Exception Handling

- Standard exception handling syntax - Order from Specific to Generic

```
try {
} catch(DmlException e) {
// DmlException handling code here.
} catch(Exception e) {
// Generic exception handling code here.
} finally {
// Final code goes here
}
```

  - Common Exception methods
    - getCause: Returns the cause of the exception as an exception object.
    - getLineNumber: Returns the line number from where the exception was thrown.
    - getMessage: Returns the error message that displays for the user.
    - getStackTraceString: Returns the stack trace as a string.
    - getTypeName: Returns the type of exception, such as DmlException, ListException, MathException, and so on.
    - Some exceptions such as DML exceptions have special methods
      - getDmlFieldNames - Returns the names of the fields that caused the error for the specified failed record.
      - getDmlId: Returns the ID of the failed record that caused the error for the specified failed record.
      - getDmlMessage: Returns the error message for the specified failed record.
      - getNumDml: Returns the number of failed records.

  - Famous DML Exceptions
    - **DmlException** - Problems with DML Statements
    - **ListException** - Any problem with a list such as index out of bounds exceptions
    - **NullPointerException** - Problems with dereferencing a null variable.
    - **QueryException** - Any problem with SOQL queries, such as assigning a query that returns no records or more than one record to a singleton sObject variable.

    - **SObjectException** - Any problem with sObject records, such as attempting to change a field in an update statement that can only be changed during insert.

- To create a custom exception use

```
publicclassMyExceptionextendsException {}

// To create an exception
newMyException();

newMyException('This is bad'); // Error Message input

newMyException(e); // Exception argument

newMyException('This is bad', e);
// The first is the message and the second is the cause.
```

===========

Triggers

- Standard Trigger Syntax

```
triggerTriggerName on ObjectName (trigger_events) { code_block
}
```

- The following are the trigger events
  - before insert
  - before update
  - before delete
  - after insert
  - after update
  - after delete
  - after undelete
- Use **addError(errorMessage)** on the Trigger object lists to add an error

Testing

- Minimum **75%** test coverage required to deploy to production
- All tests must **pass** in order to deploy to production
- **@isTest** annotation is put on the class to indicate that it is a test class
- The test classes don't count towards the 3MB org code size limit
- Test methods need to be static and are defined using the testmethod keyword or with the @isTest annotation

```
statictestmethodvoidmyTest() {
// Add test logic
}
```

or

```
static@isTestvoidmyTest() {
// Add test logic
}
```

- The tests can have only one Test.startTest() and Test.stopTest() block. This block ensures that other setup code in your test doesn't share the same limits and enables you to test the governor limits.

Async Apex

Scheduable

- Used to run apex code at specific times
- Uses the **Scheduable** interface which requires than the execute function be implemented

- Example:

```
global classMySchedulableClassimplementsSchedulable {
  global voidexecute(SchedulableContextctx) {
CronTriggerct= [SELECTId, CronExpression, TimesTriggered,
NextFireTimeFROMCronTriggerWHEREId=:ctx.getTriggerId()];
System.debug(ct.CronExpression);
System.debug(ct.TimesTriggered);
  }
}
```

- Use **System.schedule** to schedule a job. Format

```
// For Midnight on march 15
// Format is
// Seconds Minutes Hours Day_of_month Month Day_of_weekoptional_year
publicstaticStringCRON_EXP='0 0 0 15 3 ? 2022';

System.schedule(NameOfJob, CRON_EXP, newMySchedulableClass());
```

- When Test.StartTest() is used then the job run immediately instead of waiting for Cron time.
- You can only have **100** classes scheduled at one time.
- The Scheduled Jobs setup item can be used to find currently scheduled jobs

Apex Batch Processing

- Lets you process batches asynchronously
- Each invocation of a batch class results in a job being placed on the Apex job queue for execution.
- The execution logic is called once per batch
- **Default batch size is 200**, you can also specify a custom batch size.
- Each new batch leads to a new set of Governor limits
- Each batch is a descrete transaction
- A batch class has to implement the **Database.Batchable** interface
- Example:

```
globalclassCleanUpRecordsimplementsDatabase.Batchable<sObject> {

globalfinalString query;

globalCleanUpRecords(Stringq) {
query= q;
    }

// 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.
globalDatabase.QueryLocatorstart(Database.BatchableContextBC) {
returnDatabase.getQueryLocator(query);
    }

// 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.
globalvoidexecute(Database.BatchableContextBC, List<sObject>scope){
delete scope;
Database.emptyRecycleBin(scope);
    }

// The finish method is called after all batches are processed. Use this method to
send confirmation emails or execute post-processing operations.
globalvoidfinish(Database.BatchableContextBC){
AsyncApexJob a =
               [SELECTId, Status, NumberOfErrors, JobItemsProcessed,
TotalJobItems, CreatedBy.Email
FROMAsyncApexJobWHEREId=
:BC.getJobId()];
// Send an email to the Apex job's submitter
//   notifying of job completion.
Messaging.SingleEmailMessage mail =newMessaging.SingleEmailMessage();
String[] toAddresses=newString[] {a.CreatedBy.Email};
mail.setToAddresses(toAddresses);
mail.setSubject('Record Clean Up Status: '+a.Status);
mail.setPlainTextBody
           ('The batch Apex job processed '+a.TotalJobItems+
'batches with '+a.NumberOfErrors+' failures.');
Messaging.sendEmail(newMessaging.SingleEmailMessage[] { mail });
    }
}
```

