Salesforce - Platform Developer 1 Master Cheat Sheet

The purpose is not to cover the basics but to have some of the advanced topics that one might not remember.

1. Exceptions
2. Triggers
3. Testing
4. Async Apex
5. REST
6. Visualforce Basics
7. Development Lifecycle

Optional

1. Large Data Volumes
2. Single Sign On
3. Integration
4. Security
5. Siebel to Salesforce
6. Master Data Management

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

```
    public class MyException extends Exception {}

// To create an exception
new MyException();

new MyException('This is bad'); // Error Message input

new MyException(e); // Exception argument

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

Triggers

- Standard Trigger Syntax

```
trigger TriggerName 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

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

or

```
static @isTest void myTest() {
    // 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 that the execute function be implemented
- Example:

```
global class MySchedulableClass implements Schedulable {
        global void execute(SchedulableContext ctx) {
        CronTrigger ct = [SELECT Id, CronExpression, TimesTriggered, NextFireTime
FROM CronTrigger WHERE Id = :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_week optional_year
public static String CRON_EXP = '0 0 0 15 3 ? 2022';

System.schedule(NameOfJob, CRON_EXP, new MySchedulableClass());
```

- 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:

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

    global final String query;

    global CleanUpRecords(String q) {
       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.
        global Database.QueryLocator start(Database.BatchableContext BC) {
        return Database.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.
        global void execute(Database.BatchableContext BC, 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.
    global void finish(Database.BatchableContext BC){
           AsyncApexJob a =
               [SELECT Id, Status, NumberOfErrors, JobItemsProcessed,
                TotalJobItems, CreatedBy.Email
                FROM AsyncApexJob WHERE Id =
                :BC.getJobId()];
           // Send an email to the Apex job's submitter
           //   notifying of job completion.
           Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();
           String[] toAddresses = new String[] {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(new Messaging.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 = new CleanUpRecords(query);
Id BatchId = Database.executeBatch(c); //The returned Id can be used to Query
Status, Errors etc..
AsyncApexJob a = [SELECT Id, Status, NumberOfErrors, JobItemsProcessed,
```

```
                TotalJobItems, CreatedBy.Email
                FROM AsyncApexJob WHERE Id = :BatchId];
```

- Batches can be scheduled using a scheduler class

Future Methods

Used to run Apex asynchronously in its own thread at a later time when system resources become available.

- You use the @future annotation to identify methods that run asynchronously
- Future Methods must be Static
- Can only return Void Type
- Can't call one future method from another
- Can't take Standard or Custom Data Types as arguments (Typiccally, we use 'ID' or 'List' as arguments for processing records)
- Typically used for Callouts to external Web services, Non-Immediate Resource intensive operations.

```
global class utilClass {
  @future
  public static void someFutureMethod(List<Id> recordIds) {
    List<Contact> conts = [Select Id, FirstName, LastName, Email from Contact
Where Id IN :recordIds];
    // process these contact records to do any operation, like sending emails
(ofc, they will be sent at a later time not immediately)
  }
}
```

Queueable Apex

Is Similar to Future Methods but with some extra features.To use Queueable Apex, simply implement the Queueable interface.

- Your Queueable class can contain member variables of non-primitive data types, such as sObjects or custom Apex types.
- Returns an Id similar to Batch apex to Monitor the Async Apex Job.
- Can chain a Queueable Apex from another Queueable Apex.

```
public class SomeClass implements Queueable {
    public void execute(QueueableContext context) {
        // Your Code Logic Here
    }
}
SomeClass updateJob = new SomeClass(); // Create an instance of your class and
pass any arguments if required.
// enqueue the job for processing
```

```
ID jobID = System.enqueueJob(updateJob);
SELECT Id, Status, NumberOfErrors FROM AsyncApexJob WHERE Id = :jobID // Monitor
it
```

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

```
RestRequest req = 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 = new Merchandise__c(
        Name=name,
        Description__c=description,
                Price__c=price,
                Total_Inventory__c=inventory);
   insert m;
        return m.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:pageBlockTable value="{!products}" var="pitem">
    <apex:column headerValue="Product">
        <apex:outputText value="{!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:relatedList list="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:outputLink value="{!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.
    - `<apex:page>`
    - `  <h1>My Fancy Site</h1>`
      `  <apex:insert name="body"/>`
- </apex:page>
- `* Use the template by using the ```<apex:define />``` tag`

- ```html
- <apex:page sidebar="false" showHeader="false">
- <apex:composition template="BasicTemplate">
- <apex:define name="body">
- This is a simple page demonstrating that this
- text is substituted, and that a banner is created.
- </apex:define>
- </apex:composition>
- </apex:page>
    - Another way to include a page is to use the `<apex:include />` tag
- <apex:page sidebar="false" showHeader="false"> Test Before
- <apex:include pageName="MainPage"/> Test After
  </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:outputText value="{0,number,currency}">
    <apex:param value="{!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 a IPaas solution.
        c. Having to build durable and resilient integration solutions inside of Salesforce can be expensive and very complicated

Security

- Authentication
- Authorisation

Salesforce to Siebel

1. Vision
    - Business process transformation
2. KPIS -
    - What gets measured gets done.
3. Define Capabilities
    - Such as Lead Management
    - Priorities of Capabilities
4. Define Retriment Approach
    - Coexitance
    - Methodical
5. Map Entities
    - Don't replicate
6. Design Application
    - Data Model
    - Logic/Workflows
7. Design Integration
8. Change Management
    - Successful transition and high adoption rates
    - Executive sponsorship
    - Training program
9. Implement
10. Rollout
    - Low risk and methodical
    - Reduce business disruption
    - Geographical

Data Migration Framework

1. Right Data Architecture - End to end architecture
2. Legacy Data Analysis - Understand History, Identify Data quality issues. Define Archival, Purge strategy
3. Data Readyness - Standarisation, Enrichment, Transformtion. Conduct one time cleanup
4. Comprehensive Testing and Verfication - Use reports, Test for consitency, usability
5. Training and Deployment

Issues with Migration

- Address Migration - SF is more user friendly
- Households and Party Model - Extend via relationships. Use Lightning connect for insights
- Data should be profiled before moving
- Remove unnecessary picklist values
- Profile the data - Which fields are being used, which picklist values are being used, etc
- Use Data Services in the cloud to clean and enrich the data

Master Data Management

- Present Master Data in a contentual and constent manner
- Single version of truth
- Relationships between various master records. eg. influncers, hierarchies, households
- Events related to the records - Life events, merges, splits
- Why
    - Governace and Compliance
    - Consitent Customer Information
    - Eliminate Duplicate Data
    - Provide a 360 degree view of the customer
    - Tracebility

    - Accurate Reporting

- Patterns of Integration
    - Sync UI Integration - Custom link to MDM to administer data which then sends data back to Salesforce
    - Sync Web Services Integration - Visualforce page which calls seach API and then triggers updates back
    - Async Web Service - Record update in Salesforce triggers a process in MDM to identify and create a record
    - Async Batch Integration - Daily changes synchronized
- How
    - Analytical MDM - Leveraged by the BI team. Data is augmented by insights. Created async.
    - Referential - Hub contains linkage which are created async. Data resides in each system.
    - Operational - All applications directly create data in MDM.
- Implementional Approaches
    - Registry Hub - Light weight, easy to build, governance is a challenge
    - Transactional Hub - Has core attributes, long implementation cycle, single version of truth, Data Governance and Stewardship is built in.
    - Hybrid - In between the two (point in time)
    - Cloud based MDM eg. Data.com, Unified front end, Social Stewardship
- Data Stewardship
    - Process should be easy
    - Automate process through rules
    - Leverage Chatter, approval process etc
    - Mistake proof application for better data capture
- MDM and CRM complement each other. CRM enables MDM to create better master data. MDM provides better visibility to CRM related data.

Platform Development Basics

Platform Development Basics

Reference

- Platform Development Basics

Salesforce Platform

- Like any platform, the Salesforce platform is a group of technologies that supports the development of other technologies on top of it. What makes it unique is that the platform supports not only all the Salesforce clouds, but it also supports custom functionality built by our customers and partners. This functionality ranges from simple page layouts to full-scale applications.

Core Platform

- The core platform lets you develop custom data models and applications for desktop and mobile. And with the platform behind your development, you can build robust systems at a rapid pace.

Heroku Platform

- Heroku gives developers the power to build highly scalable web apps and back-end services using Python, Ruby, Go, and more. It also provides database tools to sync seamlessly with data from Salesforce.

Salesforce APIs

- These let developers integrate and connect all their enterprise data, networks, and identity information.

Mobile SDK

- The Mobile SDK is a suite of technologies that lets you build native, HTML5, and hybrid apps that have the same reliability and security as the Salesforce app.

Metadata-driven development model

- the key differences between developing on the platform and developing outside of Salesforce. Because the platform is metadata-aware, it can auto-generate a significant part of your user experience for you.
- Things like dialogs, record lists, detail views, and forms that you'd normally have to develop by yourself come for free.
- You even get all the functionality to create, read, update, and delete (affectionately referred to as CRUD) custom object records in the database.

Lightning Component Framework

- A UI development framework similar to AngularJS or React.

Apex

- Salesforce's proprietary programming language with Java-like syntax.

Visualforce

- A markup language that lets you create custom Salesforce pages with code that looks a lot like HTML, and optionally can use a powerful combination of Apex and JavaScript.

Data Security

Reference

- Data Security

Levels of Data Access

1. Organization

- For your whole org, you can maintain a list of authorized users, set password policies, and limit logins to certain hours and locations.

2. Objects

- Access to object-level data is the simplest thing to control. By setting permissions on a particular type of object, you can prevent a group of users from creating, viewing, editing, or deleting any records of that object.
- For example, you can use object permissions to ensure that interviewers can view positions and job applications but not edit or delete them.

3. Fields

- You can restrict access to certain fields, even if a user has access to the object.
- For example, you can make the salary field in a position object invisible to interviewers but visible to hiring managers and recruiters.

4. Records

- You can allow particular users to view an object, but then restrict the individual object records they're allowed to see.
- For example, an interviewer can see and edit her own reviews, but not the reviews of other interviewers. You can manage record-level access in these four ways.

