



Integration Workbook: Summer '14

# Integration Workbook



Last updated: June 14, 2014



# Table of Contents

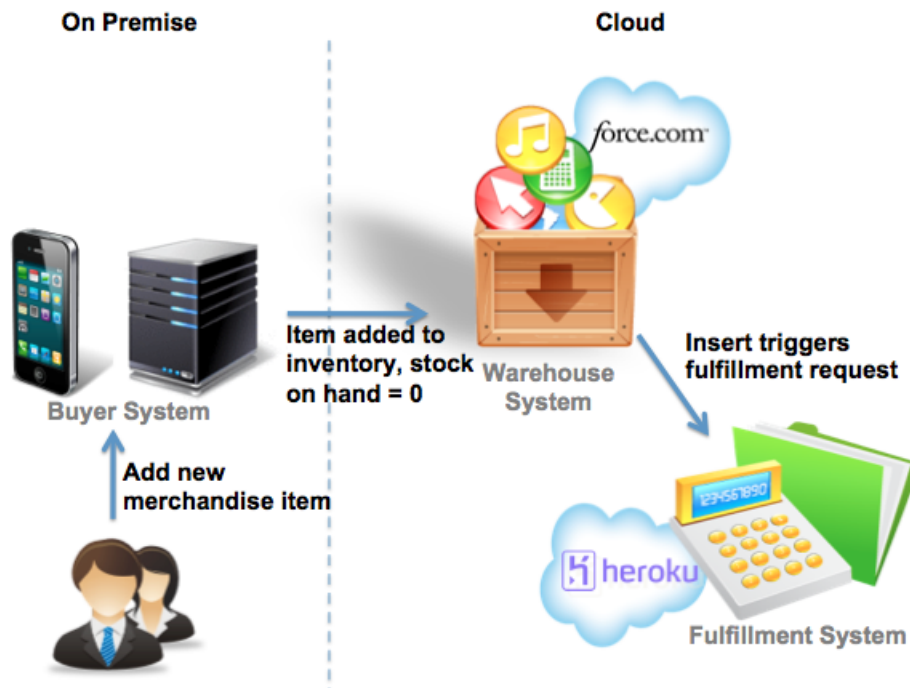
<b>Force.com Integration Workbook.....</b>	<b>1</b>
<b>Before You Begin.....</b>	<b>2</b>
<b>Tutorial #1: Create a New Heroku Application.....</b>	<b>3</b>
Step 1: Clone the Github Project.....	3
Step 2: Create a Heroku Project.....	3
Step 3: Test the Application.....	4
Summary.....	5
<b>Tutorial #2: Connect the Warehouse App with an External Service .....</b>	<b>6</b>
Step 1: Create an External ID Field on Invoice.....	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.....	9
Step 6: Test the Complete Integration Path.....	10
Summary.....	11
<b>Tutorial #3: Update the Heroku App.....</b>	<b>13</b>
Step 1: Configure Your Connected App.....	13
Step 2: Update your Application with a New Branch.....	13
Step 3: Update the User Interface with Invoice Information.....	14
<b>Tutorial #4: Add Your App to Salesforce Using Force.com Canvas.....</b>	<b>16</b>
Step 1: Update your Canvas App with a New Branch.....	16
Step 2: Edit The Connected App Details and Enable it for Force.com Canvas.....	16
Step 3: Configure Access to Your Force.com Canvas App.....	20
Step 4: Make Your Force.com Canvas App Available from the Chatter Tab.....	20
Step 5: Use Visualforce to Display the Canvas App on a Record.....	21
Summary.....	22



# 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](http://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'll need to install the Warehouse data model in your organization, create a Heroku developer account, and install the Heroku Toolbelt software on your local workstation.

## Step 1: Install the Warehouse Data Model

This workbook uses a set of objects that represent a simple warehouse management system. To install these objects into your developer organization:

1. Navigate to <https://login.salesforce.com/packaging/installPackage.apexp?p0=04ti0000000Pi7P> in your browser.
2. Log in using your developer edition organization username and password.
3. On the Package Installation Details page, click **Continue**.
4. Click **Next**. On the Security Level page click **Next**. On the following page, click **Install**.
5. You'll also want to create some sample records. Once you've deployed the Warehouse data model, select the Warehouse app from the drop-down app menu in the upper-right corner of your current Salesforce page.
6. Click the Data tab, and then click **Create Data** to create some sample records.



