Salesforce Platform Workshop: WINTER '13

Salesforce Platform Advanced Workshop

Last updated: March 12, 2014

Table of Contents

About the Workbook

This workbook shows you how to create a cloud app in a series of tutorials. While you can use the Salesforce platform to build virtually any kind of app, most apps share certain characteristics, such as:

- A database to model the information in the app
- A user interface to expose data and functionality to those logged into your app
- Business logic and workflow to carry out particular tasks under certain conditions

In addition, apps developed on the Salesforce Platform automatically support:

- A public website and mobile apps to allow access to data and functionality
- A native social environment that allows you to interact with people or data
- Built-in security for protecting data and defining access across your organization
- Multiple APIs to integrate with external systems
- The ability to install or create packaged apps

The workbook tutorials are centered around building a very simple warehouse management system. Your warehouse contains computer hardware and peripherals: laptops, desktops, tablets, monitors, that kind of thing. To keep track of how merchandise moves out of the warehouse, you use an invoice. An invoice is a list of line items. Each line item has a particular piece of merchandise, and the number of items ordered. The invoice rolls up all the prices and quantities for an invoice total. It's a very simple data model, but just enough to illustrate the basic concepts.

Development proceeds from the bottom up; that is, you first build an app and database model for keeping track of merchandise. You continue by modifying the user interface, adding business logic, etc. Each of the tutorials builds on the previous tutorial to advance the app's development and simultaneously help you learn about the platform.

Tutorial 1: Set Up Your Development Environment

The first thing you need to do is set up your development and testing environment. You'll install a package of components, so it's a good idea to get a new DE org for this workshop. Then you'll set up your browser and mobile device for testing.

Step 1: Get a New DE Org

If you already have a DE org you might not need a new one. However, if you've previously done any tutorials with the Warehouse app, the following package installation step will fail. While you could delete the Warehouse app and all of its components, this is a timely process, especially if you created related fields and business logic, a global action, or a page layout. It's far easier and faster to simply create a new org.

1. In your browser, go to `http://developer.force.com/join`.
2. Fill in the fields about you and your company.
3. In the `Email Address` field, make sure to use a public address you can easily check from a Web browser right now.
4. Type a unique `Username` like `firstname.lastname@s1workshop.com`.
5. Read and then select the checkbox for the `Master Subscription Agreement`. and then click **Submit Registration**.
6. In a moment you'll receive an email with a login link. Click the link and change your password.

Step 2: Install the Enhanced Warehouse Data Model

To prepare your developer organization for the exercises in this book, you need to import the Warehouse data model. You might be familiar with the Warehouse app if you've gone through the previous Hands-on Workshop, or from the tutorials in the *Force.com Workbook*. The Warehouse app used here is an enhanced version that will help demonstrate what Salesforce1 mobile app can do.

1. In your browser go to `http://bit.ly/1iSt8Qq`
2. If you were already logged in, you will be redirected to the Package Installation Details page. Otherwise, log in with your Developer Edition credentials.
3. Click **Continue**, **Next**, **Next**, and **Install**.
4. After the installation finishes, click the Force.com app menu and select **Warehouse**.
5. Click the **Data** tab and then click the **Create Data** button.

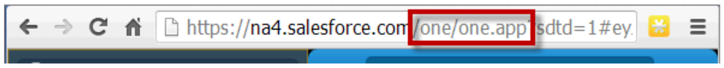
The package contains a pre-built Visualforce page, as well as some supporting resources. You'll learn about those right after your development and testing environments are set up.

Step 3: Access the Mobile Browser App

When developing Visualforce pages for the Salesforce1 mobile app, you can't do the familiar `https://<instance>/apex/<page>` hack on the URL to view the page: you must view the pages in the mobile app. The best way to test your pages is with the installed app, because it provides the most realistic experience. However, since it's a pain to grab your phone every time you want to see a change, you can open a new browser tab and use the `one.app` mobile browser version.

1. In your browser, open a new tab.

2. Copy and paste your Salesforce instance into the address bar of the new tab, and add `/one/one.app` to the end. For example, if your Salesforce instance has an URL of `https://na4.salesforce.com`, use `https://na4.salesforce.com/one/one.app`.



You should now see the mobile browser version of Saleforce1. As you go through the exercises in this workbook, you can develop in one tab and then test in the other!

Note: The `one/one.app` version is great for development, but you should always test on the actual devices and browsers that you intend to support.

Step 4: Download the Salesforce1 Mobile App

For final testing, you'll also need to install the Salesforce1 mobile app on your device. If you've already downloaded the Salesforce1 mobile app, you can skip this step.

1. Use your mobile device's browser to go to `www.salesforce.com/mobile`, select the appropriate platform, and download Salesforce1.
2. Open Salesforce1 from your mobile device.
3. Enter your Salesforce credentials and tap **Log in to Salesforce**.
4. If you're prompted to allow access to your data, tap **OK** and continue.

You're not going to use the mobile device just yet, but now that everything is set up for development and testing, you can take a look at that package you installed.

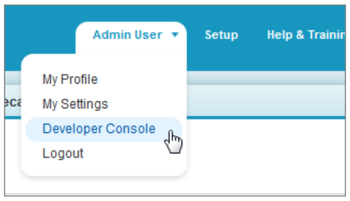
Tutorial 2: Using the Developer Console

The Developer Console lets you execute Apex code statements. It also lets you execute Apex methods within an Apex class or object. In this tutorial you open the Developer Console, execute some basic Apex statements, and toggle a few log settings.

Lesson 1: Activating the Developer Console

After logging into your Salesforce environment, the screen displays the current application you're using (in the diagram below, it's **Warehouse**), as well as your name.

1. Click ***<Your name>*** > **Developer Console**.

   The Developer Console opens in a separate window.

   Note: If you don't see the Developer Console option, you might not be using an appropriate type of Force.com environment—see Before You Begin at the beginning of this workbook for more information.

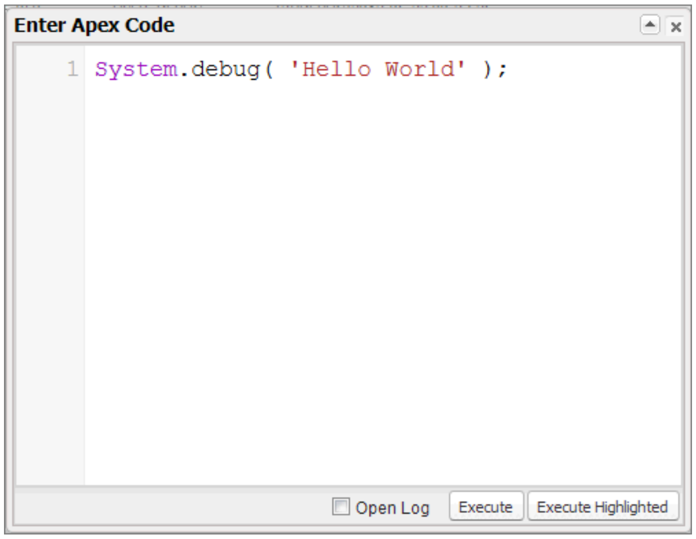
2. If this is your first time opening the Developer Console, you can take a tour of the Developer Console features. Click **Start Tour** to learn more about the Developer Console.

You can open the Developer Console at any time.

Lesson 2: Using the Developer Console to Execute Apex Code

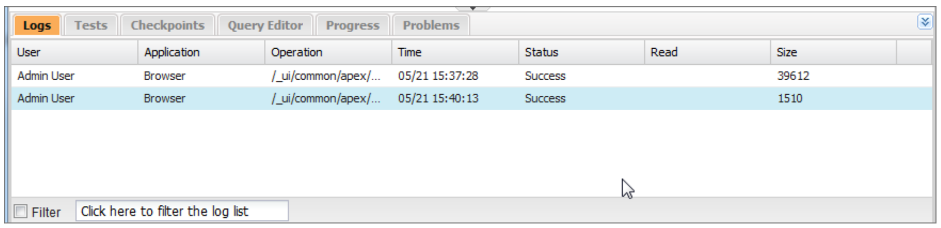
The Developer Console can look overwhelming, but it's just a collection of tools that help you work with code. In this lesson, you'll execute Apex code and view the results in the Log Inspector. The Log Inspector is the Developer Console tool you'll use most often.

1. Click **Debug** > **Open Execute Anonymous Window** or Ctrl/E.
2. In the Enter Apex Code window, enter the following text: `System.debug( 'Hello World' );`
3. Deselect **Open Log** and click **Execute**.



```
System.debug( 'Hello World' );
```

Every time you execute code, a log is created and listed in the *Logs* panel.

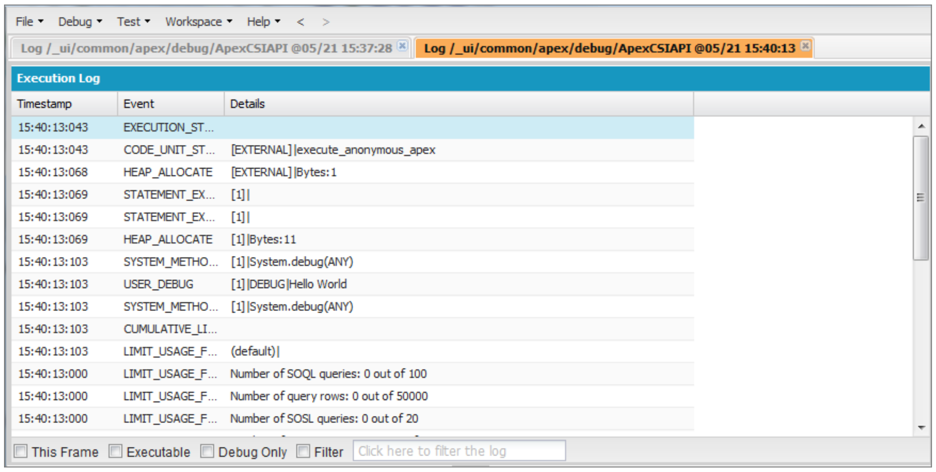


User	Application	Operation	Time	Status	Read	Size
Admin User	Browser	/_ui/common/apex/...	05/21 15:37:28	Success		39612
Admin User	Browser	/_ui/common/apex/...	05/21 15:40:13	Success		1510

Double-click a log to open it in the Log Inspector. You can open multiple logs at a time to compare results.

Log Inspector is a context-sensitive execution viewer that shows the source of an operation, what triggered the operation, and what occurred afterward. Use this tool to inspect debug logs that include database events, Apex processing, workflow, and validation logic.

The Log Inspector includes predefined perspectives for specific uses. Click **Debug** > **Switch Perspective** to select a different view, or click Ctrl/P to select individual panels. You'll probably use the Execution Log panel the most. It displays the stream of events that occur when code executes. Even a single statement generates a lot of events. The Log Inspector captures many event types: method entry and exit, database and web service interactions, and resource limits. The event type `USER_DEBUG` indicates the execution of a `System.debug()` statement.


File
Debug
Test
Workspace
Help
Log /_ui/common/apex/debug/ApexCSIAPI @05/21 15:37:28
Log /_ui/common/apex/debug/ApexCSIAPI @05/21 15:40:13
Execution Log

Timestamp	Event	Details
15:40:13:043	EXECUTION_ST...	
15:40:13:043	CODE_UNIT_ST...	[EXTERNAL]|execute_anonymous_apex
15:40:13:068	HEAP_ALLOCATE	[EXTERNAL]|Bytes:1
15:40:13:069	STATEMENT_EX...	[1]|
15:40:13:069	STATEMENT_EX...	[1]|
15:40:13:069	HEAP_ALLOCATE	[1]|Bytes:11
15:40:13:103	SYSTEM_METHO...	[1]|System.debug(ANY)
15:40:13:103	USER_DEBUG	[1]|DEBUG|Hello World
15:40:13:103	SYSTEM_METHO...	[1]|System.debug(ANY)
15:40:13:103	CUMULATIVE_LI...	
15:40:13:103	LIMIT_USAGE_F...	(default)|
15:40:13:000	LIMIT_USAGE_F...	Number of SOQL queries: 0 out of 100
15:40:13:000	LIMIT_USAGE_F...	Number of query rows: 0 out of 50000
15:40:13:000	LIMIT_USAGE_F...	Number of SOSL queries: 0 out of 20

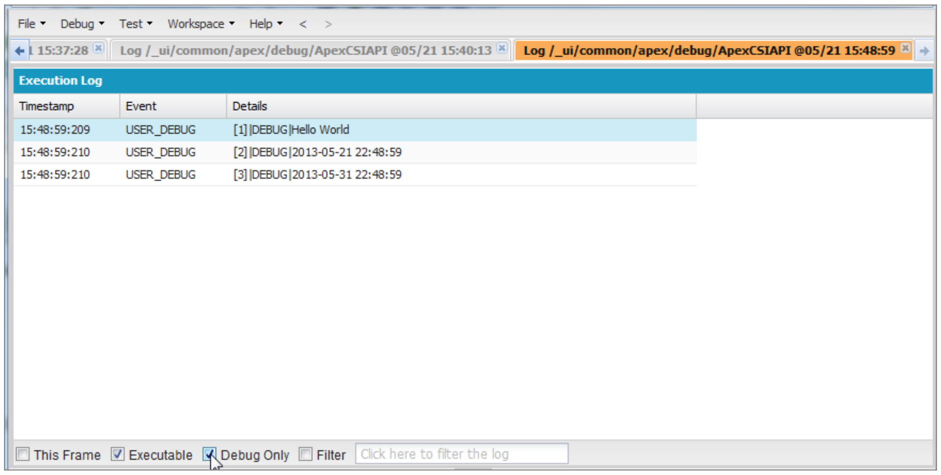
☐ This Frame ☐ Executable ☐ Debug Only ☐ Filter Click here to filter the log

1. Click **Debug** > **Open Execute Anonymous Window** or Ctrl/E and enter the following code:

```
System.debug( 'Hello World' );
System.debug( System.now() );
System.debug( System.now() + 10 );
```

2. Select **Open Log** and click **Execute**.
3. In the Execution Log panel, select **Executable**. This limits the display to only those items that represent executed statements. For example, it filters out the cumulative limits.
4. To filter the list to show only `USER_DEBUG` events, select **Debug Only** or enter `USER` in the **Filter** field.

Note: The filter text is case sensitive.


File
Debug
Test
Workspace
Help
15:37:28
Log /_ui/common/apex/debug/ApexCSIAPI @05/21 15:40:13
Log /_ui/common/apex/debug/ApexCSIAPI @05/21 15:48:59
Execution Log

Timestamp	Event	Details
15:48:59:209	USER_DEBUG	[1]|DEBUG|Hello World
15:48:59:210	USER_DEBUG	[2]|DEBUG|2013-05-21 22:48:59
15:48:59:210	USER_DEBUG	[3]|DEBUG|2013-05-31 22:48:59

☐ This Frame ☑ Executable ☑ Debug Only ☐ Filter Click here to filter the log

Congratulations—you have successfully executed code on the Force.com platform and viewed the results! You'll learn more about the Developer Console tools in later tutorials.

Tell Me More...

Help Link in the Developer Console

To learn more about a particular aspect of the Developer Console, click the Help link in the Developer Console.

Anonymous Blocks

The Developer Console allows you to execute code statements on the fly. You can quickly evaluate the results in the *Logs* panel. The code that you execute in the Developer Console is referred to as an anonymous block. Anonymous blocks run as the current user and can fail to compile if the code violates the user's object- and field-level permissions. Note that this not the case for Apex classes and triggers. .

Summary

To execute Apex code and view the results of the execution, use the Developer Console. The detailed execution results include not only the output generated by the code, but also events that occur along the execution path. Such events include the results of calling another piece of code and interactions with the database.

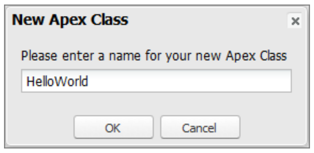
Tutorial 3: Creating and Instantiating Classes

Apex is an object-oriented programming language, and much of the Apex you write will be contained in classes, sometimes referred to as blueprints or templates for objects. In this tutorial you'll create a simple class with two methods, and then execute them from the Developer Console.

Lesson 1: Creating an Apex Class Using the Developer Console

To create a Apex classes in the Developer Console:

1. Click ***`Your Name`*** > **Developer Console** to open the Developer Console.
2. Click **File** > **New** > **Apex Class**.
3. Enter `HelloWorld` for the name of the new class and click **OK**.



