HANDS-ON ACTIVITIES

PROGRAMMATIC ARCHITECTURE

FEBRUARY, 2018

1. @FUTURE AND APEX

Describe when to apply the appropriate Apex functionality, such as custom settings, when to use synchronous vs. asynchronous patterns, and all the available execution contexts (e.g., batch, trigger, callout, etc.) when Apex should be used on the platform and articulate appropriate scenarios for each item.

USE CASE

Universal Containers has a group of users that manage cases regarding printer hardware issues. There is a lot of employee turnover in this group of users, so the administrators would like to automate the ability to add and remove users from this queue when users are created or deactivated.

DETAILED REQUIREMENTS

1. When a new user is created with a Department = ‘Printers,’ add the user to the ‘Printers Queue.’
2. When a user is de-activated and is part of the ‘Printers Queue,’ remove them from the queue.
3. When a user is updated with a Department = ‘Printers,’ add them to the ‘Printers Queue.’
4. When a user is re-activated with their Department = ‘Printers,’ add them to the ‘Printers Queue.’
5. When a user’s department changes from Printers to something else and they are part of the ‘Printers Queue,’ remove them from the queue.

PREREQUISITE SETUP STEPS

1. Case Queue named **Printers Queue**.
2. Assign one to two users to the **Printers Queue** with Department field = **Printers**.

CONSIDERATIONS

- How many triggers are needed?
- How much test coverage is required?
- What kinds of test methods should be created?
- How do you properly structure test classes to test @future methods?
- Determine whether @future is needed or not?
- Governor limits: What should you be concerned about?
- How do you want to handle any errors with executing the code?
- Should before or after update/insert triggers be used?
- What values are available with the Trigger.new and Trigger.old objects?
- When do you use each?
- When should you use with sharing versus without sharing with classes?

BEST SOLUTION OVERVIEW

- How many triggers are needed?
    - ◻ Ensure only one trigger per object to avoid sequencing issues.
- How much test coverage is required?
    - ◻ Triggers need a minimum of one line of test coverage. Recommend 100%.
    - ◻ Ensure 75% overall code coverage for the org. Recommend 100%.
- What kinds of test methods should be created?
    - ◻ Ensure positive, negative, single, and bulk record test methods are created.
- How do you properly structure test classes to test @future methods? Ensure test.startTest() & test.stopTest() are used to force future methods to fire before doing asserts on the results in the test class methods.

- Determine whether @future is needed or not?
    - ◻ Should the user wait for the code to finish? Or should it be run behind the scenes?
    - ◻ Does the user need to know what happened with this code?
- Governor limits: What should you be concerned about?
    - ◻ Ensure code is bulkified, especially important for triggers.
    - ◻ Number of @future calls in a single transaction.
- How do you want to handle any errors with executing the code?
    - ◻ Should the user receive notifications?
    - ◻ Should all records fail if one record in the batch fails?
- Should before or after update/insert triggers be used?
    - ◻ Before triggers to add to changes on the record before it is saved to the database.
    - ◻ After triggers to perform actions after a linkage or record creation/update, or access system-set fields.
    - ◻ Records in the after triggers are read-only.
- What values are available with the Trigger.new and Trigger.old objects? When do you use each?
    - ◻ Trigger.new and newMap has the new versions of the objects – available in insert and update and can only be modified in before triggers.
    - ◻ Trigger.old and oldMap has the previous versions of the objects – available in update and delete and cannot be modified.

- ▫ Other context variables: Trigger.isExecuting, isInsert, isUpdate, isDelete, isBefore, isAfter, isUndelete, size - used to determine context of the trigger code that is being executed.

- When should you use with sharing versus without sharing with classes?
  - ▫ Always use sharing; very few instances require without sharing (and with sharing is the default).
  - ▫ Without sharing is only used when the code must access/work with data that is not normally visible for the user executing the code.

