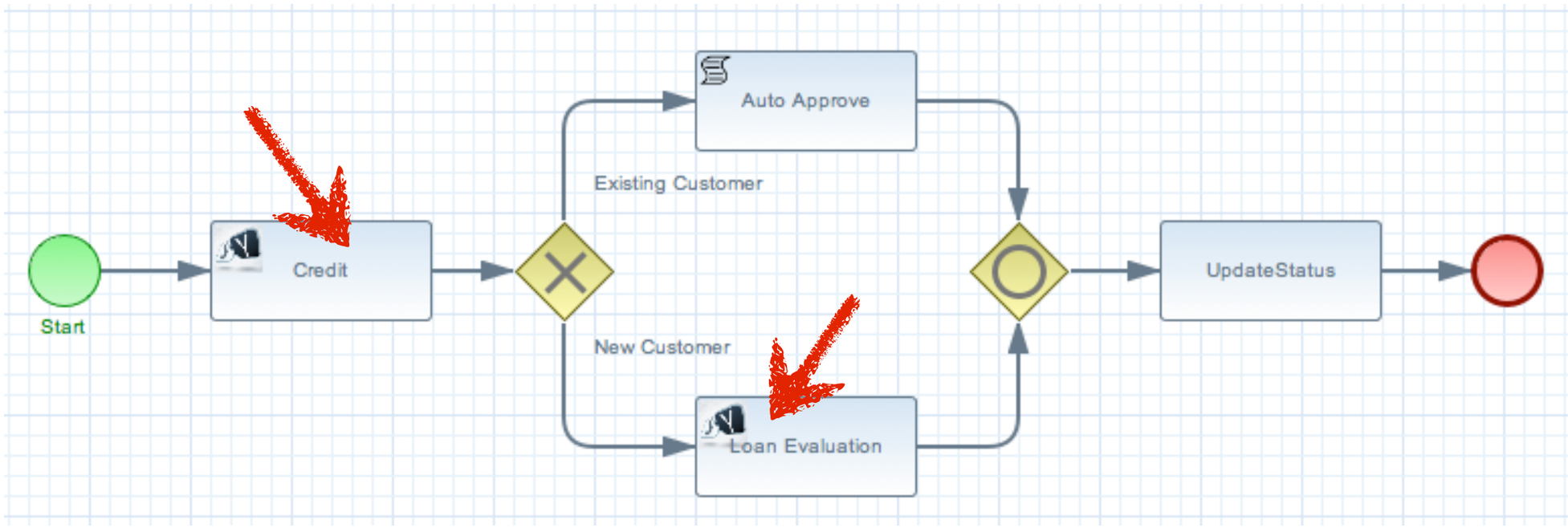


Step 2

Inspect Process

FYI

Note the two SY service tasks in our BPMN 2 workflow. Each task represents a service invocation which is mapped to a component reference. The references are currently missing from the component and need to be added.

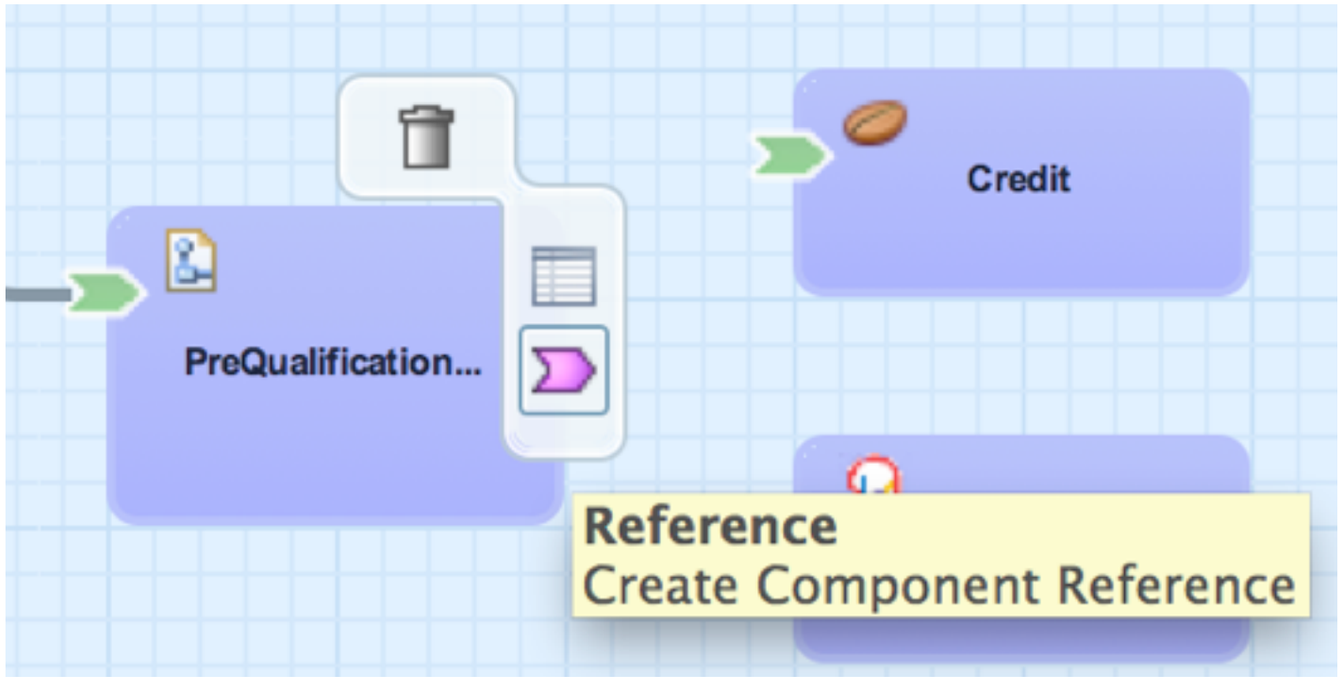


Step 2

Add CreditService Reference

TODO

1. Hover over the PreQualification component to bring up the button bar.
2. Click on the Reference icon to create a new reference.

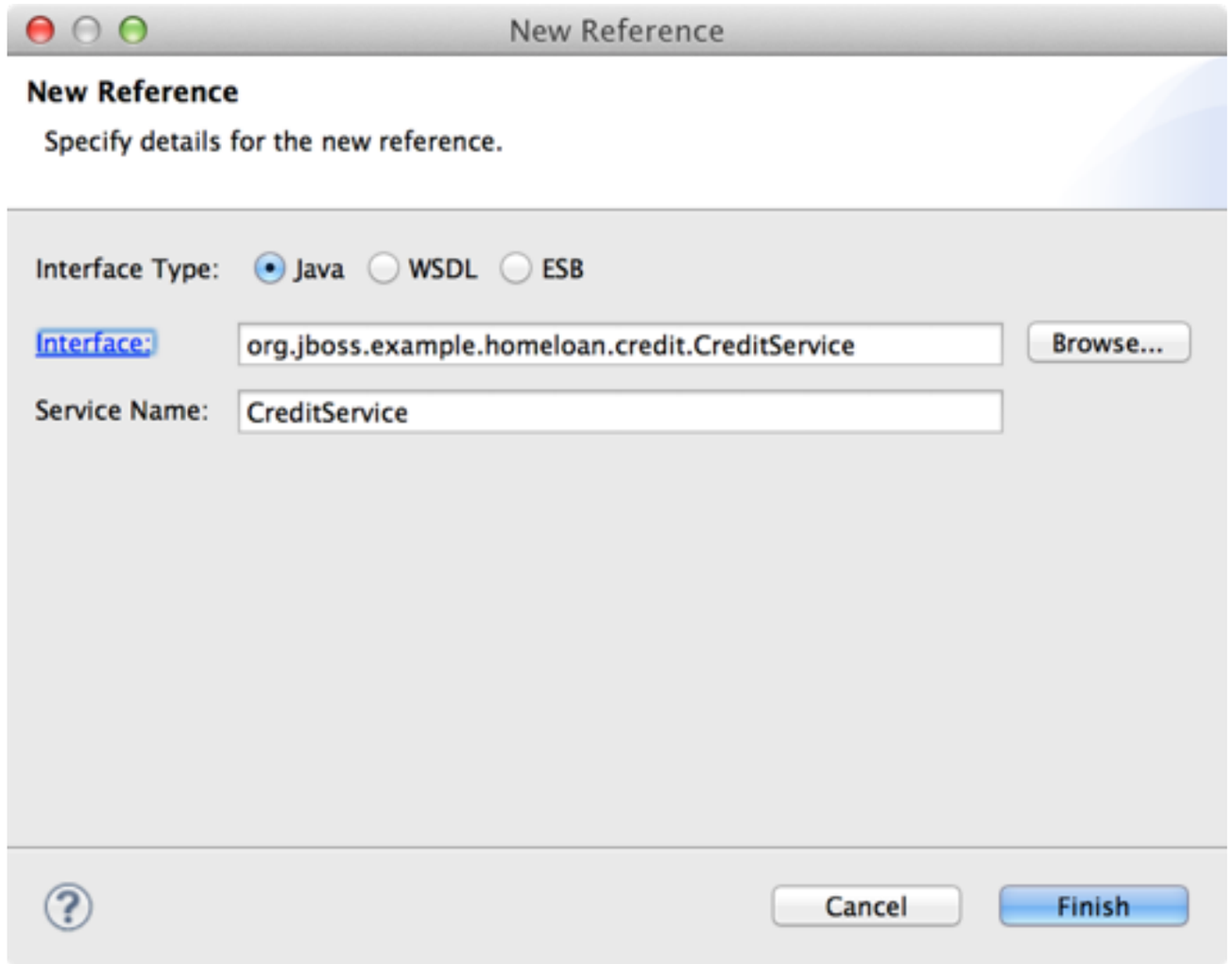


Step 2

CreditService Contract

TODO

1. Click on the Browse ... button to select the bean implementation.
2. Enter CreditService in the input field and select it from the list.
3. Click Finish.

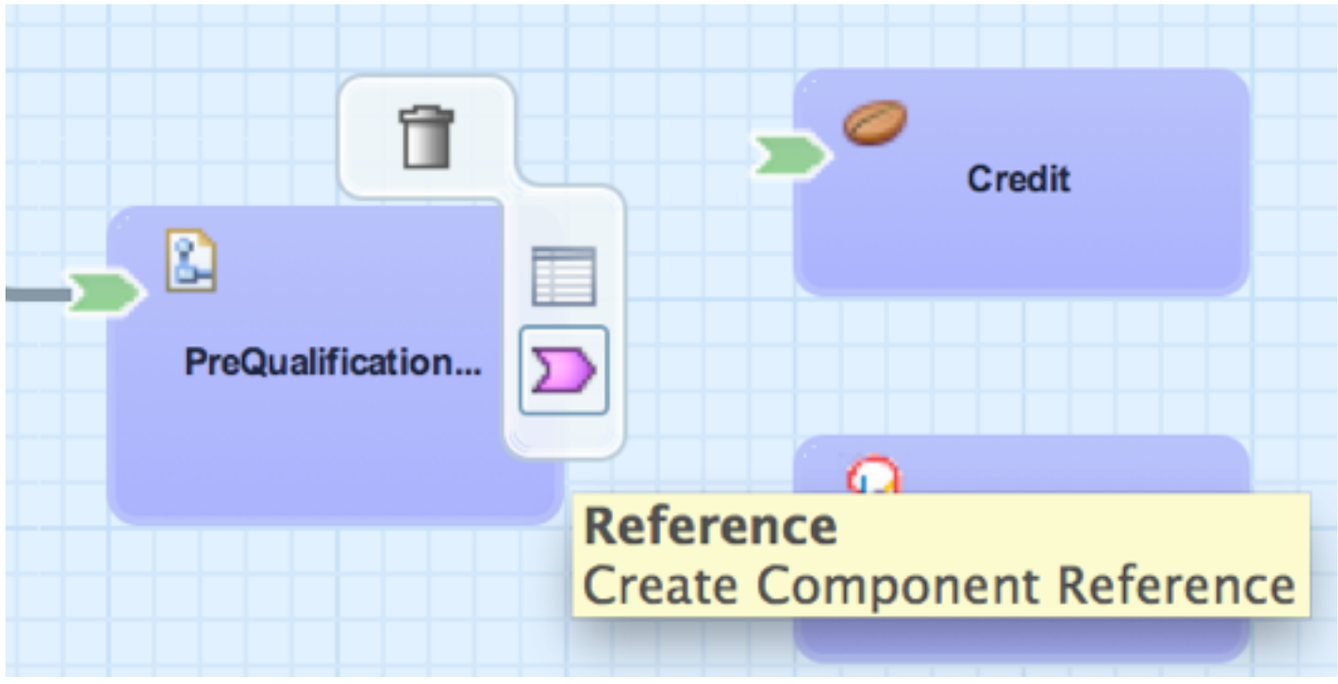


Step 2

Add LoanEvaluation Reference

TODO

1. Hover over the PreQualification component to bring up the button bar.
2. Click on the Reference icon to create a new reference.

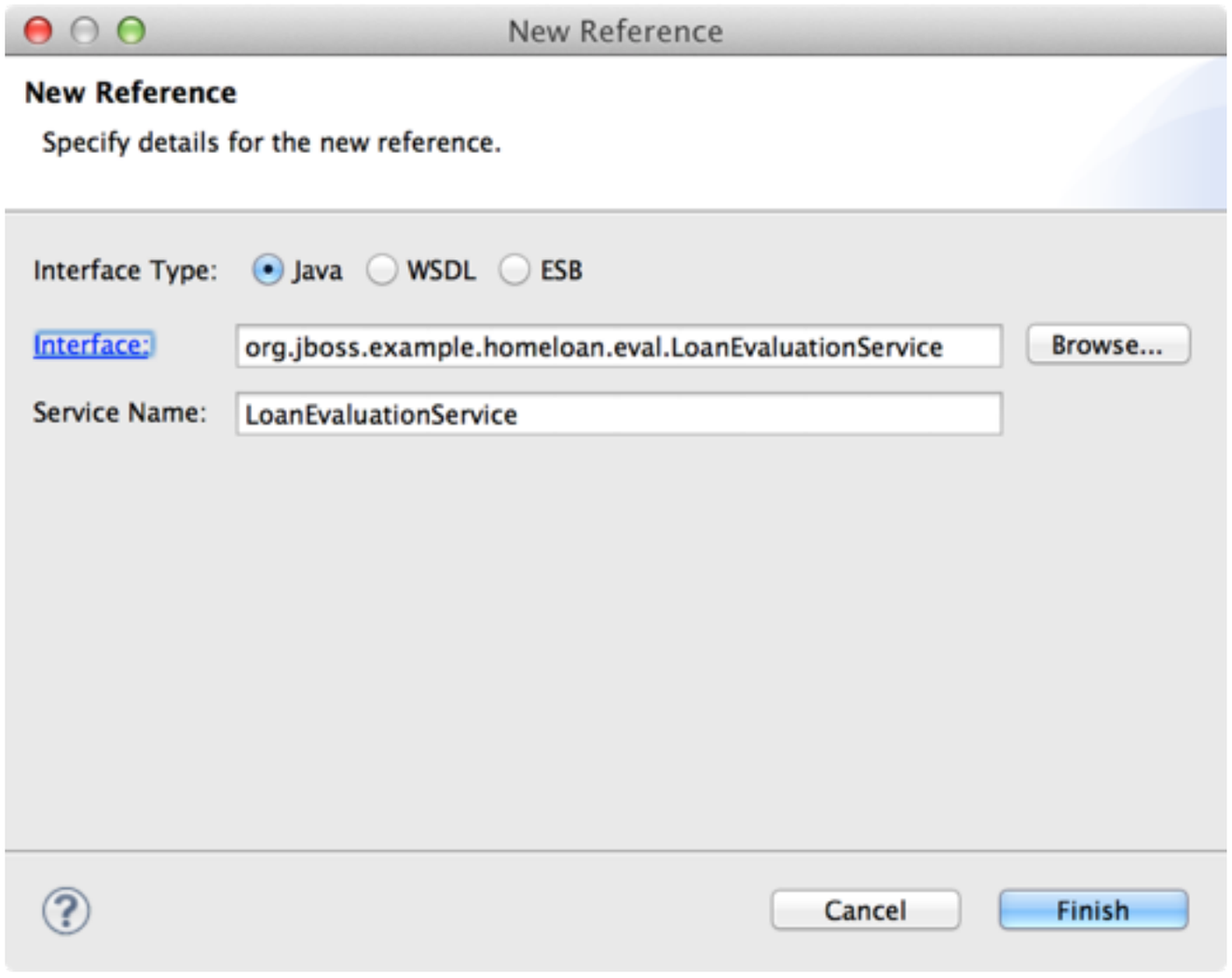


Step 2

LoanEvaluation Contract

TODO

1. Click on the Browse ... button to select the bean implementation.
2. Enter LoanEvaluationService in the input field and select it from the list.
3. Click Finish.