**Note:** After you've gone through this workbook, you can un-install the Warehouse data model and sample data from your organization by navigating to **Installed Packages** under Setup and deleting the Warehouse package.

## Step 2: 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 3: 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 Heroku repository. 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/sbob-sfdc/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 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 might 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`, use:  
`https://quiet-planet-3215.herokuapp.com/_auth`.

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

```
git push heroku master
```

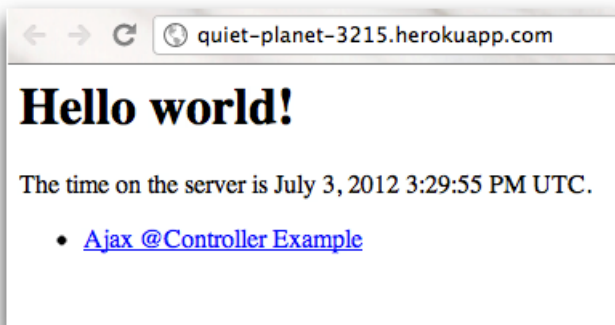
The process will take a while as it copies files, grabs any required dependencies, compiles, and then deploys your application.

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 **Invoices** and then select an existing invoice or create a new one if necessary.
5. In the browser URL bar, select the invoice 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 record ID into the field under **Id**.
8. Click **Create**. An order is created with the Invoice ID. Your page looks something like:

**Create Order**

Enter a 15 or 18 character Invoice Statement ID to create an order.

**Order Fields**

**Id**

**Create**

**Existing Orders**

Invoice ID	Order Number
<a href="#">a01E0000009Rppk</a>	<a href="#">1</a>

## Summary

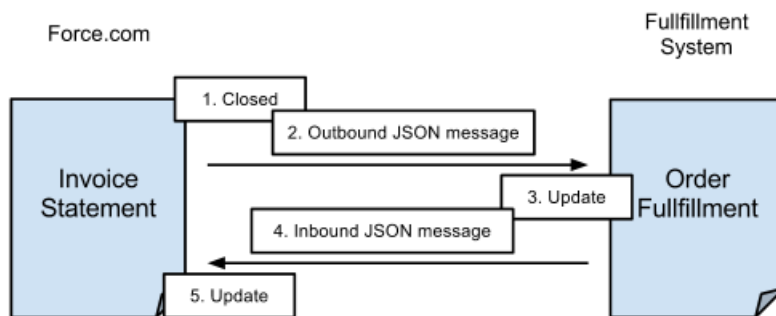
Heroku's polyglot design lets you easily deploy your applications with industry-standard tools, such as git. Typically, teams use local development environments, like Eclipse, and in fact Heroku has released an Eclipse plugin for seamless integration with Eclipse. You can also interact with Heroku on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

## Tutorial #2: Connect 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'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

To start, create a custom field in the Invoice 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 from Setup by clicking **Create > Objects > Invoice**.
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. From Setup, click **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.

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

1. From Setup, click **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();
        // iterate 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
    // with the external ID returned by the fulfillment service.
    else {
        // Retrieve all of the Invoice sObjects
        // originally passed into the method call to prep for update.
        List<Invoice__c> invoices =
            [SELECT Id FROM Invoice__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 Ids from the retrieved
        // invoices list.
        Map<Id, Invoice__c> invoiceMap =
            new Map<Id, Invoice__c>(invoices);
        // Update the order numbers in the invoices
        for ( ExternalOrder order : orders ) {
            Invoice__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 remote 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 correctly configured. 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__c for testing
Invoice__c invoice = [SELECT ID FROM Invoice__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 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.