- Batches of records are not guaranteed to execute in the order they are received from the start method.
- The maximum number of records that can be returned in the Database.QueryLocator object is **50 million**.
- Test methods can execute only one batch.
- To execute a batch job use **Database.executeBatch**

```
CleanUpRecords c =newCleanUpRecords(query);
Database.executeBatch(c);
```

- Batches can be scheduled using a scheduler class

=========

REST

- To create a rest resource annotate the class with `@RestResource(urlMapping='/Accounts/*`
- The base URL is https://instance.salesforce.com/services/apexrest/ and the rest of the mapping is appended to that.
- Class must be **global** and the methods must be **global static**
- The `@HttpGet` or `@HttpPost` annotations need to be used on the methods
- Use the following to get the compelete URL of the request

```
RestRequestreq=RestContext.request;
String merchId=req.requestURI.substring(req.requestURI.lastIndexOf('/') +1);
```

In the above example we are splitting the URL and taking the last index

- The workbench can be used to test REST APIs
- In the case of HTTP POST arguments passed to the method are automatically deserialized

```
@HttpPost
global static String createMerchandise(String name,
  String description, Decimal price, Double inventory) {
Merchandise__c m =newMerchandise__c(
      Name=name,
Description__c=description,
Price__c=price,
Total_Inventory__c=inventory);
  insert m;
returnm.Id;
}
```

The input to this is

```
{
"name" : "Eraser",
"description" : "White eraser",
"price" : 0.75,
"inventory" : 1000
}
```

Visualforce Basics

- In order to use a Standard List controller, add the **recordSetVar** attribute to the `<apex:page>` tag
- The `<apex:pageBlock/>` and `<apex:pageBlockSection />` tags create some user interface elements which match the standard UI style
- The pageBlockSection tag has a **columns** element which lets you set the number of columns in the section.
- The `<apex:pageBlockTable value="{!products}" var="pitem">` tag adds the standard visualforce table.
- The **column** tag creates a column in the table

```
<apex:pageBlockTablevalue="{!products}"var="pitem">
<apex:columnheaderValue="Product">
<apex:outputTextvalue="{!pitem.Name}"/>
</apex:column>
</apex:pageBlockTable>
```

- The **tabStyle** attribute of `apex:page` will make sure that the tab is displayed as the Object Name when a standardController is not used
- The **$User** variable is used to access user details
- The `<apex:detail />` tag is used to display the standard view page for a record
- The relatedList tag is used to display related lists

```
<apex:relatedListlist="Cases" />
```

- **rerender** attribute is used on certain tags to re-render a section
- Ways to display visualforce
    - Override Standard Pages
    - Embed a page in a standard layout
    - Create a button that links to a VF page
    - VF Pages can be used in public facing sites
- **$Page.pageName** is used to get a link reference to another page
- You can link to default actions in your pages.

```
<apex:outputLinkvalue="{!URLFOR($Action.Account.new)}">Create</apex:outputLink>
```

- `<apex:pageMessages />` is used to show errors
- `<apex:inputField />` and `<apex:outputField />` also show the field labels

- Templates
    - Use `<apex:insert>` to insert a placeholder in a template e.g. `html <apex:page><h1>My Fancy Site</h1><apex:insert name="body"/></apex:page>`
        - Use the template by using the `<apex:define />` tag

```
<apex:pagesidebar="false"showHeader="false">
<apex:compositiontemplate="BasicTemplate">
<apex:definename="body">
<p>This is a simple page demonstrating that this
          text is substituted, and that a banner is created.</p>
</apex:define>
</apex:composition>
</apex:page>
```

    - Another way to include a page is to use the `<apex:include />` tag

```
<apex:pagesidebar="false"showHeader="false"><p>Test Before</p>
<apex:includepageName="MainPage"/><p>Test After</p>
</apex:page>
```

- Use the **$Resource** variable to access static resources
- Use `Visualforce.remoting.Manager.invokeAction('{!$RemoteAction.ClassName.MethodName}')` to invoke a remote action
- Use the following to format an output text in currency format

```
<apex:outputTextvalue="{0,number,currency}">
<apex:paramvalue="{!pitem.Price}"/>
</apex:outputText>
```

Development Lifecycle

- Typical Development Lifecycle for Developing in Production
    - Plan functional requirements.
    - Develop using Salesforce Web tools, using profiles to hide your changes until they're ready to deploy.
    - Update profiles to reveal your changes to the appropriate users.
    - Notify end users of changes.
- Typical Development Lifecycle for Developing in a Sandbox
    - Create a development environment.
    - Develop using Salesforce Web and local tools.
    - Test within the development environment.
    - Replicate production changes in the development environment.
    - Deploy what you've developed to your production organization.