New Apex Class

Please enter a name for your new Apex Class

HelloWorld

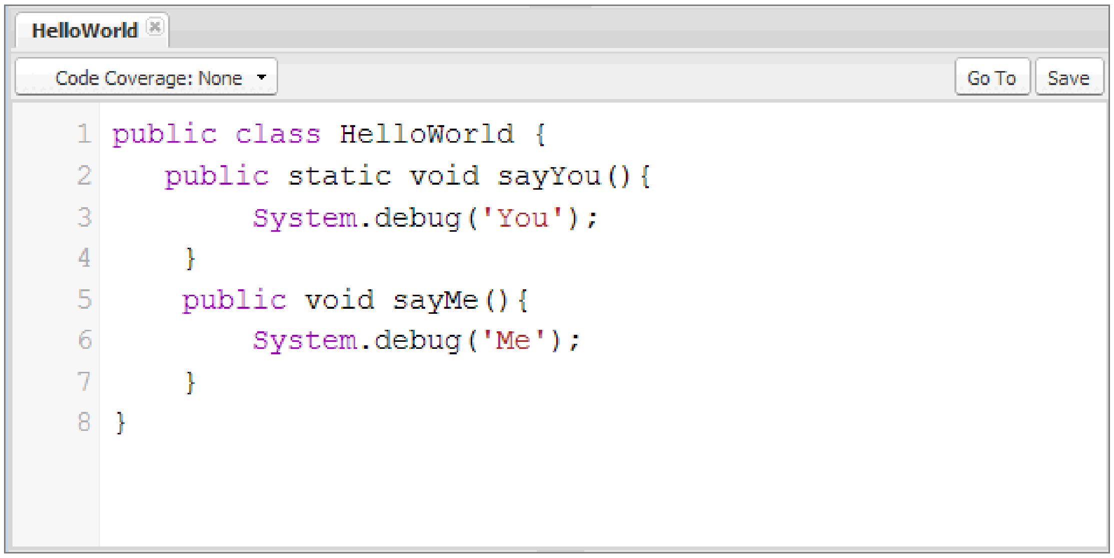
4. A new empty `HelloWorld` class is created. Add a static method to the class by adding the following text between the braces:

```
public static void sayYou() {
    System.debug( 'You' );
}
```

5. Add an instance method by adding the following text just before the final closing brace:

```
public void sayMe() {
    System.debug( 'Me' );
}
```

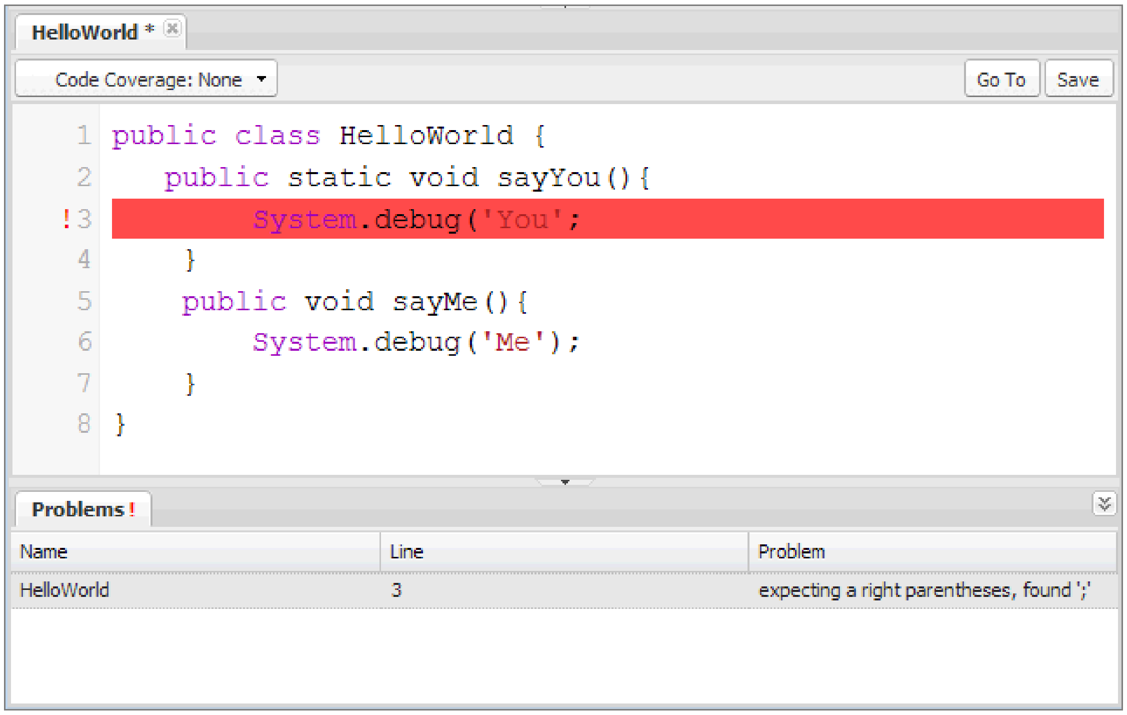
6. Click **Save**.



```
public class HelloWorld {
   public static void sayYou(){
        System.debug('You');
    }
    public void sayMe(){
        System.debug('Me');
    }
}
```

Tell Me More...

- You've created a class called `HelloWorld` with a static method `sayYou()` and an instance method `sayMe()`. Looking at the definition of the methods, you'll see that they call another class, `System`, invoking the method `debug()` on that class, which will output strings.
- If you invoke the `sayYou()` method of *your* class, it invokes the `debug()` method of the `System` class, and you see the output.
- The Developer Console validates your code in the background to ensure that the code is syntactically correct and compiles successfully. Making mistakes is inevitable, such as typos in your code. If you make a mistake in your code, errors appear in the Problems pane and an exclamation mark is added next to the pane heading: **Problems!**.
- Expand the Problems panel to see a list of errors. Clicking on an error takes you to the line of code where this error is found. For example, the following shows the error that appears after you omit the closing parenthesis at the end of the `System.debug` statement.



```
public class HelloWorld {
   public static void sayYou(){
        System.debug('You';
    }
    public void sayMe(){
        System.debug('Me');
    }
}
```

Name	Line	Problem
HelloWorld	3	expecting a right parentheses, found ';'

Re-add the closing parenthesis and notice that the error goes away.

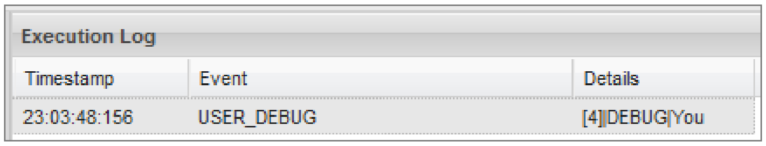
Lesson 2: Calling a Class Method

Now that you've created the `HelloWorld` class, follow these steps to call its methods.

1. Execute the following code in the Developer Console to call the `HelloWorld` class's static method. (See Tutorial 2 if you've forgotten how to do this). You might have to delete any existing code in the entry panel. Notice that to call a static method, you don't have to create an instance of the class.

```
HelloWorld.sayYou();
```

2. Open the resulting log.
3. Set the filters to show `USER_DEBUG` events. (Also covered in Tutorial 2). "You" appears in the log:



Execution Log

Timestamp	Event	Details
23:03:48:156	USER_DEBUG	[4]|DEBUG|You

4. Now execute the following code to call the `HelloWorld` class's instance method. Notice that to call an instance method, you first have to create an instance of the `HelloWorld` class.

```
HelloWorld hw = new HelloWorld();
hw.sayMe();
```

5. Open the resulting log and set the filters.
6. "Me" appears in the Details column. This code creates an instance of the `HelloWorld` class, and assigns it to a variable called *`hw`*. It then calls the `sayMe()` method on that instance.
7. Clear the filters on both logs, and compare the two execution logs. The most obvious differences are related to creating the `HelloWorld` instance and assigning it to the variable `hw`. Do you see any other differences?

Congratulations—you have now successfully created and executed new code on the Force.com platform!

Lesson 3: Creating an Apex Class Using the Salesforce User Interface

You can also create an Apex class in the Salesforce user interface.

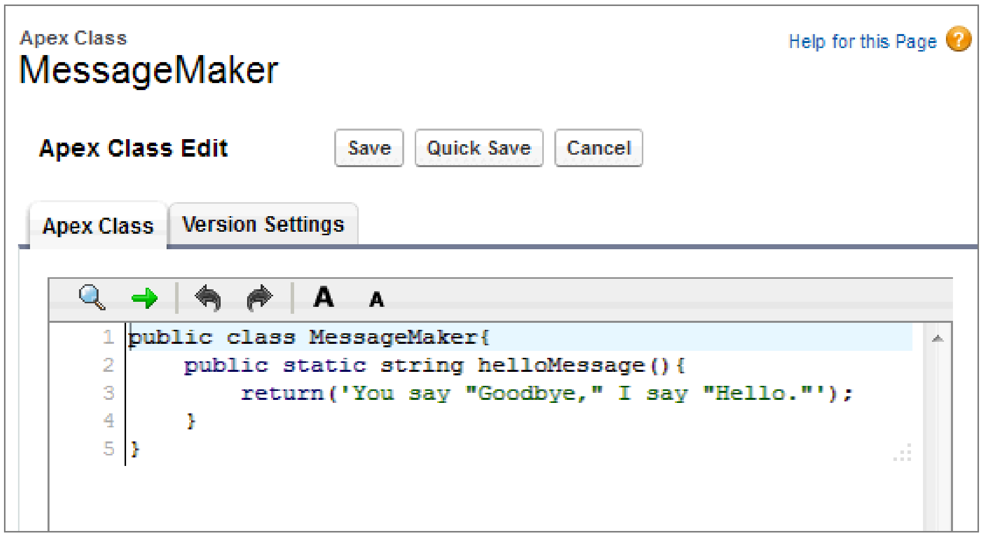
1. From Setup, click **Develop** > **Apex Classes**.
2. Click **New**.
3. In the editor pane, enter the following code:

```
public class MessageMaker {
}
```

4. Click **Quick Save**. You could have clicked **Save** instead, but that closes the class editor and returns you to the Apex Classes list. **Quick Save** saves the Apex code, making it available to be executed, yet it also lets you continue editing—making it easier to add to and modify the code.

5. Add the following code to the class:

```
public static string helloMessage() {
    return('You say "Goodbye," I say "Hello"');
}
```



```
public class MessageMaker{
    public static string helloMessage(){
        return('You say "Goodbye," I say "Hello."');
    }
}
```

6. Click **Save**.

You can also view the class you've just created in the Developer Console and edit it.

1. In the Developer Console, click **File** > **Open**.
2. In the Setup Entity Type panel, click **Classes**, and then double-click **MessageMaker** from the **Entities** panel.

   The `MessageMaker` class displays in the source code editor. You can edit the code there by typing directly in the editor and saving the class.

Tutorial 4: Adding Custom Business Logic Using Triggers

Triggers are Apex code that execute before or after an insert, update, delete or undelete event occurs on an sObject. Think of them as classes with a particular syntax that lets you specify when they should run, depending on how a database record is modified.

The syntax that introduces a trigger definition is very different to that of a class or interface. A trigger always starts with the trigger keyword, followed by the name of the trigger, the database object to which the trigger should be attached to, and then the conditions under which it should fire, for example, before a new record of that database object is inserted. Triggers have the following syntax:

```
trigger triggerName on ObjectName (trigger_events) {
   code_block
}
```

You can specify multiple trigger events in a comma-separated list if you want the trigger to execute before or after insert, update, delete, and undelete operations. The events you can specify are:

- `before insert`
- `before update`
- `before delete`
- `after insert`
- `after update`
- `after delete`
- `after undelete`

Lesson 1: Creating a Trigger

The trigger you'll create in this lesson fires before the deletion of invoices. It prevents the deletion of invoices if they contain line items.

1. In the Developer Console, click **File** > **New** > **Apex Trigger**.
2. Enter `RestrictInvoiceDeletion` for the name and select `Invoice__c` from the **sObject** drop-down list. Click **Submit**.
3. Delete the auto-generated code and add the following.

```
trigger RestrictInvoiceDeletion on Invoice__c (before delete) {
    // With each of the invoices targeted by the trigger
    //   and that have line items, add an error to prevent them
    //   from being deleted.
    for (Invoice__c invoice :
                    [SELECT Id
                    FROM Invoice__c
                    WHERE Id IN (SELECT Invoice__c FROM Line_Item__c) AND
                    Id IN :Trigger.old]){
    Trigger.oldMap.get(invoice.Id).addError('Cannot delete invoices with lineitems');
    }
}
```

4. Click **Save**.

Once you save the trigger, it is active by default.

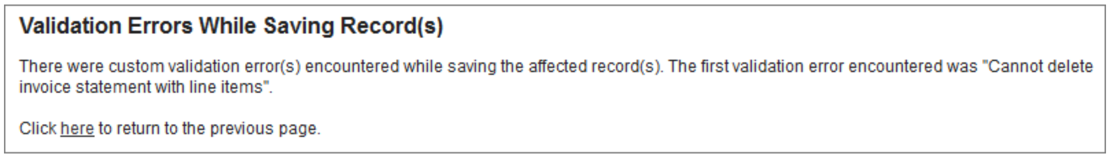
Tell Me More...

- The trigger is called `RestrictInvoiceDeletion` and is associated with the `Invoice__c` sObject.
- The trigger fires before one or more `Invoice__c` sObjects are deleted. This is specified by the `before delete` parameter.
- The trigger contains a SOQL for loop that iterates over the invoices that are targeted by the trigger and that have line items.
- Look at the nested query in the first condition in the `WHERE` clause: `(SELECT Invoice__c FROM Line_Item__c)`. Each line item has an `Invoice__c` field that references the parent invoice. The nested query retrieves the parent invoices of all line items.
- The query checks whether the Id values of the invoices are part of the set of parent invoices returned by the nested query: `WHERE Id IN (SELECT Invoice__c FROM Line_Item__c)`
- The second condition restricts the set of invoices queried to the ones that this trigger targets. This is done by checking if each Id value is contained in `Trigger.old`. `Trigger.old` contains the set of old records that are to be deleted but haven't been deleted yet.
- For each invoice that meets the conditions, the trigger adds a custom error message using the `addError` method, which prevents the deletion of the record. This custom error messages appears in the user interface when you attempt to delete an invoice with line items, as you'll see in the next lesson.

Lesson 2: Invoking the Trigger

In this lesson, you'll invoke the trigger you created in the previous lesson. The trigger fires before the deletion of invoices, so you can cause it to fire by deleting an invoice either through the user interface or programmatically. In this lesson, you'll perform the deletion through the user interface so you can see the error message returned by the trigger when the invoice has line items. You'll also create an invoice with no line items. You'll be able to delete this new invoice since the trigger doesn't prevent the deletion of invoice without line items.

1. Click the **Invoices** tab.
2. With the **View** drop-down list selected to `All`, click **Go!**.
3. Click the sample invoice, with a name like **INV-0000**.
4. On the invoice detail page, click **Delete**.
5. Click **OK** when asked for confirmation.
   A new page displays with the following error message:



Validation Errors While Saving Record(s)

There were custom validation error(s) encountered while saving the affected record(s). The first validation error encountered was "Cannot delete invoice statement with line items".

Click here to return to the previous page.

6. Click the link to go back to the invoice's page.
7. Click **Back to List: Invoices**.
8. You're now going to create another invoice that doesn't contain any line items. Click **New Invoice**.
9. Click **Save**.
10. Try to delete this invoice; click **Delete**.
11. Click **OK** when asked for confirmation.

This time the invoice gets deleted. When the trigger is invoked, the trigger query only selects the invoices that have line items and prevents those records from deletion by marking them with an error. Since this invoice doesn't have any line items, it is not part of those records that the trigger marks with an error and the deletion is allowed.

Tell Me More...

- The validation error message that appears when deleting the sample invoice is the error message specified in the trigger using the `addError` method on the invoice sObject: 'Cannot delete invoice with line items.'
- The trigger universally enforces the business rule, no matter where operations come from: a user, another program, or a bulk operation. This lesson performed the deletion manually through the user interface.

Summary

In this tutorial, you exercised the trigger by attempting to delete two invoices. You saw how the trigger prevented the deletion of an invoice with a line item, and you were able to view the error message in the user interface. In the next tutorial, Tutorial 5: Apex Unit Tests, you'll cause the trigger to be invoked programmatically. You'll add test methods that attempt to delete invoices with and without line items.

Tutorial 5: Apex Unit Tests

Writing unit tests for your code is fundamental to developing Apex code. You must have 75% test coverage to be able to deploy your Apex code to your production organization. In addition, the tests counted as part of the test coverage must pass. Testing is key to ensuring the quality of your application. Furthermore, having a set of tests that you can rerun in the future if you have to make changes to your code allows you to catch any potential regressions to the existing code.

Note:

This test coverage requirement also applies for creating a package of your application and publishing it on the Force.com AppExchange. When performing service upgrades, Salesforce executes Apex unit tests of all organizations to ensure quality and that no existing behavior has been altered for customers.

Test Data Isolation and Transient Nature

By default, Apex test methods don't have access to pre-existing data in the organization. You must create your own test data for each test method. In this way, tests won't depend on organization data and won't fail because of missing data when the data it depends on no longer exists.

Test data isn't committed to the database and is rolled back when the test execution completes. This means that you don't have to delete the data that is created in the tests. When the test finishes execution, the data created during test execution won't be persisted in the organization and won't be available.

You can create test data either in your test method or you can write utility test classes containing methods for test data creation that can be called by other tests.

There are some objects that tests can still access in the organization. They're metadata objects and objects used to manage your organization, such as `User` or `Profile`.

Lesson 1: Adding a Test Utility Class

In this lesson, you'll add tests to test the trigger that you created in the previous tutorial. Because you need to create some test data, you'll add a test utility class that contains methods for test data creation that can be called from any other test class or test method.

1. In the Developer Console, click **File** > **New** > **Apex Class**.
2. For the class name, enter `TestDataFactory` and click **OK**.
3. Delete the auto-generated code and add the following.

```
@isTest
public class TestDataFactory {
    public static Invoice__c createOneInvoiceStatement(
                                             Boolean withLineItem) {

        // Create Warehouse if one does not exist
        Warehouse__c w;
        List<Warehouse__c> warehouse = [SELECT ID from Warehouse__c];


        if(warehouse.size() == 0) {
           w = createWarehouseLocation('Warehouse 1');

        } else {
           w = warehouse[0];
        }
        // Create one invoice
        Invoice__c testInvoice = createInvoiceStatement();
  if (withLineItem == true) {


  // Create a merchandise item
  Merchandise__c m = createMerchandiseItem('Orange juice',w);
  // Create one line item and associate it with the invoice
     AddLineItem(testInvoice, m);
    }
    return testInvoice;
}

// Helper methods
//
private static Warehouse__c createWarehouseLocation(String warehouseName) {
    Warehouse__c w = new Warehouse__c(
        Name=warehouseName,
        City__c='San Francisco',
        Location__Latitude__s=37.7833,
     Location__Longitude__s=122.4167);
 insert w;
 return w;
}

private static Merchandise__c createMerchandiseItem(String merchName, Warehouse__c w) {

    Merchandise__c m = new Merchandise__c(
        Name=merchName,
        Price__c=2,
        Quantity__c=1000,
        Warehouse__c = w.Id);
 insert m;
 return m;
}

private static Invoice__c createInvoiceStatement() {
    Invoice__c inv = new Invoice__c();
     insert inv;
  return inv;
}

private static Line_Item__c AddLineItem(Invoice__c inv,
                                        Merchandise__c m) {
     Line_Item__c lineItem = new Line_Item__c(
        Invoice__c = inv.Id,
        Merchandise__c = m.Id,
  Unit_Price__c = m.Price__c,
  Quantity__c = (Double)(10*Math.random()+1));

        insert lineItem;
        return lineItem;
    }
}
```

4. Click **Save**.

Tell Me More...

