Lab Exercise

Gateways - Lab 8

Gateways allow your non-Spring Integration components to access Spring Integration systems without having to know (and import) the SI API. In other words, gateways allow your other components to be loosely coupled to SI.

In this last lab of the tutorial series, you revisit the Pig Latin transformer. You create a gateway to separate a calling component, in this case a simple class with a main method, from the rest of the SI API or any messaging system.

Specifically, in this lab you will:

- Explore a simple transformation application.
- Create a gateway interface.
- Configure a gateway component.
- Modify a simple application to use the gateway and divorce itself from SI knowledge.

Lab solution folder: ExpressSpringIntegration\lab8\lab8-gateways-solution

Scenario

Already created for you is a simple Spring Integration application that uses the transformer logic you have already seen and used. The Startup class, as it has done for all the labs, contains a main( ) method that kicks off the SI application. In this case, it is used to grab the input MessageChannel and deposits a message into the MessageChannel. Unfortunately, this requires that Startup has a lot of information of SI components and the messaging system that underlies it all.

Here is a list of the imports in Startup at the start of this lab.

```
import
org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.messaging.Message;
import org.springframework.messaging.MessageChannel;
import org.springframework.messaging.support.MessageBuilder;
```

In this lab, you modify the application to decouple Startup from SI the API using a gateway.

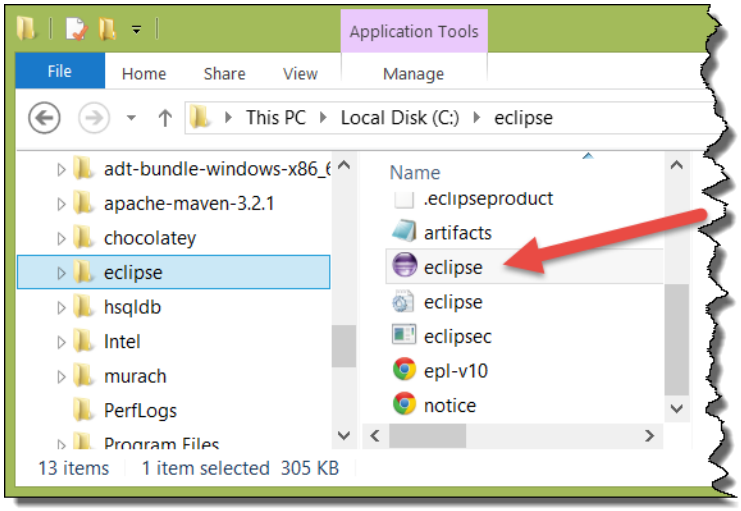
Step 1: Import the Maven Project

A Maven Project containing the tightly coupled Startup to SI APIs for Pig Latin transformation has already been created for you. You will use this project to begin your exploration of gateways.

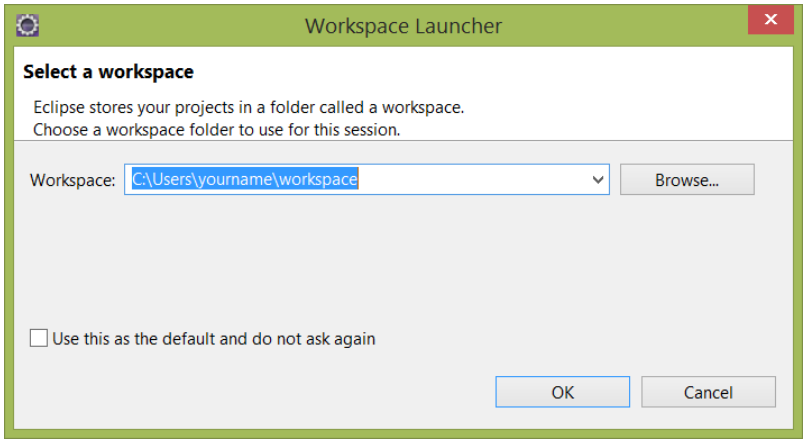
1.1 Start the Eclipse-based IDE. Locate and start Eclipse (or Eclipse-based) IDE.