SOLUTION DESCRIPTION:

1. Create an Apex trigger for after update and after insert on the user object.
   - Determine if user record matches scenario from detailed requirements or not and add to an update or remove list.
   - Pass the list to @future methods to perform the update or removal in bulk, if not empty.

2. Create an Apex class with two @future methods.
   - One method to add the users to the 'Printers Queue' queue.
   - One method to remove the users from the 'Printers Queue' queue.

APEX CLASS

```
public class CaseSync {

   @future
   public static void addMembers(List<Id> memberIds) {

      List<GroupMember> ToInsertUserList = new List<GroupMember>();
      Group groupId = [SELECT Id FROM Group WHERE Name = 'Printers Queue'
LIMIT 1];

      // for each user
      for (Id u : memberIds) {
         // user is new, add to queue
         GroupMember gma = new GroupMember( GroupId = groupId.Id,
UserOrGroupId = u );
         ToInsertUserList.add(gma);
      }

```

```
        insert ToInsertUserList;
    }

    @future
    public static void removeMembers(List<Id> memberIds) {

        List<GroupMember> ToDeleteUserList = new List<GroupMember>();
        Group groupId = [SELECT Id FROM Group WHERE Name = 'Printers Queue' LIMIT 1];

        ToDeleteUserList = [SELECT GroupId, UserOrGroupId FROM GroupMember WHERE Group.Type = 'Queue' AND GroupId = :groupId.Id AND UserOrGroupId in :memberIds];

        delete ToDeleteUserList;
    }

}

trigger CaseQueueSync on User (after update, after insert) {
    // Create an empty list of IDs
    List<Id> addUserIds = new List<Id>();
    List<Id> removeUserIds = new List<Id>();

    if (Trigger.isInsert) {
        for (User u : Trigger.new) {
            if (u.isActive == true && u.department == ‘Printers') {
                // will be an add, not a delete if either field was updated
                addUserIds.add(u.Id);
            }
        }
    } else {

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

        // Iterate through all of the *new* versions of User
        // sObjects updated by the call that fires the trigger, adding
        // corresponding IDs to the two userIds lists, but *only* when an
        // sObject's status changed from active to not active, vice-versa, or profile changes to/from Standard User.
        for (User u : Trigger.new) {
            if (u.isActive == true && u.department == 'Printers') {
                // will be an add, not a delete if either field was updated
                if ((oldMap.get(u.Id).isActive != true) || (oldMap.get(u.Id).department != 'Printers')) {
                    addUserIds.add(u.Id);
                } // else these two fields didn't change, so don't do anything
            } else if (u.isActive == false || u.department != 'Printers') {
                // will be a delete, not an add if either field was updated
                if ((oldMap.get(u.Id).isActive != false) || (oldMap.get(u.Id).department == 'Printers')) {
                    removeUserIds.add(u.Id);
                } // else these two fields didn't change, so don't do anything
            }

        }
    }
    // If the list of IDs is not empty, call CaseSync.addMembers & CaseSync.removeMembers
    // supplying the list of IDs for processing.
    if (addUserIds.size() > 0) {
        CaseSync.addMembers(addUserIds);
    }
    if (removeUserIds.size() > 0) {
        CaseSync.removeMembers(removeUserIds);
    }
}
```

2. BATCH APEX AND APEX CALLOUTS

Given a scenario, describe how batch apex and the flex queue could be applied.

USE CASE

Universal Containers has a Sales Cloud implementation. The sales team executes a routine process to close out Opportunities during the course of the day. At the end of the day, all closed-out Opportunities are required to be batched and sent to a backend service application for calculating the commission on the Opportunity for each of the Opportunity records. The commission amount is updated on each of the Opportunity records in response.