Control Access to Orgs

Manage Users

- Every Salesforce user is identified by a username, a password, and a single profile. Together with other settings, the profile determines what tasks users can perform, what data they see, and what they can do with the data.
    - Create User
    - Deactivate a User
    - Set Password Policy
    - Restrict Login Access by IP Address
    - Restrict Login Access by Time

Control Access to Objects

Manage Object Permissions

- The simplest way to control data access is to set permissions on a particular type of object. (An object is a collection of records, like leads or contacts.) You can control whether a group of users can create, view, edit, or delete any records of that object.
- You can set object permissions with profiles or permission sets. A user can have one profile and many permission sets.
    - A user's profile determines the objects they can access and the things they can do with any object record (such as create, read, edit, or delete).
    - Permission sets grant additional permissions and access settings to a user.
- Use profiles to grant the minimum permissions and settings that all users of a particular type need. Then use permission sets to grant more permissions as needed. The combination of profiles and permission sets gives you a great deal of flexibility in specifying object-level access.

Use Profiles to Restrict Access

- Each user has a single profile that controls which data and features that user has access to. A profile is a collection of settings and permissions. Profile settings determine which data the user can see, and permissions determine what the user can do with that data.
    - The settings in a user's profile determine whether she can see a particular app, tab, field, or record type.
    - The permissions in a user's profile determine whether she can create or edit records of a given type, run reports, and customize the app.
- Profiles usually match up with a user's job function (for example, system administrator, recruiter, or hiring manager), but you can have profiles for anything that makes sense for your Salesforce org. A profile can be assigned to many users, but a user can have only one profile at a time.

Standard Profiles

- Read Only
- Standard User
- Marketing User
- Contract Manager
- System Administrator

Permission Sets

- A permission set is a collection of settings and permissions that give users access to various tools and functions. The settings and permissions in permission sets are also found in profiles, but permission sets extend users' functional access without changing their profiles.
- Permission sets make it easy to grant access to the various apps and custom objects in your org, and to take away access when it's no longer needed.
- Users can have only one profile, but they can have multiple permission sets.
- You'll be using permission sets for two general purposes: to grant access to custom objects or apps, and to grant permissions—temporarily or long term—to specific fields.

Control Access to Fields

Modify Field-Level Security

- Defining field-level security for sensitive fields is the second piece of the security and sharing puzzle, after controlling object-level access.

Restrict Field Access with a Profile

- You apply field settings by modifying profiles or permission sets.

Control Access to Records

Record-Level Security

- Record access determines which individual records users can view and edit in each object they have access to in their profile. First ask yourself these questions:
    - Should your users have open access to every record, or just a subset?
    - If it's a subset, what rules should determine whether the user can access them?

- You control record-level access in four ways. They're listed in order of increasing access. You use org-wide defaults to lock down your data to the most restrictive level, and then use the other record-level security tools to grant access to selected users, as required.

i. `Org-wide defaults` - specify the default level of access users have to each other's records.

ii. `Role hierarchies` - ensure managers have access to the same records as their subordinates. Each role in the hierarchy represents a level of data access that a user or group of users needs.

iii. `Sharing rules` - are automatic exceptions to org-wide defaults for particular groups of users, to give them access to records they don't own or can't normally see.

iv. `Manual sharing` - lets record owners give read and edit permissions to users who might not have access to the record any other way.

Role Hierarchy

- A role hierarchy works together with sharing settings to determine the levels of access users have to your Salesforce data. Users can access the data of all the users directly below them in the hierarchy.

Sharing Rules

- This enables you to make automatic exceptions to your org-wide sharing settings for selected sets of users.
- Sharing rules can be based on who owns the record or on the values of fields in the record. For example, use sharing rules to extend sharing access to users in public groups or roles. As with role hierarchies, sharing rules can never be stricter than your org-wide default settings. They just allow greater access for particular users.

Public Group

- Using a public group when defining a sharing rule makes the rule easier to create and, more important, easier to understand later, especially if it's one of many sharing rules that you're trying to maintain in a large organization.
- Create a public group if you want to define a sharing rule that encompasses more than one or two groups or roles, or any individual.

Formulas & Validations

Reference

- Formulas & Validations

Formula Fields

- Fields that have calculated values based on a Formula which is entered in the Formula Editor.
- You can create custom formula fields on any standard or custom object.
- Always use the `Check Syntax` button
    - Check Syntax button in the editor is an important tool for debugging your formulas.
    - The syntax checker tells you what error it encountered and where it's located in your formula.

Roll-Up Summary Fields

- While formula fields calculate values using fields within a single record, roll-up summary fields calculate values from a set of related records, such as those in a related list.

- You can create roll-up summary fields that automatically display a value on a master record based on the values of records in a detail record. These detail records must be directly related to the master through a master-detail relationship

- You can perform different types of calculations with roll-up summary fields. You can count the number of detail records related to a master record, or calculate the sum, minimum value, or maximum value of a field in the detail records. For example, you might want:

    - A custom account field that calculates the total of all related pending opportunities.
    - A custom order field that sums the unit prices of products that contain a description you specify.

- Types of Summaries that can be used

    - `COUNT` - Totals the number of related records.
    - `SUM` - Totals the values in the field you select in the Field to Aggregate option. Only number, currency, and percent fields are available.
    - `MIN` - Displays the lowest value of the field you select in the Field to Aggregate option for all directly related records. Only number, currency, percent, date, and date/time fields are available.

- MAX - Displays the highest value of the field you select in the Field to Aggregate option for all directly related records. Only number, currency, percent, date, and date/time fields are available.

Validation Rules

- Validation rules verify that data entered by users in records meet the standards you specify before they can save it.
- A validation rule can contain a formula or expression that evaluates the data in one or more fields and returns a value of "True" or "False."
- Validation rules can also include error messages to display to users when they enter invalid values based on specified criteria. Using these rules effectively contributes to quality data.
- For example, you can ensure that all phone number fields contain a specified format or that discounts applied to certain products never exceed a defined percentage.

Lightning Flow

- Lightning Flow provides declarative process automation for every Salesforce app, experience, and portal. Included in Lightning Flow are two point-and-click automation tools: Process Builder and Cloud Flow Designer.

Reference

- Lightning Flow

With Process Builder, you build processes. With Cloud Flow Designer, you build flows.

Process Builder

- A simple but versatile tool for automating simple business processes. In Process Builder, you build processes.
- Process Builder is the simplest Lightning Flow tool for the job.
- For a given object, Process Builder lets you control the order of your business processes without any code.

Cloud Flow Designer

- A drag-and-drop tool for building more complex business processes. In the Cloud Flow Designer, you build `flows`.
- In Cloud Flow Designer, you can't define a trigger. For a flow to start automatically, you have to call it from something else, like a process or Apex trigger.

Guided visual experiences

- Business processes that need input from users, whether they're employees or customers.
- `Use Cloud Flow Designer` - If the business process you're automating requires user input, use the Cloud Flow Designer to build a flow. That way, you can provide a rich, guided experience for your users, whether in your Salesforce org or in a Lightning community.

Behind-the-scenes automation

- Business processes that get all the necessary data from your Salesforce org or a connected system. In other words, user input isn't needed.
- `Use Process Builder` Most behind-the-scenes business processes are designed to start automatically when a particular thing occurs. Generally, we refer to that "thing" as a trigger. Business processes can be automatically triggered when:
    - A record changes
    - A platform event message is received
    - A certain amount of time passes or specified time is reached

Approval automation

- Business processes that determine how a record, like a time-off request, gets approved by the right stakeholders.
- To automate your company's business processes for approving records, use Approvals to build an approval process.

In a nutshell: Process Builder vs Cloud Flow Designer vs Approval Automation

- Process Builder and Cloud Flow Designer are Lightning Flow tools. Approval Automation is just an additional tool.
- Process Builder is like a background job which doesn't need human interaction.
- Cloud Flow Designer is like a workflow which may or may not requires human input. (e.g. Screen flows requires human input, Autolaunches flows do not)
- Approval Automation is a process specifically made to request for an approval or sign-off and that requires human interaction.

Process Builder

- Process Builder is a point-and-click tool that lets you easily automate if/then business processes and see a graphical representation of your process as you build.
- Every process consists of a trigger, at least one criteria node, and at least one action. You can configure `immediate actions` or `schedule actions` to be executed at a specific time.
    - Each `immediate action` is executed as soon as the criteria evaluates to true.
    - Each `scheduled action` is executed at the specified time, such as 10 days before the record's close date or 2 days from now.

Process Types

1. Record Change - Process starts when `a record is created or edited`
2. Invocable - Process starts when `it's called by another process`
3. Platform Event - Process starts when `a platform event message is received`

Cloud Flow Designer

- Cloud Flow Designer is a point-and-click tool for building flows.

Flow Building Blocks

1. `Elements` - appear on the canvas. To add an element to the canvas, drag it there from the palette.
2. `Connectors` - define the path that the flow takes at runtime. They tell the flow which element to execute next.
3. `Resources` - are containers that represent a given value, such as field values or formulas. You can reference resources throughout your flow. For example, look up an account's ID, store that ID in a variable, and later reference that ID to update the account.

Flow Elements

1. `Screen` - Display data to your users or collect information from them with Screen elements. You can add simple fields to your screens, like input fields and radio buttons and out-of-the-box Lightning components like File Upload
2. `Logic` - Control the flow of... well, your flow. Create branches, update data, loop over sets of data, or wait for a particular time.
3. `Actions` - Do something in Salesforce when you have the necessary information (perhaps collected from the user via a screen). Flows can look up, create, update, and delete Salesforce records. They can also create Chatter posts, submit records for approval, and send emails. If your action isn't possible out of the box, call Apex code from the flow.
4. `Integrations` - In addition to connecting with external systems by calling Apex code, the Cloud Flow Designer has a couple of tie-ins to platform events. Publish platform event messages with a Record Create element. Subscribe to platform events with a Wait element.

Flow Variables

1. `Variable` - A single value.
2. `sObject Variable` - A set of field values for a single record.
3. `Collection Variable` - Multiple values of the same data type.
4. `sObject Collection Variable`- A set of field values for multiple records that have the same object.

Approvals

- An approval process automates how Salesforce records are approved in your org. In an approval process, you specify:

- The steps necessary for a record to be approved and who approves it at each step.
- The actions to take based on what happens during the approval process.

Apex Basics & Database

References

- Apex Basics & Database
- Apex Developer Guide

What is Apex?