1.1.1 Locate the Eclipse folder.

1.1.2 Start Eclipse by double clicking on the eclipse.exe icon (as highlighted in the image below).



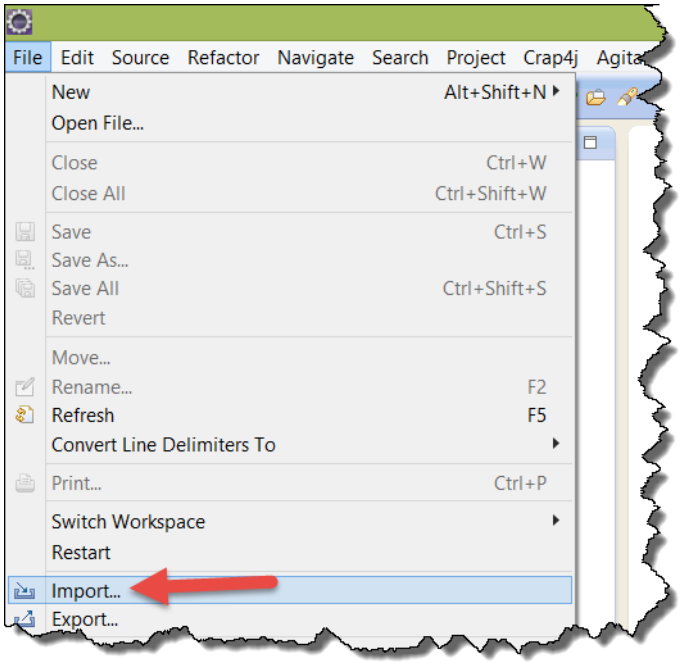
1.1.3 Open your workspace. Type in *C:\Users\<your username >\workspace* and click the *OK* button.



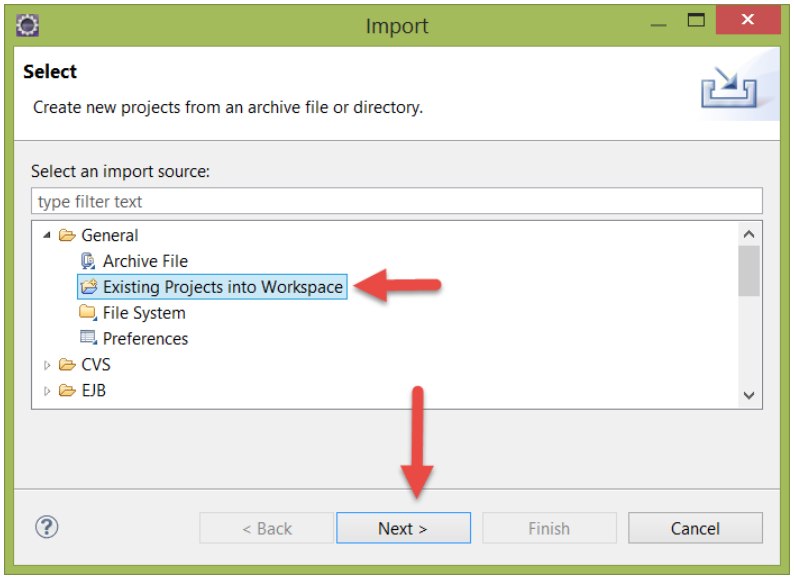
Note: As noted in the past labs, you may use an alternate location for your workspace, but the labs will always reference this location (c:\users\[your username]\workspace) as the default workspace location.