Step 2
Validate Changes

FYI

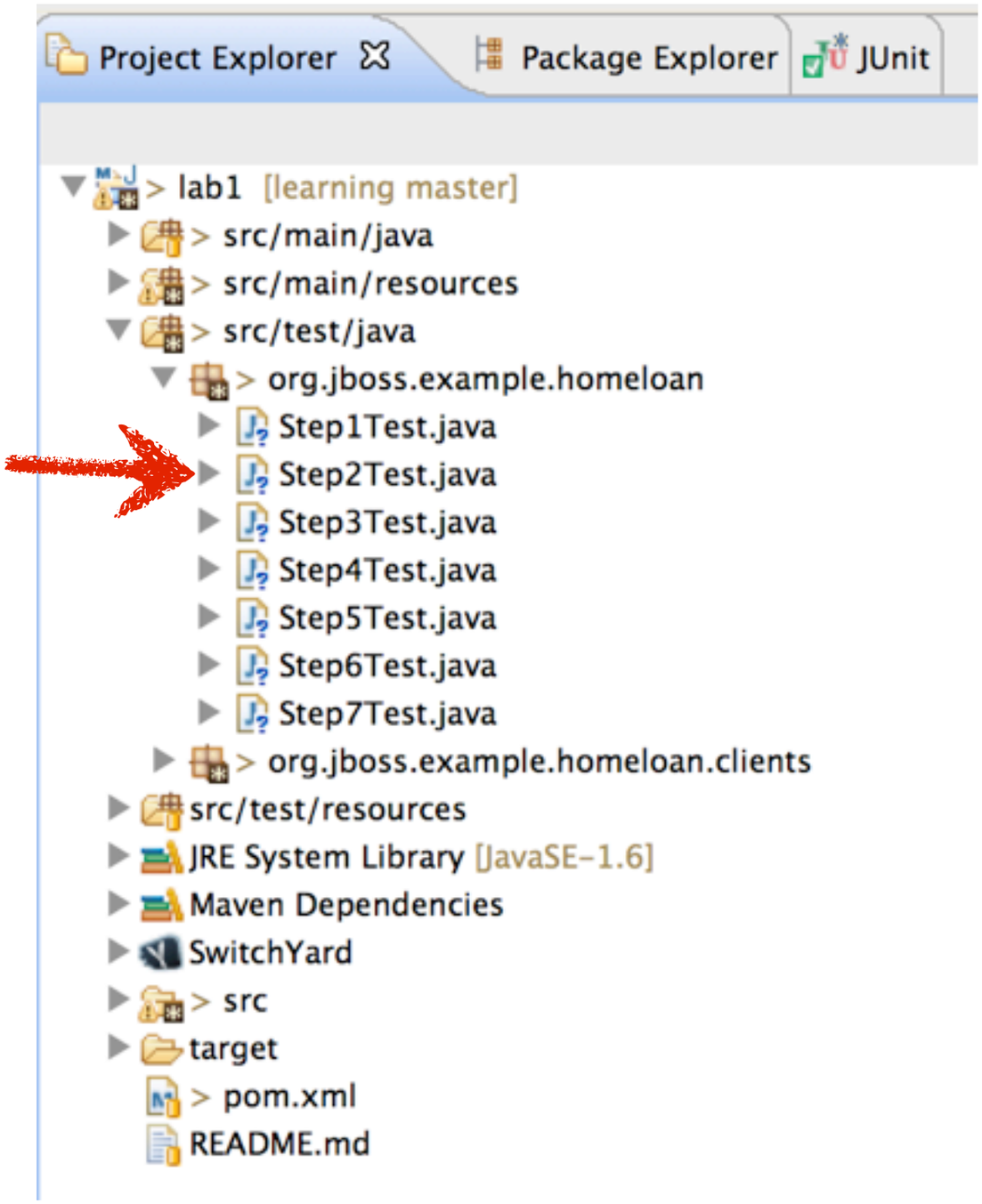
You have completed the changes required for step 2. Let's validate the changes using a service unit test.

TODO

1. Make sure the project is completely saved by selecting File -> Save All.

2. Double-click on Step2Test in the explorer to open the unit test.

3. Go to the Run menu in the main menu bar and select 'Run As -> JUnit Test' to run the unit test.

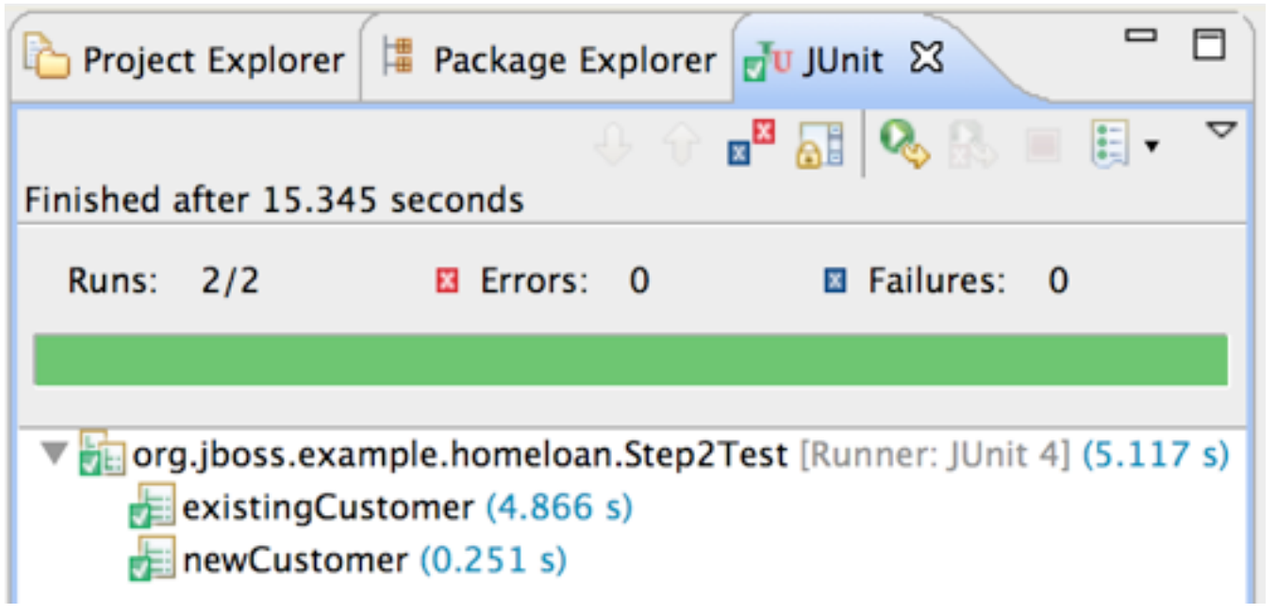


Step 2

Success?

FYI

You should see a nice green bar indicating that the test passed. If you get a red bar indicating test failure, raise your hand and someone will be by to help.



Step 3

Camel Routing

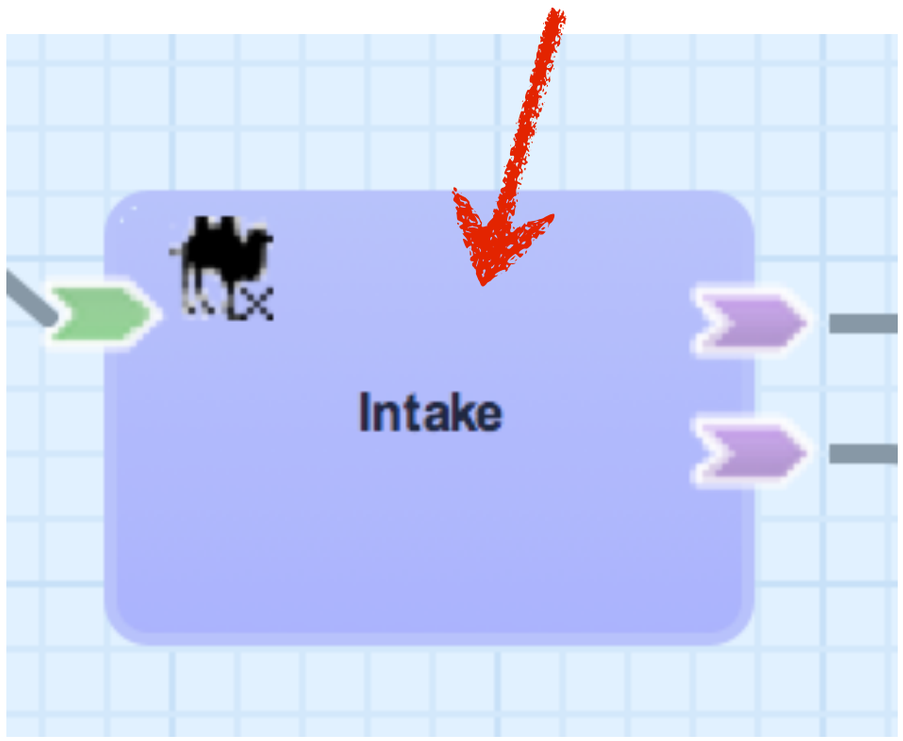
Goals

- *Inspect Camel routing implementation*
- *Update routing logic to invoke SwitchYard services*
- *Run unit tests to verify changes.*

Camel Implementation

FYI

The SwitchYard application model is implementation-agnostic, which means the bits in the digram look the same no matter what the implementation type is (which is a good thing).



TODO

1. Double-click on the Intake component to open the service implementation.

Step 3

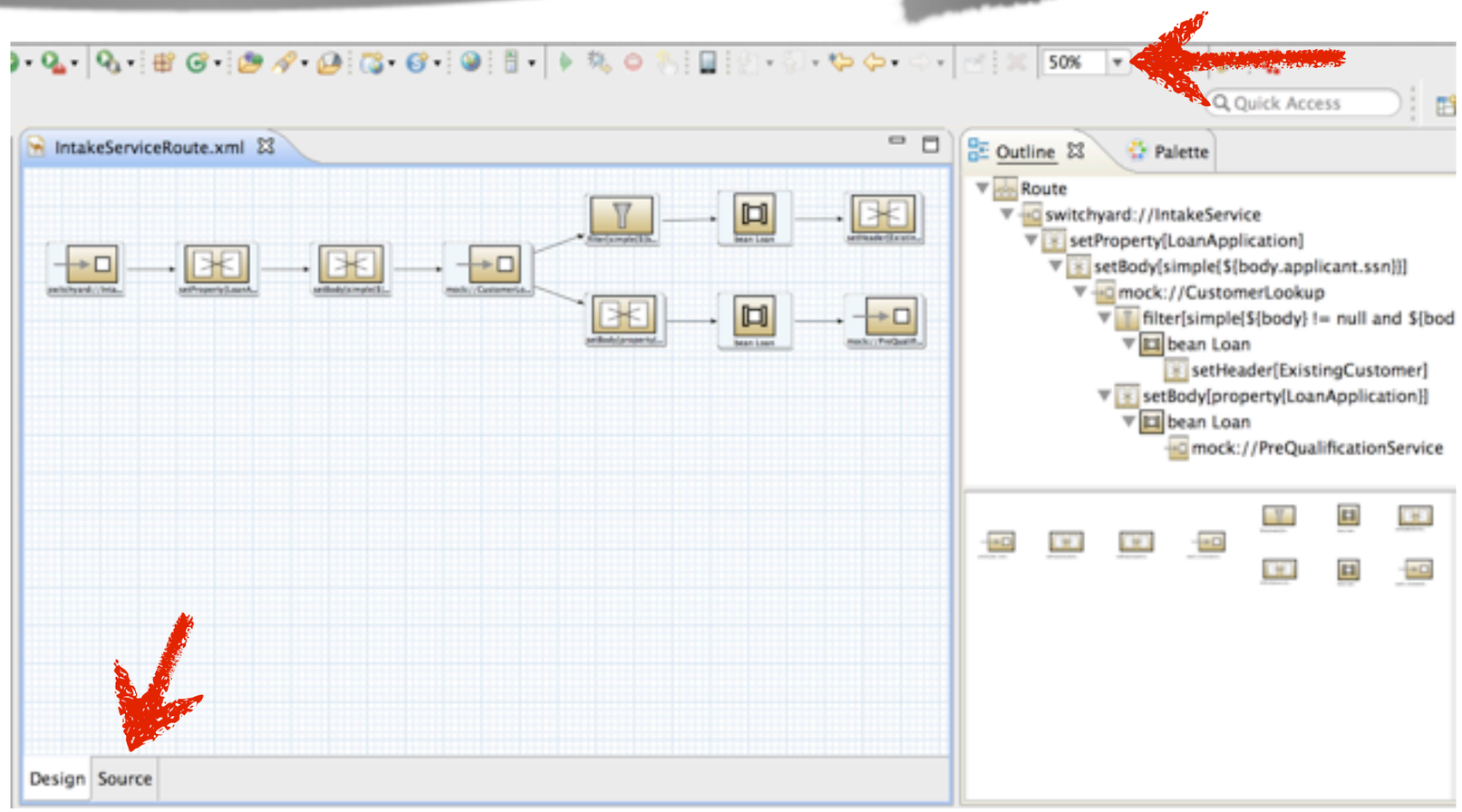
Inspect Camel Route

FYI

Routes using the Camel XML DSL are displayed in the Camel visual editor by default. Your initial view will not look like the picture below until you change the 'Zoom' setting.

TODO

1. Change the Zoom setting from 100% to 50% so you can view the entire route.
2. Click on the Source tab to see the actual XML DSL for the route.



Step 3

Invoke SwitchYard Services from Route

FYI

`switchyard://` endpoints allow you to invoke services in SwitchYard from a Camel route. Each `switchyard://` endpoint used in a route must have a corresponding reference defined on the component.

TODO

1. Replace **`mock://CustomerLookup`** with **`switchyard://CustomerLookup`**
2. Replace **`mock://PreQualificationService`** with **`switchyard://PreQualificationService`**

IntakeServiceRoute.xml

```
<?xml version="1.0" encoding="ASCII"?>
<routes xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="switchyard://IntakeService"/>
        <setProperty propertyName="LoanApplication">
            <simple>${body}</simple>
        </setProperty>
        <setBody>
            <simple>${body.applicant.ssn}</simple>
        </setBody>
        <to uri="mock://CustomerLookup"/>
        <filter>
            <simple>${body} != null and ${body.size} == 1</simple>
            <bean ref="Loan" method="customerUpdate(${property.LoanApplication}, ${bod
            <setHeader headerName="ExistingCustomer">
                <constant>true</constant>
            </setHeader>
        </filter>
        <setBody>
            <property>LoanApplication</property>
        </setBody>
        <bean ref="Loan" method="summary"/>
        <to uri="mock://PreQualificationService"/>
    </route>
</routes>
```

Design | Source

Step 3
Validate Changes

FYI

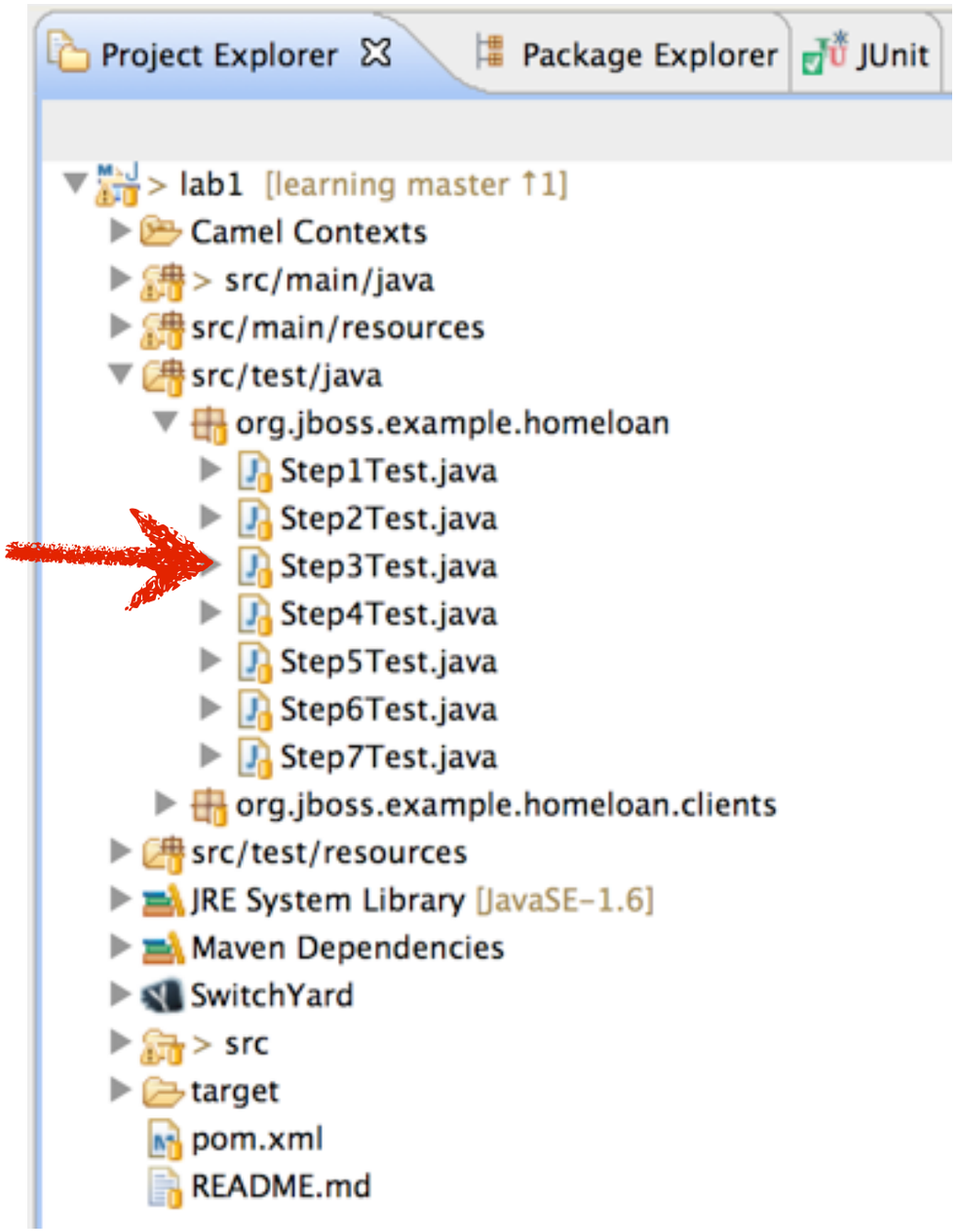
You have completed the changes required for step 3. Let's validate the changes using a service unit test.