- Apex is a programming language that uses Java-like syntax and acts like database stored procedures.
- Apex enables developers to add business logic to system events, such as button clicks, updates of related records, and Visualforce pages.
- As a language, Apex is:
  - `Hosted` — Apex is saved, compiled, and executed on the server—the Lightning Platform.
  - `Object oriented` — Apex supports classes, interfaces, and inheritance.
  - `Strongly typed` — Apex validates references to objects at compile time.
  - `Multitenant aware` — Because Apex runs in a multitenant platform, it guards closely against runaway code by enforcing limits, which prevent code from monopolizing shared resources.
  - `Integrated with the database` — It is straightforward to access and manipulate records. Apex provides direct access to records and their fields, and provides statements and query languages to manipulate those records.
  - `Data focused` — Apex provides transactional access to the database, allowing you to roll back operations.
  - `Easy to use` — Apex is based on familiar Java idioms.
  - `Easy to test` — Apex provides built-in support for unit test creation, execution, and code coverage. Salesforce ensures that all custom Apex code works as expected by executing all unit tests prior to any platform upgrades.
  - `Versioned` — Custom Apex code can be saved against different versions of the API.

Apex Data Types

Primitives

- Integer
- Double
- Long
- Date
- Datetime
- String
- ID
- Boolean

sObject

- e.g. Account, Contact, or MyCustomObject__c
- can also be a generic sObject

Collections

- List
- Set
- Map

Enum

User-defined Apex classes

System-supplied Apex classes

Data Manipulation Language (DML)

- Create and modify records in Salesforce by using the Data Manipulation Language, abbreviated as DML.
- DML provides a straightforward way to manage records by providing simple statements to insert, update, merge, delete, and restore records.

DML Statements

- insert
- update
- upsert
    - DML operation creates new records and updates sObject records within a single statement, using a specified field to determine the presence of existing objects, or the ID field if no field is specified.
- delete
- undelete
- merge
    - merges up to three records of the same sObject type into one of the records, deleting the others, and re-parenting any related records.

Insert Records

```
// Create the account sObject
Account acct = new Account(Name='Acme', Phone='(415)555-1212',
NumberOfEmployees=100);
// Insert the account by using DML
insert acct;
```

ID Field Auto-Assigned to new Records

```
// Create the account sObject
Account acct = new Account(Name='Acme', Phone='(415)555-1212',
NumberOfEmployees=100);
// Insert the account by using DML
insert acct;
// Get the new ID on the inserted sObject argument
ID acctID = acct.Id;
// Display this ID in the debug log
System.debug('ID = ' + acctID);
// Debug log result (the ID will be different in your case)
// DEBUG|ID = 001D000000JmKkeIAF
```

Bulk DML

- You can perform DML operations either on a single sObject, or in bulk on a list of sObjects.
- Performing bulk DML operations is the recommended way because it helps avoid hitting governor limits, such as the DML limit of 150 statements per Apex transaction.

Upsert Records

```
// Insert the Josh contact
```

```
Contact josh = new
Contact(FirstName='Josh',LastName='Kaplan',Department='Finance');
insert josh;
// Josh's record has been inserted
//   so the variable josh has now an ID
//   which will be used to match the records by upsert
josh.Description = 'Josh\'s record has been updated by the upsert operation.';
// Create the Kathy contact, but don't persist it in the database
Contact kathy = new
Contact(FirstName='Kathy',LastName='Brown',Department='Technology');
// List to hold the new contacts to upsert
List<Contact> contacts = new List<Contact> { josh, kathy };
// Call upsert
upsert contacts;
// Result: Josh is updated and Kathy is created.
```

Upsert Records using idLookup field matching

```
Contact jane = new Contact(FirstName='Jane',
                           LastName='Smith',
                           Email='jane.smith@example.com',
                           Description='Contact of the day');
insert jane;
// 1. Upsert using an idLookup field
// Create a second sObject variable.
// This variable doesn’t have any ID set.
Contact jane2 = new Contact(FirstName='Jane',
                           LastName='Smith',
                           Email='jane.smith@example.com',
                           Description='Prefers to be contacted by email.');
// Upsert the contact by using the idLookup field for matching.
upsert jane2 Contact.fields.Email;
// Verify that the contact has been updated
System.assertEquals('Prefers to be contacted by email.',
                    [SELECT Description FROM Contact WHERE
Id=:jane.Id].Description);
```

Delete Records

```
Contact[] contactsDel = [SELECT Id FROM Contact WHERE LastName='Smith'];
delete contactsDel;
```

DML Exception

```
try {
    // This causes an exception because
    //   the required Name field is not provided.
    Account acct = new Account();
    // Insert the account
    insert acct;
} catch (DmlException e) {
    System.debug('A DML exception has occurred: ' +
                e.getMessage());
}
```

Database Methods

- Apex contains the built-in Database class, which provides methods that perform DML operations and mirror the DML statement counterparts.
    - `Database.insert()`
    - `Database.update()`
    - `Database.upsert()`
    - `Database.delete()`
    - `Database.undelete()`
    - `Database.merge()`
- Unlike DML statements, Database methods have an optional `allOrNone` parameter that allows you to specify whether the operation should partially succeed. When this parameter is set to false, if errors occur on a partial set of records, the successful records will be committed and errors will be returned for the failed records. Also, no exceptions are thrown with the partial success option.

Database Insert with allOrNone set to false

```
Database.insert(recordList, false);
```

- By default, the `allOrNone` parameter is true, which means that the Database method behaves like its DML statement counterpart and will throw an exception if a failure is encountered.

Database.SaveResult

```
Database.SaveResult[] results = Database.insert(recordList, false);
```

- Upsert returns `Database.UpsertResult` objects, and delete returns `Database.DeleteResult` objects.

Database Insert with partial success

```
// Create a list of contacts
List<Contact> conList = new List<Contact> {
        new Contact(FirstName='Joe',LastName='Smith',Department='Finance'),
        new Contact(FirstName='Kathy',LastName='Smith',Department='Technology'),
        new Contact(FirstName='Caroline',LastName='Roth',Department='Finance'),
        new Contact()};

// Bulk insert all contacts with one DML call
Database.SaveResult[] srList = Database.insert(conList, false);
// Iterate through each returned result
for (Database.SaveResult sr : srList) {
    if (sr.isSuccess()) {
        // Operation was successful, so get the ID of the record that was
processed
```

```
        System.debug('Successfully inserted contact. Contact ID: ' + sr.getId());
    } else {
        // Operation failed, so get all errors
        for(Database.Error err : sr.getErrors()) {
            System.debug('The following error has occurred.');
            System.debug(err.getStatusCode() + ': ' + err.getMessage());
            System.debug('Contact fields that affected this error: ' +
err.getFields());
         }
    }
}
```

DML Statements vs Database Methods

- Use DML statements if you want any error that occurs during bulk DML processing to be thrown as an Apex exception that immediately interrupts control flow (by using try. . .catch blocks). This behavior is similar to the way exceptions are handled in most database procedural languages.
- Use Database class methods if you want to allow partial success of a bulk DML operation — if a record fails, the remainder of the DML operation can still succeed. Your application can then inspect the rejected records and possibly retry the operation. When using this form, you can write code that never throws DML exception errors. Instead, your code can use the appropriate results array to judge success or failure. Note that Database methods also include a syntax that supports thrown exceptions, similar to DML statements.

Insert Related Records

```
Account acct = new Account(Name='SFDC Account');
insert acct;
// Once the account is inserted, the sObject will be
// populated with an ID.
// Get this ID.
ID acctID = acct.ID;
// Add a contact to this account.
Contact mario = new Contact(
    FirstName='Mario',
    LastName='Ruiz',
    Phone='415.555.1212',
    AccountId=acctID);
insert mario;
```

Update Related Records

```
// Query for the contact, which has been associated with an account.
Contact queriedContact = [SELECT Account.Name
                          FROM Contact
                          WHERE FirstName = 'Mario' AND LastName='Ruiz'
                          LIMIT 1];
// Update the contact's phone number
```

```
queriedContact.Phone = '(415)555-1213';
// Update the related account industry
queriedContact.Account.Industry = 'Technology';
// Make two separate calls
// 1. This call is to update the contact's phone.
update queriedContact;
// 2. This call is to update the related account's Industry field.
update queriedContact.Account;
```

Delete Related Records

```
Account[] queriedAccounts = [SELECT Id FROM Account WHERE Name='SFDC Account'];
delete queriedAccounts;
```

Account Handler Challenge

```
public class AccountHandler {
        public static Account insertNewAccount(String accountName) {
                try {
                        // Create account using accountName
                        Account acct = new Account(Name=accountName);
                        insert acct;
                        // Return account object
                        return acct;
                }
                catch(DmlException e) {
                        System.debug('A DML exception has occurred: ' +
e.getMessage());
                        return null;
                }
        }
}
```

Salesforce Object Query Language (SOQL)

- used to read saved records. SOQL is similar to the standard SQL language but is customized for the Lightning Platform.
- Because Apex has direct access to Salesforce records that are stored in the database, you can embed SOQL queries in your Apex code and get results in a straightforward fashion. When SOQL is embedded in Apex, it is referred to as inline SOQL.

Inline SOQL

```
Account[] accts = [SELECT Name,Phone FROM Account];
```

Query Editor

- The Developer Console provides the Query Editor console, which enables you to run your SOQL queries and view results. The Query Editor provides a quick way to inspect the database. It is a good way to test your SOQL queries before adding them to your Apex code. When you use the Query Editor, you need to supply only the SOQL statement without the Apex code that surrounds it.