- This class contains one public method called `createOneInvoiceStatement` that creates an invoice and a merchandise item to be used as test data in test methods in the next lesson. It takes a Boolean argument that indicates whether a line item is to be added to the invoice.
- It also contains three helper methods that are used by `createOneInvoiceStatement`. These methods are all private and are used only within this class.

- Even though any Apex class can contain public methods for test data creation, this common utility class is defined with the `@isTest` annotation. The added benefit of using this annotation is that the class won't count against the 3 MB organization code size limit. The public methods in this can only be called by test code.

Lesson 2: Adding Test Methods

Now that you've added a utility class that is called by the test method to create some data used for testing, you're ready to create the class that contains the test methods. Follow this procedure to add a test class and test methods.

1. In the *Repository* tab, click `Classes` in the Setup Entity Type section, and then click **New**.
2. For the class name, enter `TestInvoiceStatementDeletion` and click **OK**.
3. Delete the auto-generated code and add the following.

```
@isTest
private class TestInvoiceStatementDeletion {

    static testmethod void TestDeleteInvoiceWithLineItem() {
        // Create an invoice with a line item then try to delete it
        Invoice__c inv = TestDataFactory.createOneInvoiceStatement(true);
        Test.startTest();
        Database.DeleteResult result = Database.delete(inv, false);
        Test.stopTest();

        // Verify invoice with a line item didn't get deleted.
        System.assert(!result.isSuccess());
    }

    static testmethod void TestDeleteInvoiceWithoutLineItems() {
        // Create an invoice without a line item and try to delete it
        Invoice__c inv = TestDataFactory.createOneInvoiceStatement(false);
        Test.startTest();
        Database.DeleteResult result = Database.delete(inv, false);
        Test.stopTest();

        // Verify invoice without line items got deleted.
        System.assert(result.isSuccess());
    }

    static testmethod void TestBulkDeleteInvoices() {
        // Create two invoices, one with and one with out line items
        // Then try to delete them both in a bulk operation, as might happen
        // when a trigger fires.
        List<Invoice__c> invList = new List<Invoice__c>();
        invList.add(TestDataFactory.createOneInvoiceStatement(true));
        invList.add(TestDataFactory.createOneInvoiceStatement(false));
        Test.startTest();
        Database.DeleteResult[] results = Database.delete(invList, false);
        Test.stopTest();

        // Verify the invoice with the line item didn't get deleted
        System.assert(!results[0].isSuccess());

        // Verity the invoice without line items did get deleted.
        System.assert(results[1].isSuccess());
    }
}
```

4. Click **Save**.

Tell Me More...

- The class is defined with the `@isTest` annotation. You saw this annotation in the previous lesson to define a common test utility class. In this lesson, this annotation is used to mark the class as a test class to contain test methods that Apex can execute. Note that any Apex class can contain test methods mixed with other code when not defined with this annotation, but it is recommended to use it.
- The class contains three test methods. Test methods are static, top-level methods that take no arguments. They're defined with the `testmethod` keyword or the `@isTest` annotation. Both of these forms are valid declarations:

  Declaration of a test method using the `testmethod` keyword:

  ```
  static testmethod void myTest() {
      // Add test logic
  }
  ```

  Declaration of a test method using the `@isTest` annotation:

  ```
  static @isTest void myTest() {
      // Add test logic
  }
  ```

- Here is a description of each test method in this class:
  - ◊ `TestDeleteInvoiceWithLineItem`: This test method verifies that the trigger does what it is supposed to do—namely it prevents an invoice with line items from being deleted. It creates an invoice with a line item using the test factory method and deletes the invoice using the `Database.delete` Apex method. This method returns a `Database.DeleteResult` object that you can use to determine if the operation was successful and get the list of errors. The test calls the `isSuccess` method to verify that it is `false` since the invoice shouldn't have been deleted.
  - ◊ `TestDeleteInvoiceWithoutLineItems`: This test method verifies that the trigger doesn't prevent the deletion of invoices that don't have line items. It inserts an invoice without any line items and then deletes the invoice. Like the previous method, it calls the test factory method to create the test data and then calls `Database.delete` for the delete operation. The test verifies that the `isSuccess` method of `Database.DeleteResult` returns `true`.
  - ◊ `TestDeleteInvoiceWithoutLineItems`: Last but not least, this third test method performs a bulk delete on a list of invoices. It creates a list with two invoices, the first of which has one line item and the second doesn't have any. It calls `Database.delete` by passing a list of invoices to delete. Notice that this time we have a second parameter. It is an optional Boolean parameter that indicates whether the delete operation on all sObjects should be rolled back if the deletion on some sObjects fails. We passed the value `false`, which means the delete DML operation shouldn't be rolled back on partial success and the sObjects that don't cause errors will be deleted. In this case, the test calls the `isSuccess` method of `Database.DeleteResult` to verify that the first invoice isn't deleted but the second one is.
- Each test method executes the delete DML operation within `Test.startTest`/`Test.stopTest` blocks. Each test method can have only one such block. All code running within this block is assigned a new set of governor limits separate from the other code in the test. This ensures that other setup code in your test doesn't share the same limits and enables you to test the governor limits. Although this isn't critical in our case since we're deleting one or two records at a time and won't be hitting any limits; however, it's good practice to enclose the actual test statements within `Test.startTest`/`Test.stopTest`. You can perform prerequisite setup and test data creation prior to calling `Test.startTest` and include verifications after `Test.stopTest`. The original set of governor limits are reverted to after the call to `Test.stopTest`.

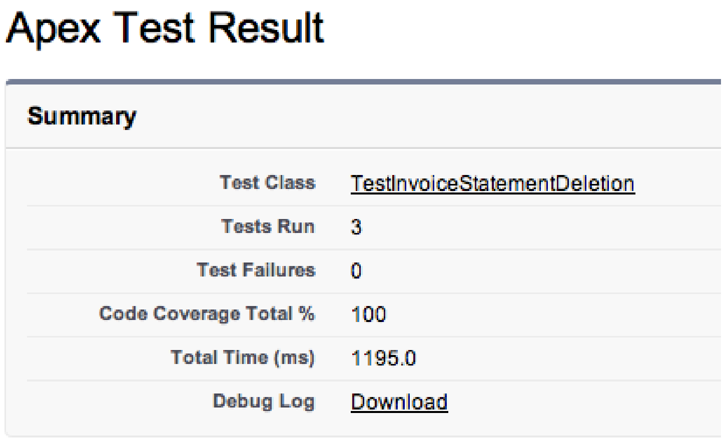
Lesson 3: Running Tests and Code Coverage

Now that you've added the tests, you need to run them and inspect their results. In addition to ensuring that the tests pass, you'll be able to find out how much of the code was covered by your tests.

There are many ways to run tests in Apex. You can either run all tests in a single class or all tests in all classes through the Apex Classes pages. You can also execute the tests asynchronously in the Apex Test Execution page, which allows you to check the results at a later time and access saved test results. Last but not least, you can also start the asynchronous execution of tests by writing Apex code that inserts and queries API objects.

For the purposes of this tutorial, you'll use the Apex Classes page in the user interface and then inspect the code coverage results.

1. From Setup, click **Develop** > **Apex Classes**.
2. Locate the **TestInvoiceStatementDeletion** link in the list of classes and click it.
3. From the detail page of the `TestInvoiceStatementDeletion` class, click **Run Test**.
4. After all test methods in the class execute, the Apex Test Result page displays, containing a summary of the test results.

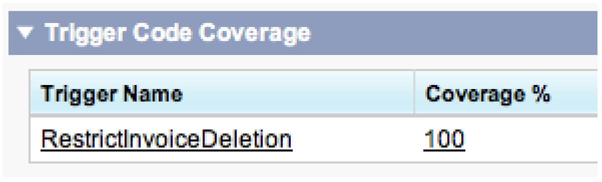

Apex Test Result

Summary

Test Class	TestInvoiceStatementDeletion
Tests Run	3
Test Failures	0
Code Coverage Total %	100
Total Time (ms)	1195.0
Debug Log	Download

The Summary section of the page shows that three tests were run with zero failures, which means that all our tests passed. The code coverage is 100%, which means that all code lines in the trigger were exercised by the tests. This section also contains a link to the debug log that contains debug log output of the executed code.

5. Scroll down to the Code Coverage section. This section contains the code coverage of all triggers and classes in the organization. You can find the code coverage of the trigger listed there.


Trigger Code Coverage

Trigger Name	Coverage %
RestrictInvoiceDeletion	100

6. Click the coverage percentage number to see the statements covered in the trigger. In this case, you'll see two blue lines corresponding to the statements that have code coverage.

   If there are statements that aren't covered by a test, they appear as red lines. In our case, there are no red lines because we have 100% coverage.
7. To update code coverage results for the trigger on the Apex Triggers page, click **Apex Classes**, then click **Calculate your organization's code coverage**.
8. Click **Apex Triggers**.
   You can view the code coverage of the trigger listed at 100% in the Code Coverage column.

Alternatively, you can click **Run All Tests**, which runs all tests in your organization and updates the code coverage results on the page at once.

Summary

In this tutorial, you learned the syntax of test classes and test methods, and the advantage of using a test class for your test methods annotated with `@isTest`. You created a test data factory class to create test data. You ran all tests and verified test results and code coverage. Last but not least, you learned the importance of having at least 75% test coverage as a requirement for deploying Apex to another organization.

Tutorial 6: Apex Batch Processing

Using batch Apex classes, you can process records in batches asynchronously. If you have a large number of records to process, for example, for data cleansing or archiving, batch Apex is your solution. Each invocation of a batch class results in a job being placed on the Apex job queue for execution.

The execution logic of the batch class is called once for each batch of records. The default batch size is 200 records. You can also specify a custom batch size. Furthermore, each batch execution is considered a discrete transaction. With each new batch of records, a new set of governor limits is in effect. In this way, it's easier to ensure that your code stays within the governor execution limits. Another benefit of discrete batch transactions is to allow for partial processing of a batch of records in case one batch fails to process successfully, all other batch transactions aren't affected and aren't rolled back if they were processed successfully.

Batch Apex Syntax

To write a batch Apex class, your class must implement the `Database.Batchable` interface. Your class declaration must include the `implements` keyword followed by `Database.Batchable<sObject>`. Here is an example:

```
global class CleanUpRecords implements Database.Batchable<sObject> {
```

You must also implement three methods:

- `start` method

  ```
  global (Database.QueryLocator | Iterable<sObject>) start(Database.BatchableContext bc)
  {}
  ```

  The `start` method is called at the beginning of a batch Apex job. It collects the records or objects to be passed to the interface method `execute`.

- `execute` method:

  ```
  global void execute(Database.BatchableContext BC, list<P>){}
  ```

  The `execute` method is called for each batch of records passed to the method. Use this method to do all required processing for each chunk of data.

  This method takes the following:

  ◊ A reference to the `Database.BatchableContext` object.
  ◊ A list of sObjects, such as `List<sObject>`, or a list of parameterized types. If you are using a `Database.QueryLocator`, the returned list should be used.

  Batches of records are not guaranteed to execute in the order they are received from the `start` method.

- `finish` method

  ```
  global void finish(Database.BatchableContext BC){}
  ```

  The `finish` method is called after all batches are processed. Use this method to send confirmation emails or execute post-processing operations.

Invoking a Batch Class

To invoke a batch class, instantiate it first and then call `Database.executeBatch` with the instance of your batch class:

```
BatchClass myBatchObject = new BatchClass();
Database.executeBatch(myBatchObject);
```

In the next steps of this tutorial, you'll learn how to create a batch class, test it, and invoke a batch job.

Lesson 2: Adding a Test for the Batch Apex Class

In this lesson, you'll add a test class for the `CleanUpRecords` batch class. The test in this class invokes the batch job and verifies that it deletes all merchandise records that haven't been purchased.

1. In the *Repository* tab, click `Classes` in the Setup Entity Type section, and then click **New**.
2. For the class name, enter `TestCleanUpBatchClass` and click **OK**.
3. Delete the auto-generated code and add the following.

```
@isTest
private class TestCleanUpBatchClass {
    static testmethod void test() {
        // The query used by the batch job.
        String query = 'SELECT Id,CreatedDate FROM Merchandise__c ' +
                   'WHERE Id NOT IN (SELECT Merchandise__c FROM Line_Item__c)';

        // Create some test merchandise items to be deleted
       //   by the batch job.
       Warehouse__c w = new Warehouse__c(
            Name='Warehouse 1',
            City__c='San Francisco',
            Location__Latitude__s=37.7833,
            Location__Longitude__s=122.4167);
        insert w;

       Merchandise__c[] ml = new List<Merchandise__c>();
       for (Integer i=0;i<10;i++) {
           Merchandise__c m = new Merchandise__c(
               Name='Merchandise ' + i,
               Price__c=2,
               Quantity__c=100,
               Warehouse__c = w.Id);
      ml.add(m);
            }
        insert ml;
        Test.startTest();
        CleanUpRecords c = new CleanUpRecords(query);
        Database.executeBatch(c);
        Test.stopTest();
        // Verify merchandise items got deleted
        Integer i = [SELECT COUNT() FROM Merchandise__c];
        System.assertEquals(i, 0);
    }
}
```

4. Click **Save**.

Tell Me More...

- The test class contains one test method called `test`. This test method starts by constructing the query string that is to be passed to the constructor of `CleanUpRecords`. Since a merchandise item that hasn't been purchased is a merchandise item that doesn't have line items associated with it, the SOQL query specifies the following:.

  ```
  WHERE Id NOT IN (SELECT Merchandise__c FROM Line_Item__c)
  ```

  The subquery

  ```
  SELECT Merchandise__c FROM Line_Item__c
  ```

  gets the set of all merchandise items that are referenced in line items. Since the query uses the NOT IN operator in the WHERE clause, this means the merchandise items that aren't referenced in line items are returned.
- The test method inserts 10 merchandise items with no associated line items to be cleaned up by the batch class method. Note that the number of records inserted is less than the batch size of 200 because test methods can execute only one batch total.
- Next, the batch class is instantiated with the query with this statement where the `query` variable is passed to the constructor of `CleanUpRecords`.:

  ```
  CleanUpRecords c = new CleanUpRecords(query);
  ```

- The batch class is invoked by calling `Database.executeBatch` and passing it the instance of the batch class:

  ```
  Database.executeBatch(c);
  ```

- The call to `Database.executeBatch` is included within the `Test.startTest` and `Test.stopTest` block. This is necessary for the batch job to run in a test method. The job executes after the call to `Test.stopTest`. Any asynchronous code included within `Test.startTest` and `Test.stopTest` gets executed synchronously after `Test.stopTest`.
- Finally, the test verifies that all test merchandise items created in this test got deleted by checking that the count of merchandise items is zero.
- Even though the batch class `finish` method sends a status email message, the email isn't sent in this case because email messages don't get sent from test methods.

Lesson 3: Running a Batch Job

You can invoke a batch class from a trigger, a class, or the Developer Console. There are times when you want to run the batch job at a specified schedule. This shows you how to submit the batch class you created in Lesson 1 through the Developer Console for immediate results. You'll also create a scheduler class that enables you to schedule the batch class.

Begin by setting up some merchandise records in the organization that don't have any associated line items. The records that the test created in the previous don't persist, so you'll create some new records to ensure the batch job has some records to process.

1. Click **Debug** > **Execute Anonymous Apex**, and then run the following:

```
// Create Warehouse if one does not exist
        Warehouse__c w;
```

```
        List<Warehouse__c> warehouse = [SELECT ID from Warehouse__c];


        if(warehouse.size() == 1) {
            w = new Warehouse__c(
                Name='Warehouse 1',
                City__c='San Francisco',
                Location__Latitude__s=37.7833,
                Location__Longitude__s=122.4167);
            insert w;

        } else {
           w = warehouse[0];
        }

Merchandise__c[] ml = new List<Merchandise__c>();
for (Integer i=0;i<250;i++) {
   Merchandise__c m = new Merchandise__c(
       Name='Merchandise ' + i,
       Price__c=2,
       Quantity__c=100,
       Warehouse__c = w.Id);
    ml.add(m);
}
    insert ml;
```

2. Click **Execute**.

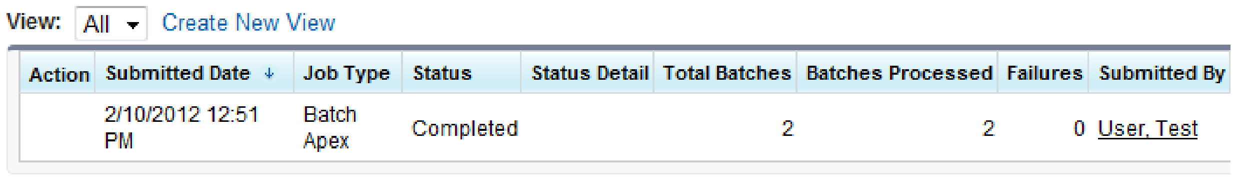
   This creates 250 merchandise items, which ensures that our batch class runs twice, once for the first 200 records, and once for the remaining 50 records.