TODO

1. Make sure the project is completely saved by selecting File -> Save All.

2. Double-click on Step3Test in the explorer to open the unit test.

3. Go to the Run menu in the main menu bar and select 'Run As -> JUnit Test' to run the unit test.

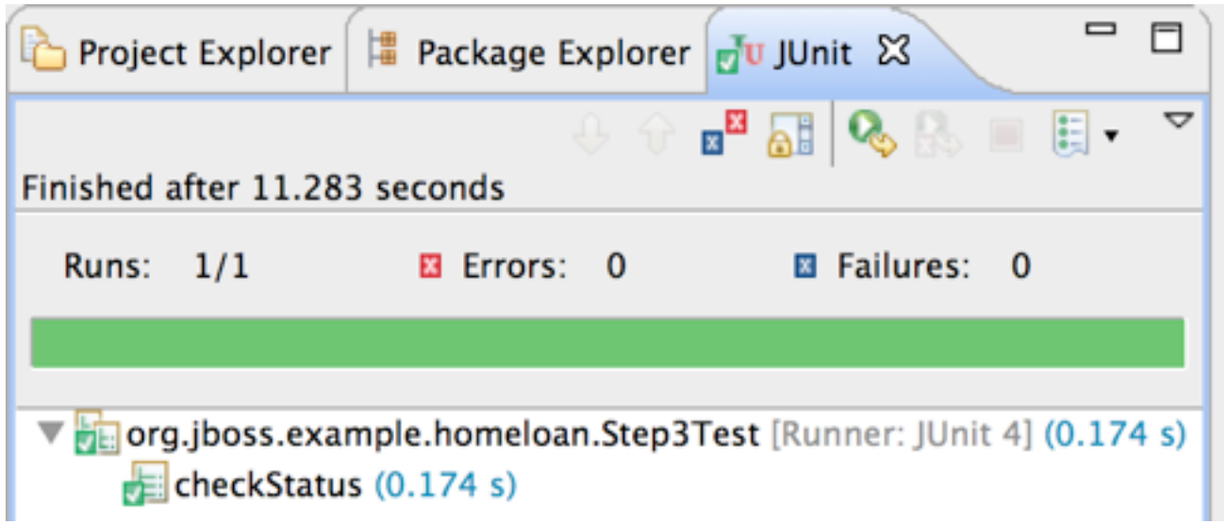


Step 3

Success?

FYI

You should see a nice green bar indicating that the test passed. If you get a red bar indicating test failure, raise your hand and someone will be by to help.



Step 4

Reference Binding

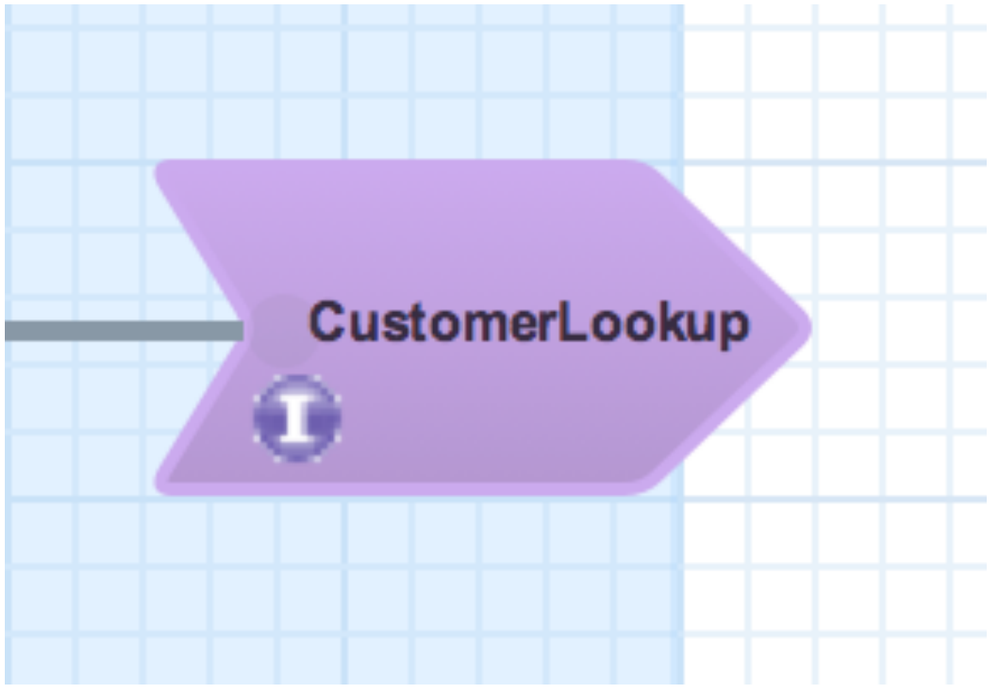
Goals

- *Add a SQL binding to the CustomerLookup reference to query the database.*
- *Run unit tests to verify changes.*

Composite Reference

FYI

A composite reference promotes a component reference to allow an implementation to invoke services outside the application through a binding. All composite references have a contract.



Step 4

Add SQL Binding

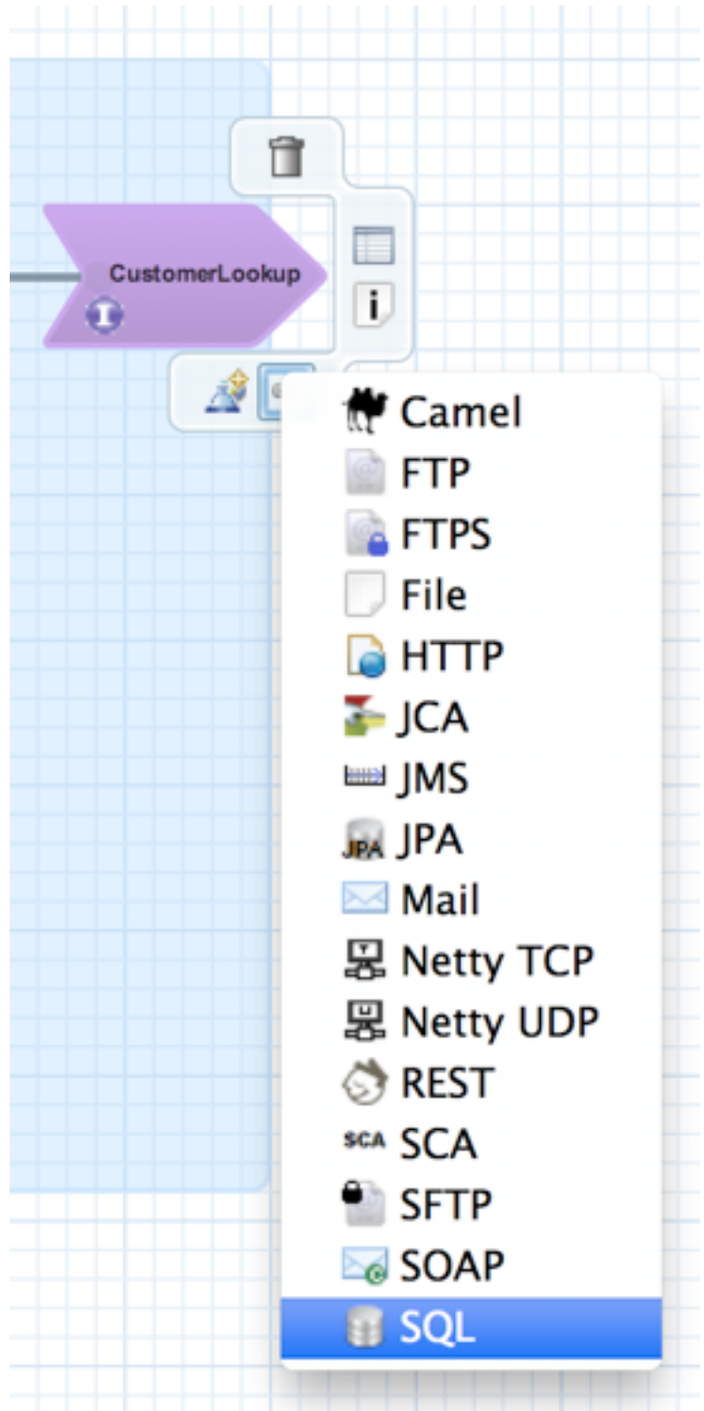
FYI

The button bar can be used to add bindings to composite services and references too. The button bar is awesome!

TODO

1. Hover over the CustomerLookup composite reference to access the button bar.

2. Click on the bindings button and select SQL.



Step 4

Configure SQL Binding

TODO

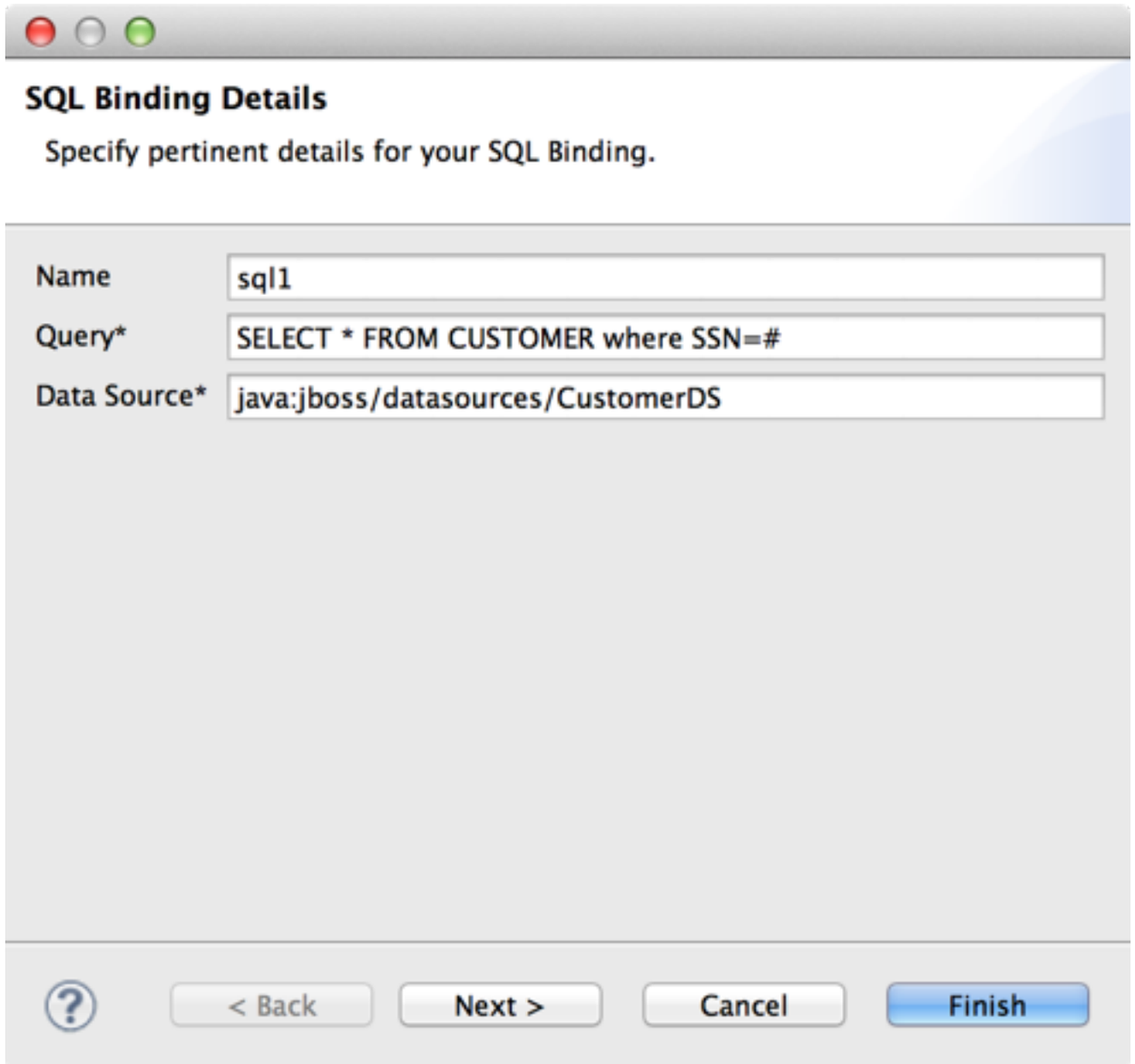
1. In the ‘Query’ field enter:

```
SELECT * FROM CUSTOMER where SSN=#
```

2. In the ‘Data Source’ field enter:

```
java:jboss/datasources/CustomerDS
```

3.Click Finish.



Step 4

Validate Changes

FYI

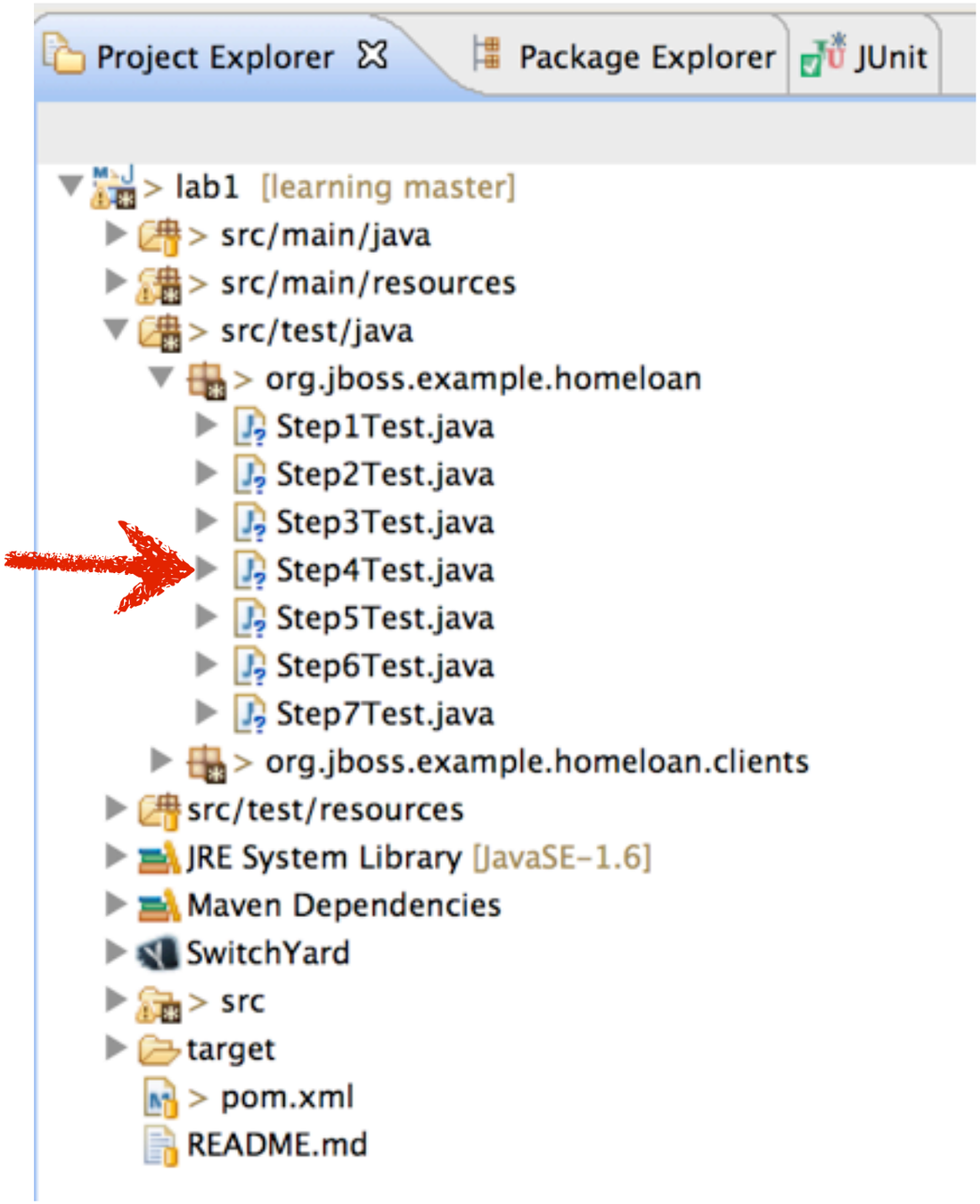
You have completed the changes required for step 4. Let's validate the changes using a service unit test.