- Ways of migrating changes between Sandboxes and Production
    - Change Sets
    - ANT Migration Toolkit
    - Force.com IDE
- Typical Development Cycle for developing in Dev and Test
    - Create a development environment.
    - Develop using Salesforce Web and local tools.
    - Create a testing environment.
    - Migrate changes from development environment to testing environment.
    - Test.
    - Replicate production changes in other environments.
    - Deploy what you've developed to your production organization.
- Developing Multiple Projects with Integration, Testing, and Staging
    - Create development environments.

- Develop using Salesforce Web and local tools.
- Create testing environments, including UAT and integration.
- Migrate changes from development environment to integration environment.
- Test.
- Migrate changes from integration environment to UAT environment.
- Perform user-acceptance tests.
- Migrate changes from UAT environment to staging environment.
- Replicate production changes in staging environment.
- Schedule the release.

- Types of Sandboxes

    - Developer - Developer sandboxes copy customization (metadata), but don't copy production data, into a separate environment for coding and testing.
    - Developer Pro - Developer Pro sandboxes copy customization (metadata), but don't copy production data, into a separate environment for coding and testing. Developer Pro has more storage than a Developer sandbox. It includes a number of Developer sandboxes, depending on the edition of your production organization.
    - Partial Copy - A Partial Copy sandbox is a Developer sandbox plus the data that you define in a sandbox template.
    - Full Copy - Full sandboxes copy your entire production organization and all its data, including standard and custom object records, documents, and attachments. Use the sandbox to code and test changes, and to train your team about the changes. You can refresh a Full sandbox every 29 days.

- Uses of Sandboxes

    - Developer - Developer or Developer Pro sandbox
    - Testing - Unit tests and Apex tests: Developer or Developer Pro sandbox, Feature tests and regression tests: Partial Copy sandbox (with a standard data set loaded)
    - Testing external integrations - Full sandbox is best when an external system expects full production data to be present. Partial Copy sandboxes may be appropriate in special cases when you want to use sample data or a subset of your actual data. Works well if you're using external IDs.
    - Staging and user-acceptance testing - Full sandbox is best for validation of new applications against production configuration and

data. Partial Copy sandboxes are appropriate if testing against a subset of production data is acceptable, for example, for regional tests.
    - Production debugging - Full Copy Sandbox

- Use Web-based importing when:
    - You are loading less than 50,000 records.
    - The object you need to import is supported by import wizards. To see what import wizards are available and thus what objects they support, from Setup, enter Data Management in the Quick Find box, then select Data Management.
    - You want to prevent duplicates by uploading records according to account name and site, contact email address, or lead email address.

- Use Data Loader when
    - You need to load 50,000 to 5,000,000 records. Data Loader is supported for loads of up to 5 million records.
    - You need to load into an object that is not yet supported by the import wizards.
    - You want to schedule regular data loads, such as nightly imports.
    - You want to export your data for backup purposes.

- Process of Developing Applications in a Release Train
    - Plan your release around Salesforce upgrades.
    - Schedule your concurrent development projects. This will help you determine if the new functionality can be done now, in the current release, or sometime in the future.
    - Establish a process for changes to the production organization.
    - Track changes in all environments. This will ensure that short-term functionality that is delivered to production does not get overwritten when you deploy from your development environment.
    - Integrate changes and deploy to staging or test environments.
    - Release to production.

Large Data Volumes

- Force.com query optimizer
  - Determines the best index from which to drive the query, if possible, based on filters in the query
  - Determines the best table to drive the query from if no good index is available
  - Determines how to order the remaining tables to minimize cost
  - Injects custom foreign key value tables as needed to create efficient join paths
  - Influences the execution plan for the remaining joins, including sharing joins, to minimize database input/output (I/O)
  - Updates statistics
- Skinny Tables -
  - These are copies of the data for a table with limited number of fields (100 columns).
  - Are kept in sync with the source data
  - Help in fast data access.
  - Can only contain field from the base object
  - Need to be enabled by support

- Indexes
  - Custom indexes can be created to optimize performance
  - Some indexes such as RecordTypeId, Division, CreatedDate, Name, Email are created by default
  - Indexes for External Id are created by default
  - By default nulls are not considered in the index
  - If a where clause narrows down the crietria to 30% for Standard fields or 10% for Custom fields the index is used
  - Formula fields which are deterministic are indexed
  - A formula field is deterministic if
    - It does not use TODAY(), NOW(), etc
    - Does not refer to a different object
    - Does not refer to a formula field which references a different object
    - Owner field
    - Standard field with special functionalities
  - Two column indexes can also be created for supporting search + sort. These are more efficient than 2 indexes.