Basic SOQL Syntax

```
SELECT Name,Phone FROM Account
```

SOQL Syntax with WHERE condition

```
SELECT Name,Phone FROM Account WHERE Name='SFDC Computing'

SELECT Name,Phone FROM Account WHERE (Name='SFDC Computing' AND
NumberOfEmployees>25)

SELECT Name,Phone FROM Account WHERE (Name='SFDC Computing' OR
(NumberOfEmployees>25 AND BillingCity='Los Angeles'))
```

SOQL Syntax with ORDER BY clause

```
SELECT Name,Phone FROM Account ORDER BY Name

SELECT Name,Phone FROM Account ORDER BY Name ASC

SELECT Name,Phone FROM Account ORDER BY Name DESC
```

SOQL Syntax with LIMIT clause

```
SELECT Name,Phone FROM Account LIMIT 1
```

SOQL Syntax combined

```
SELECT Name,Phone FROM Account
      WHERE (Name = 'SFDC Computing' AND NumberOfEmployees>25)
      ORDER BY Name
      LIMIT 10
```

SOQL Inline combined

```
Account[] accts = [SELECT Name,Phone FROM Account
      WHERE (Name='SFDC Computing' AND NumberOfEmployees>25)
      ORDER BY Name
      LIMIT 10];
System.debug(accts.size() + ' account(s) returned.');
```

```
// Write all account array info
System.debug(accts);
```

Accessing Variables in SOQL Queries

```
String targetDepartment = 'Wingo';
Contact[] techContacts = [SELECT FirstName,LastName
                          FROM Contact WHERE Department=:targetDepartment];
```

Querying Related Records

```
SELECT Name, (SELECT LastName FROM Contacts) FROM Account WHERE Name = 'SFDC
Computing'
```

Querying Related Records (inline)

```
Account[] acctsWithContacts = [SELECT Name, (SELECT FirstName,LastName FROM
Contacts)
      FROM Account
      WHERE Name = 'SFDC Computing'];
// Get child records
Contact[] cts = acctsWithContacts[0].Contacts;
System.debug('Name of first associated contact: ' + cts[0].FirstName + ', ' +
cts[0].LastName);
```

Traversing a relationship from child object

```
Contact[] cts = [SELECT Account.Name FROM Contact
                 WHERE FirstName = 'Carol' AND LastName='Ruiz'];
Contact carol = cts[0];
String acctName = carol.Account.Name;
System.debug('Carol\'s account name is ' + acctName);
```

Querying Record in Batches By Using SOQL For Loops

```
insert new Account[]{new Account(Name = 'for loop 1'),
                     new Account(Name = 'for loop 2'),
                     new Account(Name = 'for loop 3')};
// The sObject list format executes the for loop once per returned batch
// of records
Integer i=0;
Integer j=0;
for (Account[] tmp : [SELECT Id FROM Account WHERE Name LIKE 'for loop _']) {
    j = tmp.size();
    i++;
}
System.assertEquals(3, j); // The list should have contained the three accounts
                        // named 'yyy'
System.assertEquals(1, i); // Since a single batch can hold up to 200 records and,
                        // only three records should have been returned, the
                        // loop should have executed only once
```

Salesforce Object Search Language (SOSL)

- a Salesforce search language that is used to perform text searches in records. Use SOSL to search fields across multiple standard and custom object records in Salesforce. SOSL is similar to Apache Lucene.

Inline SOSL

```
List<List<SObject>> searchList = [FIND 'SFDC' IN ALL FIELDS
                                  RETURNING Account(Name),
Contact(FirstName,LastName)];
```

SOQL vs SOSL

- Like SOQL, SOSL allows you to search your organization's records for specific information.
- Unlike SOQL, which can only query one standard or custom object at a time, a single SOSL query can search all objects.
- Another difference is that SOSL matches fields based on a word match while SOQL performs an exact match by default (when not using wildcards). For example, searching for 'Digital' in SOSL returns records whose field values are 'Digital' or 'The Digital Company', but SOQL returns only records with field values of 'Digital'.
- Use SOQL to retrieve records for a single object.
- Use SOSL to search fields across multiple objects. SOSL queries can search most text fields on an object.

Query Editor

- The Developer Console provides the Query Editor console, which enables you to run SOSL queries and view results. The Query Editor provides a quick way to inspect the database. It is a good way to test your SOSL queries before adding them to your Apex code. When you use the Query Editor, you need to supply only the SOSL statement without the Apex code that surrounds it.

Basic SOSL Syntax

```
FIND 'SearchQuery' [IN SearchGroup] [RETURNING ObjectsAndFields]
```

- `SearchQuery` is the text to search for (a single word or a phrase). Search terms can be grouped with logical operators (AND, OR) and parentheses. Also, search terms can include wildcard characters (*, ?). The `* wildcard` matches

zero or more characters at the middle or end of the search term. The `?` `wildcard` matches only one character at the middle or end of the search term.

- Text searches are case-insensitive. For example, searching for Customer, customer, or CUSTOMER all return the same results.
- `SearchGroup` is optional. It is the scope of the fields to search. If not specified, the default search scope is all fields. SearchGroup can take one of the following values.
  - ALL FIELDS
  - NAME FIELDS
  - EMAIL FIELDS
  - PHONE FIELDS
  - SIDEBAR FIELDS
- `ObjectsAndFields` is optional. It is the information to return in the search result — a list of one or more sObjects and, within each sObject, list of one or more fields, with optional values to filter against. If not specified, the search results contain the IDs of all objects found.

SOSL Apex Example

```
List<List<sObject>> searchList = [FIND 'Wingo OR SFDC' IN ALL FIELDS
                    RETURNING
Account(Name),Contact(FirstName,LastName,Department)];
Account[] searchAccounts = (Account[])searchList[0];
Contact[] searchContacts = (Contact[])searchList[1];
System.debug('Found the following accounts.');
for (Account a : searchAccounts) {
    System.debug(a.Name);
}
System.debug('Found the following contacts.');
for (Contact c : searchContacts) {
    System.debug(c.LastName + ', ' + c.FirstName);
}
```

SOSL Query with WHERE condition

```
RETURNING Account(Name, Industry WHERE Industry='Apparel')
```

SOSL Query with ORDER BY clause

```
RETURNING Account(Name, Industry ORDER BY Name)
```

SOSL Query with LIMIT clause

```
RETURNING Account(Name, Industry LIMIT 10)
```

Contact and Lead Search Challenge

```
public class ContactAndLeadSearch {
        public static List<List<sObject>> searchContactsAndLeads(String
nameParam) {
                List<List<sObject>> nameList = [FIND :nameParam IN NAME FIELDS


        RETURNING Contact(FirstName,LastName),Lead(Name)];
                System.debug(nameList);
                return nameList;
        }
}
```

Apex Triggers

- Apex triggers enable you to perform custom actions before or after events to records in Salesforce, such as insertions, updates, or deletions. Just like database systems support triggers, Apex provides trigger support for managing records.

Reference

- Apex Trigger
- Triggers
- Invoking Callouts Using Apex
- Apex Callouts

Trigger Syntax

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

- Trigger Events
    - before insert
    - before update
    - before delete
    - after insert
    - after update
    - after delete
    - after undelete

Sample Trigger on Account Creation

```
trigger HelloWorldTrigger on Account (before insert) {
        System.debug('Hello World!');
}
```

- Test the trigger using `Execute Anonymous Window`

```
Account a = new Account(Name='Test Trigger');
insert a;
```

Types of Triggers

`Before triggers`

- are used to update or validate record values before they're saved to the database.

After triggers

- are used to access field values that are set by the system (such as a record's Id or LastModifiedDate field), and to affect changes in other records. The records that fire the after trigger are read-only.

Context Variables

- `Trigger.New` contains all the records that were inserted in insert or update triggers.
- `Trigger.Old` provides the old version of sObjects before they were updated in update triggers, or a list of deleted sObjects in delete triggers.

Iterate over each account in a for loop and update the Description field for each.

```
trigger HelloWorldTrigger on Account (before insert) {
    for(Account a : Trigger.New) {
        a.Description = 'New description';
    }
}
```

- The system saves the records that fired the before trigger after the trigger finishes execution. You can modify the records in the trigger without explicitly calling a DML insert or update operation. If you perform DML statements on those records, you get an error.

Boolean Context Variables

```
trigger ContextExampleTrigger on Account (before insert, after insert, after
delete) {
    if (Trigger.isInsert) {
        if (Trigger.isBefore) {
            // Process before insert
        } else if (Trigger.isAfter) {
            // Process after insert
        }
    }
    else if (Trigger.isDelete) {
        // Process after delete
    }
}
```

List of Context Variables

- `isExecuting` Returns true if the current context for the Apex code is a trigger, not a Visualforce page, a Web service, or an executeanonymous() API call.
- `isInsert` Returns true if this trigger was fired due to an insert operation, from the Salesforce user interface, Apex, or the API.
- `isUpdate` Returns true if this trigger was fired due to an update operation, from the Salesforce user interface, Apex, or the API.
- `isDelete` Returns true if this trigger was fired due to a delete operation, from the Salesforce user interface, Apex, or the API.
- `isBefore` Returns true if this trigger was fired before any record was saved.
- `isAfter` Returns true if this trigger was fired after all records were saved.
- `isUndelete` Returns true if this trigger was fired after a record is recovered from the Recycle Bin (that is, after an undelete operation from the Salesforce user interface, Apex, or the API.)
- `new` Returns a list of the new versions of the sObject records.
    - This sObject list is only available in insert, update, and undelete triggers, and the records can only be modified in before triggers.

`newMap` A map of IDs to the new versions of the sObject records.

- This map is only available in before update, after insert, after update, and after undelete triggers.

`old` Returns a list of the old versions of the sObject records.

- This sObject list is only available in update and delete triggers.

`oldMap` A map of IDs to the old versions of the sObject records.

- This map is only available in update and delete triggers.

`size` The total number of records in a trigger invocation, both old and new.

Calling a Class Method from a Trigger

```
trigger ExampleTrigger on Contact (after insert, after delete) {
    if (Trigger.isInsert) {
        Integer recordCount = Trigger.New.size();
        // Call a utility method from another class
        EmailManager.sendMail('Your email address', 'Trailhead Trigger Tutorial',
                    recordCount + ' contact(s) were inserted.');
    }
    else if (Trigger.isDelete) {
        // Process after delete
    }
}
```

Adding Related Records from a Trigger

- This version will be improved later (down below)

```
trigger AddRelatedRecord on Account(after insert, after update) {
    List<Opportunity> oppList = new List<Opportunity>();

    // Get the related opportunities for the accounts in this trigger
    Map<Id,Account> acctsWithOpps = new Map<Id,Account>(
        [SELECT Id,(SELECT Id FROM Opportunities) FROM Account WHERE Id IN
:Trigger.New]);

    // Add an opportunity for each account if it doesn't already have one.
    // Iterate through each account.
    for(Account a : Trigger.New) {
        System.debug('acctsWithOpps.get(a.Id).Opportunities.size()=' +
acctsWithOpps.get(a.Id).Opportunities.size());
        // Check if the account already has a related opportunity.
        if (acctsWithOpps.get(a.Id).Opportunities.size() == 0) {
            // If it doesn't, add a default opportunity
            oppList.add(new Opportunity(Name=a.Name + ' Opportunity',
                                        StageName='Prospecting',
                                        CloseDate=System.today().addMonths(1),
                                        AccountId=a.Id));
        }
    }
    if (oppList.size() > 0) {
        insert oppList;
    }
}
```

Trigger Exception

```
trigger AccountDeletion on Account (before delete) {

    // Prevent the deletion of accounts if they have related opportunities.
    for (Account a : [SELECT Id FROM Account
                      WHERE Id IN (SELECT AccountId FROM Opportunity) AND
                      Id IN :Trigger.old]) {
        Trigger.oldMap.get(a.Id).addError(
            'Cannot delete account with related opportunities.');
```

```
    }
}
```

Triggers and Callouts

- Apex allows you to make calls to and integrate your Apex code with external Web services. Apex calls to external Web services are referred to as callouts.
- For example, you can make a callout to a stock quote service to get the latest quotes.
- When making a callout from a trigger, the callout must be done asynchronously so that the trigger process doesn't block you from working while waiting for the external service's response.The asynchronous callout is made in a background process, and the response is received when the external service returns it.