1.2 Import the new Maven Project.

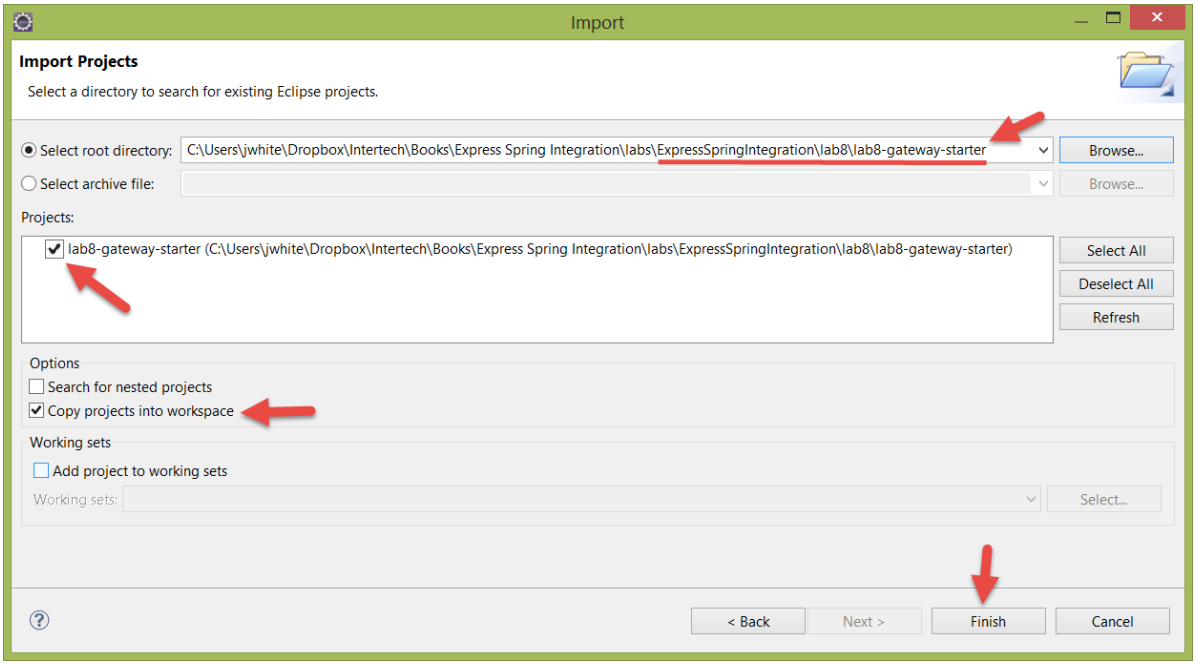
1.2.1 Select File>Import... from the Eclipse menu bar.



1.2.2 Locate and open the General folder in the Import window, and then select *Existing Projects into Workspace Project* from the options. Now push the *Next>* button.

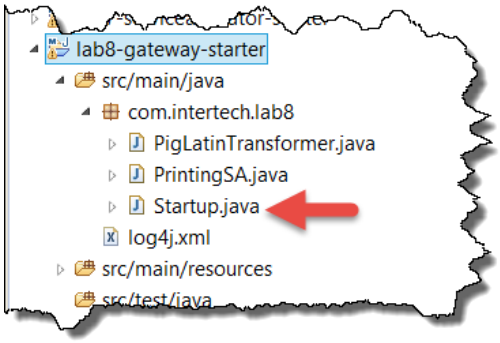


1.2.3 In the "Import" window, use the *Browse…* button to locate the lab8-gateway-starter project folder located in ExpressSpringIntegration\lab8 (this folder is located in the lab downloads). Make sure the project is selected, and the *"Copy projects into workspace"* checkbox is also checked before you hit the *Finish* button (as shown below).



1.3 Explore the project. Examine the project for the Spring Integration components that are already present.

1.3.1 Examine the Startup.java. Expand the src/main/java folder in the Project Explorer view, and open the Startup.java file by double clicking on it.

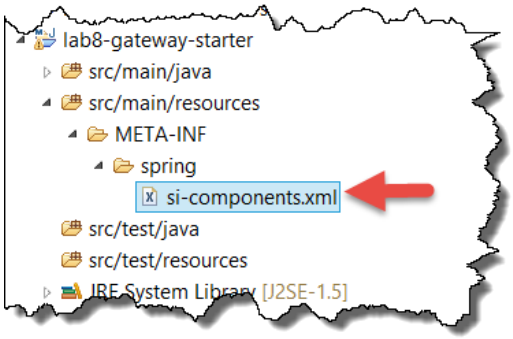


1.3.2 Examine the main( ) method of Startup. The Startup's main( ) method creates a SI message (containing a String payload) and then pushes the message onto the request channel of the SI application.

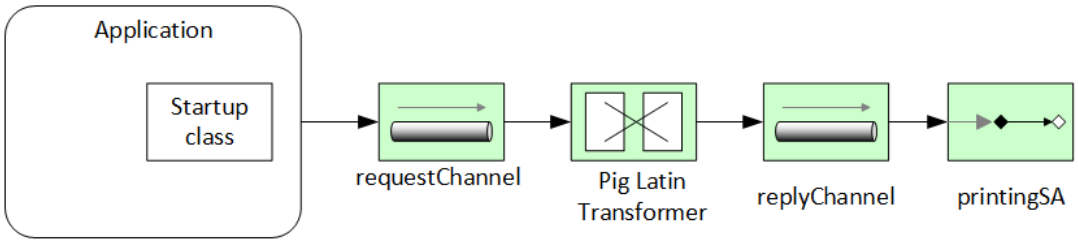
```
public static void main(String[] args) {
  ClassPathXmlApplicationContext context = new
    ClassPathXmlApplicationContext(
    "/META-INF/spring/si-components.xml");

  MessageChannel channel = context.getBean("requestChannel",
    MessageChannel.class);
  Message<String> message = MessageBuilder.withPayload(
    "Hello brave new world").build();
  channel.send(message);

  context.close();
}
```

