Integration Workbook

Last updated: February 15, 2013

© Copyright 2000–2013 salesforce.com, inc. All rights reserved. Salesforce.com is a registered trademark of salesforce.com, inc., as are other names and marks. Other marks appearing herein may be trademarks of their respective owners.

Table of Contents

Force.com Integration Workbook..1

Before You Begin..2

Tutorial #1: Create a New Heroku Application..3

Step 1: Clone the Github Project..3
Step 2: Create a Heroku Project..3
Step 3: Test the Application..5
Summary..5

Tutorial #2: Connecting the Warehouse App with an External Service 6

Step 1: Create an External ID Field on Invoice Statement..6
Step 2: Create a Remote Site Record..6
Step 3: Create an Integration Apex Class..7
Step 4: Test the @future Method..9
Step 5: Create a Trigger to Call the @future Method..10
Step 6: Test the Complete Integration Path..10
Summary..12

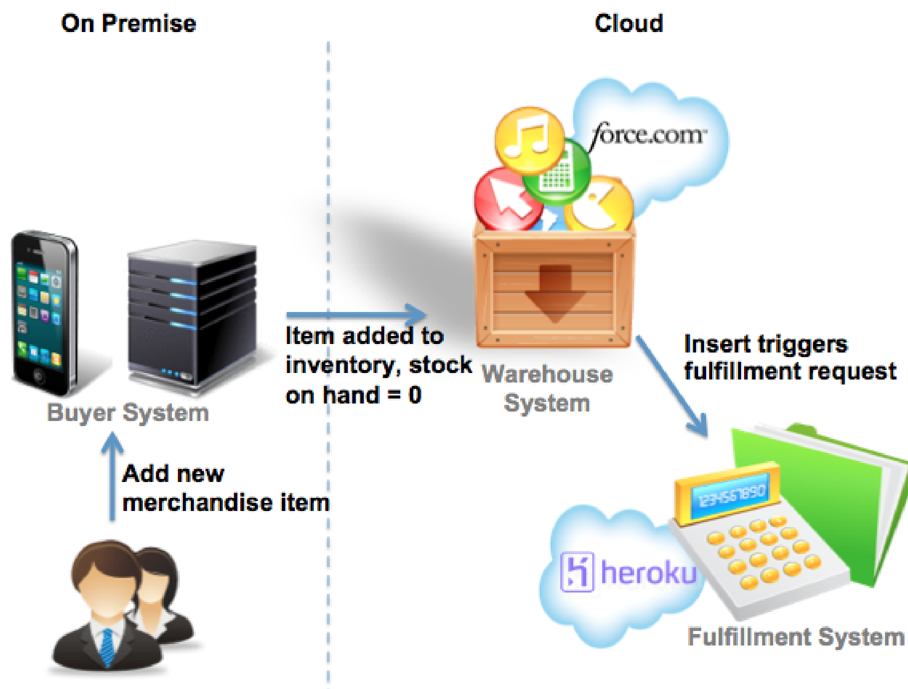
Tutorial #3: Update the Heroku App..13

Step 1: Configure Remote Access..13
Step 2: Update your Application with a New Branch..14
Step 3: Update the User Interface with Invoice Information..15
Summary..16

Force.com Integration Workbook

One of the most frequent tasks Force.com developers undertake is integrating Force.com apps with existing applications. The tutorials within this workbook are designed to introduce the technologies and concepts required to achieve this functionality.

The Force.com Integration Workbook is intended to be the companion to the Force.com Workbook. The series of tutorials provided here extend the Warehouse application by connecting it with a cloud-based fulfillment app.



Intended Audience

This workbook is intended for developers new to the Force.com platform but have basic working knowledge in Java.

Tell Me More....

This workbook is designed so that you can go through the steps as quickly as possible. At the end of some steps, there is an optional Tell Me More section with supporting information.

- You can find the latest version of this and other workbooks at `developer.force.com/workbooks`.
- To learn more about Force.com and to access a rich set of resources, visit Developer Force at `http://developer.force.com`.

Before You Begin

Before you begin the tutorials, you must have completed at least tutorials 1 through 3 in the Force.com Workbook to create the Warehouse data model. If you'd prefer to install the Warehouse data model as a package, you can install it from this link: https://login.salesforce.com/packaging/installPackage.apexp?p0=04tE0000000Pzdj.