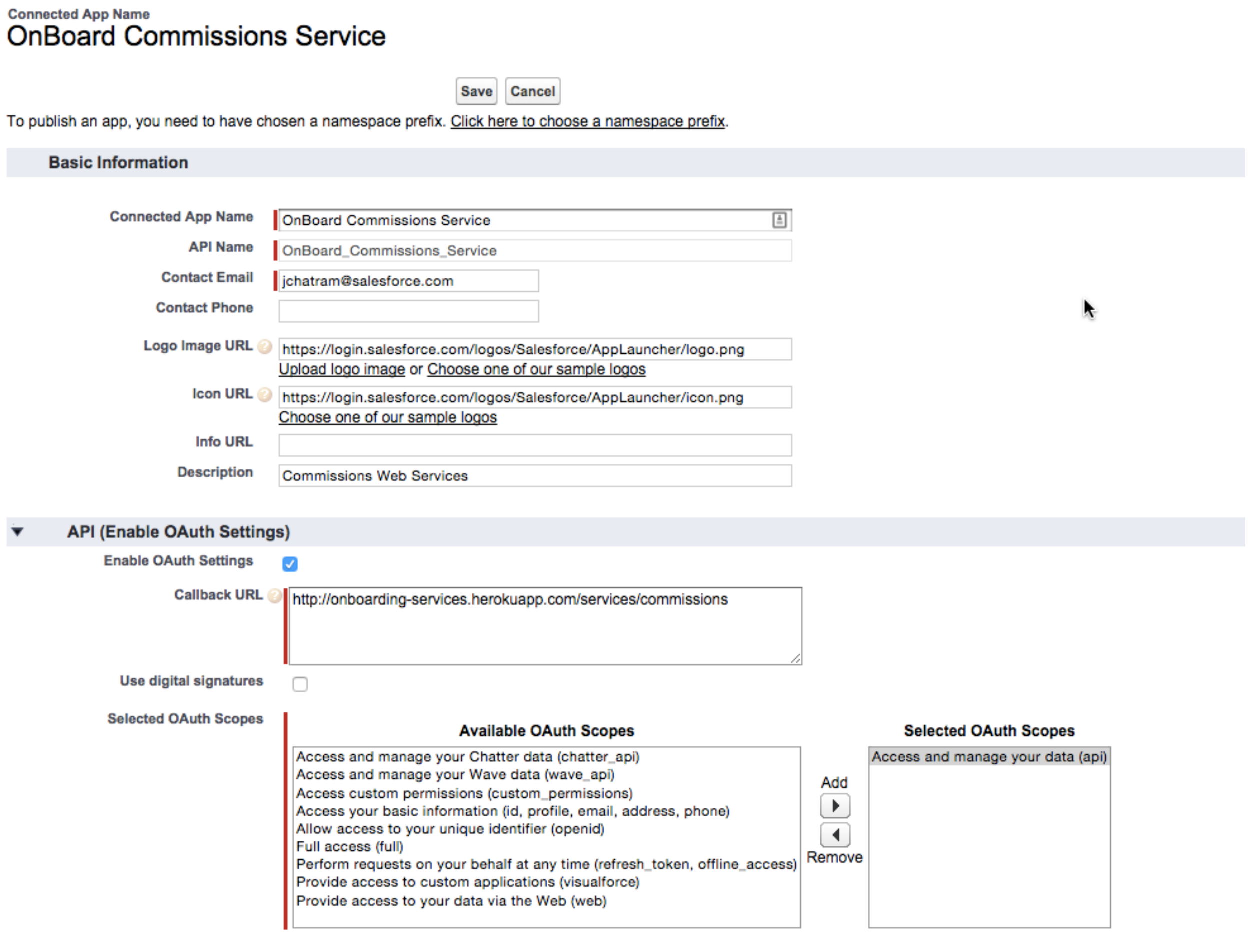
DETAILED REQUIREMENTS

The Universal Containers sales team has given a set of requirements for their development team to set up a recurring job to process all closed Opportunities for the day. The requirements:

1. To query all closed Opportunities records.
2. For Opportunities in "closed-won" status, group records in a batch and send the batch of Opportunities to a Commission Web Service:
   - Endpoint: http://onboarding-services.herokuapp.com/services/commissions
3. Responses received via web services call should update the commission amount for each of the opportunity records in scope.