1.3.3 Note, however, that the Startup class must import many of the SI types – therefore causing a type coupling to SI. Think of Startup as an application you wish to keep independent of SI. How? With a gateway.

1.3.4 Examine the SI configuration. Expand the src/main/resources/META-INF/spring folder in the Project Explorer view, and open si-components.xml file by double clicking on it. The chief component in the si-components.xml file is a Pig Latin transformer. String messages from the requestChannel (placed there by Startup) are sent to the transform where the String payload is translated to Pig Latin and then placed in a new message in the replyChannel. A service activator displays the Pig Latin translation to the Console view.



Below is the EIP diagram of the application as it currently stands.

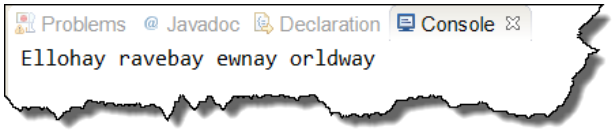


Step 2: Test the Tightly-Coupled SI Application

2.1 Test the application. Test the application to see the Startup application create a String payload message ("Hello brave new world") and send it into the SI system for Pig Latin transformation.

2.1.1 Locate the Startup.java file in the source folder. Right click on file and select Run As > Java Application from the resulting menu.

2.1.2 Check the Console view for results.



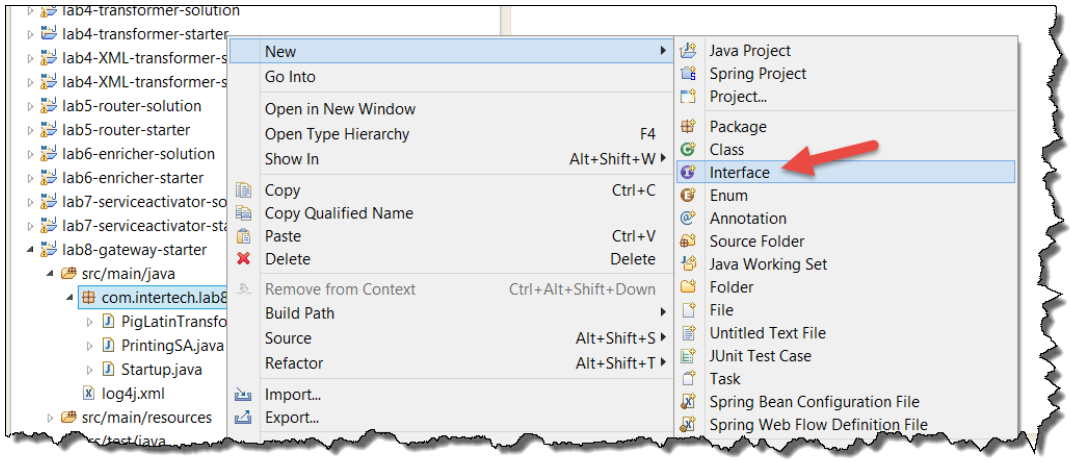
2.2 Unlike other applications, there is not infinite loop in this application. Startup closes the Spring application context and ends on its own. So, there is no need to terminate the application.