- Divisions can be used to partition huge volumes of data by business unit
- Mashups can be used to get real time data from an external system by either using Canvas or Callouts
- Sharing Calculation can be deferred by an admin to speed up data load
- Hard Deletes can be used to optimize performance so that data does not go to the recycle bin.
- Best Practices for Reporting
    - Partition data
    - Minimize no of joins (No of objects and relationships on the report)
    - Reduce amount of data returned - reduce no of fields
    - Use filters which use standard or custom indexes
    - Archive unused records
- Best Practices for Loading Data through API
    - Use the bulk api if you have more than 100,000 records
    - insert() is the fastest followed bu update() and then upsert()
    - Data should be clean before processing. Error in batches cause the records to be processed row by row.
    - Load only changed data
    - For Custom integrations - use http keep-alive, use gzip compression, authenticate only once
    - Avoid switching on sharing during the load
    - Disable Computations such as Apex Triggers, Workflows, etc
    - Group records by ParentId to avoid locking conflicts
    - Use SF Guid instead of External Id for Parent References
    - Bulk API has a limit of the number of records you can load in a day. 300 batches and 10,000 records per batch
- Best Practices for Extracting data from the API
    - getUpdated() and getDeleted() should be used
    - Where more than 1 million records are returned use the Bulk API
    - When using the bulk API chunk the queries
- Best Practices for Searching
    - Avoid Wildcards

- Reduce the no of joins
    - Partition data with Divisions
- Best Practices for SOQL and SOSL
    - Search on Indexed fields
    - Avoid quering formula fields
    - Don't use nulls in where clause
- General Best Practices
    - Consider using an Aggregate Data custom object which is populated using batch apex
    - Consider using a different value instead of null such as N/A
    - Enable Separate Loading of Related Lists setting to reduce the time taken to reduce related list rendering
    - Remove the API User from the sharing hierarchy
    - Consider using a query which reduces the data set to either 30% or 10%
    - Keep the recycle bin empty

Single Sign On

- Preferred method is Standards based SAML
- Best way is to use the My Domain Feature
- Supports mutiple identity providers
- IDP Sends single sign on messages
- A Service Provider (SP) recieves Single Sign On messages
- Trust is stablished by exchanging meta data
- Messages are digitaly signed XML documents
- Single Sign on works on mobile and desktop app as well when using OAuth
- Issuer is a unique string in the SAML
- IDP Certificate is used to validate the signed request
- Three subjects are supported
    - Salesforce.com username
    - Federation Id

    - User Id from User object
- IDP must support relay state for SP initiated SAML
- Just in time provisioning allows the IDP to create a user in Salesforce, requires to pass provisioning attributes
- Single Sign On is possible for Communities and Portal
- Firefox has a SAML Tracer plugin to debug or the SAML Assertion validator

Salesforce Identity

- Use connected apps to connect to external apps

Social Sign On

- Login and OAuth access from public sources
- Use from My Domains features
- Supported Providers
    - Salesforce
    - Facebook
    - Open Id Connect
    - Janrain
- Registration handler class is used to create, update or link to existing users

---

Integration

- The following are useful tips when coming up with a end state architecture
    - The integration architecture should align with the business strategy
    - Integration architecture should support a mix of batch and real-time
    - Architecture should be based around busines service level agreements (BSLAs)
    - Integration architecture should have clearly defined standard for implementing different use cases
- Typical methods

- Cloud to Ground (Salesforce.com originated)
    a. The message is relayed to a DMZ end point (either a firewall, reverse proxy or gateway appliance). This is where security authentication occurs. Whitelist IPs, two way SSL, basic HTTP Authentication
    b. DMZ relays the message to the On Premise infrastructure usually an ESB
    c. ESB then may push the message to the SOA infrastructure, source system, Database or EDW etc
- Ground to Cloud
    a. Most designs use an ESB to connect to Salesforce which provides session management
    b. ETL solutions can be leveraged to send large data volumes
    c. Offline data replication is another scenario
    d. ETL can pull data from Salesforce
- Cloud to Cloud
    a. Salesforce2Salesforce can be used to connect to other instances
    b. Stay away from building complex integration, instead use aIPaas solution.
    c. Having to build durable and resilient integration solutions inside of Salesforce can be expensive and very complicated