TODO

1. Make sure the project is completely saved by selecting File -> Save All.

2. Double-click on Step4Test in the explorer to open the unit test.

3. Go to the Run menu in the main menu bar and select 'Run As -> JUnit Test' to run the unit test.

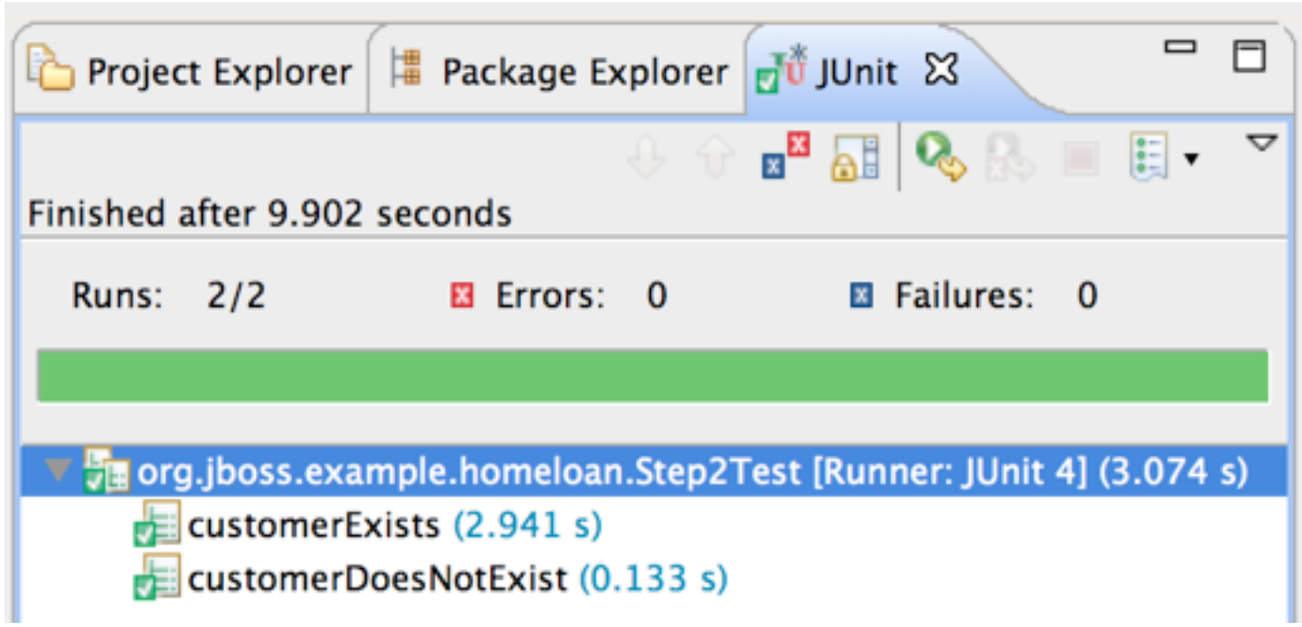


Step 4

Success?

FYI

You should see a nice green bar indicating that the test passed. If you get a red bar indicating test failure, raise your hand and someone will be by to help.



Step 5

Transformation

Goals

- *Add a transformer*
- *Run unit tests to verify changes.*

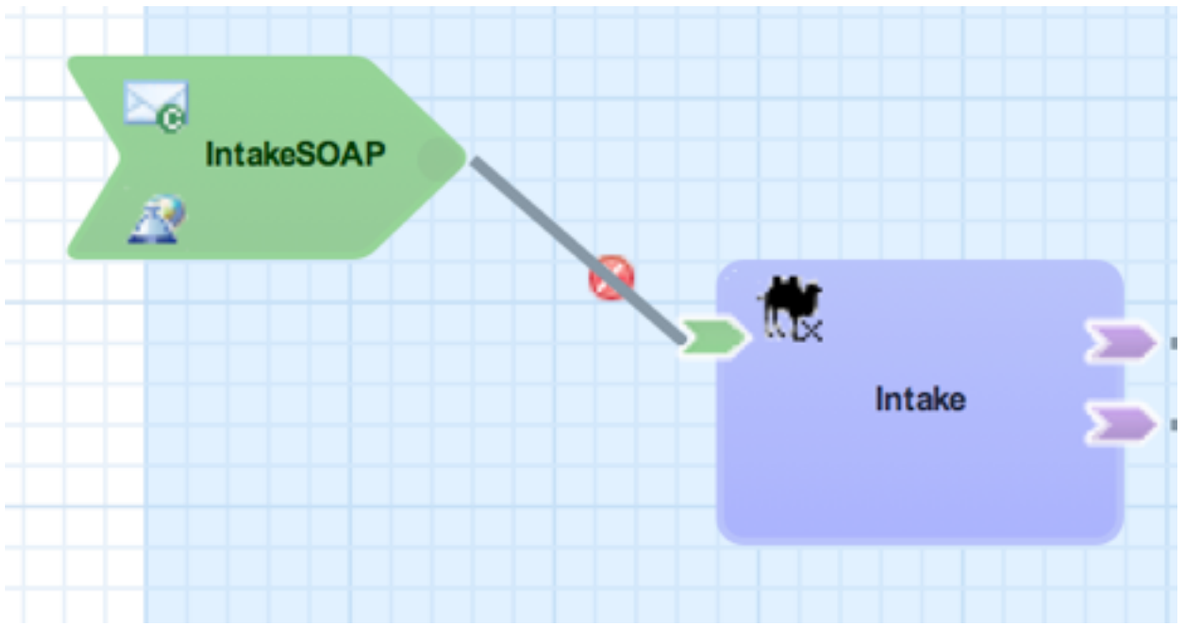
Step 5

Adding Transformers

FYI

Transformers allow you to transform between different data formats in a declarative manner outside of your implementation logic. The IntakeSOAP service is exposed to external consumers using WSDL, so the payload type is XML. The implementation of IntakeService expects a Java type of Application. We need to add a transformer which transforms between the XML and Java format of an applicant.

WSDL/XML ⟷ Java

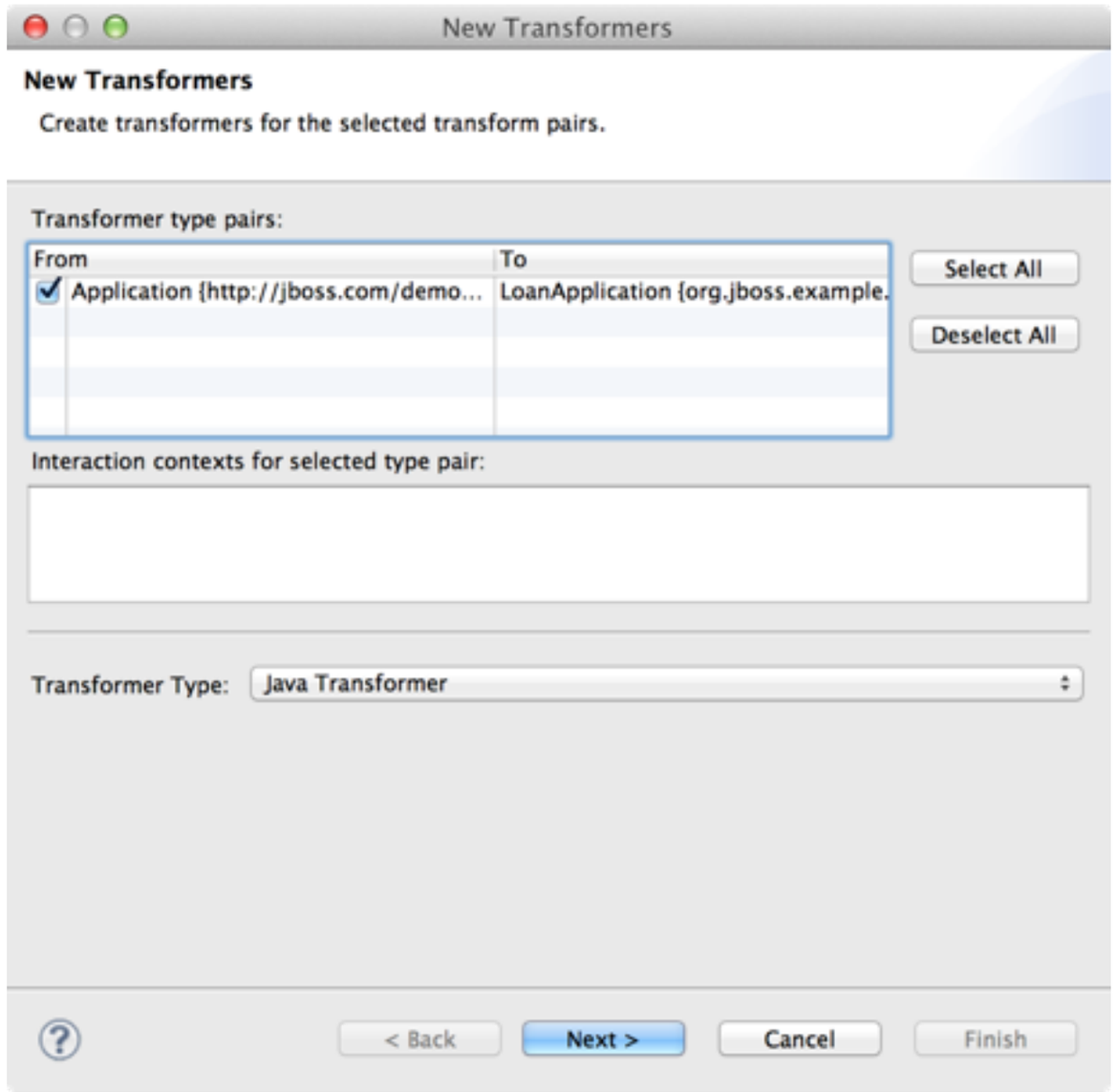


Step 5

Add Transformer

TODO

1. Right-click on the composite in the visual editor.
2. Select “Create Required Transformers” in the pop-up menu.
3. Click Next >



Step 5

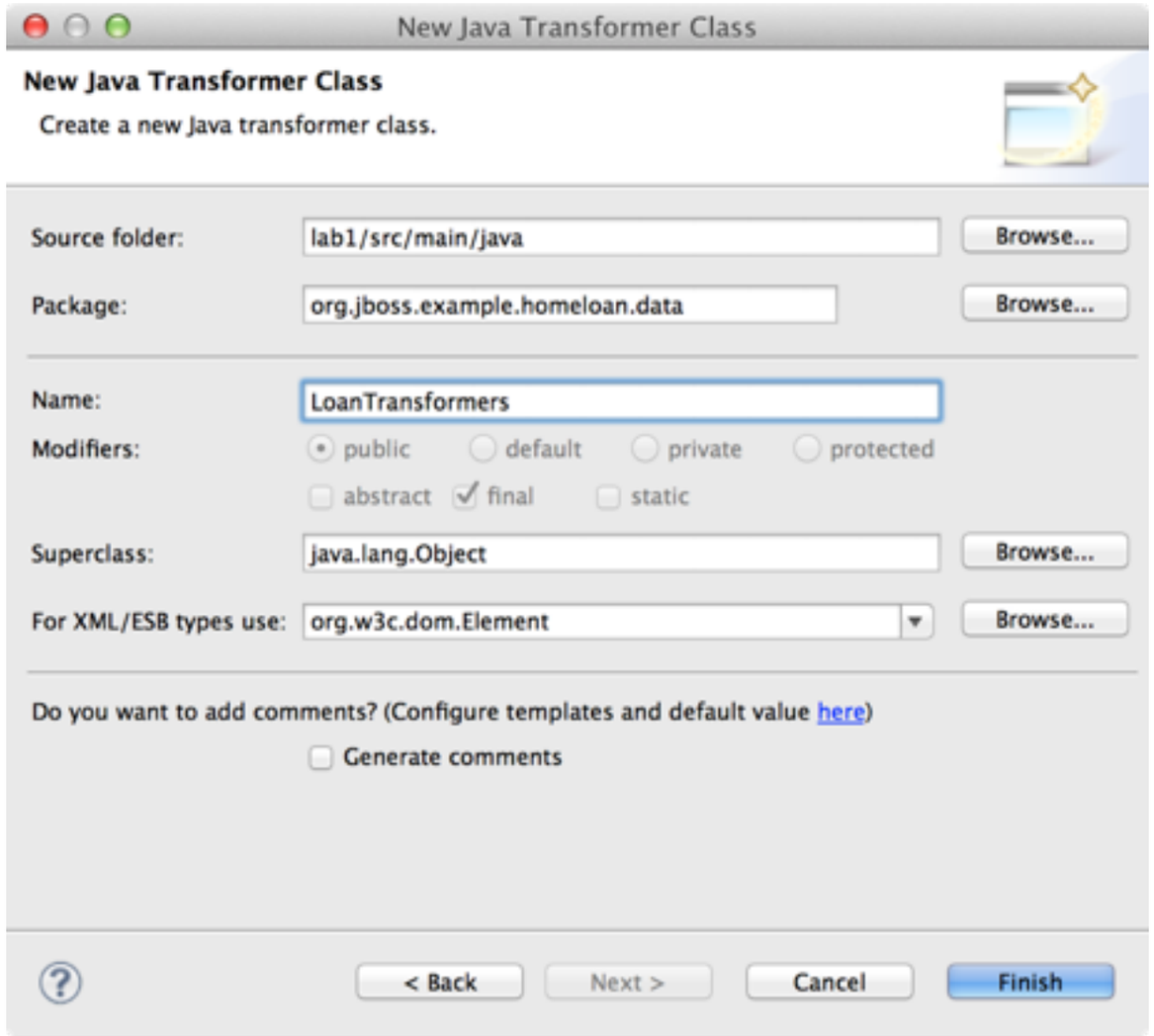
Add Transformer

TODO

1. Enter values:

```
Package : org.jboss.example.homeloan.data
Name    : LoanTransformers
```

2. Click Finish

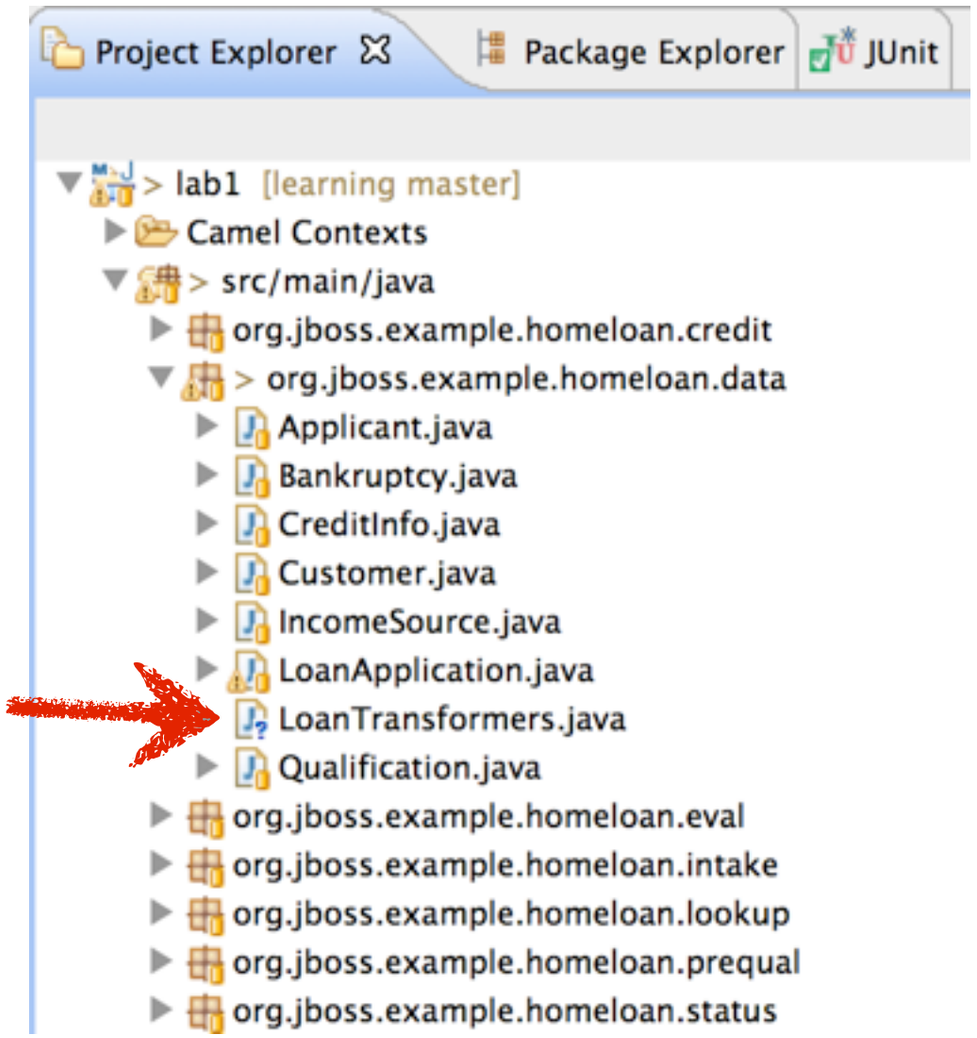


Step 5

Implement Transformer

TODO

1. Double-click on the new LoanTransformers class to edit the transformer.



Step 5

Implement Transformer

TODO

1. Update the body of the transformApplicationToLoanApplication() method to:

```
return LoanApplication.fromElement(from);
```

switchyard.xml | LoanTransformers.java

```
package org.jboss.example.homeloan.data;

import org.switchyard.annotations.Transformer;
import org.w3c.dom.Element;

public final class LoanTransformers {

    @Transformer(from = "{http://jboss.com/demo/products/fsw/6.0/Application.xsd}Application")
    public LoanApplication transformApplicationToLoanApplication(Element from) {
        return LoanApplication.fromElement(from);
    }

}
```