Step 3: Create Gateway Interface

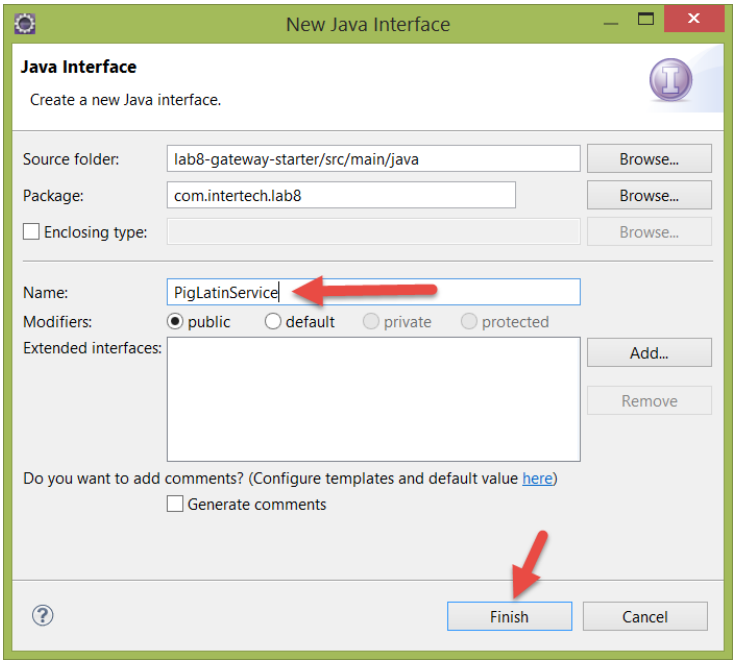
The gateway interface is meant to hide the rest of the application from SI or other messaging details. In this step, you create the gateway interface, which SI implements (as a proxy) at runtime.

3.1 Create an interface that well serve as the façade to the SI application.

3.1.1 Locate the com.intertech.lab8 package, right click on it and select New > Interface from the resulting menu.



3.1.2 In the New Java Interface window, enter PigLatinService as the interface name and then press the Finish button.



3.2 Code the PigLatinService interface.

3.2.1 Add a translate( ) method to the interface. The translate method should take the English string to be translated to Pig Latin and it should return the translated Pig Latin String.

```
String translate(String english);
```

3.2.2 Note that the interface makes no imports – in particular no imports of SI types.

3.2.3 Save the interface and make sure there are not compile errors.

Step 4: Add and Configure the Gateway

In this step, define a gateway component using the interface created above. Spring Integration provides org.springframework.integration.gateway.GatewayProxyFactoryBean that generates a proxy for that service interface.

4.1 Add the Gateway SI component.

4.1.1 Locate the si-components.xml file in src/main/resource/META-INF/spring and open it by double clicking on the file.

4.1.2 Add a new gateway component using the PigLatinService interface and defining its default request and reply channels as shown below.

```
<int:gateway id="latinService"
  service-interface="com.intertech.lab8.PigLatinService"
   default-request-channel="requestChannel" default-reply-
channel="replyChannel" />
```

Note: The reply channel is optional, but in this case, you want the Startup application to receive the translated String text.

4.2 Modify Startup.java to use the gateway.

4.2.1 Locate, open, and explore the Startup.java class in the com.intertech.lab8 package by double clicking on the file in the Package Explorer.

4.2.2 Comment out or remove the lines of code in the main( ) method that places the message into the SI request

```
public static void main(String[] args) {
  ClassPathXmlApplicationContext context = new
    ClassPathXmlApplicationContext(
    "/META-INF/spring/si-components.xml");

  // MessageChannel channel = context.getBean("requestChannel",
  // MessageChannel.class);
  // Message<String> message =
  // MessageBuilder.withPayload("Hello brave new world").build();
  // channel.send(message);

  context.close();
}
```

4.2.3 Now add lines of code right before the context close that call on the gateway to invoke the PigLatinService. Capture the return translated String from the service and display it using standard out (shown below).

```
public static void main(String[] args) {
  ClassPathXmlApplicationContext context = new
    ClassPathXmlApplicationContext(
    "/META-INF/spring/si-components.xml");

  // MessageChannel channel = context.getBean("requestChannel",
  // MessageChannel.class);
  // Message<String> message =
  // MessageBuilder.withPayload("Hello brave new world").build();
  // channel.send(message);

  PigLatinService service = context.getBean("latinService",
    PigLatinService.class);
  System.out.println(service.translate("Hello brave new world"));

  context.close();
}
```

Note: if you get stuck or feel like not typing in all the code yourself, you will find a working copy of the final Startup.java file at ExpressSpringIntegration\lab8\lab8-gateway-solution.