Sample Callout

```
public class CalloutClass {
    @future(callout=true)
    public static void makeCallout() {
        HttpRequest request = new HttpRequest();
        // Set the endpoint URL.
        String endpoint = 'http://yourHost/yourService';
        request.setEndPoint(endpoint);
        // Set the HTTP verb to GET.
        request.setMethod('GET');
        // Send the HTTP request and get the response.
        HttpResponse response = new HTTP().send(request);
    }
}
```

Sample Trigger for the Callout

```
trigger CalloutTrigger on Account (before insert, before update) {
    CalloutClass.makeCallout();
}
```

Match Billing Postal Code challenge

```
trigger AccountAddressTrigger on Account (before insert, before update) {
    for(Account acct : Trigger.New) {
        // If match billing address is checked
        if(acct.Match_Billing_Address__c) {
                // Set the shipping postal code to be same as billing postal code
                acct.ShippingPostalCode = acct.BillingPostalCode;
        }
    }
}
```

Bulk Apex Triggers

- When you use bulk design patterns, your triggers have better performance, consume less server resources, and are less likely to exceed platform limits.
- The benefit of bulkifying your code is that bulkified code can process large numbers of records efficiently and run within governor limits on the Lightning Platform. These governor limits are in place to ensure that runaway code doesn't monopolize resources on the multitenant platform.

Not Bulk

```
trigger MyTriggerNotBulk on Account(before insert) {
    Account a = Trigger.New[0];
    a.Description = 'New description';
}
```

Bulk (which is better)

```
trigger MyTriggerBulk on Account(before insert) {
    for(Account a : Trigger.New) {
        a.Description = 'New description';
    }
}
```

Bulk SOQL Queries

- Use this to avoid Query limits which are 100 SOQL queries for synchronous Apex or 200 for asynchronous Apex.

Bad Practice

```
trigger SoqlTriggerNotBulk on Account(after update) {
    for(Account a : Trigger.New) {
        // Get child records for each account
        // Inefficient SOQL query as it runs once for each account!
        Opportunity[] opps = [SELECT Id,Name,CloseDate
                              FROM Opportunity WHERE AccountId=:a.Id];

        // Do some other processing
    }
}
```

Good Practice

- Store to a list first before iteration

```
trigger SoqlTriggerBulk on Account(after update) {
    // Perform SOQL query once.
    // Get the accounts and their related opportunities.
    List<Account> acctsWithOpps =
        [SELECT Id,(SELECT Id,Name,CloseDate FROM Opportunities)
         FROM Account WHERE Id IN :Trigger.New];

    // Iterate over the returned accounts
    for(Account a : acctsWithOpps) {
        Opportunity[] relatedOpps = a.Opportunities;
        // Do some other processing
    }
}
```

Alternative, when Account parent records are not needed

```
trigger SoqlTriggerBulk on Account(after update) {
    // Perform SOQL query once.
    // Get the related opportunities for the accounts in this trigger.
    List<Opportunity> relatedOpps = [SELECT Id,Name,CloseDate FROM Opportunity
        WHERE AccountId IN :Trigger.New];

    // Iterate over the related opportunities
    for(Opportunity opp : relatedOpps) {
        // Do some other processing
    }
}
```

Improve the Alternative with a For-loop in one statement

```
trigger SoqlTriggerBulk on Account(after update) {
    // Perform SOQL query once.
    // Get the related opportunities for the accounts in this trigger,
    // and iterate over those records.
    for(Opportunity opp : [SELECT Id,Name,CloseDate FROM Opportunity
        WHERE AccountId IN :Trigger.New]) {

        // Do some other processing
    }
}
```

- Triggers execute on batches of 200 records at a time. So if 400 records cause a trigger to fire, the trigger fires twice, once for each 200 records. For this reason, you don't get the benefit of SOQL for loop record batching in triggers, because triggers batch up records as well. The SOQL for loop is called twice in this example, but a standalone SOQL query would also be called twice. However, the SOQL for loop still looks more elegant than iterating over a collection variable!

Bulk DML

- Use this because the Apex runtime allows up to 150 DML calls in one transaction.

Bad Practice

```
trigger DmlTriggerNotBulk on Account(after update) {
    // Get the related opportunities for the accounts in this trigger.
    List<Opportunity> relatedOpps = [SELECT Id,Name,Probability FROM Opportunity
        WHERE AccountId IN :Trigger.New];
    // Iterate over the related opportunities
    for(Opportunity opp : relatedOpps) {
        // Update the description when probability is greater
        // than 50% but less than 100%
        if ((opp.Probability >= 50) && (opp.Probability < 100)) {
            opp.Description = 'New description for opportunity.';
            // Update once for each opportunity -- not efficient!
            update opp;
        }
    }
}
```

Good Practice

- Store records to a list first before updating. Don't update per record.

```
trigger DmlTriggerBulk on Account(after update) {
    // Get the related opportunities for the accounts in this trigger.
    List<Opportunity> relatedOpps = [SELECT Id,Name,Probability FROM Opportunity
        WHERE AccountId IN :Trigger.New];

    List<Opportunity> oppsToUpdate = new List<Opportunity>();
    // Iterate over the related opportunities
    for(Opportunity opp : relatedOpps) {
        // Update the description when probability is greater
        // than 50% but less than 100%
        if ((opp.Probability >= 50) && (opp.Probability < 100)) {
            opp.Description = 'New description for opportunity.';
            oppsToUpdate.add(opp);
        }
    }

    // Perform DML on a collection
    update oppsToUpdate;
}
```

Adding Related Records from a Trigger (IMPROVED)

- Instead of checking an opportunities list size for each record in the loop, add the condition in the main query.

```
trigger AddRelatedRecord on Account(after insert, after update) {
```

```
    List<Opportunity> oppList = new List<Opportunity>();

    // Add an opportunity for each account if it doesn't already have one.
    // Iterate over accounts that are in this trigger but that don't have
opportunities.
    for (Account a : [SELECT Id,Name FROM Account
                      WHERE Id IN :Trigger.New AND
                      Id NOT IN (SELECT AccountId FROM Opportunity)]) {
        // Add a default opportunity for this account
        oppList.add(new Opportunity(Name=a.Name + ' Opportunity',
                                    StageName='Prospecting',
                                    CloseDate=System.today().addMonths(1),
                                    AccountId=a.Id));
    }

    if (oppList.size() > 0) {
        insert oppList;
    }
}
```

Solution to Closed Opportunity Trigger challenge

```
trigger ClosedOpportunityTrigger on Opportunity (after insert, after update) {
        List<Task> taskList = new List<Task>();

        // Check Opportunity.StageName if equal to 'Closed Won'
        for(Opportunity o : [SELECT Id FROM Opportunity

                             WHERE Id IN :Trigger.New

                                    AND StageName = 'Closed Won']) {

                // Create a task with subject 'Follow Up Test Task'
                // Associate task with Opportunity - Fill the 'WhatId' field with
the opportunity ID
                taskList.add(new Task(Subject = 'Follow Up Test Task',

                                      WhatId = o.Id));
        }

        if(taskList.size() > 0) {
                insert taskList;
        }
}
```

Apex Testing

- The Apex testing framework enables you to write and execute tests for your Apex classes and triggers on the Lightning Platform platform. Apex unit tests ensure high quality for your Apex code and let you meet requirements for deploying Apex.

Reference

- Apex Testing

Code Coverage Requirement for Deployment

- Before you can deploy your code or package it for the Lightning Platform AppExchange, at least 75% of Apex code must be covered by tests, and all those tests must pass.
- In addition, each trigger must have some coverage. Even though code coverage is a requirement for deployment, don't write tests only to meet this requirement. Make sure to test the common use cases in your app, including positive and negative test cases, and bulk and single-record processing.

Test Method Syntax

```
@isTest static void testName() {
    // code_block
}
```

or

```
static testMethod void testName() {
    // code_block
}
```

- Using the `isTest annotation` instead of the `testMethod keyword` is more flexible as you can specify parameters in the annotation.
- Visibility of a `test method` doesn't matter, so declaring a test method as public or private doesn't make a difference.
- `Test classes` can be either private or public. If you're using a test class for unit testing only, declare it as private. Public test classes are typically used for test data factory classes.

```
@isTest
private class MyTestClass {
    @isTest static void myTest() {
        // code_block
    }
}
```

Temperature Converter

```
public class TemperatureConverter {
```

```
    // Takes a Fahrenheit temperature and returns the Celsius equivalent.
    public static Decimal FahrenheitToCelsius(Decimal fh) {
        Decimal cs = (fh - 32) * 5/9;
        return cs.setScale(2);
    }
}
```

Temperature Converter Test Class

```
@isTest
private class TemperatureConverterTest {
                @isTest static void testWarmTemp() {
        Decimal celsius = TemperatureConverter.FahrenheitToCelsius(70);
        System.assertEquals(21.11,celsius);
    }

    @isTest static void testFreezingPoint() {
        Decimal celsius = TemperatureConverter.FahrenheitToCelsius(32);
        System.assertEquals(0,celsius);
    }

    @isTest static void testBoilingPoint() {
        Decimal celsius = TemperatureConverter.FahrenheitToCelsius(212);
        System.assertEquals(100,celsius,'Boiling point temperature is not
expected.');
    }

    @isTest static void testNotBoilingPoint() {
        Decimal celsius = TemperatureConverter.FahrenheitToCelsius(212);
        // Simulate failure
        System.assertNotEquals(0,celsius,'Boiling point temperature is not
expected.');
    }

    @isTest static void testNegativeTemp() {
        Decimal celsius = TemperatureConverter.FahrenheitToCelsius(-10);
        System.assertEquals(-23.33,celsius);
    }
}
```

Increase Your Code Coverage

- When writing tests, try to achieve the highest code coverage possible. Don't just aim for 75% coverage, which is the lowest coverage that the Lightning Platform platform requires for deployments and packages.