You'll also need a developer account on Heroku and have the Heroku Toolbelt software installed. The following steps show you how to do this.

Step 1: Create a Heroku Account

Heroku is a cloud application platform separate from Force.com. It provides a powerful Platform as a Service for deploying applications in a multitude of languages, including Java. It also enables you to easily deploy your applications with industry-standard tools, such as git. If you don't already have a Heroku account you can create a free account as follows:

1. Navigate to http://heroku.com.
2. Click **Sign Up**.
3. Enter your email address.
4. Wait a few minutes for the confirmation email and follow the steps included in the email.

Step 2: Install the Heroku Toolbelt

The Heroku Toolbelt is a free set of software tools that you'll need to work with Heroku. To install the Heroku Toolbelt:

1. Navigate to https://toolbelt.heroku.com/.
2. Select your development platform (Mac OS X, Windows, Debian/Ubuntu) if it isn't already auto-selected for you.
3. Click the download button.
4. After the download finishes, run the downloaded install package on your local workstation and follow the steps to install.

Tutorial #1: Create a New Heroku Application

Heroku provides a powerful Platform as a Service for deploying applications in a multitude of languages, including Java. In this tutorial, you create a Web application using the Java Spring MVC framework to mimic handling fulfillment requests from our Warehouse application.

Familiarity with Java is helpful, but not required for this exercise. The tutorial starts with an application template to get you up and running. You then walk through the steps to securely integrate the application with the Force.com platform.

Step 1: Clone the Github Project

Git is a distributed source control system with an emphasis on speed and ease of use. Heroku integrates directly into git, allowing for continuous deployment of your application by pushing changes into a repository on Heroku. Github is a Web-based hosting service for git repositories.

You start with a pre-existing Spring MVC-based application stored on GitHub, and then as you make changes you deploy them into your Heroku account and see your updates available online via Heroku's cloud framework.

1. Open a command line terminal. For Mac OS X users, this can be done by going to the Terminal program, under `Applications/Utilities`. For PC users, this can be done by going to the **Start Menu**, and typing `cmd` into the Run dialog.
2. Once in the command line terminal, change to a directory where you want to download the example app. For example, if your directory is "development", type `cd development`.
3. Execute the following command:

```
git clone https://github.com/developerforce/spring-mvc-fulfillment-base
```

Git downloads the existing project into a new folder, `spring-mvc-fulfillment-base`.

Step 2: Create a Heroku Project

Now that you have the project locally, you need a place to deploy it that is accessible on the Web. In this step you deploy the app on Heroku.

1. In the command line terminal, change directory to the `spring-mvc-fulfillment-base` folder you created in the last step:

```
cd spring-mvc-fulfillment-base
```

2. Execute the following command to login to Heroku (followed by Heroku login credentials, if necessary):

```
heroku login
```

3. Execute the following command to create a new application on Heroku:

```
heroku create
```

Heroku creates a local git repository as well as a new repository on its hosting framework where you can push applications, and adds the definition for that remote deployment for your local git repository to understand. This makes it easy to leverage git for source control, make local edits and then deploy your application to the Heroku cloud.

All application names on Heroku must be unique, so you'll see messages like the following when Heroku creates a new app:

```
Creating quiet-planet-3215... done
```

Important: The output above shows that the new application name is `quiet-planet-3215`. You may want to copy and paste the generated name into a text file or otherwise make a note of it. Throughout this workbook there are references to the application name that look like `{appname}` that should be replaced with your application name. So, if your application name is `quiet-planet-3215`, when a tutorial step prompts you to enter a URL with the format `https://{appname}.herokuapp.com/_auth`, you would use: `https://quiet-planet-3215.herokuapp.com/_auth`.

4. To deploy the local code to Heroku, execute the following command:

```
git push heroku master
```

The process will take a while as it copies files, grabs any required dependencies, compiles, and then deploys your application.

5. Once the process is complete, you can preview the existing application by executing:

```
heroku open
```

You can also simply open `https://{appname}.herokuapp.com` in a browser.

You now have a new Heroku application in the cloud. The first page should look like this:

Tell Me More...