Step 5
Validate Changes

FYI

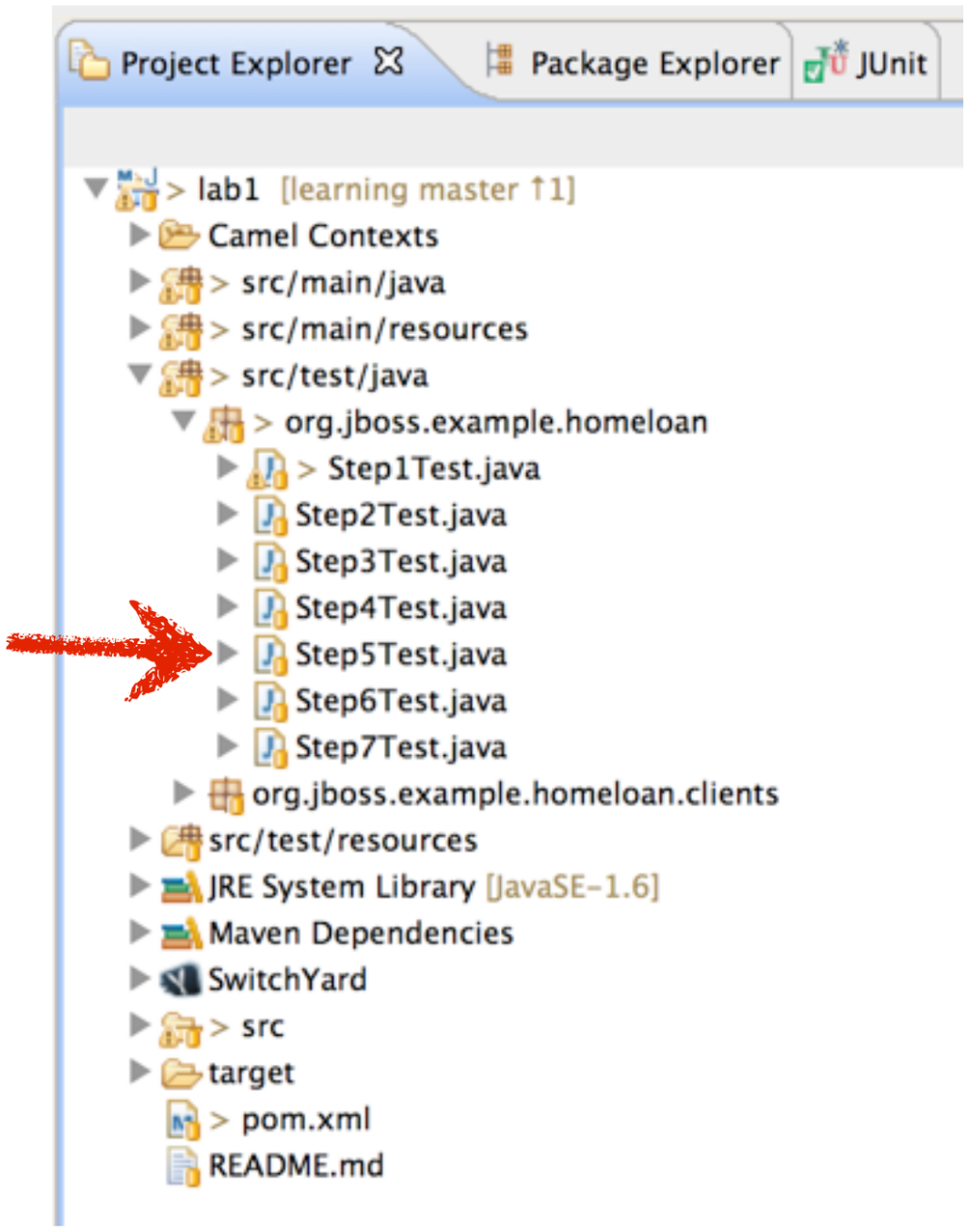
You have completed the changes required for step 5. Let's validate the changes using a service unit test.

TODO

1. Make sure the project is completely saved by selecting File -> Save All.

2. Double-click on Step5Test in the explorer to open the unit test.

3. Go to the Run menu in the main menu bar and select 'Run As -> JUnit Test' to run the unit test.

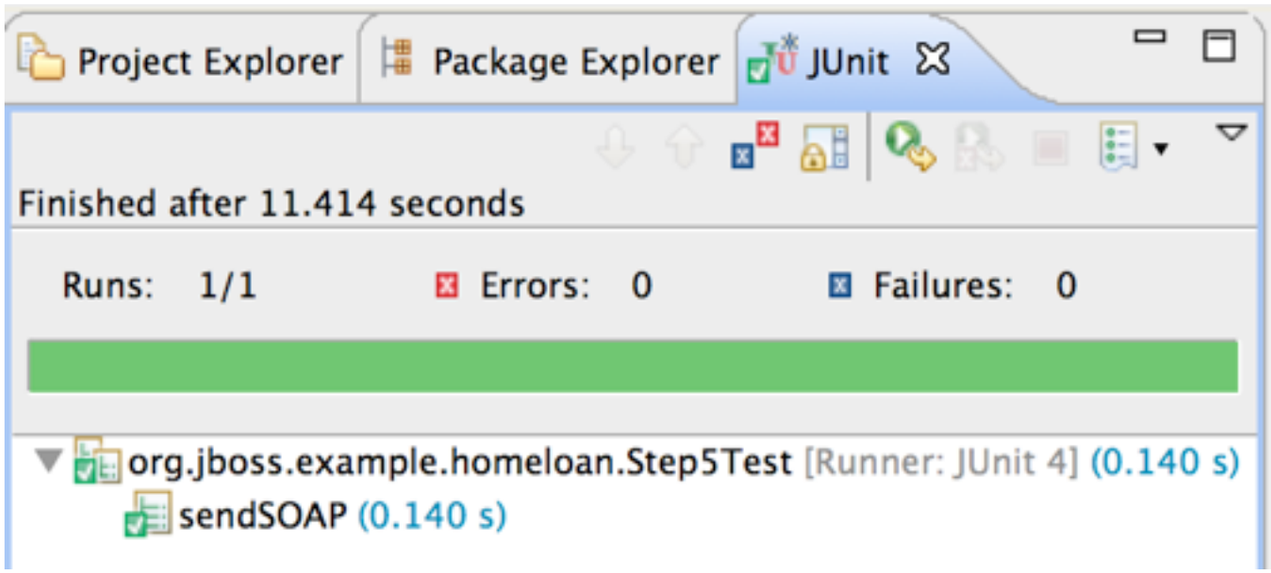


Step 5

Success?

FYI

You should see a nice green bar indicating that the test passed. If you get a red bar indicating test failure, raise your hand and someone will be by to help.



Step 6

Service Binding

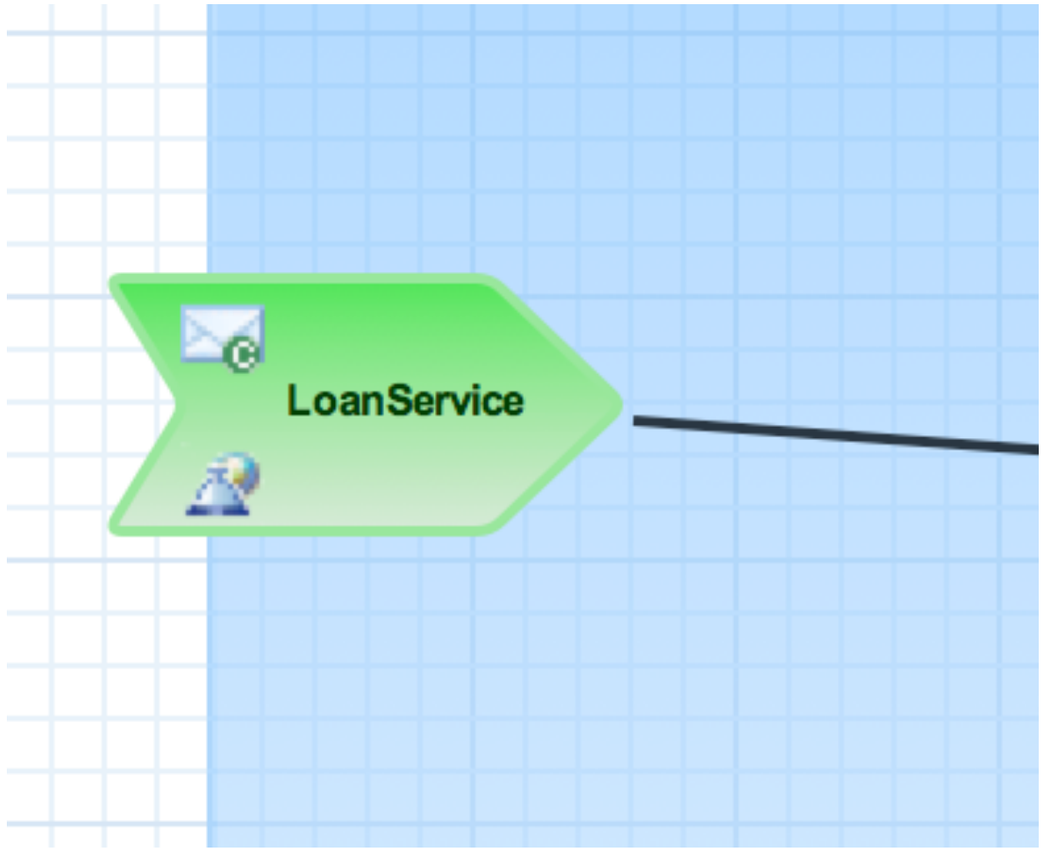
Goals

- *Add JMS binding to composite service.*
- *Run unit tests to verify changes.*

Composite Service

FYI

A composite service promotes a service implementation in your application and makes it available to external consumers through one or more service bindings. All composite services have a contract.



TODO

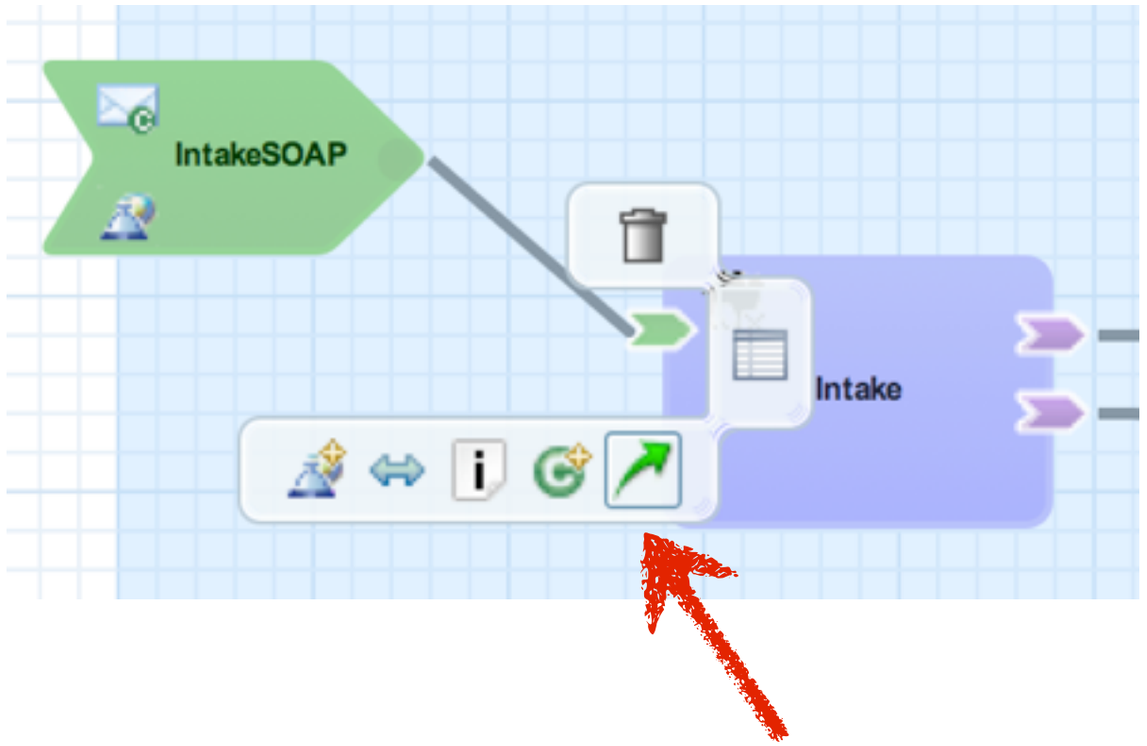
1. Access the properties view for the composite service using its button bar.

Step 6

Service Promotion

TODO

1. Hover over the green component service on the Intake component.
2. Click on the promotion icon to promote the component service to a composite service.

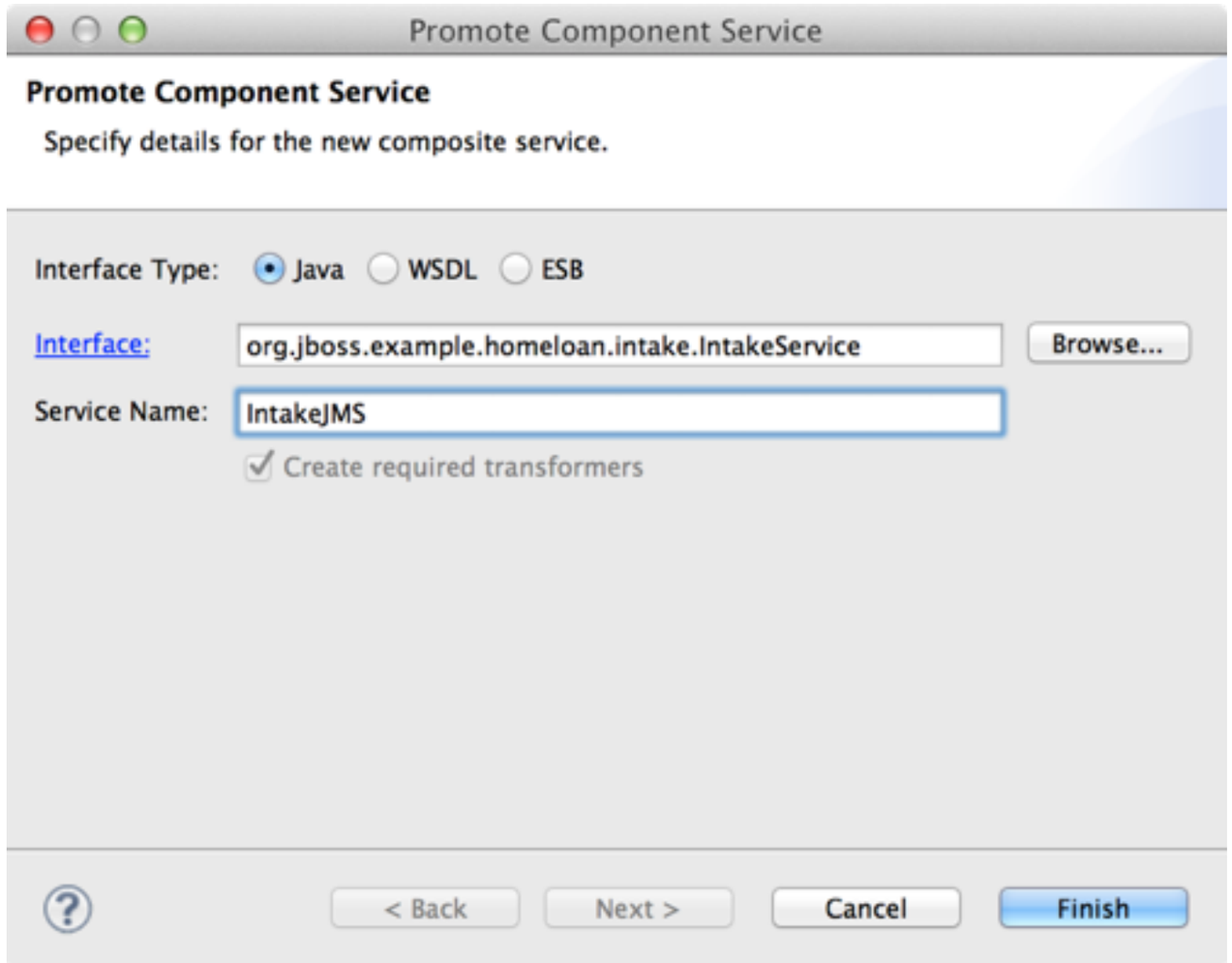


Step 6

Service Promotion

TODO

1. `Service Name : IntakeJMS`
2. Click Finish.