Sample TaskUtil Class

```
public class TaskUtil {
    public static String getTaskPriority(String leadState) {
        // Validate input
        if (String.isBlank(leadState) || leadState.length() > 2) {
            return null;
```

```
        }

        String taskPriority;

        if (leadState == 'CA') {
             taskPriority = 'High';
        } else {
             taskPriority = 'Normal';
        }

        return taskPriority;
    }
}
```

- Note: The equality operator (==) performs case-insensitive string comparisons, so there is no need to convert the string to lower case first. This means that passing in 'ca' or 'Ca' will satisfy the equality condition with the string literal 'CA'.

Sample test for TaskUtil

Not 100% Coverage

```
@isTest
private class TaskUtilTest {
    @isTest static void testTaskPriority() {
        String pri = TaskUtil.getTaskPriority('NY');
        System.assertEquals('Normal', pri);
    }
}
```

100% Coverage

```
@isTest
private class TaskUtilTest {
    @isTest static void testTaskPriority() {
        String pri = TaskUtil.getTaskPriority('NY');
        System.assertEquals('Normal', pri);
    }

    @isTest static void testTaskHighPriority() {
        String pri = TaskUtil.getTaskPriority('CA');
        System.assertEquals('High', pri);
    }

    @isTest static void testTaskPriorityInvalid() {
        String pri = TaskUtil.getTaskPriority('Montana');
        System.assertEquals(null, pri);
    }
}
```

Solution to Verify Date Challenge

- VerifyDate class

```
public class VerifyDate {

        //method to handle potential checks against two dates
        public static Date CheckDates(Date date1, Date date2) {
                //if date2 is within the next 30 days of date1, use date2.
Otherwise use the end of the month
                if(DateWithin30Days(date1,date2)) {
                        return date2;
                } else {
                        return SetEndOfMonthDate(date1);
                }
        }

        //method to check if date2 is within the next 30 days of date1
        private static Boolean DateWithin30Days(Date date1, Date date2) {
                //check for date2 being in the past
        if( date2 < date1) { return false; }

        //check that date2 is within (>=) 30 days of date1
        Date date30Days = date1.addDays(30); //create a date 30 days away from
date1
                if( date2 >= date30Days ) { return false; }
                else { return true; }
        }

        //method to return the end of the month of a given date
        private static Date SetEndOfMonthDate(Date date1) {
                Integer totalDays = Date.daysInMonth(date1.year(),
date1.month());
                Date lastDay = Date.newInstance(date1.year(), date1.month(),
totalDays);
                return lastDay;
        }

}
```

- TestVerifyDate class

```
@istest
private class TestVerifyDate {

        // When CheckDates() method is a Date1 within Date2
        @istest static void testCheckDatesWhenWithinDate1() {
                Date given1 = Date.parse('01/01/2018');
                Date given2 = Date.parse('20/01/2018');
                Date expected = Date.parse('20/01/2018');
                Date result = VerifyDate.CheckDates(given1, given2);
                System.assertEquals(result, expected);
        }

        // When CheckDates() method is a Date1 not within Date2
        @istest static void testCheckDatesWhenNotWithinDate1() {
                Date given1 = Date.parse('01/01/2018');
```

```
                Date given2 = Date.parse('15/02/2018');
                Date expected = Date.parse('31/01/2018');
                Date result = VerifyDate.CheckDates(given1, given2);
                System.assertEquals(result, expected);
        }
}
```

Test Apex Triggers

- Before deploying a trigger, write unit tests to perform the actions that fire the trigger and verify expected results.

Account Deletion trigger

- AccountDelation class

```
trigger AccountDeletion on Account (before delete) {

    // Prevent the deletion of accounts if they have related contacts.
    for (Account a : [SELECT Id FROM Account
                      WHERE Id IN (SELECT AccountId FROM Opportunity) AND
                      Id IN :Trigger.old]) {
        Trigger.oldMap.get(a.Id).addError(
            'Cannot delete account with related opportunities.');
    }

}
```

- TestAccountDeletion class

```
@isTest
private class TestAccountDeletion {
    @isTest static void TestDeleteAccountWithOneOpportunity() {
        // Test data setup
        // Create an account with an opportunity, and then try to delete it
        Account acct = new Account(Name='Test Account');
        insert acct;
        Opportunity opp = new Opportunity(Name=acct.Name + ' Opportunity',
                                          StageName='Prospecting',
                                          CloseDate=System.today().addMonths(1),
                                          AccountId=acct.Id);
        insert opp;

        // Perform test
        Test.startTest();
        Database.DeleteResult result = Database.delete(acct, false);
        Test.stopTest();
        // Verify
        // In this case the deletion should have been stopped by the trigger,
        // so verify that we got back an error.
        System.assert(!result.isSuccess());
```

```
        System.assert(result.getErrors().size() > 0);
        System.assertEquals('Cannot delete account with related opportunities.',
                             result.getErrors()[0].getMessage());
    }

}
```

`Test.startTest()` and `Test.stopTest()`

- The test method contains the Test.startTest() and Test.stopTest() method pair, which delimits a block of code that gets a fresh set of governor limits. In this test, test-data setup uses two DML statements before the test is performed.
- The `startTest` method marks the point in your test code when your test actually begins. Each test method is allowed to call this method `only once`. All of the code before this method should be used to initialize variables, populate data structures, and so on, allowing you to set up everything you need to run your test. Any code that executes after the call to startTest and before stopTest is assigned a new set of governor limits.

Solution to Restrict Contact By Name Challenge

- RestrictContactByName class

```
trigger RestrictContactByName on Contact (before insert, before update) {
        //check contacts prior to insert or update for invalid data
        For (Contact c : Trigger.New) {
                if(c.LastName == 'INVALIDNAME') {//invalidname is invalid

                        // Note: Commented because this not provide data for
e.getDmlFields in actual test
                        //c.AddError('The Last Name "'+c.LastName+'" is not
allowed for DML');

                        // Fix: e.getDmlFields[0](0) is now Contact.LastName
                        c.LastName.AddError('The Last Name "'+c.LastName+'" is
not allowed for DML');
                }

        }
}
```

- TestRestrictContactByName class

```
@istest private class TestRestrictContactByName {
        @istest static void CheckInvalidLastName() {
                // Setup
                String givenLastName = 'INVALIDNAME';
                Contact givenContact = new Contact(LastName=givenLastName);

                // Perform test
                try {
                        insert givenContact;
                }
                catch(DmlException e) {
```

```
                        //Assert Error Message
                        System.assert(
                                e.getMessage().contains('The Last Name
"'+givenLastName+'" is not allowed for DML'),
                                e.getMessage()
                        );

                        //Assert Field
                System.assertEquals(Contact.LastName, e.getDmlFields(0)[0]);

                        //Assert Status Code
                  System.assertEquals('FIELD_CUSTOM_VALIDATION_EXCEPTION',
e.getDmlStatusCode(0));
                }
        }

        @istest static void CheckValidLastName() {
                // Setup
                String givenLastName = 'Smith';
                Contact givenContact = new Contact(LastName=givenLastName);

                // Perform test
                try {
                        Test.startTest();
                        insert givenContact;
                        Contact[] resultContacts = [SELECT LastName FROM Contact
WHERE LastName=:givenLastName];
                        Test.stopTest();

                        // Verify
                        System.assertEquals(resultContacts[0].LastName,
givenLastName);
                }
                catch(DmlException e) {
                        //Assert Error Message
                        System.assert(
                                String.isEmpty(e.getMessage()),
                                e.getMessage()
                        );
                }
        }
}
```

Create Test Data for Apex Tests

- Use test utility classes to add reusable methods for test data setup.

TestDataFactory class

```
@isTest
public class TestDataFactory {
    public static List<Account> createAccountsWithOpps(Integer numAccts, Integer
numOppsPerAcct) {
```

```
        List<Account> accts = new List<Account>();

        for(Integer i=0;i<numAccts;i++) {
            Account a = new Account(Name='TestAccount' + i);
            accts.add(a);
        }
        insert accts;

        List<Opportunity> opps = new List<Opportunity>();
        for (Integer j=0;j<numAccts;j++) {
            Account acct = accts[j];
            // For each account just inserted, add opportunities
            for (Integer k=0;k<numOppsPerAcct;k++) {
                opps.add(new Opportunity(Name=acct.Name + ' Opportunity ' + k,
                                         StageName='Prospecting',
                                         CloseDate=System.today().addMonths(1),
                                         AccountId=acct.Id));
            }
        }
        // Insert all opportunities for all accounts.
        insert opps;

        return accts;
    }
}
```

Modify TestAcccountDeletion class

```
@isTest
private class TestAccountDeletion {
    @isTest static void TestDeleteAccountWithOneOpportunity() {
        // Test data setup
        // Create one account with one opportunity by calling a utility
method
        Account[] accts = TestDataFactory.createAccountsWithOpps(1,1);

        // Perform test
        Test.startTest();
        Database.DeleteResult result = Database.delete(accts[0], false);
        Test.stopTest();
        // Verify
        // In this case the deletion should have been stopped by the trigger,
        // so verify that we got back an error.
        System.assert(!result.isSuccess());
        System.assert(result.getErrors().size() > 0);
        System.assertEquals('Cannot delete account with related opportunities.',
                             result.getErrors()[0].getMessage());
    }

    @isTest static void TestDeleteAccountWithNoOpportunities() {
        // Test data setup
        // Create one account with no opportunities by calling a utility method
        Account[] accts = TestDataFactory.createAccountsWithOpps(1,0);

        // Perform test
        Test.startTest();
        Database.DeleteResult result = Database.delete(accts[0], false);
        Test.stopTest();
```

```
        // Verify that the deletion was successful
        System.assert(result.isSuccess(), result);
    }

    @isTest static void TestDeleteBulkAccountsWithOneOpportunity() {
        // Test data setup
        // Create accounts with one opportunity each by calling a utility method
        Account[] accts = TestDataFactory.createAccountsWithOpps(200,1);

        // Perform test
        Test.startTest();
        Database.DeleteResult[] results = Database.delete(accts, false);
        Test.stopTest();
        // Verify for each record.
        // In this case the deletion should have been stopped by the trigger,
        // so check that we got back an error.
        for(Database.DeleteResult dr : results) {
            System.assert(!dr.isSuccess());
            System.assert(dr.getErrors().size() > 0);
            System.assertEquals('Cannot delete account with related
opportunities.',
                                dr.getErrors()[0].getMessage());
        }
    }

    @isTest static void TestDeleteBulkAccountsWithNoOpportunities() {
        // Test data setup
        // Create accounts with no opportunities by calling a utility method
        Account[] accts = TestDataFactory.createAccountsWithOpps(200,0);

        // Perform test
        Test.startTest();
        Database.DeleteResult[] results = Database.delete(accts, false);
        Test.stopTest();
        // For each record, verify that the deletion was successful
        for(Database.DeleteResult dr : results) {
            System.assert(dr.isSuccess());
        }
    }

}
```

Solution to RandomContactFactory challenge

```
public class RandomContactFactory {
        public static List<Contact> generateRandomContacts(Integer num, String
name) {
                Contact[] contactList = new List<Contact>();
                for(Integer i=0; i<num; i++) {
                        contactList.add(new Contact(FirstName=name + ' ' + i));
                }
                return contactList;
        }
}
```

Developer Console

- The Developer Console is an integrated development environment (more typically called an IDE) where you can create, debug, and test apps in your org.
- The Developer Console is connected to one org and is browser-based.