User	Application	Operation	Time	Status	Read	Size
Joshua Birk	Browser	/_ui/common/apex/deb...	06/13 08:42:07	Success		3334
Joshua Birk	Unknown	FutureHandler	06/13 08:42:07	Success		5400

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 object that calls the `Integration.postOrder` method, complete the following steps:

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

```
trigger HandleOrderUpdate on Invoice__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__c> oldMap = new Map<ID,
```

```
Invoice__c>(Trigger.old);

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

// Iterate through all of the *new* versions of Invoice__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__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 invoice IDs 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 **Invoices** 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 Invoices tab before any changes have been made:

The screenshot shows the 'Invoice Statements' page for 'INV-0001'. The 'Invoice Statement Detail' section shows the following information:

Invoice Number	INV-0001	Owner	Steve Bobrowski [Change]
Description	First invoice		
Status	Open		
Invoice Value	\$3,500.00		
Orderid	1		
Created By	Steve Bobrowski, 6/23/2012 11:12 PM	Last Modified By	Steve Bobrowski, 6/23/2012 11:47 PM

The 'Line Items' table shows two items:

Action	Line Item: Line Item Number	Merchandise Name	Units Sold	Value
Edit   Del	1	Laptop	1	\$500.00
Edit   Del	2	Desktop	3	\$3,000.00

Red arrows point to the 'Status' field (Open) and the 'Orderid' field (1).

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

The screenshot shows the 'Invoice Statements' page for 'INV-0001'. The 'Invoice Statement Detail' section shows the following information:

Invoice Number	INV-0001	Owner	Steve Bobrowski [Change]
Description	First invoice		
Status	Closed		
Invoice Value	\$3,500.00		
Orderid	3		
Created By	Steve Bobrowski, 6/23/2012 11:12 PM	Last Modified By	Steve Bobrowski, 6/24/2012 11:21 AM

The 'Line Items' table shows two items:

Action	Line Item: Line Item Number	Merchandise Name	Units Sold	Value
Edit   Del	1	Laptop	1	\$500.00
Edit   Del	2	Desktop	3	\$3,000.00

Red arrows point to the 'Status' field (Closed) and the 'Orderid' field (3).

## 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 Invoices 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 Your Connected App

Before moving on, let's go back to your Salesforce organization so that we can configure your connected app. At a high-level, the configuration steps include:

- Adding your app to the available connected apps in your organization.
- Enabling OAuth. External applications must authenticate remotely before they can access data. Force.com supports OAuth 2.0 (hereafter referred to as OAuth) as an authentication mechanism.

Let's go ahead and begin.

1. From Setup, click **Create > Apps**.
2. In the Connected Apps section, click **New**.
3. For Connected App Name, enter your app name.
4. Enter the API Name, used when referring to your app from a program. It defaults to a version of the name without spaces.
5. Provide your Contact Email.
6. Under API (Enable OAuth Settings) select **Enable OAuth Settings**.
7. For Callback URL, enter `https://{appname}.herokuapp.com/_auth`.



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

8. In the Selected OAuth Scopes field, select **Full access (full)**.
9. Click **Save**.

### Step 2: Update your Application with a New Branch

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

1. Return to the command line, and make sure you're 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
```

- a. Before continuing, go back to your org.
- b. From Setup, click **Create > Apps**.
- c. In the Connected App Settings section, select your canvas app.
- d. Next to Consumer Secret, click **Click to reveal**.
- e. Use your keyboard controls to copy the number that appears.

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

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

Replace **PUBLICKEY** with the Consumer Key. Similarly, replace **PRIVATEKEY** with the Consumer Key. 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/sbob-sfdc/spring-mvc-fulfillment-base/compare/master...full>. Notice that the changes use the Force.com REST API to manipulate Invoice 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 to access data from the Force.com platform, but the Web page hasn't been updated to display the Invoice 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="${invoice.number}"/></p>
<p>Status: <c:out value="${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:

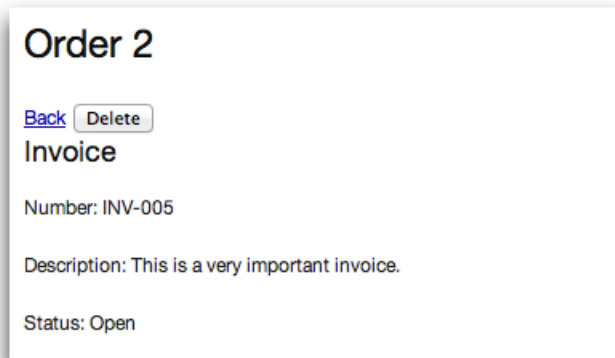
```
git commit -am "Updating UI"
```

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`) 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.