Look carefully as the `git push` happens and you'll see some magic. Early on, Heroku detects that the push is a Spring MVC app, so it installs Maven, builds the app, and then gets it running for you, all with just a single command.

Step 3: Test the Application

This step shows you how to take your application for a quick test run to verify it's working.

1. In a browser tab or window, navigate to `https://{appname}.herokuapp.com`.
2. Click **Ajax @Controller Example**.
3. In another browser tab or window, open the Warehouse application on your Force.com instance.
4. Click **Invoice Statements** and then select an existing invoice statement or create a new one if necessary.
5. In the browser URL bar, select the invoice statement record ID, which is everything after `salesforce.com` in the URL. It should look something like `a01E0000000diKc`. Copy the ID to your clipboard.
6. Return to the browser window or tab showing your Heroku application.
7. Paste the invoice statement record ID into the field under **Id**.
8. Click **Create**. An order is created with the Invoice ID. Your page looks something like:

Summary

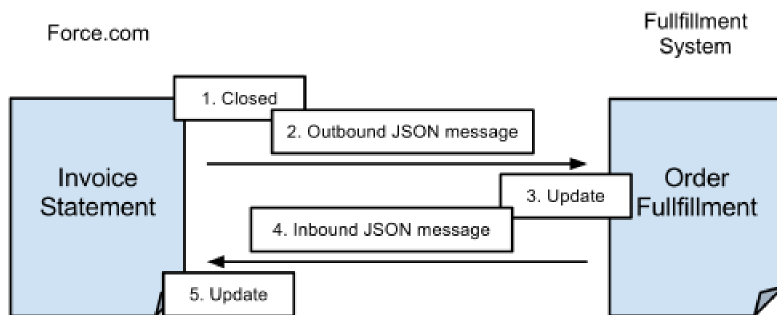
Heroku's polyglot Platform as a Service design lets you easily deploy your applications with industry-standard tools, such as git. Normally, teams would use local development environments, like Eclipse, and in fact Heroku has released an Eclipse plugin for seamless integration with Eclipse. You can also interact with Heroku on the command line and directly access logs and performance tools for your applications.

Tutorial #2: Connecting the Warehouse App with an External Service

Force.com offers several ways to integrate with external systems. For example, without writing any code, you can declare workflow rules that send outbound email alerts and make simple Web service calls. You can implement more complex scenarios programmatically with Apex code.

This tutorial teaches you how to create a Web service callout to integrate the Warehouse app with the fulfillment application you deployed in Tutorial 1. This fulfillment system, written in Java, is hosted on Heroku, but it could be any application with a Web service interface.

The following diagram illustrates the example scenario requirements: When an invoice statement's status changes to Closed in your Force.com system, the system should send a JSON-formatted message to the order fulfillment service running on Heroku, which in turn returns an order ID to the Force.com system that then records this in the Invoice.



Step 1: Create an External ID Field on Invoice Statement

To start, create a custom field in the Invoice Statement custom object that can store the order ID returned by the Java app running on Heroku. The field is an index into an external system, so it makes sense to make it an External ID.

1. Log in to your Salesforce organization.
2. Go to the Invoice Statement custom object by clicking ***Your Name*** > **Setup** > **Create** > **Objects** > **Invoice Statement**.
3. Scroll down to `Custom Fields & Relationships`, and click **New**.
4. Select the **Text** field type and click **Next**.
5. Enter `OrderId` as the field label, and enter `6` as the field length. Leave the field name as the default `OrderId`.
6. Click the **External ID** checkbox and click **Next**.
7. Click **Next** to accept the defaults, then click **Save**.

Step 2: Create a Remote Site Record

The Force.com platform implements very conservative security controls. By default, Force.com prohibits callouts to external sites. This step teaches you how to register the Heroku Java site in the Remote Site Settings page.

1. Click ***Your Name*** > **Setup** > **Security Controls** > **Remote Site Settings**.
2. Click **New Remote Site** and fill in the site settings.
3. In the Remote Site Name field, enter `FulfillmentWebService` (no spaces).
4. In the Remote Site URL field, enter (exactly) `https://`*`{appname}`*`.herokuapp.com`.
5. Leave all other values as they are and click **Save**.