Reference

- Developer Console Basics

Workspaces

- Workspaces in the Developer Console help you organize information to show just what you need as you work on each of your development tasks. Although it sounds like a fancy term, a workspace is simply a collection of resources, represented by tabs, in the main panel of the Developer Console. You can create a workspace for any group of resources that you use together.
- Workspaces reduce clutter and make it easier to navigate between different resources.

Navigate/Edit Source Codes

Create and Execute Apex Class

- Apex is a Salesforce programming language that you can use to customize business processes in your org.
- File | New | Apex Class

```
public class EmailMissionSpecialist {
   // Public method
   public void sendMail(String address, String subject, String body) {
      // Create an email message object
      Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();
      String[] toAddresses = new String[] {address};
      mail.setToAddresses(toAddresses);
      mail.setSubject(subject);
      mail.setPlainTextBody(body);
      // Pass this email message to the built-in sendEmail method
      // of the Messaging class
      Messaging.SendEmailResult[] results = Messaging.sendEmail(
                                 new Messaging.SingleEmailMessage[] { mail });

      // Call a helper method to inspect the returned results
      inspectResults(results);
   }
```

```
   // Helper method
   private static Boolean inspectResults(Messaging.SendEmailResult[] results) {
      Boolean sendResult = true;
      // sendEmail returns an array of result objects.
      // Iterate through the list to inspect results.
      // In this class, the methods send only one email,
      // so we should have only one result.
      for (Messaging.SendEmailResult res : results) {
         if (res.isSuccess()) {
            System.debug('Email sent successfully');
         }
         else {
            sendResult = false;
            System.debug('The following errors occurred: ' + res.getErrors());
         }
      }
      return sendResult;
   }
}
```

- Debug | Open Execute Anonymous Window

```
EmailMissionSpecialist em = new EmailMissionSpecialist();
em.sendMail('Enter your email address', 'Flight Path Change',
   'Mission Control 123: Your flight path has been changed to avoid collision '
   + 'with asteroid 2014 QO441.');
```

Create Lightning Components

- Lightning Components is a framework for developing mobile and desktop apps. You can use it to create responsive user interfaces for Lightning Platform apps. Lightning components also make it easier to build apps that work well across mobile and desktop devices.
- Using the Developer Console, you can create a `Lightning component bundle`. A component bundle acts like a folder in that it contains components and all other related resources, such as style sheets, controllers, and design.
- To create lightning components, File | New | Lightning Component
- To open lightning components, File | Open Lightning Resources

Create Visualforce Pages and Components

- Visualforce is a web development framework for building sophisticated user interfaces for mobile and desktop apps. These interfaces are hosted on the Lightning Platform platform. Your user interface can look like the standard Salesforce interface, or you can customize it.
- To create a visual force page, File | New | Visualforce Page

- To open a visual force page, File | Open | Entity Type > Pages

---

Generate/Analyze Logs

- Logs are one of the best places to identify problems with a system or program. Using the Developer Console, you can look at various debug logs to understand how your code works and to identify any performance issues.

Two Ways to open debug logs

- `Before execution` - enable Open Log in the Enter Apex Code window. The log opens after your code has been executed.
- `After execution` - double-click the log that appears in the Logs tab.

Debug Log columns

1. `Timestamp` — The time when the event occurred. The timestamp is always in the user's time zone and in HH:mm:ss:SSS format.
2. `Event` — The event that triggered the debug log entry. For instance, in the execution log that you generated, the FATAL_ERROR event is logged when the email address is determined to be invalid.
3. `Details` — Details about the line of code and the method name where the code was executed.

Log Inspector

- The handy Log Inspector exists to make it easier to view large logs! The Log Inspector uses log panel views to provide different perspectives of your code. - To open, Debug | View Log Panels

Log Inspector Panels

1. `Stack Tree` — Displays log entries within the hierarchy of their objects and their execution using a top-down tree view. For instance, if one class calls a second class, the second class is shown as the child of the first.
2. `Execution Stack` — Displays a bottom-up view of the selected item. It displays the log entry, followed by the operation that called it.
3. `Execution Log` — Displays every action that occurred during the execution of your code.

4. `Source` — Displays the contents of the source file, indicating the line of code being run when the selected log entry was generated.
5. `Source List` — Displays the context of the code being executed when the event was logged. For example, if you select the log entry generated when the faulty email address value was entered, the Source List shows execute_anonymous_apex.
6. `Variables` — Displays the variables and their assigned values that were in scope when the code that generated the selected log entry was run.
7. `Execution Overview` — Displays statistics for the code being executed, including the execution time and heap size.

Perspective Manager

- A perspective is a layout of grouped panels. For instance, the predefined Debug perspective displays the Execution Log, Source, and Variables, while the Analysis perspective displays the Stack Tree, Execution Log, Execution Stack, and Execution Overview.
- To choose a perspective, Debug | Switch Perspectives or Debug | Perspective Manager

Log Categories

- `ApexCode`, which logs events related to Apex code and includes information about the start and end of an Apex method.
- `Database`, which includes logs related to database events, including Database Manipulation Language (DML), SOSL, and SOQL queries.

Log Levels

- Log levels control how much detail is logged for each log category.
- From the least amount of data logged (level = NONE) to the most (level = FINEST).
    - NONE
    - ERROR
    - WARN
    - INFO
    - DEBUG
    - FINE
    - FINER
    - FINEST

- To change, Debug | Change Log Levels

Inspect Objects at Checkpoints

- Checkpoints show you snapshots of what's happening in your Apex code at particular points during execution.
- You can set checkpoints only when the Apex log level is set to `FINEST`.

Checkpoint Inspector

- The Checkpoint Inspector has two tabs: Heap and Symbols.
    i. `Heap` — Displays all objects present in memory at the line of code where your checkpoint was executed.
    ii. `Symbols` — Displays all symbols in memory in tree view.

Execute SOQL and SOSL Queries

- SOQL and SOSL Queries can also be executed using the Anonymous Execute Window, Debug | Open Execute Anonymous Window

Visualforce Basics

- `Visualforce` is a web development framework that enables developers to build sophisticated, custom user interfaces for mobile and desktop apps that can be hosted on the Lightning Platform platform. You can use Visualforce to build apps that align with the styling of Lightning Experience, as well as your own completely custom interface.

Reference

- Visualforce Basics

Example Visualforce Page

```
<apex:page standardController="Contact" >
    <apex:form >

        <apex:pageBlock title="Edit Contact">
            <apex:pageBlockSection columns="1">
                <apex:inputField value="{!Contact.FirstName}"/>
                <apex:inputField value="{!Contact.LastName}"/>
                <apex:inputField value="{!Contact.Email}"/>
                <apex:inputField value="{!Contact.Birthdate}"/>
            </apex:pageBlockSection>
            <apex:pageBlockButtons >
                <apex:commandButton action="{!save}" value="Save"/>
            </apex:pageBlockButtons>
        </apex:pageBlock>

    </apex:form>
</apex:page>
```

Create & Edit Visualforce Pages

- Visualforce pages are basic building blocks for application developers.
- A Visualforce page is similar to a standard Web page, but includes powerful features to access, display, and update your organization's data. Pages can be referenced and invoked via a unique URL, just as they would be on a traditional web server.
- Visualforce uses a tag-based markup language that's similar to HTML. Each Visualforce tag corresponds to a coarse or fine-grained user interface component, such as a section of a page, a list view, or an individual field.
- Visualforce boasts nearly 150 built-in components, and provides a way for developers to create their own components.
- Visualforce markup can be freely mixed with HTML markup, CSS styles, and JavaScript libraries, giving you considerable flexibility in how you implement your app's user interface.

Hello World

```
<apex:page>
    <h1>Hello World</h1>
</apex:page>
```

sidebar and showHeader options have no effect in lightning experience

```
<apex:page sidebar="false" showHeader="false">
    <h1>Hello World</h1>
    <p>
        This is my first Visualforce Page!
    </p>
</apex:page>
```

Removing standard Salesforce Stylesheets (CSS)

```
<apex:page standardStylesheets="false">
    <h1>Hello World</h1>
    <p>
        This is my first Visualforce Page!
    </p>
</apex:page>
```

Page Blocks and Page Block Sections

- You cannot use page blocks sections without parent page blocks.

```
<apex:page>
    <h1>Hello World</h1>

    <apex:pageBlock title="A Block Title">
        <apex:pageBlockSection title="A Section Title">
            I'm three components deep!
        </apex:pageBlockSection>

        <apex:pageBlockSection title="A New Section">
            This is another section.
        </apex:pageBlockSection>
    </apex:pageBlock>
</apex:page>
```

Two other ways to build Visualforce pages

- The development mode `quick fix` and footer is a fast way to quickly create a new page, or make short edits to an existing page. It's great for quick changes or when you want to create a short page to test out some new code on a blank slate, before incorporating it into your app pages.
- The `Setup editor`, available in Setup by entering Visualforce Pages in the Quick Find box, then selecting `Visualforce Pages`, is the most basic editor, but it provides access to page settings that aren't available in the Developer Console or development mode footer.

Solution to Visualforce page challenge

```
<apex:page showHeader="false">
    <apex:image id="salesforceLogo"
value="https://developer.salesforce.com/files/salesforce-developer-network-
logo.png" width="220" height="55" alt="Salesforce Logo"/>
</apex:page>
```

Global Variables and Visualforce Expressions

- Visualforce pages can display data retrieved from the database or web services, data that changes depending on who is logged on and viewing the page, and so on. This dynamic data is accessed in markup through the use of global variables, calculations, and properties made available by the page's controller.