3. Let's now submit the batch class by calling `Database.executeBatch` from the Developer Console. Run the following in the Execute window:

```
String query = 'SELECT Id,CreatedDate FROM Merchandise__c ' +
    'WHERE Id NOT IN (SELECT Merchandise__c FROM Line_Item__c)';
CleanUpRecords c = new CleanUpRecords(query);
Database.executeBatch(c);
```

   You'll receive an email notification for the job's completion. It might take a few minutes for the email to arrive. The email should state that two batches were run.

4. To view the status of the batch job execution, from Setup, click **Monitoring** > **Apex Jobs** or **Jobs** > **Apex Jobs**. Since the job finished, its status shows as completed and you can see that two batches were processed.



Action	Submitted Date	Job Type	Status	Status Detail	Total Batches	Batches Processed	Failures	Submitted By
	2/10/2012 12:51 PM	Batch Apex	Completed		2	2	0	User, Test

5. To schedule the batch job programmatically, you need to create a class that implements the `Schedulable` interface which invokes the batch class from its `execute` method. First, from Setup, click **Develop** > **Apex Classes** > **New**.
6. In the code editor box, add the following class definition.

```
global class MyScheduler implements Schedulable {

   global void execute(SchedulableContext ctx) {
        // The query used by the batch job.
        String query = 'SELECT Id,CreatedDate FROM Merchandise__c ' +
                        'WHERE Id NOT IN (SELECT Merchandise__c FROM Line_Item__c)';

        CleanUpRecords c = new CleanUpRecords(query);
        Database.executeBatch(c);
    }
}
```

7. Follow steps similar to the ones Lesson 3: Scheduling and Monitoring Scheduled Jobs to schedule the `MyScheduler` class.

Summary

In this tutorial, you created a batch Apex class for data cleanup. You then tested the batch class by writing and running a test method. You also learned how to schedule the batch class.

Batch Apex allows to process records in batches and is useful when you have a large number of records to process.

Tutorial 7: Apex REST

You can create custom REST Web service APIs on top of the Force.com platform or Database.com by exposing your Apex classes as REST resources. Client applications can call the methods of your Apex classes using REST to run Apex code in the platform.

Apex REST supports both XML and JSON for resource formats sent in REST request and responses. By default, Apex REST uses JSON to represent resources.

For authentication, Apex REST supports OAuth 2.0 and the Salesforce session. This tutorial uses Workbench to simulate a REST client. Workbench uses the session of the logged-in user as an authentication mechanism for calling Apex REST methods.

Note: Workbench is a free, open source, community-supported tool (see the Help page in Workbench). Salesforce.com provides a hosted instance of Workbench for demonstration purposes only—salesforce.com recommends that you do not use this hosted instance of Workbench to access data in a production database. If you want to use Workbench for your production database, you can download, host, and configure it using your own resources.

Lesson 1: Adding a Class as a REST Resource

Let's add a class with two methods and expose it through Apex REST.

1. In the Developer Console, click **File** > **New** > **Apex Class**.
2. For the class name, enter `MerchandiseManager` and click **OK**.
3. Delete the auto-generated code and add the following.

```
@RestResource(urlMapping='/Merchandise/*')
global with sharing class MerchandiseManager {


    @HttpGet
    global static Merchandise__c getMerchandiseById() {
        RestRequest req = RestContext.request;
        String merchId = req.requestURI.substring(
                                  req.requestURI.lastIndexOf('/')+1);
        Merchandise__c result =
                       [SELECT Name,Price__c,Quantity__c
                        FROM Merchandise__c
                        WHERE Id = :merchId];
        return result;
    }

    @HttpPost
   global static String createMerchandise(String name, Decimal price, Double inventory,
 String warehouse) {
        List<Warehouse__c> warehouses = [SELECT ID from Warehouse__c WHERE Name =:
warehouse LIMIT 1];

       if(warehouses.size() > 0) {
            Merchandise__c m = new Merchandise__c(
                Name=name,
                Price__c=price,
                Quantity__c=inventory,
                Warehouse__c=warehouses[0].Id);
            insert m;
            return m.Id;
        } else {
          throw new RESTException('No warehouse found by that name');
          return null;
        }

        }

    class RESTException extends Exception {}
}
```

4. Click **Save**.

Tell Me More...

- The class is global and defined with the `@RestResource(urlMapping='/Invoice__c/*')` annotation. Any Apex class you want to expose as a REST API must be global and annotated with the `@RestResource` annotation. The parameter of the `@RestResource` annotation, `urlMapping`, is used to uniquely identify your resource and is relative to the base URL `https://`***`instance`***`.salesforce.com/services/apexrest/`. The base URL and the `urlMapping` value form the URI that the client sends in a REST request. In this case, the URL mapping contains the asterisk wildcard character, which means that the resource URI can contain any value after `/Merchandise/`. In Step 3 of this tutorial, we'll be appending an ID value to the URI for the record to retrieve.
- The class contains two global static methods defined with Apex REST annotations. All Apex REST methods must be global static.
- The first class method, `getMerchandiseById`, is defined with the `@HttpGet` annotation.
  - ◊ The `@HttpGet` annotation exposes the method as a REST API that is called when an HTTP `GET` request is sent from the client.
  - ◊ This method returns the merchandise item that corresponds to the ID sent by the client in the request URI.
  - ◊ It obtains the request and request URI through the Apex static `RestContext` class.
  - ◊ It then parses the URI to find the value passed in after the last `/` character and performs a SOQL query to retrieve the merchandise record with this ID. Finally, it returns this record.
- The second class method, `createMerchandise`, is defined with the `@HttpPost` annotation. This annotation exposes the method as a REST API and is called when an HTTP `POST` request is sent from the client. This method creates a merchandise record using the specified data sent by the client. It calls the `insert` DML operation to insert the new record in the database and returns the ID of the new merchandise record to the client.

Lesson 2: Creating a Record Using the Apex REST POST Method

In this lesson, you'll use REST Explorer in Workbench to send a REST client request to create a new merchandise record. This request invokes one of the Apex REST methods you've just implemented.

Workbench's REST Explorer simulates a REST client. It uses the session of the logged-in user as an authentication mechanism for calling Apex REST methods.

You might be able to skip the first few steps in this procedure if you already set up sample data with Workbench in a previous tutorial.

1. Navigate to: `workbench.developerforce.com`.
2. If prompted for your credentials, enter your login information and click **Login**.
3. For **Environment**, select **Production**.
4. Accept the terms of service and click **Login with Salesforce**.

5. Click **Allow** to allow Workbench to access your information.
6. After logging in, click **utilities** > **REST Explorer**.
7. Click **POST**.
8. The URL path that REST explorer accepts is relative to the instance URL of your org, so you only have to provide the path that is appended to the instance URL. In the relative URL box, replace the default URL with `/services/apexrest/Merchandise/`
9. For the request body, insert the following JSON string representation of the object to insert:

```
{
  "name" : "CRT Monitor",
  "price" : 99,
  "inventory" : 1000,
  "warehouse" : "Aloha Warehouse"
}
```

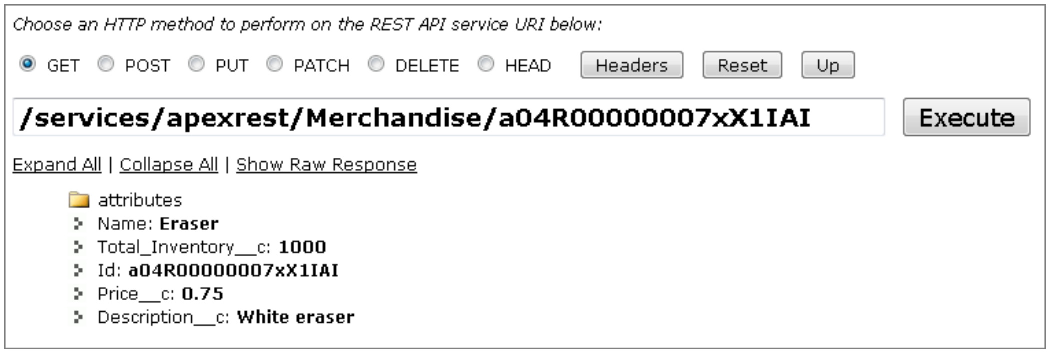
   Note that the field names for the object to create must match and must have the same case as the names of the parameters of the method that will be called.

10. Click **Execute**.
    This causes the `createMerchandise` method to be called. The response contains the ID of the new merchandise record.
11. To obtain the ID value from the response, click **Show Raw Response**, and then copy the ID value, without the quotation marks, that is displayed at the bottom of the response. For example, `"a04R00000007xX1IAI"`, but your value will be different.
    You'll use this ID in the next lesson to retrieve the record you've just inserted.

Lesson 3: Retrieving a Record Using the Apex REST GET Method

In this lesson, you'll use Workbench to send a REST client request to retrieve the new merchandise record you've just created in the previous lesson. This request invokes one of the Apex REST methods you've just implemented.

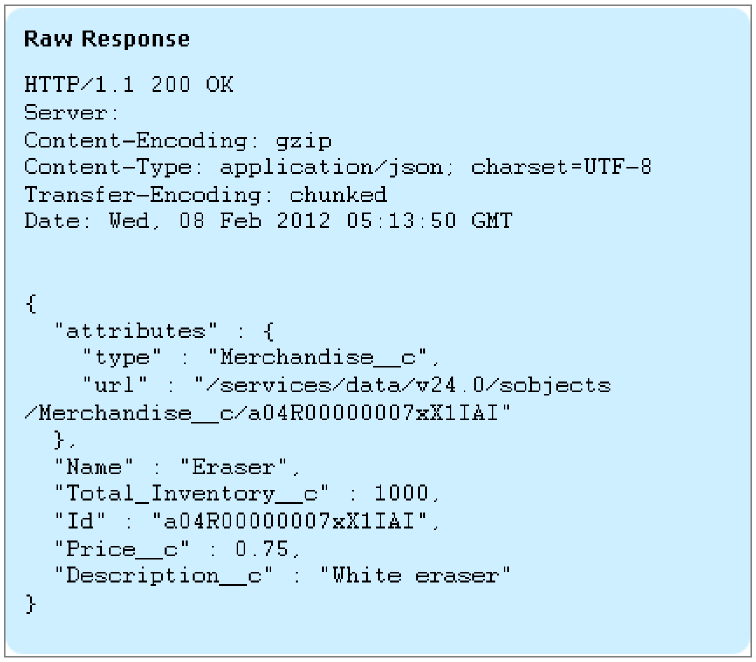
1. In the REST Explorer, Click **GET**.
2. In the relative URL box, append the ID of the record you copied from Lesson 2 of this tutorial to the end of the URL: `/services/apexrest/Merchandise/`.
3. Click **Execute**.
   This causes the `getMerchandiseById` method to be called. The response returned contains the fields of the new merchandise record.


Choose an HTTP method to perform on the REST API service URI below:
GET
POST
PUT
PATCH
DELETE
HEAD
Headers
Reset
Up
Execute
Expand All | Collapse All | Show Raw Response

/services/apexrest/Merchandise/a04R00000007xX1IAI

attributes
Name: Eraser
Total_Inventory__c: 1000
Id: a04R00000007xX1IAI
Price__c: 0.75
Description__c: White eraser

4. Optionally, click **Show Raw Response** to view the entire response, including the HTTP headers and the response body in JSON format.


Raw Response

```
HTTP/1.1 200 OK
Server:
Content-Encoding: gzip
Content-Type: application/json; charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 08 Feb 2012 05:13:50 GMT


{
  "attributes" : {
    "type" : "Merchandise__c",
    "url" : "/services/data/v24.0/sobjects
/Merchandise__c/a04R00000007xX1IAI"
  },
  "Name" : "Eraser",
  "Total_Inventory__c" : 1000,
  "Id" : "a04R00000007xX1IAI",
  "Price__c" : 0.75,
  "Description__c" : "White eraser"
}
```

Summary

In this tutorial, you created a custom REST-based API by writing an Apex class and exposing it as a REST resource. The two methods in the class are called when HTTP `GET` and `POST` requests are received. You also used the methods that you implemented using the REST Explorer in Workbench and saw the raw JSON response.

Tutorial 8: Use Visualforce In a Salesforce1 Mobile App

In this hands-on workshop you learn how to add Visualforce pages to the Salesforce1 mobile app. To streamline the development process, you'll install a package that has three pre-built pages that are ready to run.

Visualforce pages can be made available from a number of places in the Salesforce1 user interface. What the page does will determine where you put it.

- Navigation menu—For pages that aren't related to other objects or that require a full screen, add the page to the navigation menu. These pages are available when you tap ☰ from the Salesforce1 mobile app. In this workshop, **FindNearbyWarehousesPage** is an example of a Visualforce page used in the navigation menu.
- Publisher—Pages that appear within the context of a record should go on the object's page layout. These *custom actions* are available when you tap ➕ from the Salesforce1 mobile app. In this workshop you'll add the **QuickOrderPage** as a custom action.
- Record Home page (as a mobile card)—If you want a Visualforce page to appear on the related items page of an object record, add the page as a mobile card. Mobile cards are mobile only; they don't appear in the full Salesforce site. In this workshop you use **AccountLocation** as a mobile card.

Lesson 1: Add a Visualforce Page to the Navigation Menu

In this tutorial you take the Visualforce page from the package and make it available from the navigation menu. The first thing you'll do is examine the code that makes this page work. The code uses the location of the current user and integrates with Google Maps to display a map with warehouses located within 20 miles. For each nearby warehouse, the map displays a pin along with the warehouse name, address, and phone number. In order to access the page, you need to add it to a Tab, and then you can add it to the Salesforce1 mobile app's navigation menu.

Step 1: Examine the `FindNearby` Apex Class

1. On the package detail page, click `FindNearby`.
2. Scroll down and examine the code after the `//SOQL` comment

This snippet is a dynamic SOQL query that uses variables passed in from the Visualforce page to find warehouses within 20 miles of the device accessing the page. This page works on any mobile device and HTML5–enabled desktop browser. If the page is unable to obtain the location, the search is centered on San Francisco.

```
String queryString =
   'SELECT Id, Name, Location__Longitude__s,
    Location__Latitude__s, ' +
   Street_Address__c, Phone__c, City__c ' +
   'FROM Warehouse__c ' +
   'WHERE DISTANCE(Location__c,
   GEOLOCATION('+lat+','+lon+'), \'mi\') < 20 ' +
   'ORDER BY DISTANCE(Location__c,
   GEOLOCATION('+lat+','+lon+'), \'mi\') ' +
   'LIMIT 10';
```

Step 2: Examine the `Initialize` Function in `FindNearbyWarehousesPage`

1. Navigate back to the package detail page and click `FindNearbyWarehousesPage`.
2. Scroll down and find the `initialize()` function.

The initialize function in the Visualforce page uses the HTML5 geolocation API to get the coordinates of the user. The browser gets the position without using any plug-ins or external libraries, and then uses JavaScript remoting to invoke the `getNearby` function in the Apex controller and passes in the coordinates.

```
function initialize() {
    var lat, lon;
    // Check to see if the device has geolocation
    // detection capabilities with JavaScript
    if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(
            function(position){
            lat = position.coords.latitude;
            lon = position.coords.longitude;
            //Use VF Remoting to send values to be
            //queried in the associated Apex Class
            Visualforce.remoting.Manager.invokeAction(
                '{!$RemoteAction.FindNearby.getNearby}', lat, lon,
                function(result, event){
if (event.status) {
console.log(result);
                        createMap(lat, lon, result);
                    } else if (event.type === 'exception') {
                        //exception case code
                    } else {
} },
                {escape: true}
            );
}); } else {
    //Set default values for map if the device doesn't
    //have geolocation capabilities
         /** San Francisco **/
         lat = 37.77493;
         lon = -122.419416;
         var result = [];
         createMap(lat, lon, result);
    }
}</apex:page>
```

Step 3: Examine the Redirect Code

`FindNearbyWarehousesPage` uses the Google Maps JavaScript API v3 to plot the nearby warehouses on a map. The map is resized based on the records returned by the SOQL query and then each record is plotted as a marker on the map.

The most important piece of the code is determining whether or not the page is being viewed in the Salesforce1 mobile app. If it is, the redirect link to the warehouse record must be coded slightly differently. If the page runs in the Salesforce1 mobile app, you must use the `navigateToSobjectRecord` method to go to the record detail page but still stay in the app.

1. Stay on the same `FindNearbyWarehousesPage` Visualforce page and scroll down to the `setupMarker()` function.

2. Notice how the try/catch construct is used to set the redirect link accordingly.

```
try{
    if(sforce.one){
    warehouseNavUrl =
        'javascript:sforce.one.navigateToSObject(
        \'' + warehouse.Id + '\')';
    }
} catch(err) {
    console.log(err);
    warehouseNavUrl = '\\' + warehouse.Id;
}
    var warehouseDetails =
        '<a href="' + warehouseNavUrl + '">' +
        warehouse.Name + '</a><br/>' +
        warehouse.Street_Address__c + '<br/>' +
        warehouse.City__c + '<br/>' +
        warehouse.Phone__c;
```

Step 4: Create a Tab

Since the package you installed already has a Visualforce page, you don't need to create one. You can simply start by creating a tab.

1. From Setup, click **Create** > **Tabs**.
2. In the Visualforce Tabs section, click **New**.
3. In the Visualforce Page drop-down list, select **FindNearbyWarehousesPage**.
4. In the Tab Label field, enter `Find Nearby Warehouses`.

   The label field is what users see both on the full site and the mobile app. With that in mind, keep your labels no longer than this.
5. Click into the Tab Style field, and select the **Globe** style.