## Tutorial #4: Add Your App to Salesforce Using Force.com Canvas

You've done a lot already. Let's go one step further and make your app accessible for your users right from within Salesforce. Force.com Canvas enables you to easily integrate a third-party application in Salesforce. Force.com Canvas is a set of tools and JavaScript APIs that you can use to expose an application as a canvas app. This means you can take your new or existing applications and make them available to your users as part of their Salesforce experience.

### Step 1: Update your Canvas App with a New Branch

Earlier, we added a branch to the codebase named "full." Now we'll add one named "canvas."

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

```
git pull origin canvas
```

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

```
git push heroku master
```

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

#### Tell Me More...

You can review all of the changes brought in by this branch on GitHub at <https://github.com/sbob-sfdc/spring-mvc-fulfillment-base/compare/full...canvas>.

Notice that the changes use the signed request from the Force.com Canvas API and not the Heroku-initiated OAuth from [Tutorial 3, Step 2](#) and also use the [Force.com](#) REST API to manipulate invoice records. Look at `CanvasUiController.java` in particular to see how it retrieves, parses, and sets the signed request for use by the app. Also, `order.jsp` has changed to present an easier-to-use screen on the invoice page layout. This tutorial has only set the signed request for use on the `canvasui` page and the orders page in the app.

### Step 2: Edit The Connected App Details and Enable it for Force.com Canvas

We've already configured your connected app. Now we need to enable and configure it for Force.com Canvas.

1. From Setup, click **Create > Apps**.
2. In the Connected App Settings section, select your application and click **Edit**.
3. In the Canvas App URL field, enter `https://{appname}.herokuapp.com/canvasui`.
4. In the Access Method field, select Signed Request (Post).
5. In the Locations field, select Chatter Tab and Visualforce Page.
6. Click **Save**.