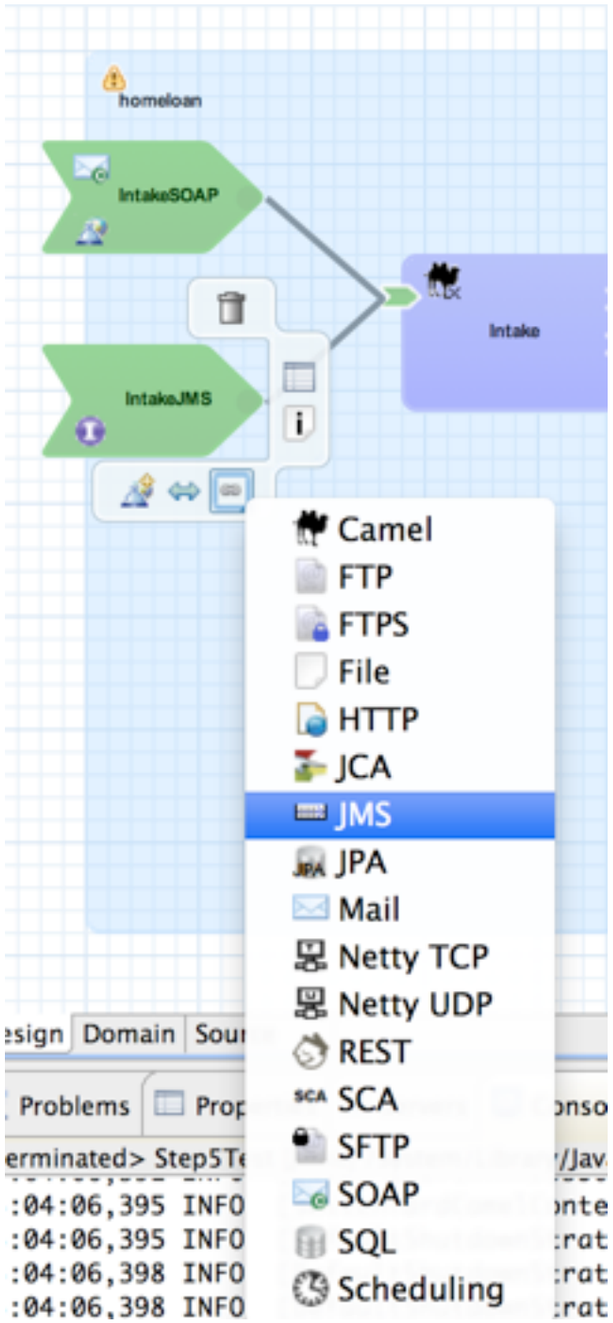
Step 6

Add JMS Service Binding

TODO

1. Hover over the IntakeJMS composite service to access the button bar.

2. Click on the bindings button and select JMS.



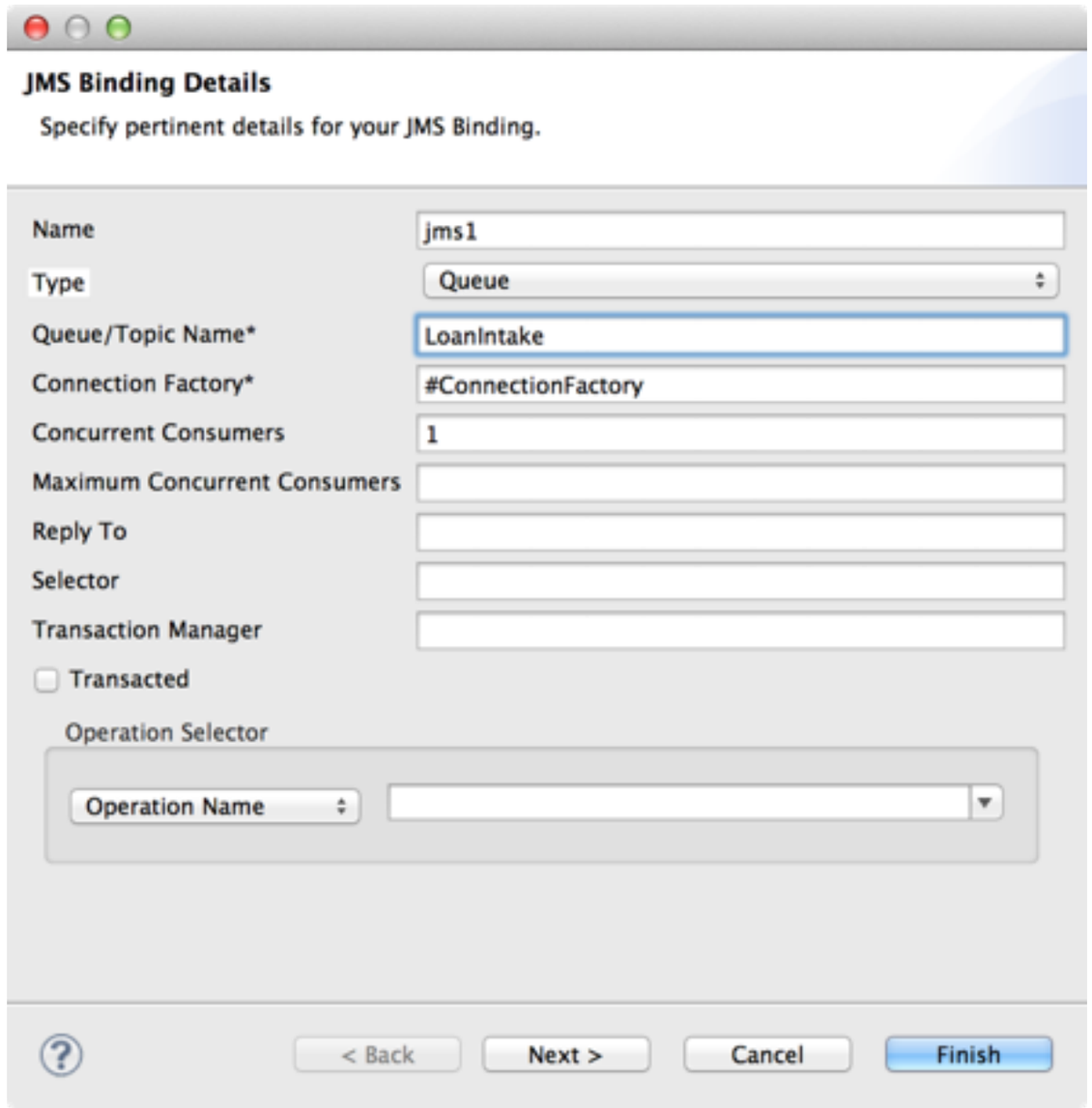
Step 6

Configure JMS Binding

TODO

1. `Queue Name : LoanIntake`

2. Click Finish.



Step 6

Validate Changes

FYI

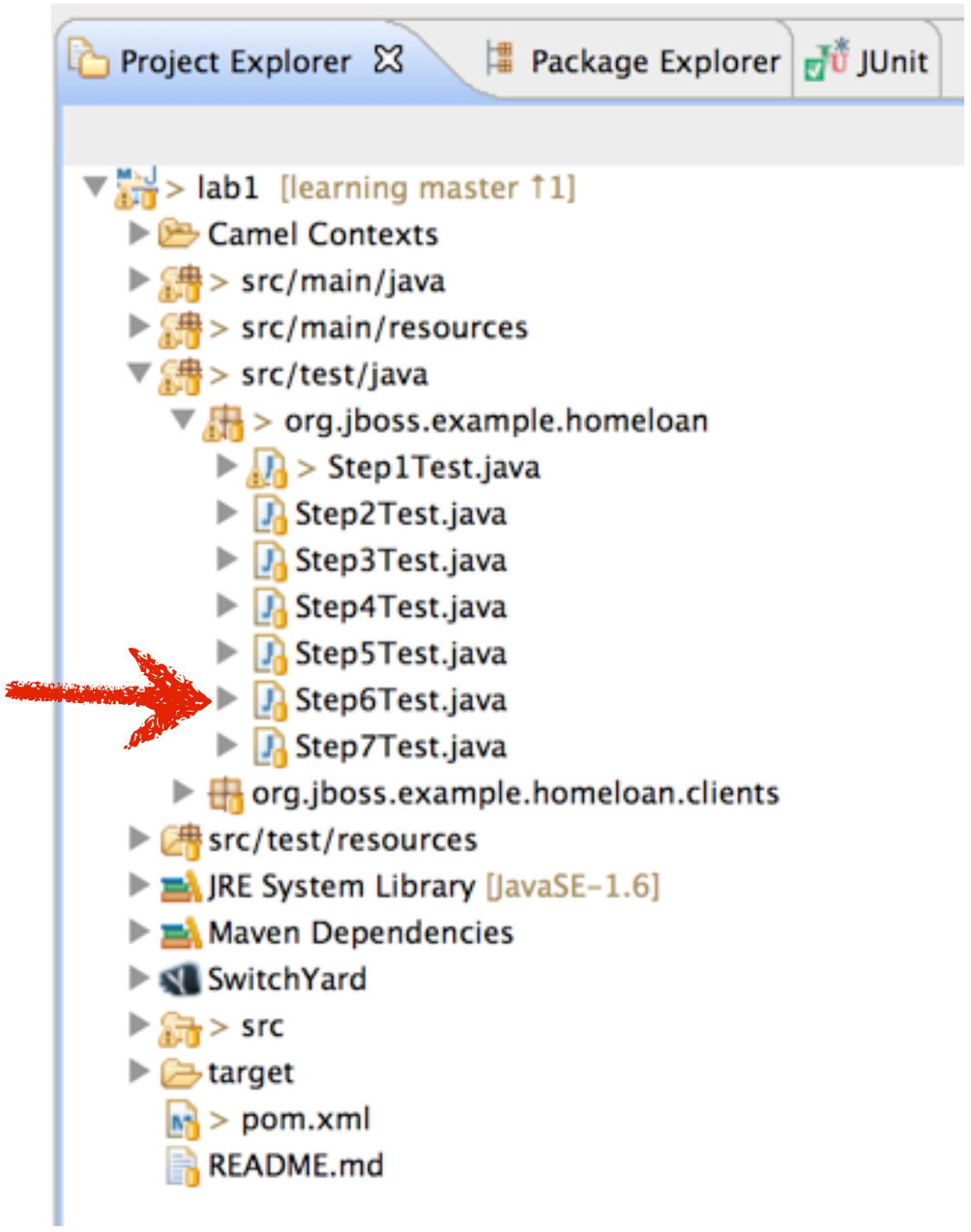
You have completed the changes required for step 6. Let's validate the changes using a service unit test.

TODO

1. Make sure the project is completely saved by selecting File -> Save All.

2. Double-click on Step6Test in the explorer to open the unit test.

3. Go to the Run menu in the main menu bar and select 'Run As -> JUnit Test' to run the unit test.

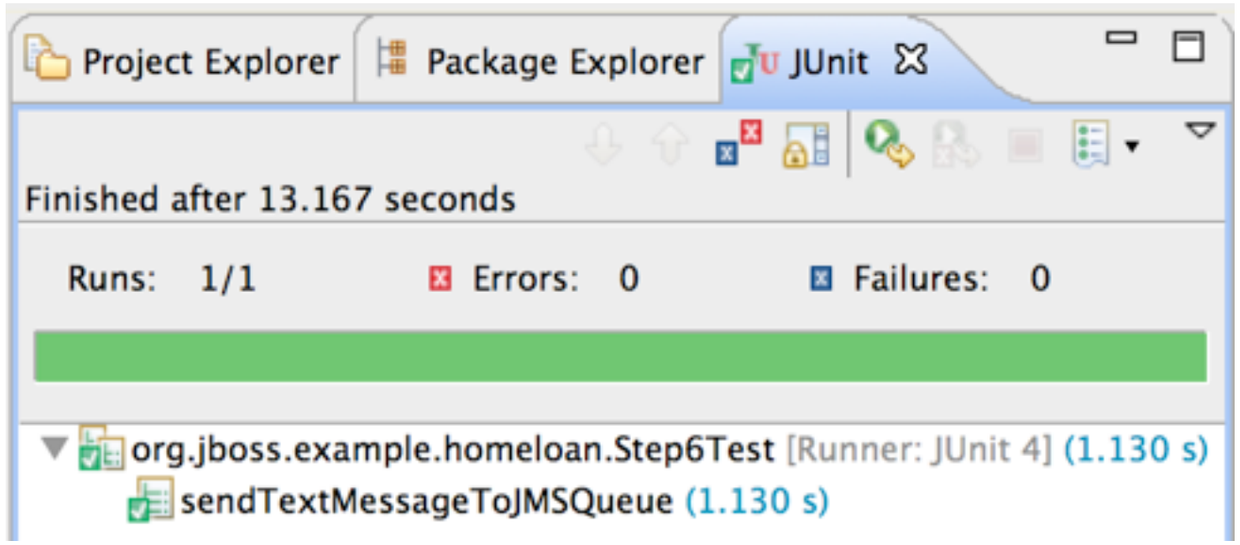


Step 6

Success?

FYI

You should see a nice green bar indicating that the test passed. If you get a red bar indicating test failure, raise your hand and someone will be by to help.



Step 7

REST Status Service

Goals

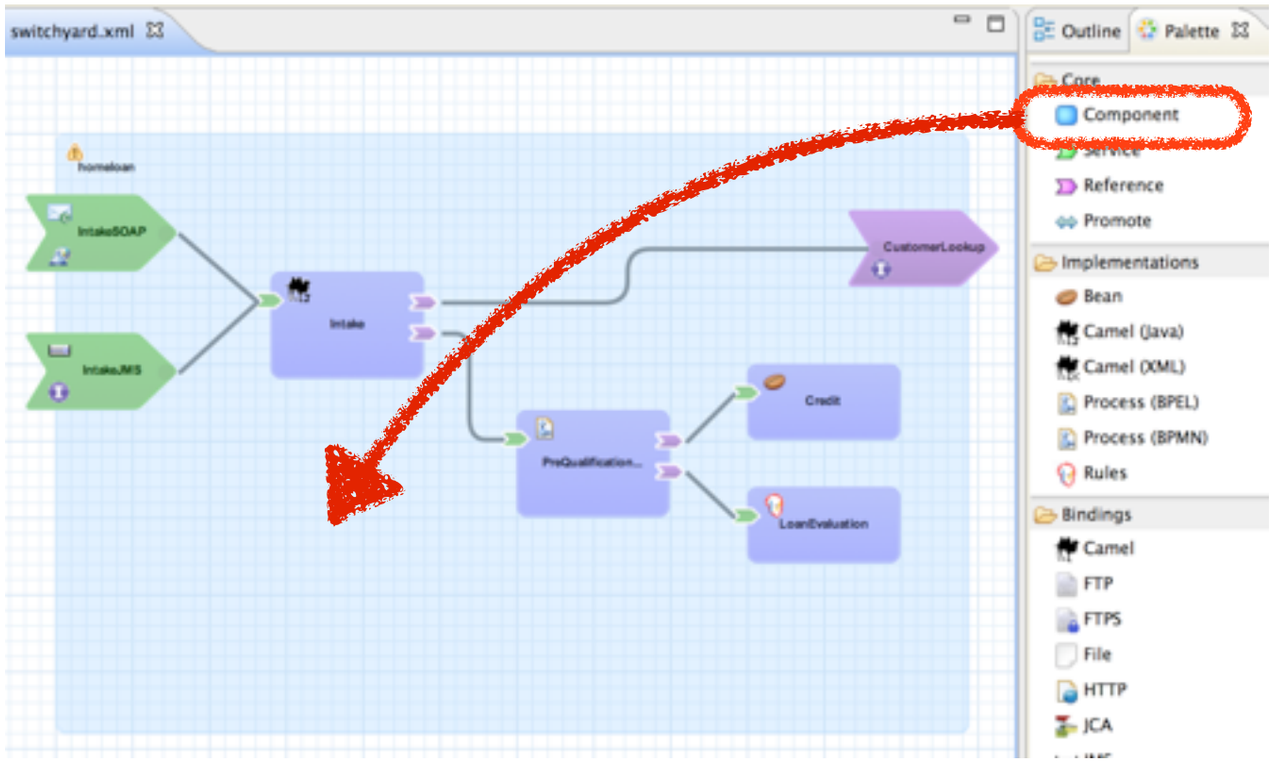
- *Create status service implementation using CDI Bean.*
- *Promote StatusService using Java contract.*
- *Define RESTful interface using JAX-RS.*
- *Add REST binding to StatusService.*
- *Run unit tests to verify changes.*