Now any Apex code in your app can call the fulfillment Web service.

Tell Me More...

Just for fun, you can skip Step 2 and create and test the callout in Step 3 and Step 4 below to observe the error message that is generated when an app attempts to callout to a URL without permission. Don't forget to come back and add the remote site record, though!

Step 3: Create an Integration Apex Class

Now that your app can access an external URL, it's time to implement the callout. Apex triggers are not permitted to make synchronous Web service calls. This restriction ensures a long running Web service does not hold a lock on a record within your Force.com app.

The steps in this tutorial teach you how to build out the correct approach, which is to create an Apex class with an asynchronous method that uses the `@future` annotation, and then build a trigger to call the method as necessary. When the trigger calls the asynchronous method, Force.com queues the call, execution of the trigger completes, and Force.com releases any record locks. Eventually, when the asynchronous call reaches the top of the queue, Force.com executes the call and posts the invoice to the order fulfillment Web service running on Heroku.

You start by adding the code for the asynchronous method in a new Apex class.

1. Click ***Your Name*** > **Setup** > **Develop** > **Apex Classes**.
2. Click **New** and paste in the following code:

```
public class Integration {

    // The ExternalOrder class holds a string and integer
    // received from the external fulfillment system.

    public class ExternalOrder {
        public String id {get; set;}
        public Integer order_number {get; set;}
    }

    // The postOrder method integrates the local Force.com invoicing system
    // with a remote fulfillment system; specifically, by posting data about
    // closed orders to the remote system. Functionally, the method 1) prepares
    // JSON-formatted data to send to the remote service, 2) makes an HTTP call
    // to send the prepared data to the remote service, and then 3) processes
    // any JSON-formatted data returned by the remote service to update the
    // local Invoices with the corresponding external Ids in the remote system.

    @future (callout=true) // indicates that this is an asynchronous call
    public static void postOrder(List<Id> invoiceIds) {

        // 1) see above

        // Create a JSON generator object
        JSONGenerator gen = JSON.createGenerator(true);
```

```
        // open the JSON generator
        gen.writeStartArray();
        // interate through the list of invoices passed in to the call
        // writing each invoice Id to the array
        for (Id invoiceId : invoiceIds) {
            gen.writeStartObject();
            gen.writeStringField('id', invoiceId);
            gen.writeEndObject();
        }
        // close the JSON generator
        gen.writeEndArray();
        // create a string from the JSON generator
        String jsonOrders = gen.getAsString();
        // debugging call, which you can check in console logs
        System.debug('jsonOrders: ' + jsonOrders);

        // 2) see above

        // create an HTTPrequest object
        HttpRequest req = new HttpRequest();
        // set up the HTTP request with a method, endpoint, header, and body
        req.setMethod('POST');
        // DON'T FORGET TO UPDATE THE FOLLOWING LINE WITH YOUR appid
        req.setEndpoint('https://{appname}.herokuapp.com/order');
        req.setHeader('Content-Type', 'application/json');
        req.setBody(jsonOrders);
        // create a new HTTP object
        Http http = new Http();
        // create a new HTTP response for receiving the remote response
        // then use it to send the configured HTTPrequest
        HTTPResponse res = http.send(req);
        // debugging call, which you can check in console logs
        System.debug('Fulfillment service returned '+ res.getBody());

        // 3) see above

        // Examine the status code from the HTTPResponse
        // If status code != 200, write debugging information, done
        if (res.getStatusCode() != 200) {
            System.debug('Error from ' + req.getEndpoint() + ' : ' +
              res.getStatusCode() + ' ' + res.getStatus());
        }
        // If status code = 200, update each Invoice Statement
        // with the external ID returned by the fulfillment service.
        else {
            // Retrieve all of the Invoice Statement sObjects
            // originally passed into the method call to prep for update.
            List<Invoice_Statement__c> invoices =
              [SELECT Id FROM Invoice_Statement__c WHERE Id IN :invoiceIds];
            // Create a list of external orders by deserializing the
            // JSON data returned by the fulfillment service.
            List<ExternalOrder> orders =
              (List<ExternalOrder>)JSON.deserialize(res.getBody(),
                List<ExternalOrder>.class);
            // Create a map of Invoice Statement Ids from the retrieved
            // invoices list.
            Map<Id, Invoice_Statement__c> invoiceMap =
              new Map<Id, Invoice_Statement__c>(invoices);
            // Update the order numbers in the invoices
            for ( ExternalOrder order : orders ) {
              Invoice_Statement__c invoice = invoiceMap.get(order.id);
              invoice.OrderId__c = String.valueOf(order.order_number);
            }
            // Update all invoices in the database with a bulk update
            update invoices;
        }
```

```
    }
}
```

Don't forget to replace `{appname}` with your Heroku application name.