   The icon for this style appears as the icon for the page in the Salesforce1 mobile app's navigation menu.
6. Click **Next**, and **Next** again.
7. Deselect the **Include Tab** checkbox so that the tab isn't included in any of the apps in the full site. You only want this tab to appear when users are viewing it on their mobile device.
8. Click **Save**.

Now that you've created the Visualforce page and the tab, you're ready to add the new tab to the navigation menu.

Step 5: Add the Tab to the Navigation Menu

In this step you add the tab as a navigation menu item in the Salesforce1 mobile app. The menu item will instantly become available to mobile app users that have access to it.

1. From Setup, click **Mobile Administration** > **Mobile Navigation**.
2. Move **Find Nearby Warehouses** to the Selected list and then **Save**.

Step 6: Try Out the App

Now you can search nearby warehouses on your device.

1. Open the Salesforce1 app on your mobile device.
2. Tap ☰ to access the navigation menu. You should see Find Nearby Warehouses under the Apps section.

Note:

- If you're using the `one/one.app` browser version, you may need to refresh the browser to see the page in the navigation menu.
- If you're using the installed mobile app, you may need to log out and log in again to see the change.

3. Tap **Find Nearby Warehouses**.
4. Click **OK** when you see a prompt that asks to use your current location. A map that contains all the nearby warehouse locations within 20 miles appears.

Note: If you don't receive a prompt, this may be related to your device settings. If that's the case, the geographical area should default to San Francisco.

The warehouses in the package sample data are all located in the San Francisco area. If you're testing this from another location, be sure to add a warehouse located within 20 miles. That's it! You can see how easy it is to make standard pages and tabs available to your mobile users.

Lesson 2: Add a Visualforce Custom Action

Actions appear in the *publisher* in the Salesforce1 mobile app, and are a quick way for mobile users to access commonly used tasks or functionality. You can use Visualforce pages as actions and add them to the publisher.

For this tutorial, imagine that you have mobile technicians on site at a customer account. The technicians need a way they can quickly create an order and find the closest warehouse that has the parts they need.

To satisfy this request, you'll create a custom action that will let the technician enter the part name and a radius, and the Visualforce page will search all warehouses within the given radius for that part. Once a warehouse is located, the technician simply enters the quantity and clicks a button to create the order.

Step 1: Examine the Package Contents

1. Click **Setup** and then under the Build section, click **Installed Packages**.
2. Click the **Warehouse** package.
3. On the package detail page, click **View Components**.

The custom action uses these elements from the package that you installed.

- QuickOrderPage Visualforce page — the Visualforce page.
- QuickOrderController Apex class — the controller class.
- Mobile_Design_Templates — a static resource used for styling the page.

Step 2: Examine `QuickOrderController`

`QuickOrderController` uses the `@RemoteAction` annotation on the methods to wrap logic in a JavaScript-friendly way. Visualforce remoting allows for quick and tight integration between Apex and JavaScript.

This communication model works asynchronously as opposed to the synchronous model in the traditional Visualforce/Apex MVC paradigm. So after passing parameters into your controller, you can get the result from a response handler function, and write any additional client-side logic before doing any DOM manipulation or building your page with mobile templates or frameworks.

Take a look at the following snippet to see how that works:

```
global class QuickOrderController{
    public static List<Merchandise__c> merchandise;
    public static Line_Item__c quickOrder;


    public QuickOrderController(ApexPages.
        StandardController controller){
    }

    @RemoteAction
    global static List<Merchandise__c> findWarehouses(String accId,
        String merchName, String warehouseDist){
         merchandise = new List<Merchandise__c>();
         String queryString = '';

         Account acc = [Select Location__Longitude__s,
         Location__Latitude__s, Name, Id
         from Account where Id =: accId];

         //Finds warehouses nearby if you have location
         //specified on the Account
         if(acc.Location__Latitude__s != null &&
             acc.Location__Longitude__s != null){
             queryString = 'SELECT Id, (SELECT Id, Name, Quantity__c,
                 Warehouse__r.Name, Warehouse__r.Id,
                 Warehouse__r.Street_Address__c,
                 Warehouse__r.City__c '+
                 'FROM Merchandise__r WHERE Name
                 like \'%'+merchName+'%\') '
                 +'FROM Warehouse__c WHERE '
                 +'DISTANCE(Location__c, GEOLOCATION('
                 +acc.Location__Latitude__s+','
                 +acc.Location__Longitude__s+'), \'mi\')';
             if(warehouseDist != null){
                 queryString += ' <'+ warehouseDist;
             }

         }
         //If no location defined on the Account, this will run
         //query against the merchandise name only
         else {
             queryString = 'SELECT Id, Name,
                 Location__Longitude__s,
                 Location__Latitude__s, '
                 +'(SELECT Id, Name, Warehouse__r.Name,
                 Quantity__c
                 FROM Merchandise__r WHERE Name
                 like \'%'+merchName+'%\') '
                 +'FROM Warehouse__c limit 25';

         }
...
```

Visualforce remoting is ideal for mobile developers on the Salesforce1 Platform because it simplifies the direct server-side access to Salesforce objects and allows you to use Apex tools such as SOQL, Apex methods, and so on for rapid platform development. And, you don't have to deal with view state, which makes pages perform better.

The Apex controller also has an `insertQuickOrder` method that creates a feed item about the new order in the account feed as shown in this code snippet. The feed item is a link post that links to the invoice.

```
FeedItem post = new FeedItem();
post.ParentId = aId;
post.Body = UserInfo.getName() + ' just created a quick order';
post.type = 'LinkPost';
post.LinkUrl = '/' + li.Invoice__c;
post.Title = li.Merchandise__r.Name + ': ' + li.quantity__c;
insert post;
```

Step 3: Examine the `QuickOrderPage`

Now take a look at `QuickOrderPage`.

This page calls the controller with the user input and then displays the merchandise and warehouse information to the user. If the user wants to create an order, this page also calls the controller to create the order associated with the customer account and add a line item. At the beginning of the page, the code also does some styling of the page using the Salesforce mobile design templates.

```
<apex:page standardController="Account"
    extensions="QuickOrderController" standardStylesheets="false"
    showheader="false" sidebar="false">

    <!--Include Stylsheets for the Mobile look and feel -->
    <apex:stylesheet value="{!URLFOR(
        $Resource.Mobile_Design_Templates,
        'Mobile-Design-Templates-master/
            common/css/app.min.css')}"/>
    <apex:includeScript value="{!URLFOR(
        $Resource.Mobile_Design_Templates,
        'Mobile-Design-Templates-master/common/js/
            jQuery2.0.2.min.js')}"/>
    <apex:includeScript value="{!URLFOR(
        $Resource.Mobile_Design_Templates,
        'Mobile-Design-Templates-master/common/js/
            jquery.touchwipe.min.js')}"/>
    <apex:includeScript value="{!URLFOR(
        $Resource.Mobile_Design_Templates,
        'Mobile-Design-Templates-master/common/
            js/main.min.js')}"/>

      <style>
          /* Default S1 color styles */
          .list-view-header, .data-capture-buttons a {
                background: -webkit-linear-gradient(
                    #2a93d5,#107abb);
                background: linear-gradient(#2a93d5,#107abb);
                box-shadow: 0 1px 3px rgba(0,0,0,.2),
                    inset 0 1px 0 rgba(255,255,255,.21);
                color: white;
                font-weight: bold;
          }

          #resultPage, #searchPage {
              padding-bottom: 50px;
          }
      </style>
```

The `QuickOrderPage` also calls the Force.com Canvas SDK to enable the publisher Submit button and close the publisher window.

First, it includes a reference to the SDK:

```
<!-- Required so the publisher can be used to submit the action -->
<script type='text/javascript'
src='/canvas/sdk/js/publisher.js'></script>
```

Then it calls the setValidForSubmit method to enable the publisher Submit button:

```
//This method will activate the publish button so the form can be submitted
Sfdc.canvas.publisher.publish({
    name: "publisher.setValidForSubmit",
    payload:"true"});
```

After the `setValidForSubmit` is called and the user clicks Submit, this subscribe method fires. This method invokes the final JavaScript function which uses JavaScript remoting to insert the line item (thus completing the quick order) and then post a feed item to the account:

```
<script type='text/javascript'>
    Sfdc.canvas.publisher.subscribe({name: "publisher.post",
        onData:function(e) {
    //This subscribe fires when the user taps Submit in the publisher
    insertQuickOrder();
    }});
</script>
```

Finally, after the callback from the remoting method returns successfully, this method closes the publisher window:

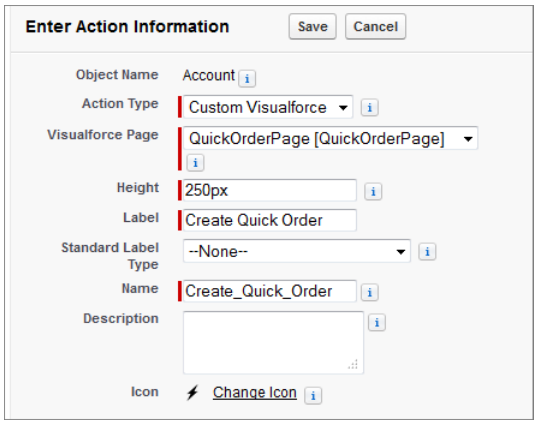
```
// Success - close the publisher and refresh the feed
Sfdc.canvas.publisher.publish({name: "publisher.close",
    payload:{ refresh:"true"}});
```

Step 4: Create a Visualforce Custom Action

The Visualforce code for this page uses the location of the current customer account to find warehouses located within the specified distance (in miles) that also have a particular part in stock.

1. In the Salesforce application, from Setup, click **Customize** > **Accounts** > **Buttons, Links and Actions**.
2. Click **New Action**.
3. In the Action Type drop-down list, select **Custom Visualforce**.
4. In the Visualforce Page drop-down list, select **QuickOrderPage**.
5. In the Label field, enter **Create Quick Order**.

   Remember this label is used for mobile, so you don't want labels much longer than this.


Enter Action Information

Object Name	Account
Action Type	Custom Visualforce
Visualforce Page	QuickOrderPage [QuickOrderPage]
Height	250px
Label	Create Quick Order
Standard Label Type	--None--
Name	Create_Quick_Order
Description	
Icon	Change Icon

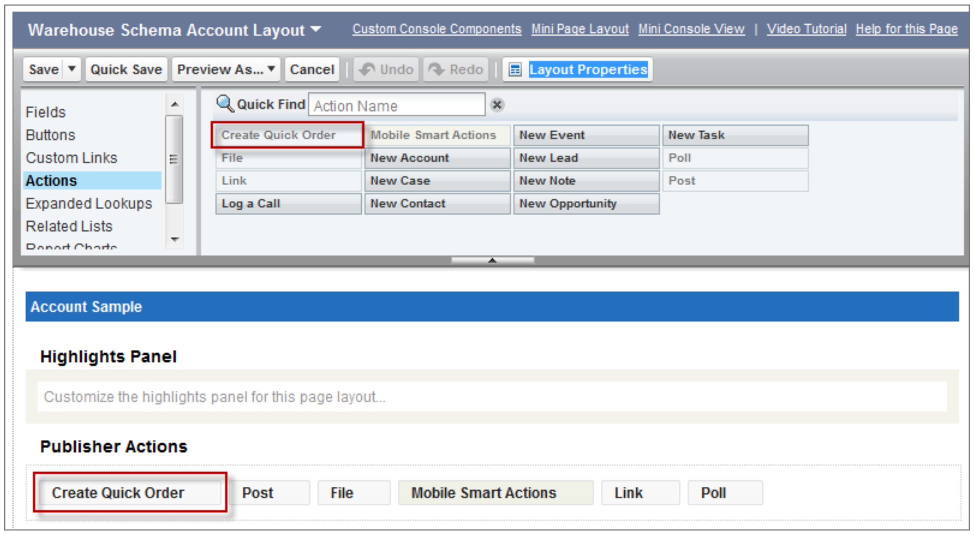
6. Click **Save**.

Now that you created the custom action, you can add the action to the Account page layout.

Step 5: Add the Custom Action to the Layout

Add the action and customize the layout.

1. From Setup, click **Customize** > **Accounts** > **Page Layouts**.
2. Click **Edit** next to the Warehouse Schema Account Layout.
3. Click the Actions category in the palette.
4. In the Publisher Actions section, click override the global publisher layout.
5. Drag the **Create Quick Order** element into the Publisher Actions section so that it appears as the first element on the left.



6. Click **Save**.

Step 6: Assign the Layout to Your Profile

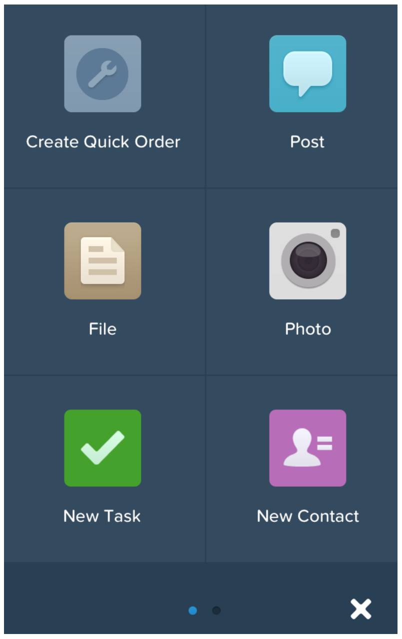
In order for users to see these changes, you need to assign the page layout to a user's profile. You're logged in as the System Administrator, so it's easiest to assign the layout to that profile.

1. On the Account Page Layout page click **Page Layout Assignment**.
2. Scroll down and click **Edit Assignment**.
3. Click **System Administrator** (your current profile).
4. In the Page Layout to Use list, choose **Warehouse Schema Account Layout** and then **Save**.

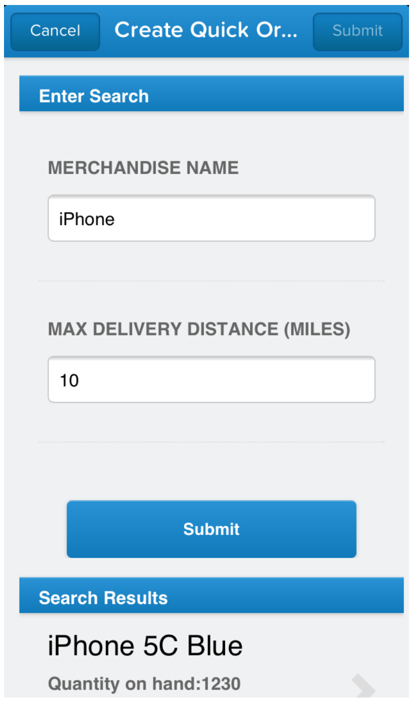
Step 7: Test the Custom Action

Test the custom action in the Salesforce1 mobile app.

1. Open the Salesforce1 app on your mobile device.
2. Tap ☰ to access the navigation menu.
3. Tap **Accounts** and navigate to an account.
4. Tap to + access the publisher.
5. Tap **Create Quick Order**.



6. In the `Merchandise Name` field, enter the name of an item, such as `iPhone 5`.
7. In the `Max Delivery Distance` (miles) field, enter `10` and then tap **Submit**.



8. In the Search Results section at the bottom, you'll see a list of merchandise that matches what you searched for. The list shows parts contained in warehouses within 10 miles of the current account. Tap **iPhone 5S Gold**.
9. In the `Quantity` field, enter `1`. After identifying nearby warehouses that contain the part you're looking for, this screen lets you enter a quantity and creates an order for the part. The order is associated with the current account.

Tip: If you want to return to the search screen, be sure to use the **Back** button. If you tap **Cancel**, an invoice with no line items is created.

10. Tap **Submit**. The order is created.
11. Swipe to the third page indicator, and you'll see the **Invoices** related list.
12. Tap the **Invoices** related list, and you'll see the new invoice.
13. You can tap the invoice and then see the line item that was created for the iPhone 5S Gold part.

Success! You've gone through the entire process of creating a custom action, which enables a mobile user to quickly search for a part in a warehouse and create an order on a specific account.

Lesson 3: Use a Visualforce Page as Mobile Card

This tutorial shows you how to add a Visualforce page as a mobile card. Once again, the page is already in the package you installed, so you can get right to the fun stuff.

Step 1: Examine the AccountLocation Page

As you've done before, open the `AccountLocation` page and take a look at the code. It doesn't do much more than call the Google API and send coordinates, which is all it really needs to do.

```
<apex:page standardController="Account" showHeader="false" standardStylesheets="false">
```

```
<apex:image url="https://maps.googleapis.com/maps/api/staticmap?markers=
   {! account.Location__Latitude__s },
   {! account.Location__Longitude__s }
   &zoom=13&size=320x240&sensor=false"/>

</apex:page>
```

Note: The content of mobile cards can't be scrolled, so make sure it fits in the space you provide it. You control the height of the mobile card by setting the height in pixels in the page layout editor. The mobile card area uses exactly that height, even if the mobile card's content is shorter. In that case, the extra area is blank. If the card's content is taller, the content is clipped.

Step 2: Add a Mobile Card to the Page Layout

Mobile cards are added to an object's page layout.

1. In the Salesfoce site, click **Setup** > **Customize** > **Account** > **Page Layouts**.
2. Click the **Edit** link next to Warehouse Schema Account Layout.
3. In the editor palette, scroll down and click **Visualforce Pages**.
4. In the Preview area, scroll down to the Mobile Cards section.
5. Drag **AccountLocation** from the palette into the Mobile Cards area.
6. Click **Save** in the Page Layout Editor.

Step 4: Test the Mobile Card

Try out the mobile card on your phone.

1. On your mobile device, tap ☰ and then **Accounts**.
2. Tap an account to get to the detail view.
3. Swipe left to get to the record related information page, and you should see your mobile card at the top.

Note: Mobile cards display before other related information, so don't use too many mobile cards or users will have to scroll a lot.