PREREQUISITE SETUP STEPS

1. Create a new custom field of type "Currency" called "Commissions Amount" on the Opportunity object. Add this field to an appropriate Page layout.
2. Create a new custom field of type "Text" called "Opportunity Number" on the Opportunity object. Add this field to an appropriate Page layout.
3. Verify http://onboarding-services.herokuapp.com/services/ - services URL is available and active in a browser.


Connected App Name

OnBoard Commissions Service

To publish an app, you need to have chosen a namespace prefix. Click here to choose a namespace prefix.

Basic Information

Connected App Name	OnBoard Commissions Service
API Name	OnBoard_Commissions_Service
Contact Email	jchatram@salesforce.com
Contact Phone	
Logo Image URL	https://login.salesforce.com/logos/Salesforce/AppLauncher/logo.png Upload logo image or Choose one of our sample logos
Icon URL	https://login.salesforce.com/logos/Salesforce/AppLauncher/icon.png Choose one of our sample logos
Info URL	
Description	Commissions Web Services

API (Enable OAuth Settings)

Enable OAuth Settings	✓
Callback URL	http://onboarding-services.herokuapp.com/services/commissions
Use digital signatures	☐

Selected OAuth Scopes

Available OAuth Scopes:
- Access and manage your Chatter data (chatter_api)
- Access and manage your Wave data (wave_api)
- Access custom permissions (custom_permissions)
- Access your basic information (id, profile, email, address, phone)
- Allow access to your unique identifier (openid)
- Full access (full)
- Perform requests on your behalf at any time (refresh_token, offline_access)
- Provide access to custom applications (visualforce)
- Provide access to your data via the Web (web)

Selected OAuth Scopes:
- Access and manage your data (api)

CONSIDERATIONS

1. Test class to have 75% or greater code coverage for this scenario.
2. Verify Opportunities in the processed batch were successfully updated with Commissions Amount.
3. How do you address limits considerations? For schedulable batch apex?
4. How do you chunk the response size?
5. Implications of writing a test class?
6. Do you need to use a database that allows callouts?
7. How are you grouping the records for callouts?

SOLUTION

Log in to your Developer Org where this service setup is required.

1. Download and save the WSDL from this URL below:

http://onboarding-services.herokuapp.com/services/commissions?wsdl

2. Create a "Connected App" in the Org for the endpoint web services URL.

- Log in to your Developer Org where this service is required to be setup.
- Navigate to **"Your Name" | Setup | App Setup | Create | Apps.**
  - Under Connected Apps, click **New**.
  - Create new Connected App as shown in the example screen shot below by providing following information:
  - Click **Save**, to complete to create the Connected App:
    - Basic Information
    - API (Enable OAuth Settings)

3. Generate an Apex class from the WSDL.

- Navigate to **"Your Name" | Setup | App Setup | Develop | Apex Classes**.
- Click **Generate from WSDL**. Choose the downloaded WSDL file.
- Follow along with steps to generate a web service stub class. Provide a meaningful name for the service Apex class, for example, "Commissions Service."

4. Create a Batch Apex class.

- Navigate to **"Your Name" | Setup | App Setup | Develop | Apex Classes**.
- Give the Apex Class a name, "BatchUpdateCommissions."

5. Create an Apex class that implements schedulable.

- Navigate to **"Your Name" | Setup | App Setup | Develop | Apex Classes**.
- Give the Apex Class a name, "ExecuteCommissionsBatch."

6. Create test Apex class to:

- Execute the batch and set a mock web service call.
- Execute the commissions update with the mock service callout.