If you look at `CanvasUiController.java`, you'll see something like the following, which shows Heroku obtaining a signed request and validating it. We're leveraging the `OAUTH_CLIENT_SECRET` Heroku key set in [Tutorial 3, Step 2](#) to validate the signed request. After the validation, the signed request is passed to `order.jsp` where the browser can access it.

```
@Controller
@RequestMapping(value="/canvasui")
public class CanvasUiController {

    private static final String SIGNED_REQUEST = "signedRequestJson";
    private CanvasContext cc = new CanvasContext();

    @Autowired
    private OrderService orderService;

    @Autowired
    private InvoiceService invoiceService;

    private Validator validator;

    @Autowired
    public CanvasUiController(Validator validator) {
        this.validator = validator;
    }

    @RequestMapping(method= RequestMethod.POST)
    public String postSignedRequest(Model model,
    @RequestParam(value="signed_request")String signedRequest, HttpServletRequest request){

        String srJson = SignedRequest.verifyAndDecodeAsJson
            (signedRequest, getConsumerSecret());
        CanvasRequest cr = SignedRequest.verifyAndDecode(signedRequest, getConsumerSecret());

        HttpSession session = request.getSession(true);
        model.addAttribute(SIGNED_REQUEST, srJson);
        cc = cr.getContext();
        CanvasEnvironmentContext ce = cc.getEnvironmentContext();
        Map<String, Object> params = ce.getParameters();
        if (params.containsKey("orderId")) {
            invoiceService.setSignedRequest(cr);
            Integer orderId = Integer.parseInt(params.get("orderId").toString());
            if(orderId != null) {
                Order order = orderService.findOrder(orderId);
                if (order == null) {
                    throw new ResourceNotFoundException(orderId);
                }
                model.addAttribute("order", order);

                Invoice invoice;
                try {
                    invoice = invoiceService.findInvoice(order.getId());
                } catch (ApiException ae) {
                    // No match
                    invoice = new Invoice();
                }
                model.addAttribute("invoice", invoice);

                return "order";
            }
        }
        return getOrdersPage(model);
    }

    @RequestMapping(method=RequestMethod.GET)
    public String getOrdersPage(Model model) {
        model.addAttribute("order", new Order());
        model.addAttribute("orders", orderService.listOrders());

        return "orders";
    }
}
```

```
    }

    private static final String getConsumerSecret(){
        String secret = System.getenv("OAUTH_CLIENT_SECRET");
        if (null == secret){
            throw new IllegalStateException("Client secret not found in environment.
            You must define the OAUTH_CLIENT_SECRET environment variable.");
        }
        return secret;
    }
}
```

```
<%@ page session="false" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

<html>
<head>
    <title>Order</title>
    <link rel="stylesheet" href="<c:url value="/resources/blueprint/screen.css" />
        " type="text/css" media="screen, projection">
    <link rel="stylesheet" href="<c:url value="/resources/blueprint/print.css" />
type="text/css" media="print">
    <!--[if lt IE 8]>
        <link rel="stylesheet" href="<c:url value="/resources/blueprint/ie.css" />
        " type="text/css" media="screen, projection">
    <![endif]-->
    <script type="text/javascript" src="<c:url value="/resources/jquery-1.4.min.js" /> ">
    </script>
    <script type="text/javascript" src="<c:url value="/resources/json.min.js" /> ">
    </script>
    <script type="text/javascript" src="<c:url value="/resources/canvas-all.js" /> ">
    </script>
    <script>
        // Get the Signed Request from the CanvasUIController
        var sr = JSON.parse('${not empty signedRequestJson?signedRequestJson: "{}"}');

        // Set handlers for the various buttons on the page
        Sfdc.canvas(function() {
            $('#finalizeButton').click(finalizeHandler);
            $('#deleteButton').click(deleteHandler);
        });

        // This function will be called when the "Finalize" button is clicked.
        // This shows using the Canvas Cross Domain API to hit the REST API
        // for the invoice that the user is viewing. The call updates the
        // Status__c field to "Shipped". If successful, the page is refreshed,
        // and if there is an error it will alert the user.
        function finalizeHandler(){
            var invoiceUri=sr.context.links.sobjectUrl + "Invoice__c/${order.id}";
            var body = {"Status__c":"Shipped"};
            Sfdc.canvas.client.ajax(invoiceUri,{
                client : sr.client,
                method: 'PATCH',
                contentType: "application/json",
                data: JSON.stringify(body),
                success : function() {
                    window.top.location.href = getRoot() + "/" + "${order.id}";
                },
                error: function(){
                    alert("Error occurred updating local status.");
                }
            });
        }

        // This function will be called when the "Delete Order" button is clicked.
```

```

        // It will delete the record from the Heroku database.
        function deleteHandler(){
            $.deleteJSON("/order/${order.orderId}", function(data) {
                alert("Deleted order ${order.orderId}");
                location.href = "/orderui";
            }, function(data) {
                alert("Error deleting order ${order.orderId}");
            });
            return false;
        }

        // This function gets the instance the user is on for a page referesh
        function getRoot() {
            return sr.client.instanceUrl;
        }
    </script>

</head>
<body>
    <div id="bodyDiv" style="width:inherit;">
        <div id="myPageBlockTable">
            <h2 id="OrderTitle">
                Order Number: <c:out value="${order.orderId}"/>
            </h2>
            <table id="myTable" width="100%">
                <col width="20%">
                    <tr><td class="myCol">Invoice Id:</td><td class="valueCol">
                        <c:out value="${invoice.id}"/></td></tr>
                    <tr><td class="myCol">Invoice Number:</td><td class="valueCol">
                        <c:out value="${invoice.number}"/></td></tr>
                    <tr><td class="myCol">Status:</td><td class="valueCol" valign="center">

                        <c:out value="${invoice.status}"/>
                        <!-- Display a green check if the order is Shipped, or a red x if
not shipped -->

                        <c:choose>
                            <c:when test="${invoice.status == 'Shipped'}">
                                
                            </c:when>
                            <c:otherwise>
                                
                            </c:otherwise>
                        </c:choose>
                    </td></tr>
                </table>
                <!-- Display the Back and Delete Order Button if viewed outside
of salesforce (no signed request). -->
                <!-- Display the Finalize Button if viewed inside of salesforce and
the Status is not Shipped. -->
                <c:choose>
                    <c:when test="${empty signedRequestJson}">
                        <button onclick="location.href='/orderui'">Back</button>
                        <button id="deleteButton">Delete Order</button>
                    </c:when>
                    <c:otherwise>
                        <c:if test="${invoice.status ne 'Shipped'}">
                            <button id="finalizeButton">Finalize</button>
                        </c:if>
                    </c:otherwise>
                </c:choose>
            </div>
        </div>
    </body>
</html>