Integrating Your Web Applications in Salesforce1 with Force.com Canvas

Force.com Canvas lets you easily integrate a third-party Web application in Salesforce1.

The Force.com Canvas SDK is an open-source suite of JavaScript libraries that provide simple methods that use existing Salesforce1 APIs (REST API, SOAP API, Chatter REST API) so you can build a seamless end-user experience for your mobile users.

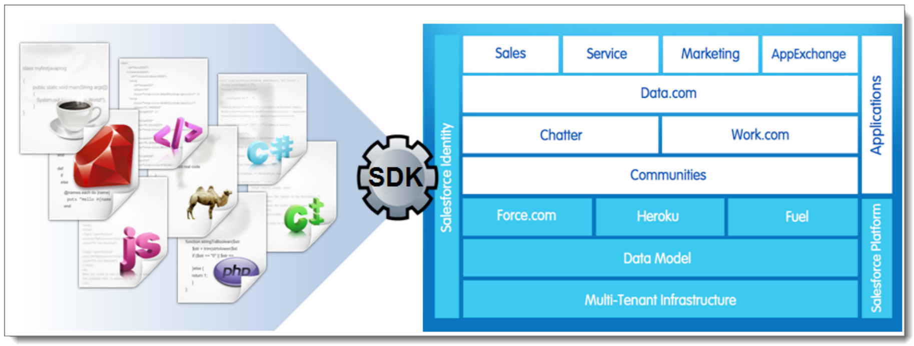
About Force.com Canvas

Force.com Canvas features include:

- Language Independence—You develop it and we display it. With Force.com Canvas you can develop in the language of your choice and easily surface the app inside of the Salesforce1 app.

  The third-party app that you want to expose as a canvas app can be written in any language. The only requirement is that the app has a secure URL (HTTPS).

- JavaScript SDK—Lightweight and easy-to-use JavaScript libraries allow your app to authenticate and communicate without having to deal with cross-domain network issues. This gives your users a single command center to drive all their apps.
- Simplified Authentication—Allows you to authenticate by using OAuth 2.0 or a signed request. This means that your app can connect to Salesforce at the data layer while remaining seamless for users.
- App Registration and Management—Developers can create apps and allow their customers to install them with a single click. Administrators can easily install apps from developers, and quickly manage who in their organizations can use the app.



Extending Salesforce1 with Canvas Custom Actions

Canvas custom actions appear in the publisher in the Salesforce1 app, and provide a way to make canvas apps available from the publisher.

In this chapter, we'll further extend the Acme Wireless organization by integrating a third-party app with the platform using Force.com Canvas. Acme Wireless has a Web application running on Heroku called Shipify. They use it to process orders. We'll copy the Shipify application that runs on Heroku and make it available as a canvas custom action from the publisher.

In this scenario, warehouse workers with mobile devices can bring up a list of open customer orders. They can select an order and then process it for shipping. After the order is processed, Shipify sets the order status in Salesforce and posts a feed item to the associated customer account. You'll see how easy it is to integrate a Web application and enable it to communicate with Salesforce. This creates a seamless experience for your mobile users.

Try It Out: Clone the Shipify Web Application

We'll start the process of creating a canvas custom action by cloning the Shipify Web application.

In addition to the prerequisites for this book listed in Development Prerequisites, you'll also need:

- "Customize Application" and "Modify All Data" user permissions. If you're an administrator, you most likely already have these permissions. Otherwise, you need to add them so that you can see the Canvas App Previewer and create canvas apps.
- Git installed. Go here to install and configure Git: `https://help.github.com/articles/set-up-git`.
- A GitHub account to clone the code example. Go here to set up a GitHub account: `https://github.com/plans`.
- A Heroku account because our Web application will run on Heroku. Go here to create a Heroku account: `https://api.heroku.com/signup`.
- Heroku Toolbelt to manage the Heroku app from the command line. Go here to download and install Heroku Toolbelt: `https://toolbelt.heroku.com`.

Shipify is a Web application that Acme Wireless uses to track the status of customer orders, customer shipments, and deliveries. The warehouse workers use this application to check for outstanding orders, fulfill those orders, and then update the shipping status. The Shipify Web application contains some order processing logic, but it's not a full order processing application. Its purpose is to demonstrate how you can integrate your Web applications with the Salesforce1 app.

Shipify is a Node.js application that runs on Heroku. Each running instance of the application must reference the consumer secret for the connected app that you create in your organization. Therefore, you'll need to have your own instance of Shipify on Heroku that you can then add as a canvas app. In this step, we'll clone the application, which is the first step in the process.

1. Open a command window and navigate to the directory where you want to download Shipify. When you clone the application, it will create a directory called `Shipify-Node-App` from wherever you run the clone command.
   - From a computer running Windows, open a command window by clicking **Start** > **Run...** and entering in `cmd`.
   - From a computer running Mac OS, open a command window by pressing Command + Space and entering `terminal`.
2. Enter this command: `git clone https://github.com/forcedotcom/Shipify-Node-App`
3. Navigate to the `Shipify-Node-App` directory and enter this command to initialize the repository: `git init`
4. Enter this command to get the files ready to commit: `git add –A`
5. Enter this command to commit the changes along with a comment: `git commit –m '`***`MyChangeComments`***`'`

   If you're working from a Windows computer, you may need to replace the single quotes with double quotes (").
6. Enter this command to log in to Heroku: `heroku login`

   When prompted, enter your email and password.
7. Enter this command to create a new Heroku app: `heroku apps:create`

   Confirmation that the app was created looks like this:

```
Creating deep-samurai-7923... done, stack is cedar
http://deep-samurai-7923.herokuapp.com/ | git@heroku.com:deep-samurai-7923.git
Git remote heroku added
```

8. Copy the URL of the Heroku app because we'll use it in the next task. In this example, the URL is `http://deep-samurai-7923.herokuapp.com`, but you'll want to copy the URL of your own Heroku app.

We've created the Shipify Web application in Heroku, but it won't work yet because we still need to deploy it, which we'll do in a later step.

The next step is to add it as a canvas app in Salesforce.

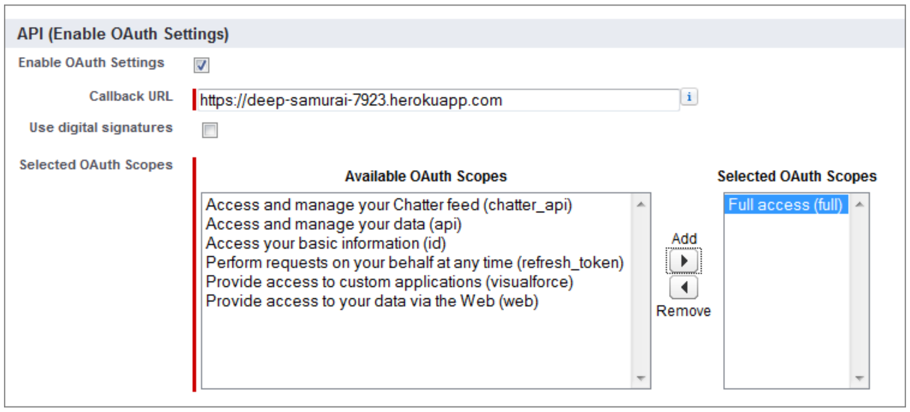
Create the Shipify Canvas App

In this step, we'll expose the Shipify Web application as a canvas app.

You'll need user permissions "Customize Application" and "Modify All Data" to create a canvas app.

1. In the Salesforce application, from Setup, click **Create** > **Apps**.
2. In the Connected Apps related list, click **New**.
3. In the `Connected App Name` field, enter `Shipify`.
4. In the `Contact Email` field, enter your email address.
5. In the API (Enable OAuth Settings) section, select `Enable OAuth Settings`.
6. In the `Callback URL` field, paste the URL of the Heroku app you just created and change the protocol to `https`. For example, your final URL should look something like `https://deep-samurai-7923.herokuapp.com`.
7. In the `Selected OAuth Scopes` field, select Full access and click **Add**.

   As a best practice, you'll want to keep the OAuth scopes as limited as needed for your canvas app functionality.



API (Enable OAuth Settings)

Enable OAuth Settings ✓

Callback URL https://deep-samurai-7923.herokuapp.com

Use digital signatures

Selected OAuth Scopes

Available OAuth Scopes:
- Access and manage your Chatter feed (chatter_api)
- Access and manage your data (api)
- Access your basic information (id)
- Perform requests on your behalf at any time (refresh_token)
- Provide access to custom applications (visualforce)
- Provide access to your data via the Web (web)

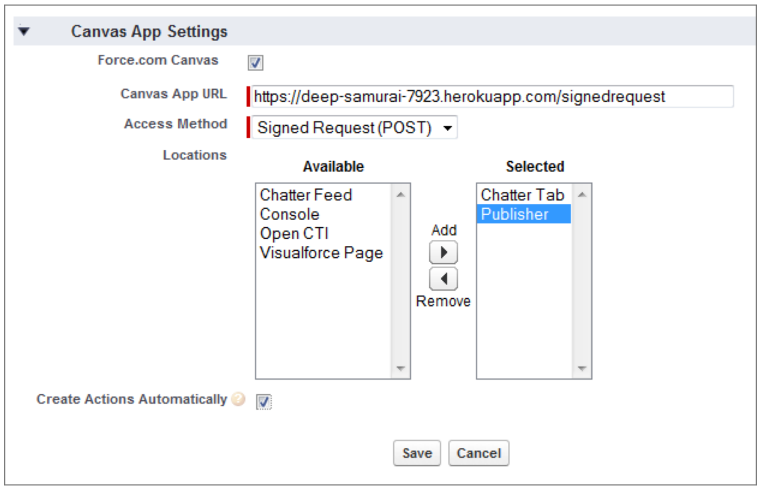
Selected OAuth Scopes:
- Full access (full)

8. In the Canvas App Settings section, select `Force.com Canvas`.
9. In the `Canvas App URL` field, enter the same URL you entered in the `Callback URL` field with `/signedrequest` appended to it. For example, the URL might look like `https://deep-samurai-7923.herokuapp.com/signedrequest`.
10. In the Access Method drop-down list, select Signed Request (POST).
11. In the `Locations` field, select Chatter Tab and Publisher and click **Add**.

    For a canvas app to appear in the publisher, you only need to specify a location of Publisher. In this scenario, we also chose Chatter Tab to be able to easily test the canvas app by looking at the Chatter tab.

12. Select `Create Actions Automatically`.

    Selecting this field creates a global custom action for the canvas app.


Canvas App Settings
Force.com Canvas ✓
Canvas App URL: https://deep-samurai-7923.herokuapp.com/signedrequest
Access Method: Signed Request (POST)
Locations
Available: Chatter Feed, Console, Open CTI, Visualforce Page
Selected: Chatter Tab, Publisher
Create Actions Automatically ✓

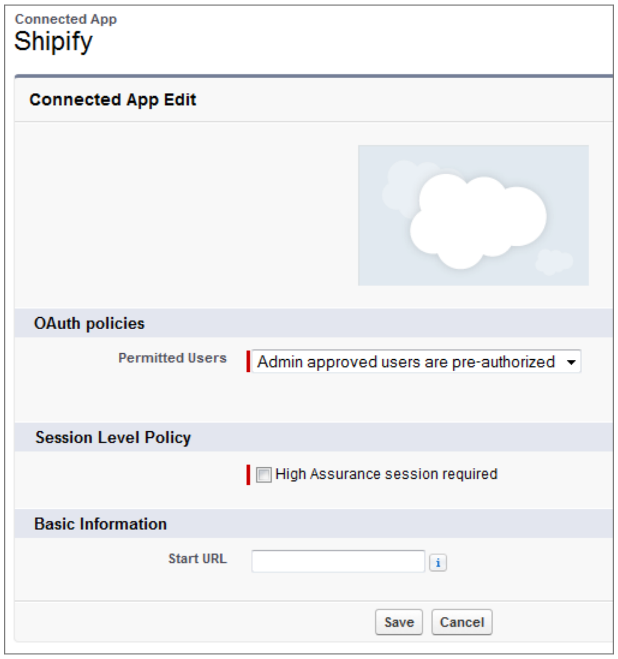
13. Click **Save**. After the canvas app is saved, the detail page appears.

We added the canvas app, and now we'll specify who can access it.

Configure Who Can Access the Shipify Canvas App

You've created the canvas app, but no one can see it until you configure user access.

1. From Setup, click **Manage Apps** > **Connected Apps**.
2. Click the Shipify app, and then click **Edit**.
3. In the Permitted Users drop-down list, select Admin approved users are pre-authorized. Click **OK** on the pop-up message that appears.


Connected App
Shipify
Connected App Edit
OAuth policies
Permitted Users: Admin approved users are pre-authorized
Session Level Policy
☐ High Assurance session required
Basic Information
Start URL

4. Click **Save**.

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

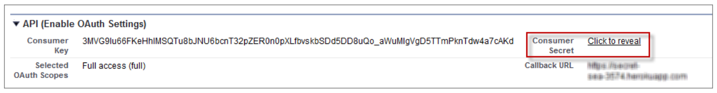
5. On the Connected App Detail page, in the Profiles related list, click **Manage Profiles**.
6. Select the `System Administrator` profile and click **Save**.

That's it! The next step is to set up some environment variables on Heroku.

Configure the Heroku Environment Variables

Now that we've created the canvas app, we need to set up an environment variable for the consumer secret.

1. From Setup, click **Create** > **Apps**.
2. In the Connected Apps related list, click **Shipify**.
3. Next to the `Consumer Secret` field, click the link **Click to reveal**.



▼ API (Enable OAuth Settings)

Consumer Key	3MVG9lu66FKeHhlMSQTu8bJNU6bcnT32pZER0n0pXLfbvskbSDd5DD8uQo_aWuMlgVgD5TTmPknTdw4a7cAKd	Consumer Secret	Click to reveal
Selected OAuth Scopes	Full access (full)	Callback URL	[redacted]

4. Copy the consumer secret.
5. Open a command window, navigate to the `Shipify-Node-App` directory, and enter this command to create the environment variable: `heroku config:add APP_SECRET='`***`Your_Consumer_Secret`***`'`

   If you're working from a Windows computer, you may need to replace the single quotes with double quotes (").
6. Enter this command: `heroku config:add RUNNING_ON_HEROKU='true'`

   This specifies that the application is running on Heroku. This is helpful for testing so you don't need to re-deploy every time you make a change. If you're working from a Windows computer, you may need to replace the single quotes with double quotes (").
7. Enter this command to deploy the app to Heroku: `git push heroku master`

   If the process completes successfully, you'll see something like this:

```
-----> Compiled slug size: 11.2MB
-----> Launching... done, v6
       http://deep-samurai-7923.herokuapp.com deployed to Heroku

To git@heroku.com:deep-samurai-7923.git
 * [new branch]      master -> master
```

   If you receive a "permission denied" error message, you may need to set up your SSH key and add it to Heroku. See `https://devcenter.heroku.com/articles/keys`.

You can quickly test that the canvas app is working in the full Salesforce site by clicking the Chatter tab. Click the **Shipify** link on the left, and you'll see the Heroku application appear right in Salesforce and display a list of open orders, including the one you created in the previous step. Now we'll add the canvas custom action to the publisher layout so our users can see it.

Add the Action to the Global Publisher Layout

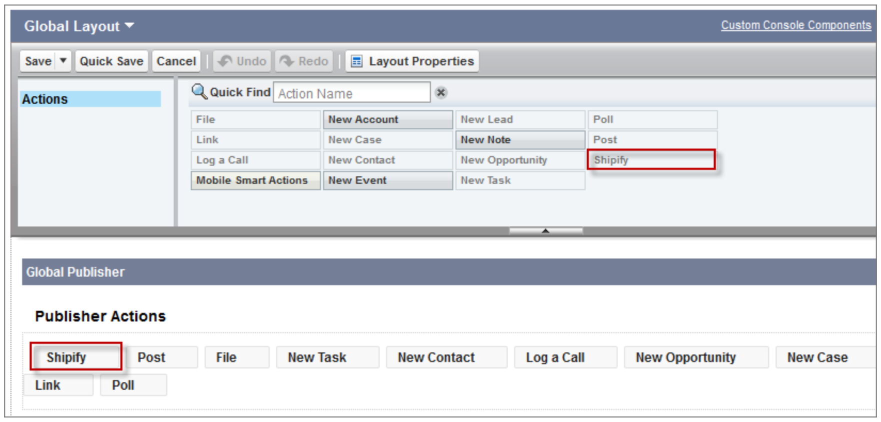
When we created the canvas app, we opted to automatically create the action. In this step, we'll add the action to the global publisher layout so it shows up in the publisher in the Salesforce1 app.

1. In the Salesforce application, from Setup, click **Create** > **Global Actions** > **Publisher Layouts**.
2. Click **Edit** next to the Global Layout.

   This is the default layout for the publisher.
3. Drag the Shipify element into the Publisher Actions section so that it appears as the first element on the left.

   Our warehouse workers are going to use this custom action a lot, so we're putting it at the start of the list so it'll be one of the first actions they see when they open the publisher.

   Note: The canvas custom action was created automatically, so it used the name of the canvas app for the name of the action (in this case, Shipify). In a production scenario, you'll want to change the action name to represent what the user can do. For example, we could change the name of this action to Ship Orders.



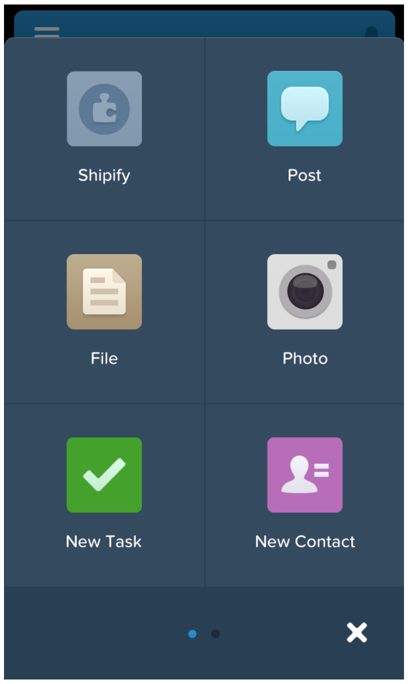
4. Click **Save**.