4.2.4 Remove the unnecessary Spring Integration imports in the Startup class by hitting Control-Shift-O. The class (representing your application that wants to be loosely coupled from SI or any messaging system) should now contain only an import for the Spring container.

```
import
org.springframework.context.support.ClassPathXmlApplicationContext;
```

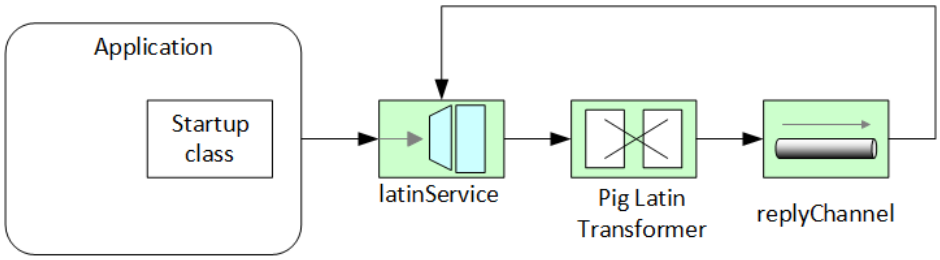
4.2.5 Save the class and make sure there are no compile errors in the project.

4.3 Remove the standard-output-displaying service activator.

4.3.1 Since the gateway returns the translated String to the Startup application, the service activator is no longer needed. Comment out or remove the service activator and associated Spring bean in si-components.xml.

```
<!-- <int:service-activator ref="printingSA" -->
<!-- input-channel="replyChannel" /> -->

<!-- <bean id="printingSA" class="com.intertech.lab8.PrintingSA"/> -->
```

4.3.1 With the completion of the gateway and removal of the service activator, the SI application can now be represented by the Wolfe and Hohpe EIP model below.



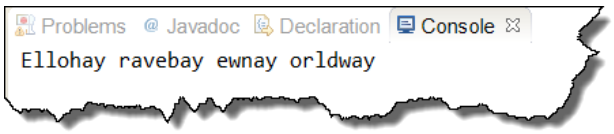
4.4 Save the configuration file and make sure there are no errors in the file.

Step 5: Test the Loosely-Coupled SI Application and the Gateway

5.1 Test the application. Again, test the application to see the Startup application create a String payload message ("Hello brave new world") and send it into the SI system for Pig Latin transformation.

5.1.1 Locate the Startup.java file in the source folder. Right click on file and select Run As > Java Application from the resulting menu.

5.1.2 Check the Console view for results. The application should behave the same way and produce the same results. The difference is that the application is now removed from the SI API.



5.2 As mentioned previously, there is not infinite loop in this application. Startup closes the Spring application context and ends on its own. So, there is no need to terminate the application.

Step 6: Asynchronous Gateway

While the application (Startup.java) is now void of SI knowledge, there is a new problem to resolve. The gateway, as designed to this point in the lab, is synchronous. That is, the application makes a request of the gateway (and the SI system under the covers) and blocks waiting for a return from the gateway. In this simple example, that is not to terribly bad since the String sent to the translation service is small and the work accomplished by the translation server and the SI components gets accomplished quickly. Imagine that the Spring Integration path was much larger with many more tasks (filtering, enriching, transforming, etc.) to be accomplished in the path. In this case, the application could be waiting for quite a while. To change this what is needed is a gateway that operates asynchronously. That is, the application can make a request of the gateway service and allow the SI system to take as long as it needs to respond without hampering the application from continuing to work. At a designated time of the application's choosing, it should be able check for and get the response from the gateway service. In this step, you make the PigLatinService asynchronous.

6.1 Modify the gateway. Make the gateway service method return a Future.

6.1.1 Locate, open, and explore the PigLatinService.java interface in the com.intertech.lab8 package by double clicking on the file in the Package Explorer.

6.1.2 Alter the return type of the translate( ) method from String to Future<String> as shown below. A Future is not a Spring Integration type but rather a Java concurrency type that represents a result of an asynchronous computation to be returned in the future.

```
Future<String> translate(String english);
```

6.1.1 Add the necessary import for Future to the interface by hitting Control-Shift-O.


```
import java.util.concurrent.Future;
```

6.1.2 Save the interface and make sure there are no compile errors in the project.

6.2 Modify the application to use the Future.

6.2.1 Locate, open, and explore the Startup.java class in the com.intertech.lab8 package by double clicking on the file in the Package Explorer.

6.2.2 In the main( ) method, capture the Future object returned by the gateway service. Then, call on the Future for return the results with a call to get( ). Alter the System.out.println( ) call to print that return String to the standard output as shown below.

```
PigLatinService service = context.getBean("latinService",
  PigLatinService.class);
Future<String> future = service.translate("Hello brave new world");
// do more work here in a real application

String serviceOutput = future.get(5000, TimeUnit.SECONDS);
System.out.println(serviceOutput);
```

Note: In a real application, the application is free to go about doing other work and is not blocked when the call to translate( ) is made. The call to get( ) takes parameters to inform the system how long to wait, if necessary, to retrieve the results from the asynchronous process – which in this case is the return from the SI components.