3. Click **Save**.

This code collects the necessary data for the remote service, makes the remote service HTTP call, and processes any data returned by the remove service to update local invoices with the corresponding external IDs. See the embedded comments in the code for more information.

Step 4: Test the `@future` Method

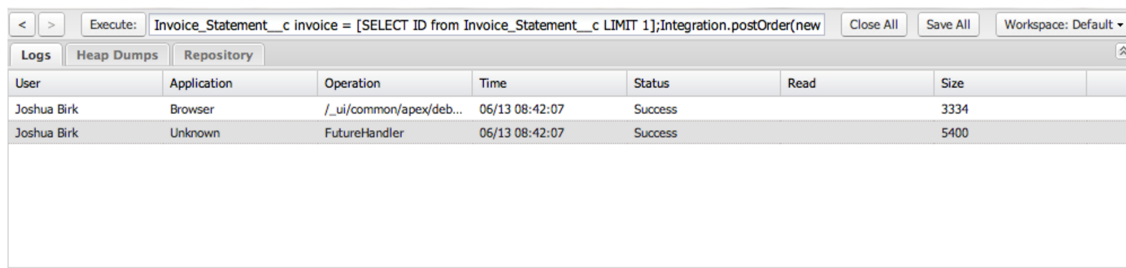
Before creating a trigger that calls a `@future` method, it's best practice to interactively test the method by itself and validate that the remote site settings are configured correctly. To test the method interactively, you can use the Developer Console.

1. Go to the Developer Console by clicking ***Your Name*** > **Developer Console**.
2. Click the box marked **Click here to enter Apex Code** and enter the following code.

```
// Get an Invoice_Statement__c for testing
Invoice_Statement__c invoice = [SELECT ID FROM Invoice_Statement__c LIMIT 1];
// Call the postOrder method to test the asynchronous call
Integration.postOrder(new List<Id>{invoice.id});
```

   This small snippet of Apex code retrieves the ID for a single invoice statement and calls your `@future` method using this ID.

3. Click the **Open Log** checkbox.
4. Click **Execute**. You should see two entries appear in the logs. Double click the second line — it should have `Future Handler` as its operation and a status of `Success`.



5. Click the **Filter** checkbox under the Execution Log and type `DEBUG` as the filter text. Scroll down and double click the last line of the execution log. You should see a popup with the response from the fulfillment Web service that looks something like:

```
08:08:42:962 USER_DEBUG [58]|DEBUG|Fulfillment service returned
[{"order_number":2,"id":"a01E0000009RpppIAC"}]
```

Now that you have a functional `@future` method that can call the fulfillment Web service, it's time to tie things together with a trigger.

Step 5: Create a Trigger to Call the `@future` Method

To create a trigger on the Invoice Statement object that calls the `Integration.postOrder` method, complete the following steps:

1. Go to the Invoice Statement custom object by clicking ***Your Name*** > **Setup** > **Create** > **Objects** > **Invoice Statement**.
2. Scroll down to `Triggers`, click **New** and paste the following code in place of the trigger skeleton:

```
trigger HandleOrderUpdate on Invoice_Statement__c (after update) {

    // Create a map of IDs to all of the *old* versions of sObjects
    // updated by the call that fires the trigger.
    Map<ID, Invoice_Statement__c> oldMap =    new Map<ID,
    Invoice_Statement__c>(Trigger.old);

    // Create an empty list of IDs
    List<Id> invoiceIds = new List<Id>();

    // Iterate through all of the *new* versions of Invoice_Statement__c
    // sObjects updated by the call that fires the trigger, adding
    // corresponding IDs to the invoiceIds list, but *only* when an
    // sObject's status changed from a non-"Closed" value to "Closed".
    for (Invoice_Statement__c invoice: Trigger.new) {
        if (invoice.status__c == 'Closed' &&  oldMap.get(invoice.Id).status__c !=
'Closed'){
            invoiceIds.add(invoice.Id);
        }
    }
    // If the list of IDs is not empty, call Integration.postOrder
    // supplying the list of IDs for fulfillment.
    if (invoiceIds.size() > 0) {
        Integration.postOrder(invoiceIds);
    }
 }
```