```

## Step 3: Configure Access to Your Force.com Canvas App

Since this app is designed for use by a specific audience, let's go ahead and give access only to the users who need it.

1. In Salesforce, from Setup, click **Manage Apps > Connected Apps**.
2. Click on your app, and then click **Edit**.
3. In the `Permitted Users` field, select **Admin approved users are pre-authorized**. Click **OK** on the pop-up message that appears.
4. Click **Save**.

Now you'll use profiles and permission sets to define who can see your canvas app. In this example, we'll allow anyone with the System Administrator profile to access the app.

5. In the Connected App Detail page's Profiles related list, click **Manage Profiles**.
6. Select the `System Administrator` profile and click **Save**.

Your app is now available to anyone with the `System Administrator` profile.

## Step 4: Make Your Force.com Canvas App Available from the Chatter Tab

The values you selected in the `Locations` field when creating the connected app in [Step 2: Edit The Connected App Details and Enable it for Force.com Canvas](#) on page 16 determine where an installed canvas app appears. When an app is made available to the Chatter tab, there's nothing we need to do for this step. If you log into your Salesforce org and select the Chatter tab, you'll see that your canvas app appears in the app navigation list.



**Note:** When displaying the list of orders on the Chatter Tab, remember that `orders.jsp` has been set up to handle the signed request POST. However, if you click into a record from this page, you are redirected to `orderui`, which uses the OAuth. If the Heroku OAuth flow is inactive, you may receive an error when viewing the individual order.

Click on it and yours should look something like this:

Invoice ID	Order Number
<a href="#">a00U00000006CXrlIAW</a>	<a href="#">2</a>



## Step 5: Use Visualforce to Display the Canvas App on a Record

While you can certainly use the app based on the work completed so far, let's take one more step and use Visualforce to display invoice information on the invoice record.

1. From Setup, click **Develop** > **Pages**.
2. Click **New**.
3. In **Label**, enter the text that should be used to identify the page in Setup tools, such as when defining custom tabs or overriding standard buttons.
4. In **Name**, enter the URL's page name.
5. Add the following markup to the Visualforce Markup box and then click **Save**.

```
<apex:page standardController="Invoice__c">
  <apex:canvasApp developerName="CanvasFulfillment"
    parameters="{ 'orderId': '{!Invoice__c.OrderId__c}' }" width="100%" />
</apex:page>
```

Notice how the parameters tag in the apex:canvasApp component is set to parameters="{ 'orderId': '{!Invoice\_\_c.OrderId\_\_c}' }". This code sends a JSON object as part of the signed request to the Heroku app when the page is loaded. In the signed request, the parameters object will look something like parameters : { 'orderId': '5' }, where '5' is the OrderId from the Invoice record. Remember that this value is an external Id field which connects the record in the Heroku database to the Salesforce invoice record. By delivering the OrderId to the Heroku app with the signed request, the Heroku app can display the correct record on the invoice page layout.