SAMPLE CODE

1. Generated Web Services Stub Class

```
//Generated by wsdl2apex

public class CommissionsService {
    public class getCommissionsResponse {
        public Double return_x;
        private String[] return_x_type_info = new String[]{'return','http://onboarding.salesforce.com/',null,'0','1','false'};
        private String[] apex_schema_type_info = new String[]{'http://onboarding.salesforce.com/','false','false'};
        private String[] field_order_type_info = new String[]{'return_x'};
    }
    public class getCommissions {
        public String arg0;
        public String arg1;
        private String[] arg0_type_info = new String[]{'arg0','http://onboarding.salesforce.com/',null,'0','1','false'};
        private String[] arg1_type_info = new String[]{'arg1','http://onboarding.salesforce.com/',null,'0','1','false'};
        private String[] apex_schema_type_info = new String[]{'http://onboarding.salesforce.com/','false','false'};
        private String[] field_order_type_info = new String[]{'arg0','arg1'};
    }
    public class GetCommisionsPort {
        public String endpoint_x = 'http://onboarding-services.herokuapp.com/services/commissions';
        public Map<String,String> inputHttpHeaders_x;
        public Map<String,String> outputHttpHeaders_x;
        public String clientCertName_x;
        public String clientCert_x;
        public String clientCertPasswd_x;
        public Integer timeout_x;
        private String[] ns_map_type_info = new String[]{'http://onboarding.salesforce.com/', 'Onboard_OppsCommission'};
        public Double getCommissions(String arg0,String arg1) {
            CommissionsService.getCommissions request_x = new CommissionsService.getCommissions();
            request_x.arg0 = arg0;
            request_x.arg1 = arg1;
            CommissionsService.getCommissionsResponse response_x;
            Map<String, CommissionsService.getCommissionsResponse> response_map_x = new Map<String, CommissionsService.getCommissionsResponse>();
            response_map_x.put('response_x', response_x);
            WebServiceCallout.invoke(
              this,
              request_x,
              response_map_x,
              new String[]{endpoint_x,
              '',
              'http://onboarding.salesforce.com/',
              'getCommissions',
              'http://onboarding.salesforce.com/',
              'getCommissionsResponse',
              CommissionsService.getCommissionsResponse'}
            );
            response_x = response_map_x.get('response_x');
            return response_x.return_x;
        }
    }
}
```

2. Batch Class

```
global class BatchUpdateCommissions implements Database.Batchable<sObject>, Database.AllowsCallouts {

global final String AsOfTime;

global BatchUpdateCommissions(String AsOf) {
// AsOf should be something like:
// LAST_WEEK, THIS_WEEK, NEXT_WEEK, LAST_MONTH, TODAY, LAST_N_DAYS:N, etc
this.AsOfTime = AsOf;
}

    global BatchUpdateCommissions() {
        this.AsOfTime = null;
    }

// start method should find all Closed Opportunities that are Closed-Won
global Database.QueryLocator start(Database.BatchableContext BC){
//String lastfewdays = this.AsOfDate.format('YYYY-MM-DDThh:mm:ssZ');
        //System.debug(lastfewdays);
String query = 'SELECT Id, Account.AccountNumber, Opportunity_Number__c, StageName, Commissions_Amount__c FROM Opportunity WHERE IsClosed = true';
        if (AsOfTime != null) {
            query += ' AND CreatedDate '+AsOfTime;
        }
        System.debug(query);
return Database.getQueryLocator(query);
    }

// execute method should call the CommissionsService to get & update the commission amount field
    global void execute(Database.BatchableContext BC, List<sObject> scope){
        System.debug(scope);
    List<Opportunity> opplist = new List<Opportunity>();
 // create stub
CommissionsService.GetCommisionsPort cs = new CommissionsService.GetCommisionsPort();
```

```

        for(Sobject s : scope){
        Opportunity o = (Opportunity)s;
        if (o.StageName == 'Closed Won') {
        Double value = cs.getCommissions(o.Opportunity_Number__c, o.Account.AccountNumber);
        o.Commissions_Amount__c = value;
        opplist.add(o);
        } // else do nothing!  or should I remove it from scope?
        }
        update opplist;
    }

    global void finish(Database.BatchableContext BC){
// do nothing
System.debug(BC);
    }

}
```