Now that we've created the custom action and added it to the publisher layout, we're ready to see it in action (no pun intended).

Test Out the Canvas Custom Action

Now you'll play the part of an Acme Wireless warehouse worker processing orders, and we'll test out the custom action in the Salesforce1 app.

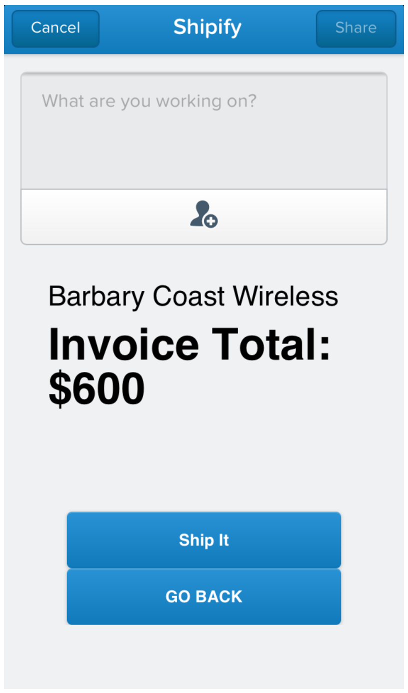
1. On your mobile device, tap + to access the publisher.



2. Tap **Shipify**.

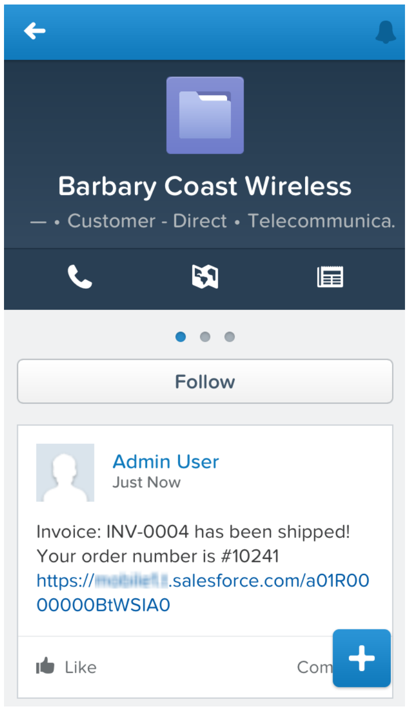
   A list of all open invoices along with the customer account and warehouse information appears.

3. Tap one of the invoices, and a screen appears that shows the total amount of the invoice and a **Ship It** button.



4. Tap **Ship It**.

   This kicks off order processing in the Shipify Web application. In addition, Shipify sets the invoice status to Closed and creates a feed item for the account that shows that the invoice has shipped along with the related order number and a link to the invoice.



Barbary Coast Wireless
— • Customer - Direct • Telecommunica.

Admin User
Just Now

Invoice: INV-0004 has been shipped!
Your order number is #10241
https://[redacted].salesforce.com/a01R0000000BtWSIA0

You did it! You've gone through the entire process of integrating a Web application in the Salesforce1 app using Force.com Canvas and canvas custom actions. Now warehouse workers can find customer orders, ship them out, and then update the order and the account feed.

For more information about development guidelines for canvas apps in the publisher, see Canvas Apps in the Publisher.

Tell Me More: Get Context in your Canvas App

The Force.com Canvas SDK provides calls and objects that let you retrieve context information about the application and the current user from Salesforce.

Getting Context

When you authenticate your canvas app using signed request, you get the CanvasRequest object (which contains the Context object) as part of the POST to the canvas app URL. If you're authenticating using OAuth, or if you want to make a call to get context information, you can do so by making a JavaScript call. You can use this information to make subsequent calls for information and code your app so that it appears completely integrated with the Salesforce1 user interface.

The following code sample is an example of a JavaScript call to get context. This code creates a link with the text "Get Context" which then calls the `Sfdc.canvas.client.ctx` function.

```
<script>
    function callback(msg) {
       if (msg.status !== 200) {
          alert("Error: " + msg.status);
          return;
       }
       alert("Payload: ", msg.payload);
    }

    var ctxlink = Sfdc.canvas.byId("ctxlink");
    var client = Sfdc.canvas.oauth.client();
    ctxlink.onclick=function() {
       Sfdc.canvas.client.ctx(callback, client)};
    }
</script>

<a id="ctxlink" href="#">Get Context</a>
```

Context Objects

When you make a call to get context in your canvas app, you get a CanvasRequest object back in the response. This object contains all the contextual information about the application and the user. The context objects include:

Object	Description
`CanvasRequest`	Returns the Context and Client objects.
`Client`	Returns context information about the client app.
`Context`	Returns information about the consumer of the canvas app. Contains the Application, Environment, Links, Organization, and User objects.
`Application`	Returns information about the canvas app, such as version, access method, URL, and so on.
`Environment`	Returns information about the environment, such as location, UI theme, and so on.
`Links`	Returns links, such as the metadata URL, user URL, Chatter groups URL, and so on. You can use these links to make calls into Salesforce from your app.
`Organization`	Returns information about the organization, such as name, ID, currency code, and so on.
`User`	Returns information about the currently logged-in user, such as locale, name, user ID, email, and so on.

This code snippet shows an example of the CanvasRequest object:

```
{
    "context":
    {
        "application":
        {
            "applicationId":"06Px000000003ed",
            "authType":"SIGNED_REQUEST",
            "canvasUrl":"http://instance.salesforce.com:8080
                /canvas_app_path/canvas_app.jsp",
            "developerName":"my_java_app",
            "name":"My Java App",
            "namespace":"org_namespace",
            "referenceId":"09HD00000000AUM",
            "version":"1.0.0"
        },
        "user":
        {
            "accessibilityModeEnabled":false,
            "currencyISOCode":"USD",
            "email":"admin@6457617734813492.com",
            "firstName":"Sean",
            "fullName":"Sean Forbes",
            "isDefaultNetwork":false,
            "language":"en_US",
            "lastName":"Forbes",
            "locale":"en_US",
            "networkId":"0DBxx000000001r",
            "profileId":"00ex0000000jzpt",
            "profilePhotoUrl":"/profilephoto/005/F",
            "profileThumbnailUrl":"/profilephoto/005/T",
            "roleId":null,
            "siteUrl":"https://mydomain.force.com/",
            "siteUrlPrefix":"/mycommunity",
            "timeZone":"America/Los_Angeles",
            "userId":"005x0000001SyyEAAS",
            "userName":"admin@6457617734813492.com",
            "userType":"STANDARD"
        },
        "environment":
        {
            "parameters":
            {
                "complex":
                {
                    "key1":"value1",
                    "key2":"value2"
                },
                "integer":10,
                "simple":"This is a simple string.",
                "boolean":true
            },
            "dimensions":
            {
                "height": "900px",
                "width": "800px",
                "maxWidth":"1000px",
                "maxHeight":"2000px"
            },
            "displayLocation":"Chatter",
            "locationUrl": "http://www.salesforce.com
                /some/path/index.html",
            "uiTheme":"Theme3",
            "version":
            {
                "api":"31.0",
                "season":"WINTER"
            },
        },
        "organization":
        {
            "currencyIsoCode":"USD",
            "multicurrencyEnabled":true,
            "name":"Edge Communications",
            "namespacePrefix":"org_namespace",
            "organizationId":"00Dx00000001hxyEAA"
        },
        "links":
        {
            "chatterFeedItemsUrl":"/services/data/v31.0/
                chatter/feed-items",
            "chatterFeedsUrl":"/services/data/v31.0/
                chatter/feeds",
            "chatterGroupsUrl":"/services/data/v31.0/
                chatter/groups",
            "chatterUsersUrl":"/services/data/v31.0/
                chatter/users",
            "enterpriseUrl":"/services/Soap/c/31.0/
                00Dx00000001hxy",
            "loginUrl":"http://login.salesforce.com",
            "metadataUrl":"/services/Soap/m/31.0/00Dx00000001hxy",
            "partnerUrl":"/services/Soap/u/31.0/00Dx00000001hxy",
            "queryUrl":"/services/data/v31.0/query/",
            "recentItemsUrl":"/services/data/v31.0/recent/",
            "restUrl":"/services/data/v31.0/",
            "searchUrl":"/services/data/v31.0/search/",
            "sobjectUrl":"/services/data/v31.0/sobjects/",
            "userUrl":"/005x0000001SyyEAAS"
        }
    },
    "client":
    {
        "instanceId":"06Px000000002JZ",
        "instanceUrl":"http://instance.salesforce.com:
            8080",
        "oauthToken":"00Dx0000X00Or4J!ARQAKowP65p8FDHkvk.Uq5...",
        "targetOrigin":"http://instance.salesforce.com:
            8080"
    },
"algorithm":"HMACSHA256",
"userId":"005x0000001SyyEAAS",
"issuedAt":null
}
```

For more information about context objects and the Force.com Canvas SDK, see the *Force.com Canvas Developer's Guide*.

Extending Salesforce1 with Canvas Apps in the Feed

Force.com Canvas enables you to add even more functionality to the feed by exposing your canvas apps as feed items in the Salesforce1 app.

You can use this functionality to:

- Post to the feed from a canvas app exposed from a canvas custom action in the publisher or post to the feed through the Chatter API.
- Display a canvas app directly inside a feed item.

In this chapter, we'll further extend the Acme Wireless organization by integrating a third-party Web app in the feed using Force.com Canvas.

Acme Wireless has a Web application that runs on Heroku called DeliveryTrakr that they use to process deliveries of customer orders. We'll copy the DeliveryTrackr application that runs on Heroku and make it available as a global action from the publisher.

In this scenario, warehouse workers equipped with mobile devices can bring up a list of deliveries. When warehouse workers access the app from the publisher, they can use it to create feed posts of different types. The app can create a text post which is a feed item that contains information about a delivery, a link post which is a feed item that contains a link to the DeliveryTrakr app, or a canvas post which is a feed item that contains a link to another canvas app. The user can click on this link to access a canvas app that lets them approve or deny a delivery.

Try It Out: Clone the DeliveryTrakr Web Application

We'll start the process of integrating a canvas app in the feed by cloning the DeliveryTrakr Web application.

In addition to the prerequisites for this book listed in Development Prerequisites, you'll also need:

- "Customize Application" and "Modify All Data" user permissions. If you're an administrator, you most likely already have these permissions. Otherwise, you need to add them so that you can see the Canvas App Previewer and create canvas apps.
- Git installed. Go here to install and configure Git: `https://help.github.com/articles/set-up-git`.
- A GitHub account to clone the code example. Go here to set up a GitHub account: `https://github.com/plans`.
- A Heroku account because our Web application will run on Heroku. Go here to create a Heroku account: `https://api.heroku.com/signup`.
- Heroku Toolbelt to manage the Heroku app from the command line. Go here to download and install Heroku Toolbelt: `https://toolbelt.heroku.com`.

The steps to create and run a canvas app are the same, regardless of the app's functionality. So the steps in this chapter are similar to those of the previous chapter, Extending Salesforce1 with Canvas Custom Actions on page 41.

DeliveryTrakr is a Web application that Acme Wireless uses to track deliveries of customer orders. The warehouse workers use this application to check for orders that have been delivered and post delivery information to the feed. The DeliveryTrakr Web application contains some delivery processing logic, but it's not a full application. Its purpose is to demonstrate how you can integrate your Web applications with the Salesforce1 app.

DeliveryTrakr is a Java application that runs on Heroku. Each running instance of the application must reference the consumer secret for the connected app that you create in your organization. Therefore, you'll need to have your own instance of DeliveryTrakr on Heroku that you can then add as a canvas app. In this step, we'll clone the application, which is the first step in the process.

1. Open a command window and navigate to the directory where you want to download DeliveryTrakr. When you clone the application, it will create a directory called `Delivery-Tracker-Java-App` from wherever you run the clone command.
    - From a computer running Windows, open a command window by clicking **Start** > **Run...** and entering in `cmd`.
    - From a computer running Mac OS, open a command window by pressing Command + Space and entering `terminal`.
2. Enter the command: `git clone https://github.com/forcedotcom/Delivery-Tracker-Java-App`
3. Navigate to the `Delivery-Tracker-Java-App` directory and enter this command to initialize the repository: `git init`
4. Enter this command to get the files ready to commit: `git add —A`
5. Enter this command to commit the changes along with a comment: `git commit —m '`***`MyChangeComments`***`'`

    If you're working from a Windows computer, you may need to replace the single quotes with double quotes (").
6. Enter this command to log in to Heroku: `heroku login`

    When prompted, enter your email and password.
7. Enter this command to create a new Heroku app: `heroku apps:create`

    Confirmation that the app was created looks like this:

    ```
    Creating deep-samurai-7923... done, stack is cedar
    http://deep-samurai-7923.herokuapp.com/ | git@heroku.com:deep-samurai-7923.git
    Git remote heroku added
    ```

8. Copy the URL of the Heroku app because we'll use it in the next task. In this example, the URL is `http://deep-samurai-7923.herokuapp.com`, but you'll want to copy the URL of your own Heroku app.

    We've created the DeliveryTrakr Web application in Heroku, but it won't work yet because we still need to deploy it, which we'll do in a later step.

The next step is to add it as a canvas app in Salesforce.

Create the DeliveryTrakr Canvas App

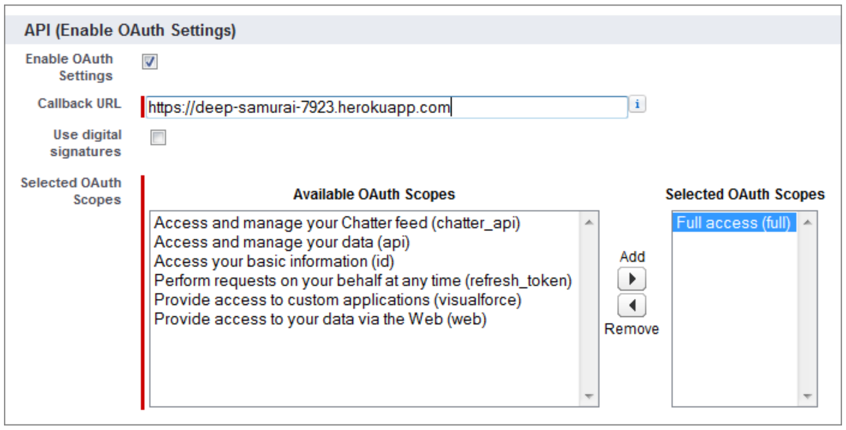
In this step, we'll expose the DeliveryTrakr Web application as a canvas app.

You'll need user permissions "Customize Application" and "Modify All Data" to create a canvas app.

1. In the Salesforce application, from Setup, click **Create** > **Apps**.
2. In the Connected Apps related list, click **New**.
3. In the `Connected App Name` field, enter `DeliveryTrakr`.
4. In the `Contact Email` field, enter your email address.
5. In the API (Enable OAuth Settings) section, select `Enable OAuth Settings`.

6. In the `Callback URL` field, paste the URL of the Heroku app you just created and change the protocol to `https`. For example, your final URL might look something like `https://deep-samurai-7923.herokuapp.com`.
7. In the `Selected OAuth Scopes` field, select Full access and click **Add**.

   As a best practice, you'll want to keep the OAuth scopes as limited as needed for your canvas app functionality.



API (Enable OAuth Settings)

Enable OAuth Settings: ✓

Callback URL: https://deep-samurai-7923.herokuapp.com

Use digital signatures: ☐

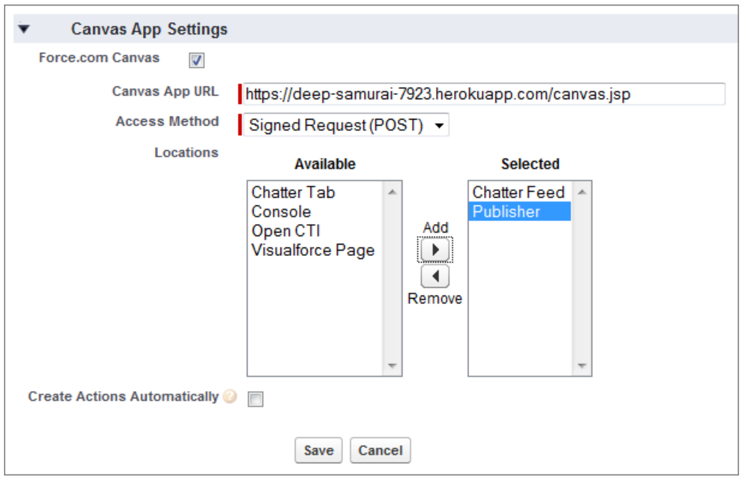
Selected OAuth Scopes

Available OAuth Scopes:
Access and manage your Chatter feed (chatter_api)
Access and manage your data (api)
Access your basic information (id)
Perform requests on your behalf at any time (refresh_token)
Provide access to custom applications (visualforce)
Provide access to your data via the Web (web)

Selected OAuth Scopes:
Full access (full)

8. In the Canvas App Settings section, select `Force.com Canvas`.
9. In the `Canvas App URL` field, enter the same URL you entered in the `Callback URL` field with `/canvas.jsp` appended to it. For example, the URL might look like `https://deep-samurai-7923.herokuapp.com/canvas.jsp`.
10. In the Access Method drop-down list, select Signed Request (POST).
11. In the `Locations` field, select Chatter Feed and Publisher and click **Add**.

    We selected these values because the canvas app appears in the publisher as well as in the feed.