3. Click **Save**.

The comments in the code explain what is happening. In particular, understand that Force.com triggers must be able to handle both single row and bulk updates because of the varying types of calls that can fire them (single row or bulk update calls). The trigger creates a list of IDs of invoices that have been closed in this update, and then calls the `@future` method once, passing the list of IDs.

Step 6: Test the Complete Integration Path

With the trigger in place, test the integration by firing the trigger.

1. Select the Warehouse app.
2. Click the **Invoice Statements** tab.
3. Click into one of the recent invoices and notice that there is no OrderId for the invoice.
4. If the Status is already Closed, double-click the word **Closed**, change it to **Open** and click **Save**.
5. Double-click the **Status** value, change it to **Closed** and click **Save**. This triggers the asynchronous callout.
6. Wait a few seconds and refresh the page in the browser.
7. You should see an external order ID appear in the `OrderId` field.

The following screen shows the Invoice Statements tab before any changes have been made:

The following screen shows the Invoice Statements tab after the asynchronous call has returned the new order ID:

Summary

Congratulations! Your app is sending invoices for fulfillment. You have successfully created an asynchronous Apex class that posted invoice details to your fulfillment app hosted on Heroku. Of course, your external application could reside anywhere as long as you have access via Web services. Your class uses open standards including JSON and REST to transmit data, and a trigger on Invoice Statements to execute the process.

Tutorial #3: Update the Heroku App

You now have two sides of an integration in place: one running a Java endpoint on Heroku, and another in Force.com which communicates with the endpoint when the appropriate changes take place. Now that you've got the connection in place, update the Heroku application to retrieve the pertinent information and display it to the user.

Step 1: Configure Remote Access

External applications must authenticate remotely before they may access data. Force.com supports OAuth 2.0 (hereafter referred to as OAuth) as an authentication mechanism.

With OAuth, a client application delegates the authentication to a provider, in this case Force.com, which in turn issues an access token if the user successfully authenticates. As a result, your application doesn't need to handle authentication explicitly; it simply needs to ensure that a valid access token accompanies all interactions with the API. You can typically download an OAuth library to do the heavy lifting.

Before an application can use OAuth, you have to configure your environment. Log in to your Force.com organization as an administrator and configure a remote access application.

1. Navigate to ***Your Name*** > **Setup** > **Develop** > **Remote Access**.
2. Click **New**.
3. For Application, enter `Buyer Client`.
4. For Callback URL, enter `https://{appname}.herokuapp.com/_auth`.

Note: Be sure to replace *{appname}* with your actual Heroku app name.

5. For Email, enter your email address.
6. Click **Save**.

The resulting page should look similar to this:

The detail page for your remote access configuration will display a consumer key as well as a consumer secret (which you have to click to reveal). You'll need these in later steps.

Step 2: Update your Application with a New Branch

While you were creating a new Apex trigger on your Force.com instance, other developers have added new functionality to the original project and placed it into a specific branch on Github. Git allows for easy branching and merging of code for your projects. Using this branch, you can test out the new features, specifically the ability for your Heroku application to access your Force.com records directly. It's easy to add this branch, called "full", to your codebase:

1. Return to the command line, and make sure your in the `spring-mvc-fulfillment-base` folder.
2. Enter the following command to fetch the "full" branch and merge it with your master branch, all in one step:

```
git pull origin full
```

3. You need to set your Remote Access keys to your Heroku application. Enter:

```
heroku config:add OAUTH_CLIENT_KEY=PUBLICKEY OAUTH_CLIENT_SECRET=PRIVATEKEY
```

   Replace `PUBLICKEY` with the `Consumer Key` from your remote access application you defined in Step 1. Similarly, replace `PRIVATEKEY` with the `Consumer Secret`. It may be helpful to do this in a text editor before putting it on the command line.