3. Schedule Apex Class

```
global class ExecuteCommissionsBatch implements Schedulable {

    public void execute(SchedulableContext sc) {
        Database.executeBatch(new BatchUpdateCommissions(), 1);
    }

}
```

4. Apex Test Class

```
@isTest public with sharing class BatchUpdateCommissions_Test {
// Execute the commissions update with mock service callout
@isTest public static void testbatch() {

BatchUpdateCommissions buc = new BatchUpdateCommissions();
createOpty();
Test.startTest();
CommissionsService.mock = new MockCommissionsCall();

Database.executeBatch(buc, 1);

// verify not set yet
List<Opportunity> olist = [SELECT Id, Name, Commissions_Amount__c FROM Opportunity];
System.assertEquals(olist[0].Commissions_Amount__c, null);
Test.stopTest();

olist = [SELECT Id, Name, Commissions_Amount__c FROM Opportunity];

// should be only one opportunity that we can see, so check that the commissions amount matches our mock call
System.assertEquals(olist[0].Commissions_Amount__c, 3245.4455);
}

// CRON expression: sample set to midnight on March 15. Set the time as appropriate for test run
// Because this is a test, job executes
// immediately after Test.stopTest().
public static String CRON_EXP = '0 0 0 15 3 ? 2022';

@isTest public static void createOpty() {
Account a = new Account(Name='Test Acct', AccountNumber='3456');
insert a;
Opportunity o = new Opportunity(Name='Test Opty 1', StageName='Closed Won', AccountId = a.Id, CloseDate=Date.today());
insert o;
}

// Execute batch and set up a mock web service call
@isTest public static void test() {
createOpty();
ExecuteCommissionsBatch ex = new ExecuteCommissionsBatch();
Test.startTest();
Test.setMock(WebServiceMock.class, new MockCommissionsCall());
// Schedule the test job
String jobId = System.schedule('ScheduleApexClassTest',
                                1. CRON_EXP,
                                2. ex);

// Get the information from the CronTrigger API object
CronTrigger ct = [SELECT Id, CronExpression, TimesTriggered,
        a.  NextFireTime
        b.  FROM CronTrigger WHERE id = :jobId];

// Verify the expressions are the same
System.assertEquals(CRON_EXP,
        a.  ct.CronExpression);

// Verify the job has not run
System.assertEquals(0, ct.TimesTriggered);

// Verify the next time the job will run
System.assertEquals('2022-03-15 00:00:00',
        a.  String.valueOf(ct.NextFireTime));
// Verify the scheduled job hasn't run yet.
Opportunity[] olist = [SELECT Id, Commissions_Amount__c FROM Opportunity
WHERE Commissions_Amount__c = NULL];
System.assertEquals(olist.size(),1);
Test.stopTest();

// Now that the scheduled job has executed after Test.stopTest(),
//   fetch the new updates (which won't happen since the schedule won't run the
batch process).
olist = [SELECT Id, Commissions_Amount__c FROM Opportunity WHERE
Commissions_Amount__c = NULL];
System.assertEquals(olist.size(), 1);

}
}
```

MOCK SERVICE CLASS

```
@isTest global class MockCommissionsCall implements WebServiceMock {
   global void doInvoke(
           Object stub,
           Object request,
           Map<String, Object> response,
           String endpoint,
           String soapAction,
           String requestName,
           String responseNS,
           String responseName,
           String responseType) {

        // Create response element from the autogenerated class.
        CommissionsService.getCommissionsResponse responseElement = new
CommissionsService.getCommissionsResponse();
        // Populate response element.
        responseElement.return_x = 3245.4455;
        // Add response element to the response parameter, as follows:
        response.put('response_x', responseElement);
   }
}
```

3. NEW INVOICE BUTTON

Describe types, considerations, and tradeoffs when designing Apex controllers.