6.2.3 Add the necessary Java concurrent imports to the class by hitting Control-Shift-O.

```
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
```

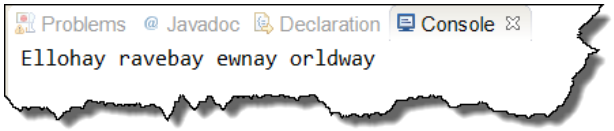
6.2.4 Save the class and make sure there are no compile errors in the project.

Step 7: Test the Asynchronous Gateway

7.1 Test the application. Again, test the application to see the Startup application create a String payload message ("Hello brave new world") and send it into the SI system for Pig Latin transformation – this time in an asynchronous fashion.

7.1.1 Locate the Startup.java file in the source folder. Right click on file and select Run As > Java Application from the resulting menu.

7.1.2 Check the Console view for results. The application should behave the same way and produce the same results.



To see the impact of your change, have the application do more work (for example a long running loop) in the toPigLatin( ) method of the PigLatinTransformer class. The application can continue to work after calling on the service and still receive the SI return without impact.

Lab Solution

si-components.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:int="http://www.springframework.org/schema/integration"
  xmlns:int-
file="http://www.springframework.org/schema/integration/file"
  xmlns:int-
mail="http://www.springframework.org/schema/integration/mail"
  xmlns:int-
xml="http://www.springframework.org/schema/integration/xml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:int-
stream="http://www.springframework.org/schema/integration/stream"
  xsi:schemaLocation="
  http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/spring-
integration.xsd
  http://www.springframework.org/schema/integration/stream
http://www.springframework.org/schema/integration/stream/spring-
integration-stream.xsd
  http://www.springframework.org/schema/integration/file
http://www.springframework.org/schema/integration/file/spring-
integration-file.xsd
  http://www.springframework.org/schema/integration/xml
http://www.springframework.org/schema/integration/xml/spring-
integration-xml.xsd">

  <int:gateway id="latinService" service-
interface="com.intertech.lab8.PigLatinService"
    default-request-channel="requestChannel" default-reply-
channel="replyChannel" />

  <int:channel id="requestChannel" />

  <int:transformer input-channel="requestChannel"
    output-channel="replyChannel" ref="pigLatinTransformer" />
  <bean id="pigLatinTransformer"
class="com.intertech.lab8.PigLatinTransformer" />

  <int:channel id="replyChannel" />

  <!-- <int:service-activator ref="printingSA" input-
channel="replyChannel"/> -->

  <!-- <bean id="printingSA" class="com.intertech.lab8.PrintingSA" />
-->
</beans>
```


Startup.java (pre-Asynchronous work)

```
package com.intertech.lab8;

import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class Startup {

  public static void main(String[] args) {
    ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext(
        "/META-INF/spring/si-components.xml");

    // MessageChannel channel = context.getBean("requestChannel",
    // MessageChannel.class);
    // Message<String> message =
    // MessageBuilder.withPayload("Hello brave new world").build();
    // channel.send(message);

    PigLatinService service = context.getBean("latinService",
        PigLatinService.class);
    System.out.println(service.translate("Hello brave new world"));

    context.close();
  }
}
```

PigLatinService (pre-Asynchronous work)

```
package com.intertech.lab8;

public interface PigLatinService {
  String translate(String english);
}
```


Startup.java (post-Asynchronous work)

```
package com.intertech.lab8;

import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class Startup {

  public static void main(String[] args) throws Exception {
    ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext(
        "/META-INF/spring/si-components.xml");

    PigLatinService service = context.getBean("latinService",
        PigLatinService.class);
    Future<String> future = service.translate("Hello brave new
world");
    // do more work here in a real application
    String serviceOutput = future.get(5000, TimeUnit.SECONDS);
    System.out.println(serviceOutput);
    context.close();
  }
}
```

PigLatinService (post-Asynchronous work)

```
package com.intertech.lab8;

import java.util.concurrent.Future;

public interface PigLatinService {

   Future<String> translate(String english);
}
```