4. Execute the following command to push the local changes to Heroku:

```
git push heroku master
```

5. In a browser tab or window, navigate to `https://{appname}.herokuapp.com` to see the changes (refresh your browser if needed).

By adding an OAuth flow to your app, your app can get a user's permission to have session information without requiring the third-party server to handle the user's credentials. With this added to the project, the fulfillment application can use the Force.com REST API to access information directly from the user's instance.

Tell Me More...

You can review all of the changes brought in by this branch on github at: https://github.com/developerforce/spring-mvc-fulfillment-base/compare/master...full. Notice that the changes use the Force.com REST API to manipulate Invoice Statement records. Look at `InvoiceServiceImpl.java` in particular to see how it creates, queries, retrieves and deletes invoices. This tutorial only uses the `findOrder()` method. The others are included for your reference.

Step 3: Update the User Interface with Invoice Information

In the previous steps you added brand new functionality by merging a branch into your local code, but there's a feature still missing. The application now understands how to use OAuth and how access data from the Force.com platform, but the Web page hasn't been updated to display the Invoice Statement fields.

1. In your local `spring-mvc-fulfillment-base` folder, open `src/main/webapp/WEB-INF/views/order.jsp` in the text or code editor of your choice.
2. Just before the following line:

```
<a href="javascript:history.back()">Back</a> <input id="delete" type="submit"
value="Delete" />
```

add:

```
<h3>Invoice</h3>
<p>Number: <c:out value="${invoice.number}"/></p>
<p>Description: <c:out value="${invoice.description}"/></p>
<p>Status: <c:out value="${invoice.status}"/></p>
```

3. Save the file.
4. Return to the command line and make sure you're in the `spring-mvc-fulfillment-base` folder.
5. Execute the following:

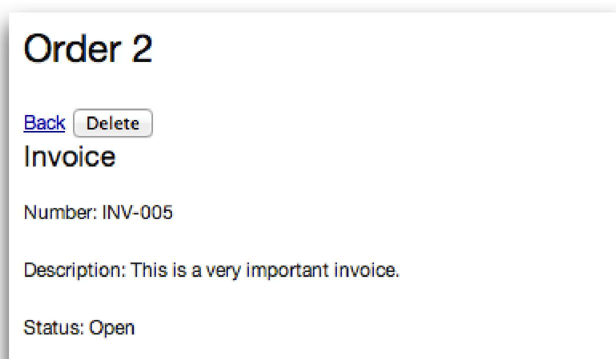
```
git commit -am "Updating UI"
```

6. After the commit process finishes, execute the following:

```
git push heroku master
```

7. Wait for the push to finish, and navigate to your fulfillment app in the browser (refreshing if necessary) to see the changes.

Notice that, given an ID, this code retrieves the corresponding invoice record. Because there might be mock ID's in the database that are not in Force.com, the app handles the corresponding exception by showing default data. Adding the invoice to the model makes it available to the view. Now when you test the fulfillment application, it will show the invoice information currently in your Force.com instance by grabbing the information via the REST API using the record ID. Your order detail page might look something like:



Tell Me More...

To really see the application in motion, you can edit information on Force.com (like the `Description` of the Invoice Statement) and see it represented the next time you load that order in your Heroku app.

Notice that the Web service call from Force.com to create orders is not secured. Securing this call goes beyond the scope of this workbook, but a simple solution would be to set up a shared secret between the Force.com app and the fulfillment app. The Force.com app would create an HMAC of the parameters in the request, using the secret, and the fulfillment app would verify the HMAC.

Summary

Congratulations! With a combination of OAuth authentication, the REST API, Apex triggers, `@future` callouts and the polyglot framework of the Heroku platform you have created and deployed a bi-directional integration between two clouds.

This workbook touches upon just one example of the many ways to integrate your applications with Salesforce. One integration technology not touched upon in this workbook is the Streaming API that lets your application receive notifications from Force.com whenever a user changes Salesforce data. You could use this in the fulfillment application to monitor when the user updates invoice statements and automatically update the application pages accordingly. Make sure to visit `http://developer.force.com` to learn more about all the different ways you can integrate your application with Force.com.