USE CASE

Universal Containers uses Sales Cloud. They use Opportunities to track their sales orders. Once an Opportunity is closed, they can create multiple invoices from the Sales Cloud application from the Opportunity. The typical process for invoice creation is:

1 Search for the Opportunity.
2 Click **Add new Invoice** from the Invoice related list under Opportunity.
3 When they are in the new Invoice screen, they would like to default the following fields on the Invoice from the Account and Opportunity:
    a. Account Billing Address (Street 1, 2, City, State and Zip code).
    b. A comma-separated list of Product codes that are associated with the Opportunity.
    c. Default Discount code that is a field in the account.

4 The sales rep should have the option to change the values in the invoice once they are defaulted. If the rep clicks **Cancel**, the rep should be taken back to the Opportunity that they navigated from.

DETAILED REQUIREMENTS:

- Search for the Opportunity: either global search or select an **Opportunity** from Opportunity List views.
- User clicks on the **New Invoice** button from a related list on the Opportunity page.
- Populate details from the Opportunity into the Invoices.
- Let users edit the values in the invoice edit page.
- User clicks **Save**, invoice is inserted into Salesforce and returned back to the Opportunity read page.
- When users click on the **Cancel** button, they are redirected to the Opportunity read page.

CONSIDERATIONS

- How many controllers and controller extensions are required?
- How much test coverage is required?
- What kinds of test methods should be written?
- When should you use with sharing and without sharing classes?
- Governor limits: what should you be concerned about?
- How do you want to handle errors with executing the code?

BEST SOLUTION OVERVIEW

1 How many controllers and controller extensions are needed?
   a. One controller extension is required. If needed, Helper classes can be created.

2 How much test coverage is required?
   a. A minimum of 75% test coverage is required.

3 What kind of test methods should be written?
   a. Ensure that positive, negative, single, and bulk record test methods are created.

4 When should you use with sharing and without sharing classes?
   a. Always use sharing; very few instances require without sharing (and with sharing is the default).
   b. Without sharing is only applicable when the code must access work with data that is not normally visible for the user executing the code.

5 Governor limits: what should you be concerned about?
   a. Ensure code is bulkified.
   b. No SOQL queries within loops.
   c. Keep viewstate to minimal.

6 How do you handle errors with executing the code?
   a. Should the user receive notifications?
   b. Should all records fail if one record in the batch fails?

SOLUTION DESCRIPTION:

1 Create a custom invoice button for the invoice related list under Opportunity.

2 Create a VF page for the invoice and custom buttons for the Save and Cancel operation. Navigate to **Setup | Develop | Visualforce Pages**.

```
<apex:page controller=”Invoice” extensions=”InvoiceExtension” showHeader="false" >
<apex:pageblock>
  <apex:commandButton value="Next" action="{!save}" id="savebtn" />
  <apex:commandButton value="Next" action="{!cancel}" id="savebtn" />
  <apex:pageblock>
          <apex:inputfield name=”Opportunity” value=”Opportunity” />
          <apex:inputfield name=”Street1” value=”Street1” />
          <apex:inputfield name=”Street2” value=”Street2” />
          <apex:inputfield name=”City” value=”City” />
                    <apex:inputfield name=”State” value=”State” />
                    <apex:inputfield name=”Zipcode” value=”Zipcode” />
                    <apex:inputfield name=”productcodes” value=”ProductCodes” />
                    <apex:inputfield name=”DiscountCode” value=”DiscountCode” />

          </apex:pageblock>
</apex:pageblock>
</apex:page>
```