Canvas App Settings

Force.com Canvas: ✓

Canvas App URL: https://deep-samurai-7923.herokuapp.com/canvas.jsp

Access Method: Signed Request (POST)

Locations

Available:
Chatter Tab
Console
Open CTI
Visualforce Page

Selected:
Chatter Feed
Publisher

Create Actions Automatically: ☐

12. Click **Save**. After the canvas app is saved, the detail page appears.

We added the canvas app, and now we'll specify who can access it.

Configure Who Can Access the DeliveryTrakr Canvas App

You've created the canvas app, but no one can see it until you configure user access.

1. From Setup, click **Manage Apps** > **Connected Apps**.

2. Click the DeliveryTrakr app, and then click **Edit**.
3. In the Permitted Users drop-down list, select Admin approved users are pre-authorized. Click **OK** on the pop-up message that appears.
4. Click **Save**.

   Now you'll define who can see your canvas app. This can be done using profiles and permission sets. In this example, we'll allow anyone with the System Administrator profile to access the app.
5. On the Connected App Detail page, in the Profiles related list, click **Manage Profiles**.
6. Select the `System Administrator` profile and click **Save**.

That's it! The next step is to set up some environment variables on Heroku.

Configure the Heroku Environment Variables

Now that we've created the canvas app, we need to set up an environment variable for the consumer secret.

1. From Setup, click **Create** > **Apps**.
2. In the Connected Apps related list, click **DeliveryTrakr**.
3. Next to the `Consumer Secret` field, click the link **Click to reveal**.
4. Copy the consumer secret.
5. Open a command window, navigate to the `Delivery-Tracker-Java-App` directory, and enter this command to create the environment variable: `heroku config:add APP_SECRET='`***`Your_Consumer_Secret`***`'`

   If you're working from a Windows computer, you may need to replace the single quotes with double quotes (").
6. Navigate to the `Delivery-Tracker-Java-App\src\main\webapp\scripts` directory.
7. Open `shipment.js` in an editor. In the `onGetPayload` function, replace YOUR_APP_URL with the URL of your DeliveryTrakr app on Heroku: `p.url = "https://[YOUR_APP_URL]/signed-request.jsp?shipment=" + shipment;`

   This code is located on or around line 231. Using our example, the line of code should look like:

   ```
   p.url = "https://deep-samurai-7923.herokuapp.com/
       signed-request.jsp? shipment=" + shipment;
   ```
8. Save the changes to `shipment.js` and navigate back to the `Delivery-Tracker-Java-App` directory.
9. Enter this command: `heroku config:add RUNNING_ON_HEROKU='true'`

   This specifies that the application is running on Heroku. This is helpful for testing so you don't need to re-deploy every time you make a change. If you're working from a Windows computer, you may need to replace the single quotes with double quotes (").
10. Enter this command to get the file ready to commit: `git add –A`
11. Enter this command to commit the changes along with a comment: `git commit –m '`***`MyChangeComments`***`'`

    If you're working from a Windows computer, you may need to replace the single quotes with double quotes (").
12. Enter this command to deploy the app to Heroku: `git push heroku master`

    If the process completes successfully, you'll see something like this:

    ```
    -----> Compiled slug size: 11.2MB
    -----> Launching... done, v6
           http://deep-samurai-7923.herokuapp.com deployed to Heroku

    To git@heroku.com:deep-samurai-7923.git
     * [new branch]      master -> master
    ```

If you receive a "permission denied" error message, you may need to set up your SSH key and add it to Heroku. See `https://devcenter.heroku.com/articles/keys`.

You can quickly test that the canvas app is working from Setup in the full Salesforce site by clicking **Canvas App Previewer**. Click the **DeliveryTrakr** link on the left, and you'll see the Heroku application running in the previewer and displaying a list of shipments and their delivery status. Now we'll create a global action for the canvas app so that your mobile users can starting using what you've created.

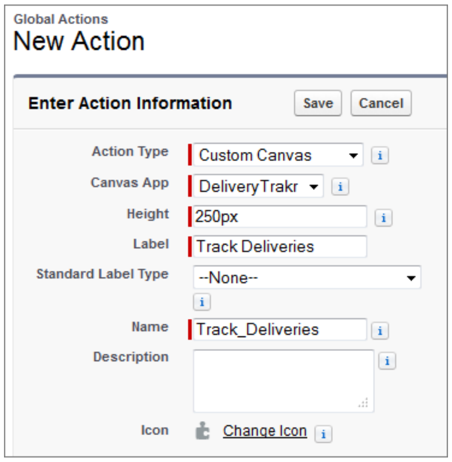
Create a Global Action

We'll create a global action for the DeliveryTrakr app so that we can then add it to the global publisher layout.

In the previous canvas example, we selected `Create Actions Automatically` when we created the canvas app in Salesforce. So the global action was created for us. In this scenario, we didn't select that field, so we'll step through creating the action manually.

1. If the Canvas Previewer is still open, click **Close**. From Setup, click **Create** > **Global Actions** > **Actions**.
2. Click **New Action**.
3. In the Action Type drop-down list, select Custom Canvas.
4. In the Canvas App drop-down list, select DeliveryTrakr.
5. In the `Label` field, enter `Track Deliveries`.

   This is the label that appears to the user in the publisher in the Salesforce1 app or in the publisher menu in the full Salesforce site.



Action Type	Custom Canvas
Canvas App	DeliveryTrakr
Height	250px
Label	Track Deliveries
Standard Label Type	--None--
Name	Track_Deliveries
Description	
Icon	Change Icon

6. Click **Save**.

Now that we've created the global action, we'll add the action to the global publisher layout. This ensures that the action appears in the publisher where mobile users can access it.

Add the Action to the Global Publisher Layout

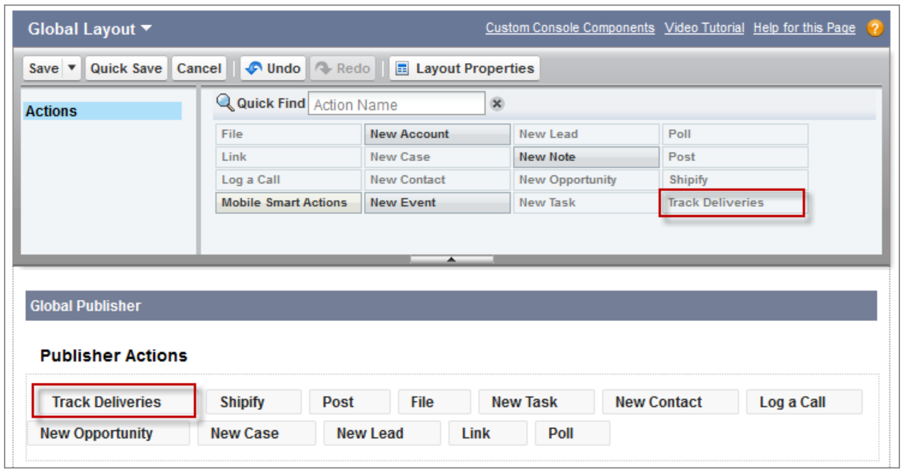
We just created the global action for the canvas app so now we'll add the action to the global publisher layout. Then it will appear in the publisher in the Salesforce1 app.

1. From Setup, click **Create** > **Global Actions** > **Publisher Layouts**.
2. Click **Edit** next to the Global Layout.

   This is the default layout for the publisher.

3. Drag the Track Deliveries element into the Publisher Actions section so that it appears as the first element on the left.

   Our warehouse workers are going to use this custom action a lot, so we're putting it at the start of the list so it'll be one of the first actions they see when they open the publisher.



4. Click **Save**.

Now that we've created the custom action and added it to the publisher layout, we're ready to see it in action.

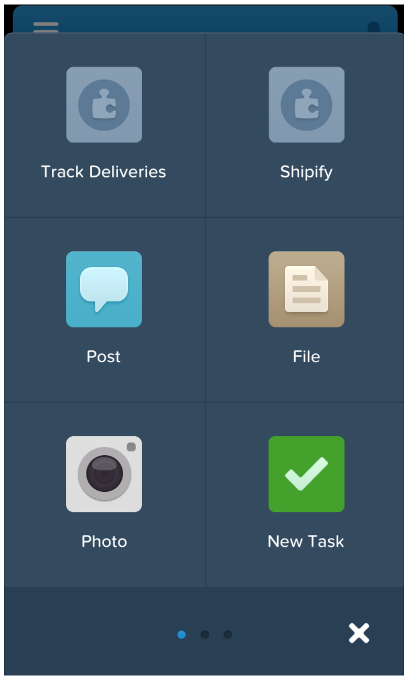
Test Out the DeliveryTrakr Canvas App

Now you'll play the part of an Acme Wireless warehouse worker processing deliveries, and we'll test out the canvas app in the Salesforce1 app.

The DeliveryTrakr app demonstrates how a canvas app can create one of three types of posts in the feed:

- Text post—Feed item that contains information about a delivery.
- Link post—Feed item that contains a link to the DeliveryTrakr app.
- Canvas post—Feed item that contains a link to another canvas app. This app is used by warehouse workers to approve or deny a delivery.

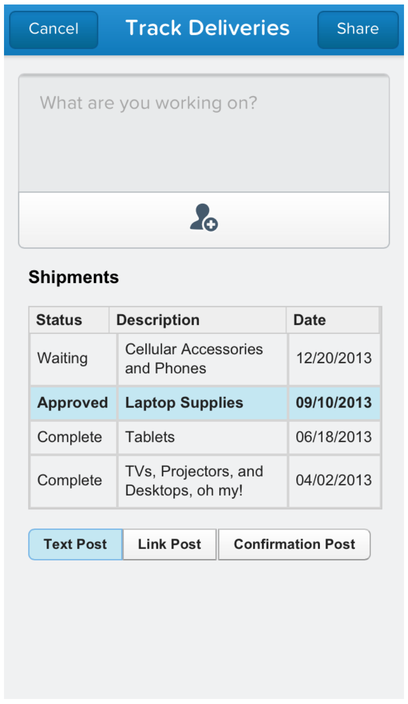
1. On your mobile device, tap **Feed** from the navigation menu.
2. Tap ➕ to access the publisher.



3. Tap **Track Deliveries**.

   The DeliveryTrakr app displaying a list of shipments appears.
4. Tap one of the deliveries in the grid.
5. Tap **Text Post**.

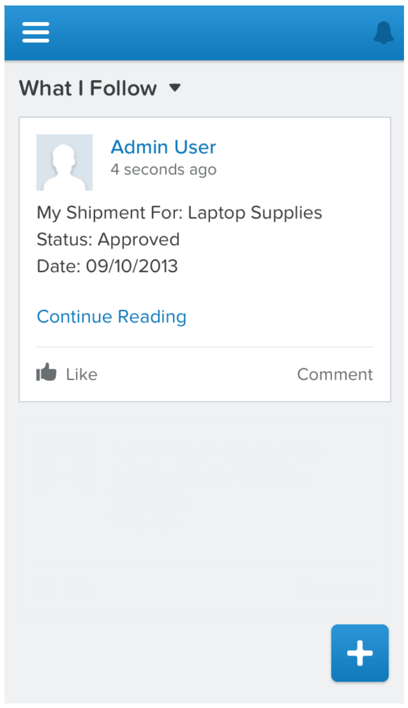
   You can also add some of your own text to the post by entering it in the **What are you working on?** pane.



Status	Description	Date
Waiting	Cellular Accessories and Phones	12/20/2013
Approved	Laptop Supplies	09/10/2013
Complete	Tablets	06/18/2013
Complete	TVs, Projectors, and Desktops, oh my!	04/02/2013

6. Tap **Share**.

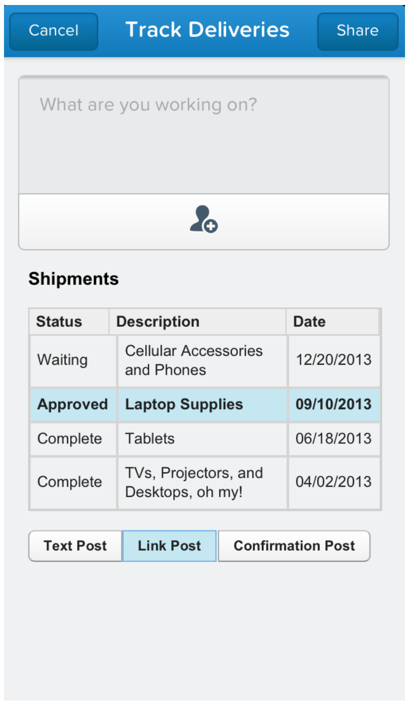
This creates a text feed post about the shipment, and that post appears in the current user's feed.



My Shipment For: Laptop Supplies
Status: Approved
Date: 09/10/2013

7. To test out creating a link post, tap + and then **Track Deliveries**.
8. Tap one of the deliveries in the grid.
9. Tap **Link Post**.

   You can also add some of your own text to the post by entering it in the **What are you working on?** pane.

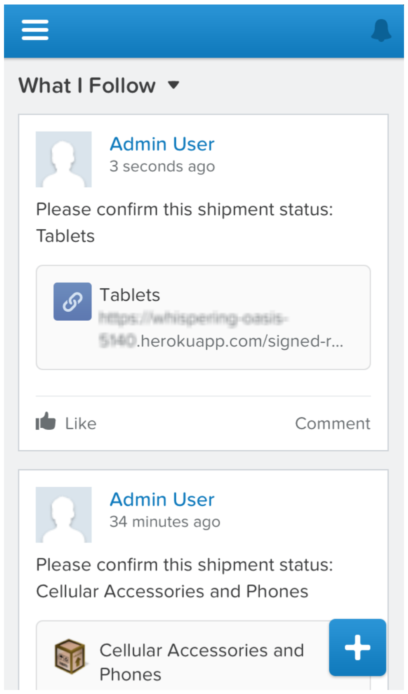


Shipments

Status	Description	Date
Waiting	Cellular Accessories and Phones	12/20/2013
Approved	Laptop Supplies	09/10/2013
Complete	Tablets	06/18/2013
Complete	TVs, Projectors, and Desktops, oh my!	04/02/2013

10. Tap **Share**.

    This creates a link feed post about the shipment, and that post appears in the current user's feed. When you click the link, it brings you to the DeliveryTrakr app running on Heroku.



What I Follow

Admin User
3 seconds ago

Please confirm this shipment status: Tablets

Tablets
https://whispering-oasis-5140.herokuapp.com/signed-r...

Like Comment

Admin User
34 minutes ago

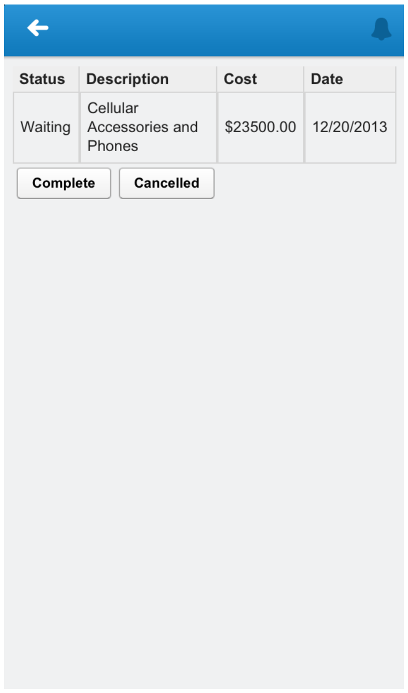
Please confirm this shipment status: Cellular Accessories and Phones

Cellular Accessories and Phones

11. To test out creating a canvas post, tap + and then **Track Deliveries**.
12. Tap one of the deliveries in the grid.
13. Tap **Confirmation Post**.

    You can also add some of your own text to the post by entering it in the **What are you working on?** pane.
14. Tap **Share**.

    This creates a canvas feed post about the shipment with a link. When you click the link, it actually takes you to a canvas app where you can complete or cancel a shipment.



Status	Description	Cost	Date
Waiting	Cellular Accessories and Phones	$23500.00	12/20/2013

You did it! You've gone through the entire process of creating a canvas app that posts a text feed item, a link feed item, and—most importantly—a canvas app in the feed.

For more information about development guidelines for canvas apps in the feed, see Canvas Apps in the Feed.

Tell Me More: About the Code to Create Feed Items

In the code for the DeliveryTrakr Web application, the `shipment.js` file contains the logic to create the various feed items.

Creating a Canvas Feed Item

This code snippet from the `onGetPayload` function creates a feed item of type `CanvasPost`. As you can see, it dynamically sets all the post values and retrieves information about the canvas app, such as the `namespace` and `developerName`, from the signed request. You can also pass in parameters that are appended to the URL and used in the canvas app feed item.

```
else if ("approval" === action)
{
    p.feedItemType = "CanvasPost";
    p.auxText = "Please confirm this shipment status: " +
        shipments[shipment].description;
    p.namespace =  sr.context.application.namespace;
    p.developerName =  sr.context.application.developerName;
    p.thumbnailUrl = "https://cdn1.iconfinder.com/data/icons/
        VISTA/project_managment/png/48/deliverables.png";
    p.parameters =  "{\"shipment\":\"" + shipment + "\"}";
    p.title = shipments[shipment].description;
    p.description = "This is a travel shipment for Shipment - " +
        shipments[shipment].description +
        ".  Click the link to open the canvas app.";
}
```

Enabling the Share Button

This code snippet is from the `draw` function. After this statement is called, the Share button is enabled. This ensures that the user can't create a feed item until they have selected all the required elements for the canvas app.

```
$$.client.publish(sr.client, {
    name : 'publisher.setValidForSubmit',
    payload : true});
```

Publishing the Feed Item

When the user clicks the Share button, this code snippet publishes the feed item to the feed. The type of feed item it creates depends on which type you select: text, link, or canvas.

```
$$.client.publish(sr.client, {name : 'publisher.setPayload',
    payload : p});
```