Step 7

StatusService Implementation

TODO

1. Click and hold on the Component icon in the palette and drag it onto an empty area of the canvas.



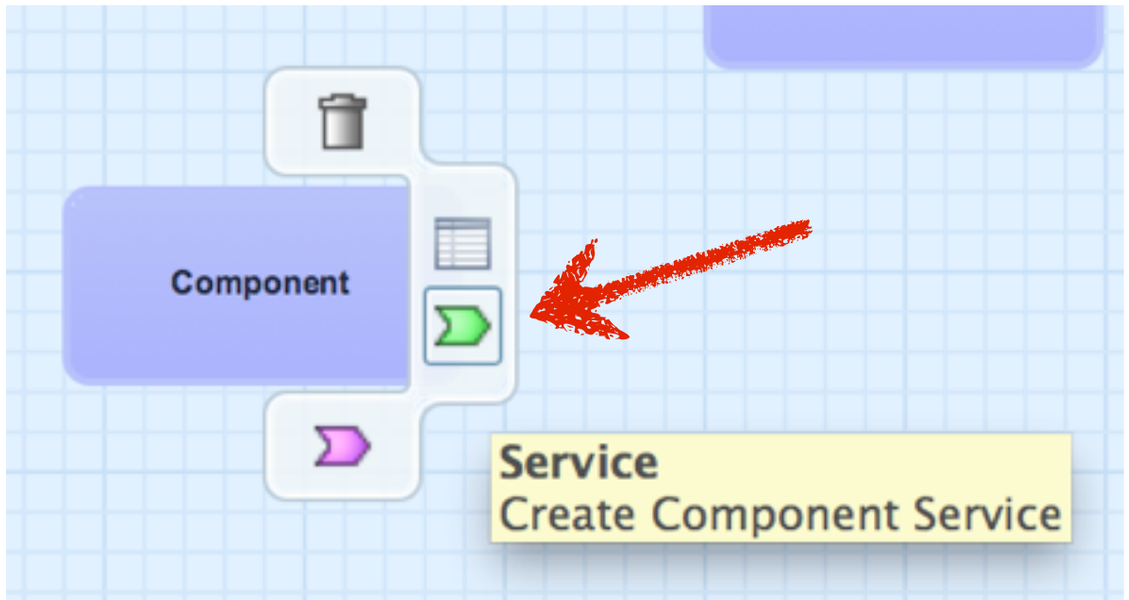
Step 7

StatusService Implementation

TODO

1. Hover over the new Component to access the button bar.

2. Click on the green Service icon to add a service to the component.

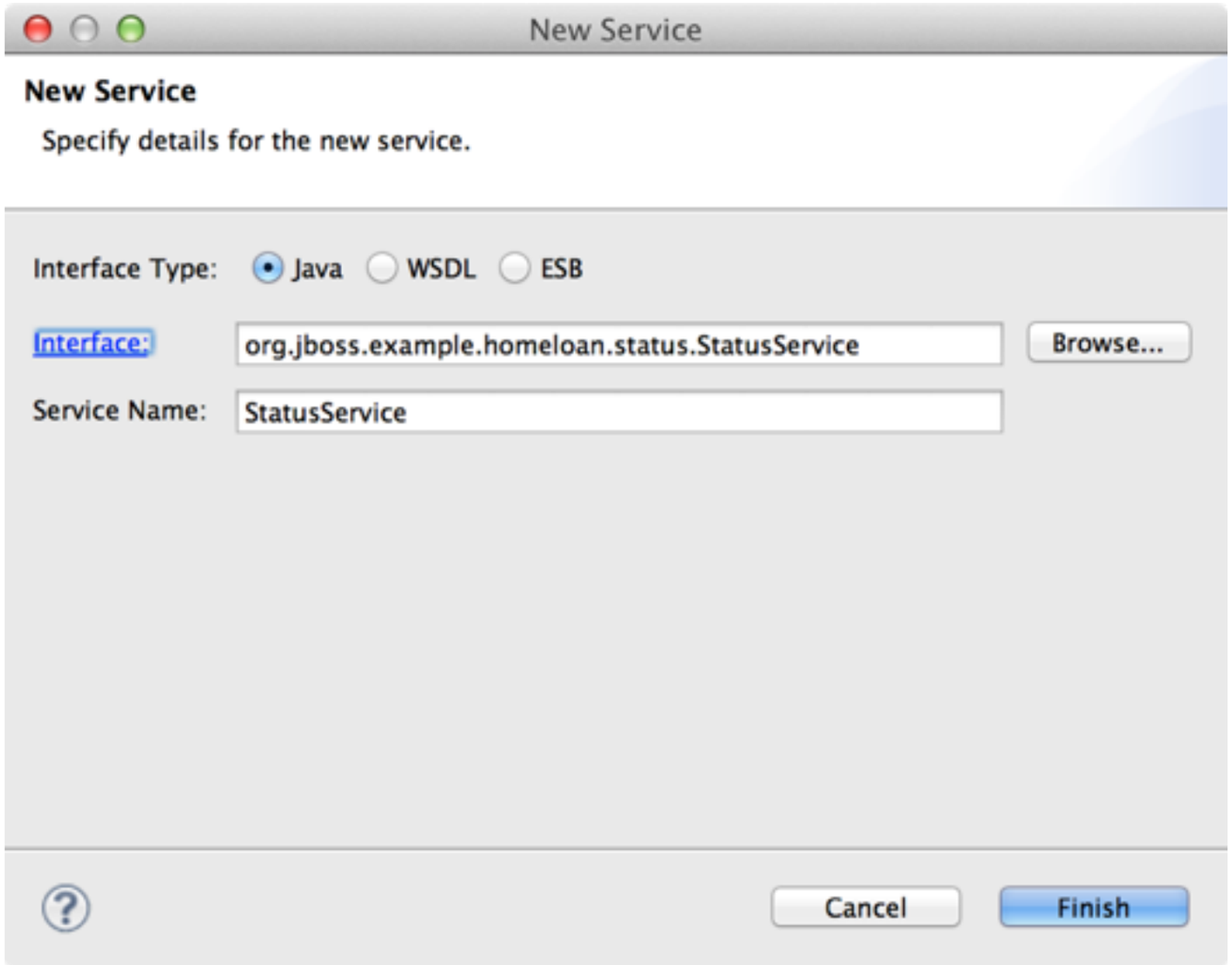


Step 7

StatusService Implementation

TODO

1. Click on the Browse ... button to select the service interface.
2. Begin typing ‘StatusService’ in the resulting dialog and select the StatusService interface when you see it in the list.
3. Click Finish.

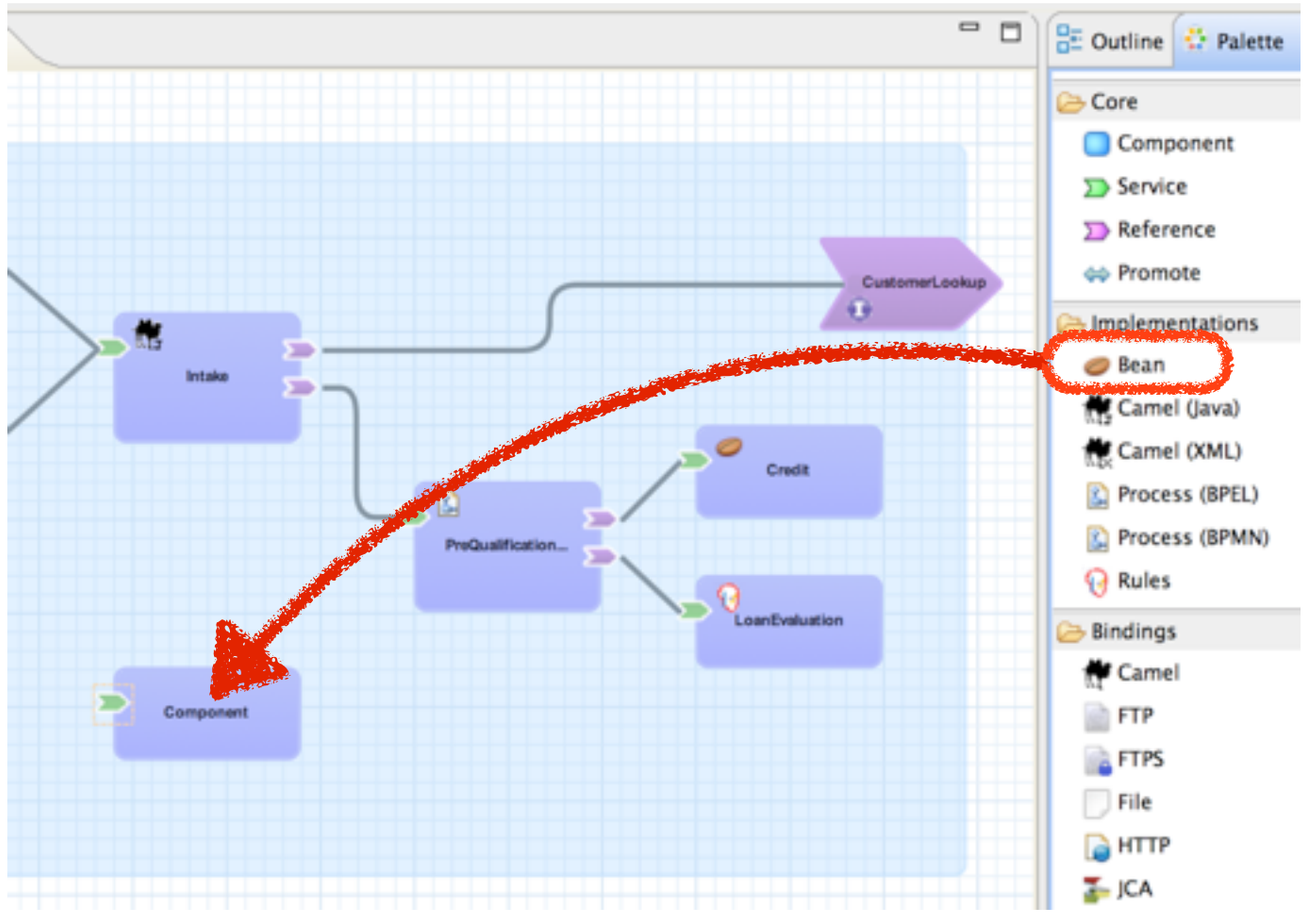


Step 7

StatusService Implementation

TODO

1. Click and hold on the Bean implementation in the palette and drag it on top of the new Component.

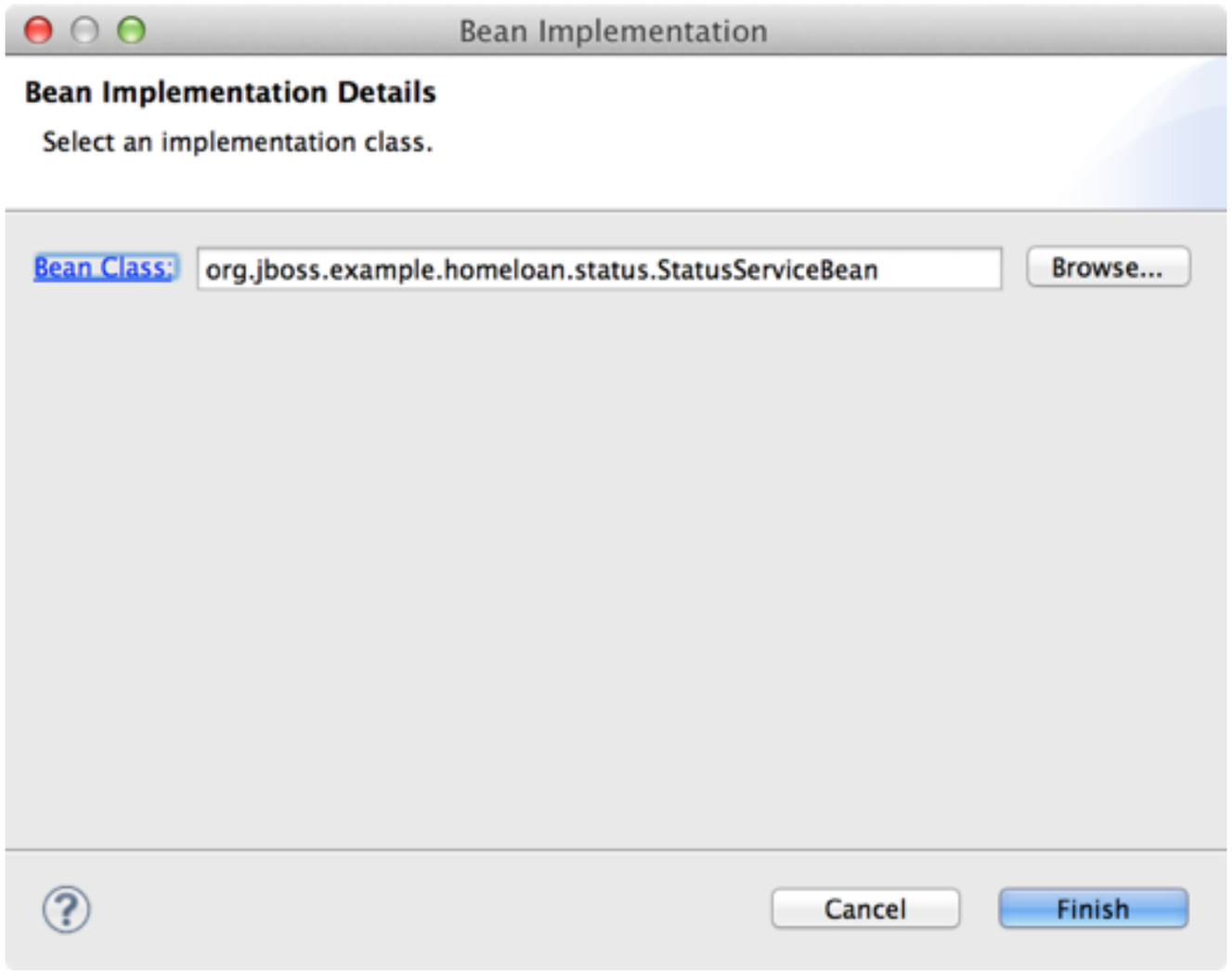


Step 7

StatusService Implementation

TODO

1. Click on the Browse ... button to select the bean implementation to use.
2. Enter StatusServiceBean in the input field and select it from the list.
3. Click Finish.

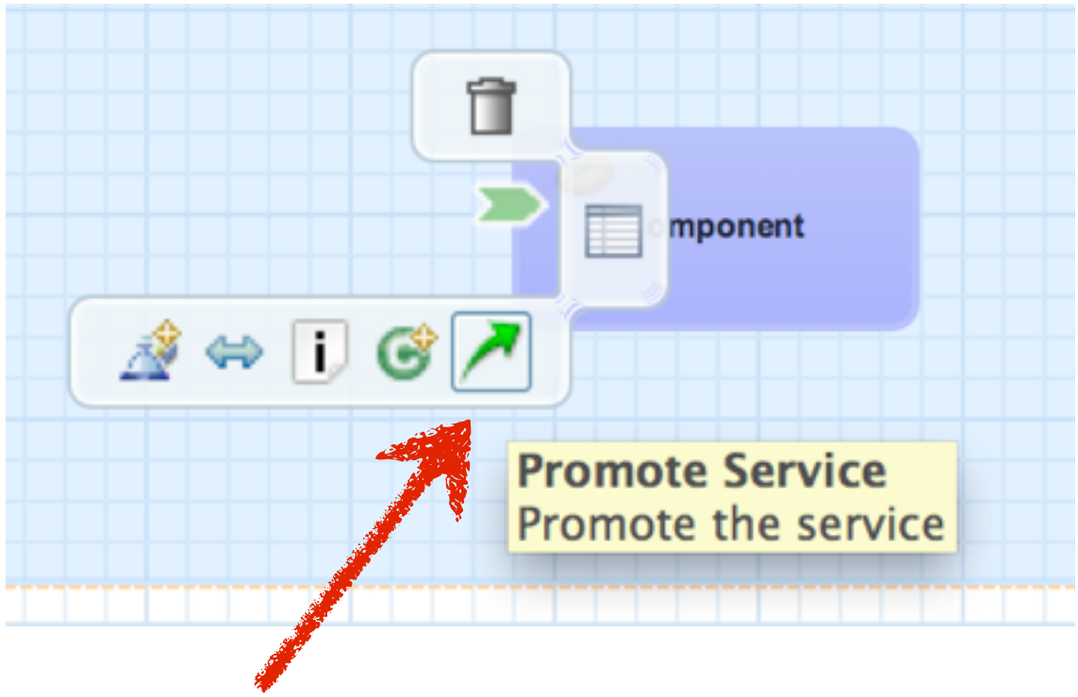


Step 7

Service Promotion

TODO

1. Hover over the green component service you just created.
2. Click on the promotion icon to promote the component service to a composite service.