3 Create a controller extension for the Invoice controller. Navigate to **Setup | Develop | Apex Classes.**

```
public class with Sharing InvoiceControllerExtension{
          private final Invoice inv;
          public String StreetName1{get;set;}
          public String StreetName2{get;set;}
          public String City{get;set;}
          public String State{get;set;}
public String Zipcode{get;set;}
public string opportunityProductCodes{get;set;}
          public InvoiceControllerExtension(ApexPages.StandardController stdController){
this.inv = (Invoice)stdController.getRecord();
                              Opportunity opp = [select id, name, Account, Account.BillingAddress from Opportunity where Id =: inv.Opportunity];
                              OpportunityProduct OppProducts = [select id, productcode from OpportunityProduct where Opportunity =: inv.Opportunity];
```

```
        inv.Street1 = opp.BillingAddress.Street1;
        inv.Street2 = opp.BillingAddress.Street2;
        inv.City = opp.BillingAddress.City;
        inv.State = opp.BillingAddress.state;
        inv.Zipcode = opp.BillingAddress.Zipcode;

        for(OpportunityProduct oppproduct : OppProducts){
            if (OpportunityProductCodes == null || OpportunityProductCodes.length == 0){
                OpportunityProductCodes = OpportunityProduct.ProductCode;
            }else{
                OpportunityProductCodes += “,” + OpportunityProduct.ProductCode;
            }
        }
        inv.OpportunityProductCodes = OpportunityProductCodes;
    }

    public PageReference Save(){
        //Insert code to customize the Save logic as required
        return Pagereference;
    }

    public PageReference Cancel(){
        //Insert code to customize the Cancel logic as required
        return Pagereference;
    }

```

4 Get the Opportunity related information and assign it to the variables in the controller extension. Please refer to the above in which Opportunity, Account and OpportunityProduct information are retrieved from the database and stored in the variables via a constructor.

5 Keep the fields in the Invoice VF page editable and map the values to the variables in the controller extension. Please refer to the VF page above where all fields are listed as input fields that will let the user edit the values populated through the controller.

6 Override the Save() and Cancel() methods for the custom logic and URL redirection.

ADDITIONAL ACTIVITIES

1. 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 a useful tool you'll use 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 then click **Execute**.

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

   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.

4. 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);

5. Select **Open Log** and click **Execute**.

6. 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.

7. To filter the list to show only USER_DEBUG events, select Debug Only or enter *USER* in the Filter field.

NOTE:

NOTE:

The filter text is case sensitive.

Congratulations–you have successfully executed code on the Force.com platform and viewed the results!

2. CREATING AND LISTING VISUALFORCE PAGES

In this tutorial, you'll learn how to create and edit your first Visualforce page. The page will be really simple, but this is the start, and we'll soon expand on it. Along the way you'll familiarize yourself with the editor and automatic page creation.

Enable Visualforce Development Mode

Development mode embeds a Visualforce page editor in your browser that allows you to see code and preview the page at the same time. Development mode also adds an Apex editor for editing controllers and extensions.

1 At the top of any Salesforce page, click the down arrow next to your name. From the menu under your name, select Setup or My Settings–whichever one appears.

2 From the left panel, select one of the following:
- If you clicked Setup, select **My personal information** > **Personal Information**.
- If you clicked **My Settings**, select **Personal** > **Advanced User Details**.

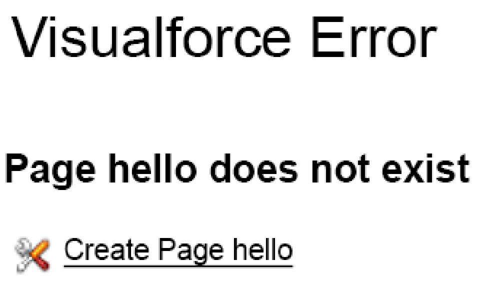
3 Click **Edit**.

4 Select the Development Mode checkbox, then click **Save**.

Create a Visualforce Page

1. In your browser, add */apex/hello* to the URL for your Salesforce instance. For example, if your Salesforce instance is:

   https://na1.salesforce.com, the new URL is *https://na1.salesforce.com/apex/hello*. You will see the following error:



Visualforce Error

Page hello does not exist

Create Page hello

2. Click the **Create Page hello** link to create the new page. You will see your new page with some default markup.

NOTE:

NOTE:
If you don't see the Page Editor below the page, just click the ***hello*** *tab in the status bar.*

That's it! The page includes some default text, as well as an embedded page editor displaying the source code. This is the primary way you'll be creating pages in this section of the workbook.