Your page should look something like this:

The screenshot shows the Salesforce Visualforce Page editor interface. At the top, it says 'Visualforce Page' and 'FulfillmentCanvas'. There are buttons for 'Edit', 'Delete', 'Clone', 'Where is this used?', and 'Show Dependencies'. Below this is a 'Page Detail' section with a table showing the page's metadata. The table has columns for 'Label', 'Name', 'Namespace Prefix', 'Available In Touch', 'Require CSRF protection on GET requests', 'Description', 'Last Modified By', and 'Created By'. The 'Label' and 'Name' are both 'FulfillmentCanvas'. The 'Namespace Prefix' is empty. 'Available In Touch' is a checkbox that is currently unchecked. 'Require CSRF protection on GET requests' is a checkbox that is currently checked. The 'Description' is empty. 'Last Modified By' is 'Admin User' and 'Created By' is 'Admin User' with a timestamp of '10/9/2013 5:58 PM'. Below the table is a 'Visualforce Markup' section with a 'Version Settings' tab. The markup box contains the following code: 

```
<apex:page standardController="Invoice__c">
  <apex:canvasApp developerName="TestFulFillment" parameters="{ 'orderId': '{!Invoice__c.OrderId__c}' }" width="100%" />
</apex:page>
```

 At the bottom of the markup box are buttons for 'Edit', 'Delete', 'Clone', 'Where is this used?', and 'Show Dependencies'.

Label	Name
FulfillmentCanvas	FulfillmentCanvas

Namespace Prefix: Available In Touch: ☐

Require CSRF protection on GET requests: ☒ Description:

Last Modified By: Admin User, 10/10/2013 3:22 PM Created By: Admin User, 10/9/2013 5:58 PM

Visualforce Markup Version Settings

```
<apex:page standardController="Invoice__c">
  <apex:canvasApp developerName="TestFulFillment" parameters="{ 'orderId': '{!Invoice__c.OrderId__c}' }" width="100%" />
</apex:page>
```

Now let's add your Visualforce page to the page layout.

6. From the Invoice tab, select a record.
7. Click **Edit Layout** and then **Visualforce Pages**.
8. Drag a section down to your page and name it Canvas Fulfillment.
  - a. Make sure to deselect **Edit Page**.

- b. Select 1-Column for the layout.
  - c. Click the wrench to update the section properties.
9. Drag your Visualforce onto your new section.
10. Click the wrench to update your page properties. The width should set to 100% and height set to 165 pixels.
11. Ensure that both `Show scrollbars` and `Show label` are deselected and click **Save**.
12. From Setup, click **Customize > Invoice > Fields** and click **Edit** next to the Status field.
13. Add another picklist item named Shipped.

Now when users go to an invoice record, they'll see the canvas app right on the record detail page:

The screenshot shows a Salesforce record for an Invoice (INV-0006). The record details include:

- Invoice Number: INV-0006
- Owner: Admin User (Change)
- Status: Closed
- Created By: Admin User, 10/9/2013 3:02 PM
- Last Modified By: Admin User, 10/10/2013 3:23 PM
- Invoice Total: \$0.00
- OrderId: 7

Below the record details, the Canvas Fulfillment App is embedded. The app displays:

- Order Number: 7
- Invoice Id: a00U00000006CXrlIAW
- Invoice Number: INV-0006
- Status: Closed (with a red 'X' icon)
- A **Finalize** button

The app also has Edit, Delete, and Clone buttons at the bottom.

Notice the **Finalize** button in the canvas app. If the invoice is not in a shipped status, the red "X" and **Finalize** will show in the app. If you click **Finalize**, Heroku uses the Force.com Canvas API to call the REST API and update the invoice status field. Once the status is set to 'Shipped', the red "X" is replaced and **Finalize** is hidden.

## Summary

Congratulations! With a combination of OAuth authentication, the REST API, Apex triggers, `@future` callouts, the polyglot framework of the Heroku platform, Force.com Canvas, and Visualforce, you 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 on is the Streaming API that lets your application receive notifications from Force.com whenever a user changes Salesforce data. You can use this in the fulfillment application to monitor when a user updates invoices and to automatically update the application pages accordingly. Visit <http://developer.force.com> to learn more about all the ways you can integrate your application with Force.com.