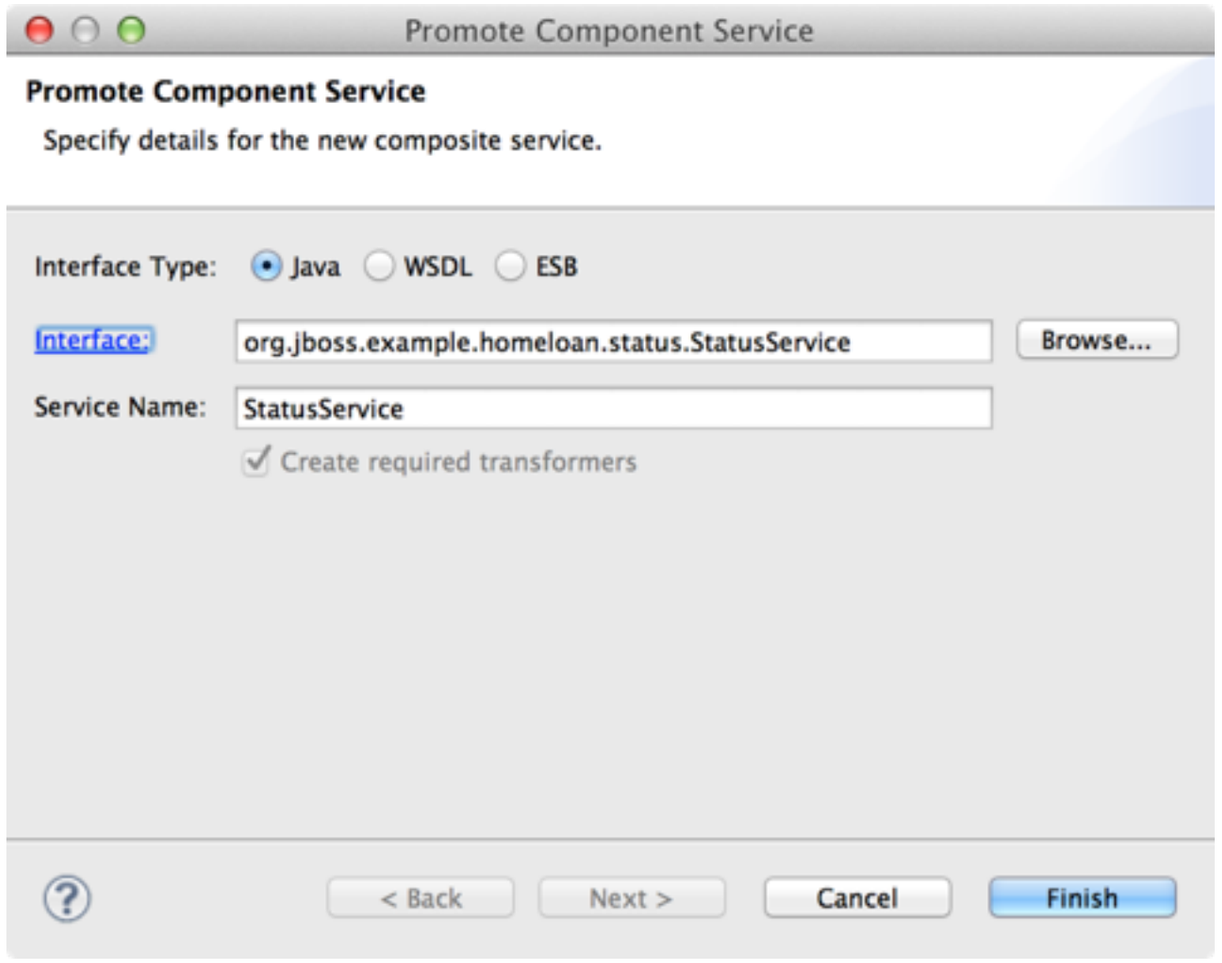


Step 7

Service Promotion

TODO

1. Click Finish.

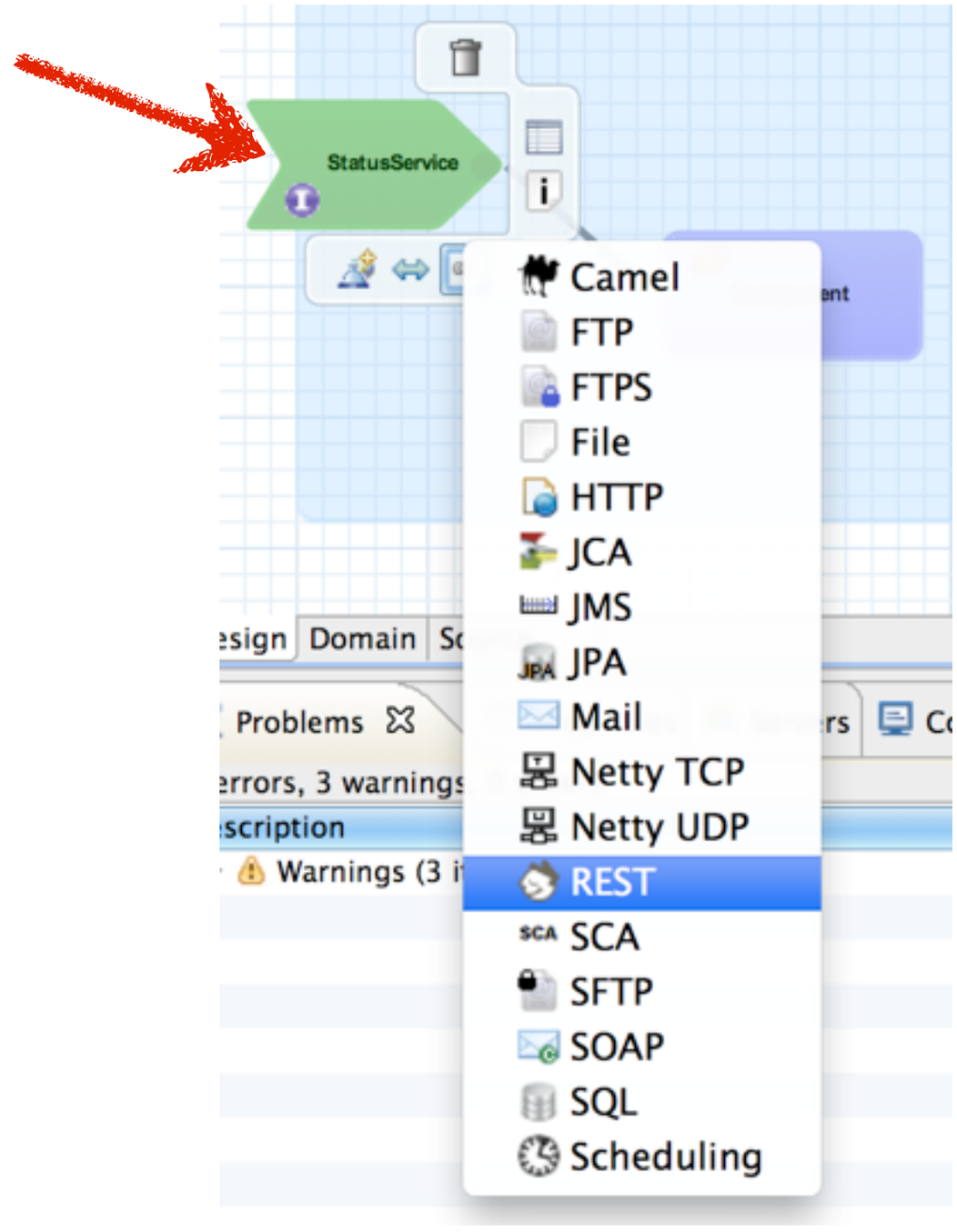


Step 7

Adding a REST Binding

TODO

1. Hover over the new composite service StatusService to bring up the button bar.
2. Click on the binding icon and select REST.

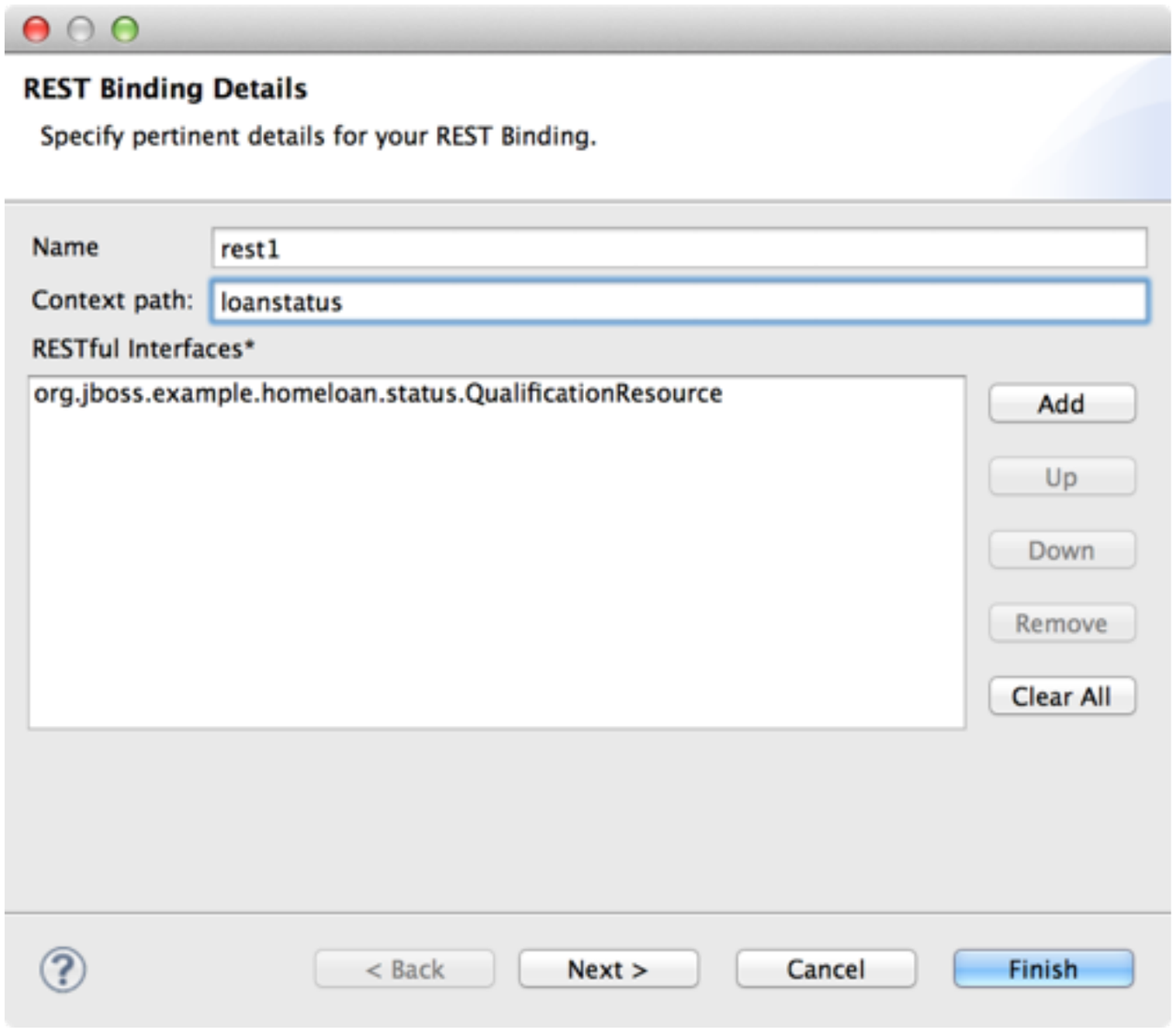


Step 7

Adding a REST Binding

TODO

1. `Context path : loanstatus`
2. Click Add to select the JAX-RS interface definition for the RESTful endpoint.
3. Enter QualificationResource in the input field and select it from the list.
4. Click Finish.



Step 7
Validate Changes

FYI

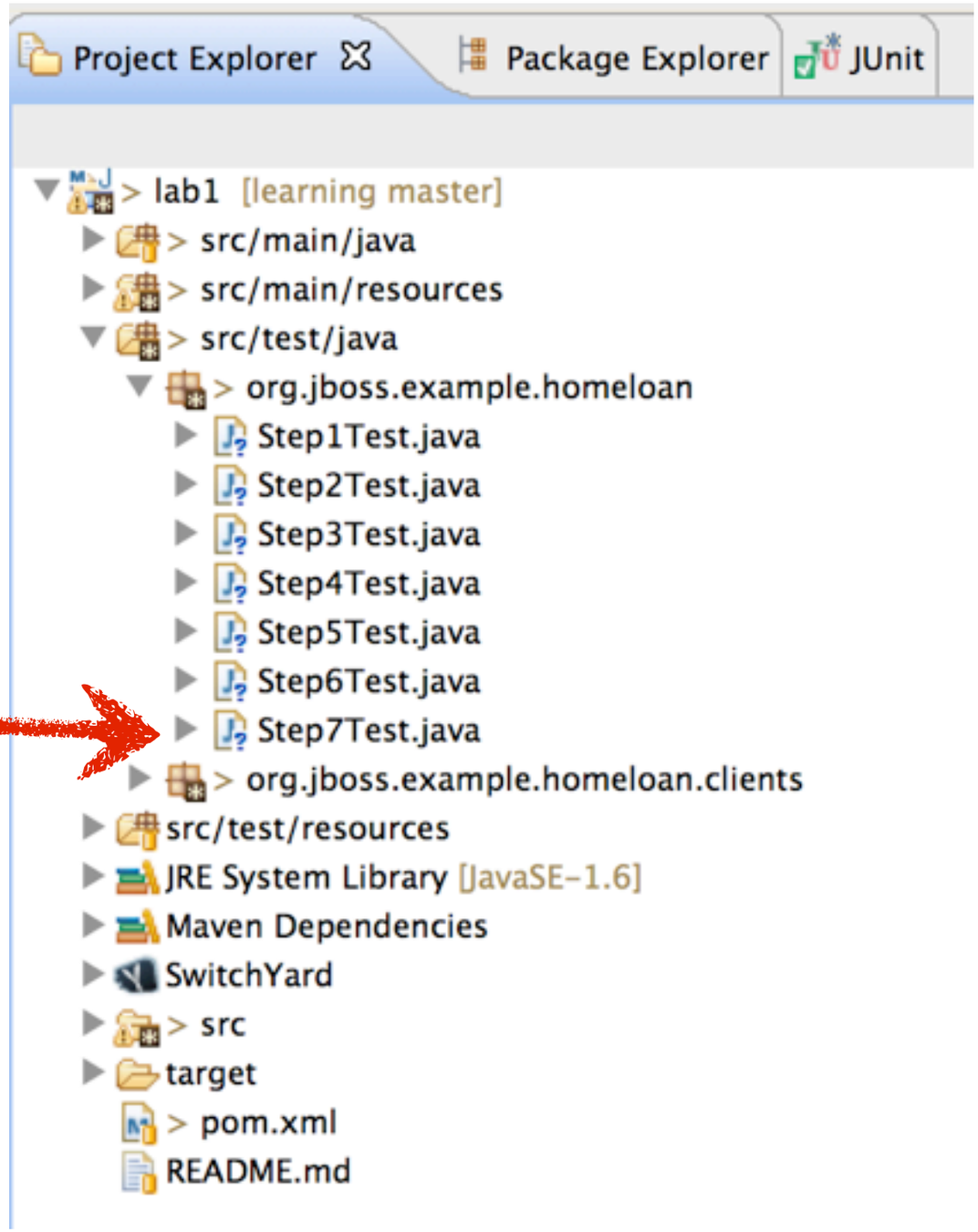
You have completed the changes required for step 7. Let's validate the changes using a service unit test.

TODO

1. Make sure the project is completely saved by selecting File -> Save All.

2. Double-click on Step7Test in the explorer to open the unit test.

3. Go to the Run menu in the main menu bar and select 'Run As -> JUnit Test' to run the unit test.

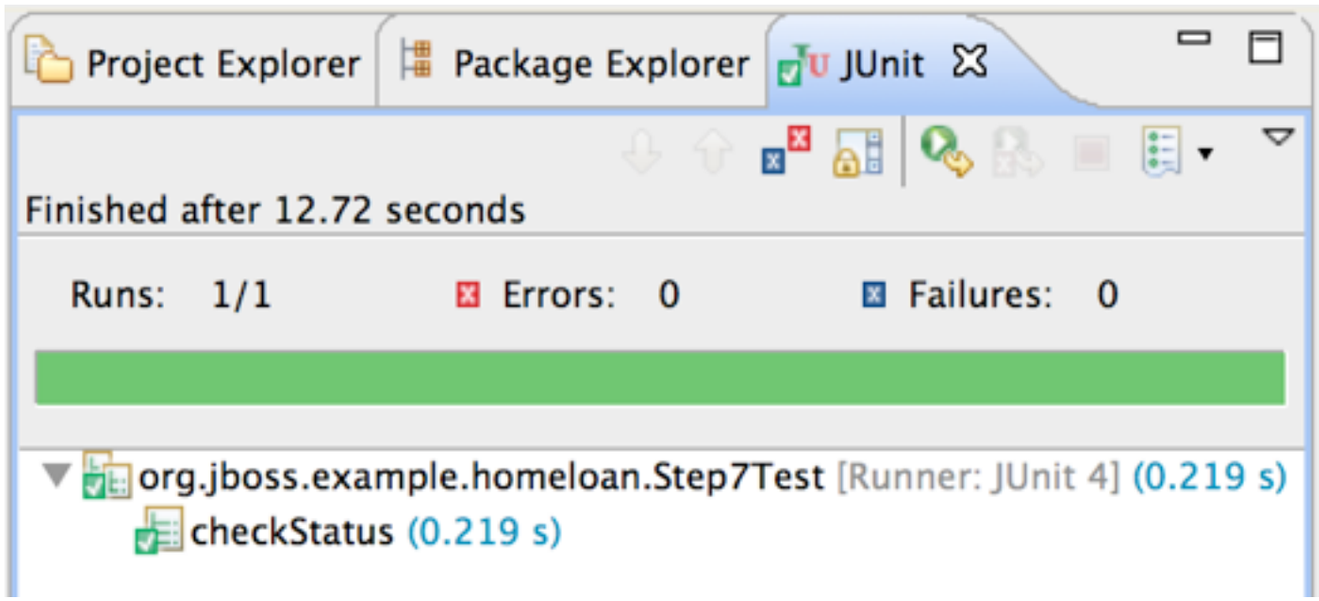


Step 7

Success?

FYI

You should see a nice green bar indicating that the test passed. If you get a red bar indicating test failure, raise your hand and someone will be by to help.



Lab 1 Complete!