- Together these are described generally as `Visualforce expressions`. Use expressions for dynamic output or passing values into components by assigning them to attributes.

Visualforce Expressions

- A Visualforce expression is any set of literal values, variables, sub-expressions, or operators that can be resolved to a single value. Method calls aren't allowed in expressions.

- The expression syntax in Visualforce is: `{! expression }`
- Anything inside the `{! }` delimiters is evaluated and dynamically replaced when the page is rendered or when the value is used. Whitespace is ignored.

- The resulting value can be a primitive (integer, string, and so on), a Boolean, an sObject, a controller method such as an action method, and other useful results.

Global Variables

- Use global variables to access and display system values and resources in your Visualforce markup.

- For example, Visualforce provides information about the logged-in user in a global variable called `$User`. You can access fields of the $User global variable

(and any others) using an expression of the following form: `{! $GlobalName.fieldName }`.

Sample visualforce expressions with global variables

```
<apex:page>

    <apex:pageBlock title="User Status">
        <apex:pageBlockSection columns="1">

            {! $User.FirstName } {! $User.LastName }
           ({! $User.Username })

        </apex:pageBlockSection>
    </apex:pageBlock>

</apex:page>
```

Formula Expressions

- Visualforce lets you use more than just global variables in the expression language. It also supports formulas that let you manipulate values.
- For example, the & character is the formula language operator that concatenates strings.
- Sample snippets

```
{! $User.FirstName & ' ' & $User.LastName }
<p> Today's Date is {! TODAY() } </p>
<p> Next week it will be {! TODAY() + 7 } </p>
Today's Date is Thu Sep 18 00:00:00 GMT 2014
Next week it will be Thu Sep 25 00:00:00 GMT 2014
<p>The year today is {! YEAR(TODAY()) }</p>
<p>Tomorrow will be day number  {! DAY(TODAY() + 1) }</p>
<p>Let's find a maximum: {! MAX(1,2,3,4,5,6,5,4,3,2,1) } </p>
<p>The square root of 49 is {! SQRT(49) }</p>
<p>Is it true?  {! CONTAINS('salesforce.com', 'force.com') }</p>
```

- UserStatus page so far

```
<apex:page>

    <apex:pageBlock title="User Status">
        <apex:pageBlockSection columns="1">

            {! $User.FirstName & ' ' & $User.LastName }
           ({! $User.Username })

            <p>The year today is {! YEAR(TODAY()) }</p>
            <p>Tomorrow will be day number  {! DAY(TODAY() + 1) }</p>
            <p>Let's find a maximum: {! MAX(1,2,3,4,5,6,5,4,3,2,1) } </p>
            <p>The square root of 49 is {! SQRT(49) }</p>
```

```
            <p>Is it true?  {! CONTAINS('salesforce.com', 'force.com') }</p>
        </apex:pageBlockSection>
    </apex:pageBlock>

</apex:page>
```

Conditional Expressions

- Use a `conditional expression` to display different information based on the value of the expression.
- You can do this in Visualforce by using a conditional formula expression, such as `IF()`. The `IF()` expression takes three arguments:

- Sample snippets

```
<p>{! IF( CONTAINS('salesforce.com','force.com'),
     'Yep', 'Nope') }</p>
<p>{! IF( DAY(TODAY()) < 15,
     'Before the 15th', 'The 15th or after') }</p>
({! IF($User.isActive, $User.Username, 'inactive') })
```

- UserStatus page so far

```
<apex:page>

    <apex:pageBlock title="User Status">
        <apex:pageBlockSection columns="1">

            {! $User.FirstName & ' ' & $User.LastName }
           ({! IF($User.isActive, $User.Username, 'inactive') })

            <p>The year today is {! YEAR(TODAY()) }</p>
            <p>Tomorrow will be day number  {! DAY(TODAY() + 1) }</p>
            <p>Let's find a maximum: {! MAX(1,2,3,4,5,6,5,4,3,2,1) } </p>
            <p>The square root of 49 is {! SQRT(49) }</p>
            <p>Is it true?  {! CONTAINS('salesforce.com', 'force.com') }</p>
            <p>{! IF( CONTAINS('salesforce.com','force.com'),
                'Yep', 'Nope') }</p>
            <p>{! IF( DAY(TODAY()) < 15,
                'Before the 15th', 'The 15th or after') }</p>
        </apex:pageBlockSection>
    </apex:pageBlock>

</apex:page>
```

Solution to global variables challenge

```
<apex:page >
    <apex:pageBlock title="User Status">
        <apex:pageBlockSection columns="1">
            <apex:pageBlockSectionItem>
                <p>
                   {! $User.FirstName & ' ' & $User.LastName }
                   ({! IF($User.isActive, $User.Username, 'inactive') })
```

```
                </p>
                <p>
                    First Name: {! $User.FirstName }
                </p>
                <p>
                    Last Name: {! $User.LastName }
                </p>
                                <p>
                    Alias: {! $User.Alias }
                </p>
                <p>
                    Company: {!$User.CompanyName}
                </p>
            </apex:pageBlockSectionItem>
        </apex:pageBlockSection>
        </apex:pageBlock>
</apex:page>
```

Visualforce Standard Controller

- Visualforce uses the traditional `model-view-controller (MVC) paradigm`, and includes sophisticated built-in controllers to handle standard actions and data access, providing simple and tight integration with the Lightning Platform database. These built-in controllers are referred to generally as `standard controllers`, or even `the` standard controller.

JavaScript to open a Visualforce page from a standard page

- using browser console

```
$A.get("e.force:navigateToURL").setParams(
    {"url": "/apex/pageName?&id=<ID of Object e.g. Account>"}).fire();
```

Sample Account Summary

```
<apex:page standardController="Account">

    <apex:pageBlock title="Account Summary">
        <apex:pageBlockSection>

            Name: {! Account.Name } <br/>
            Phone: {! Account.Phone } <br/>
            Industry: {! Account.Industry } <br/>
            Revenue: {! Account.AnnualRevenue } <br/>

        </apex:pageBlockSection>
    </apex:pageBlock>

</apex:page>
```

- Append `id` in URL as such: `https://<salesforce-instance>/apex/AccountSummary?core.apexpages.request.devconsole=1&id=<id of an account>`

Solution to Contact View Challenge

```
<apex:page standardController="Contact">

    <apex:pageBlock title="Contact Summary">
        <apex:pageBlockSection>

            First Name: {! Contact.FirstName } <br/>
            Last Name: {! Contact.LastName } <br/>
            Owner Email: {! Contact.Owner.Email } <br/>

        </apex:pageBlockSection>
    </apex:pageBlock>

</apex:page>
```

Display Records, Fields, and Tables

Output Components

- Components that output data from a record and enable you to design a view-only user interface.
- Visualforce includes nearly 150 built-in components that you can use on your pages. Components are rendered into HTML, CSS, and JavaScript when a page is requested.
    - `Coarse-grained components` provide a significant amount of functionality in a single component, and might add a lot of information and user interface to the page it's used on.
    - `Fine-grained components` provide more focused functionality, and enable you to design the page to look and behave the way you want.

Display Record Details

```
<apex:page standardController="Account">

    <apex:detail />

</apex:page>
```

- `<apex:detail>` is a coarse-grained output component that adds many fields, sections, buttons, and other user interface elements to the page in just one line of markup.

Display Related Lists

```
<apex:page standardController="Account">

    <apex:detail relatedList="false" />
    <apex:relatedList list="Contacts"/>
          <apex:relatedList list="Opportunities" pageSize="5"/>

</apex:page>
```

- `relatedList="false"` removes the default related lists sections
- `<apex:relatedList>` adds specific related lists sections
- Use higher level components when they offer the functionality you need, and use lower level components when you need more control over what's displayed on the page.

Display Individual Fields

- Replacing `<apex:detail>` with specific fields wrapped in page block and page block section.

```
<apex:page standardController="Account">

        <apex:pageBlock title="Account Details">
      <apex:pageBlockSection>
          <apex:outputField value="{! Account.Name }"/>
          <apex:outputField value="{! Account.Phone }"/>
          <apex:outputField value="{! Account.Industry }"/>
          <apex:outputField value="{! Account.AnnualRevenue }"/>
      </apex:pageBlockSection>
  </apex:pageBlock>

  <apex:relatedList list="Contacts"/>
        <apex:relatedList list="Opportunities" pageSize="5"/>

</apex:page>
```

Display a Table

- `<apex:pageBlockTable>` is an iteration component that generates a table of data, complete with platform styling. Here's what's going on in your markup.
- An `iteration component` works with a collection of similar items, instead of on a single value.

```
<apex:page standardController="Account">

        <apex:pageBlock title="Account Details">
```

```
        <apex:pageBlockSection>
            <apex:outputField value="{! Account.Name }"/>
            <apex:outputField value="{! Account.Phone }"/>
            <apex:outputField value="{! Account.Industry }"/>
            <apex:outputField value="{! Account.AnnualRevenue }"/>
        </apex:pageBlockSection>
    </apex:pageBlock>

        <apex:pageBlock title="Contacts">
       <apex:pageBlockTable value="{!Account.contacts}" var="contact">
          <apex:column value="{!contact.Name}"/>
          <apex:column value="{!contact.Title}"/>
          <apex:column value="{!contact.Phone}"/>
       </apex:pageBlockTable>
    </apex:pageBlock>

    <apex:pageBlock title="Opportunities">
       <apex:pageBlockTable value="{!Account.opportunities}" var="opportunity">
          <apex:column value="{!opportunity.Name}"/>
          <apex:column value="{!opportunity.CloseDate}"/>
          <apex:column value="{!opportunity.StageName}"/>
       </apex:pageBlockTable>
    </apex:pageBlock>
</apex:page>
```

- Replaced `<apex:relatedList>` with specific fields using page block table and columns wrapped in page blocks.

Coarse-grained vs Fine-grained

- `Coarse-grained components` let you quickly add lots of functionality to a page, while `Fine-grained components` give you more control over the specific details of a page.

Solution to Opportunity View challenge

```
<apex:page standardController="Opportunity">

    <apex:pageBlock title="Opportunity Summary">
        <apex:pageBlockSection>
                        <apex:outputField value="{! Opportunity.Name }"/>
            <apex:outputField value="{! Opportunity.Amount }"/>
            <apex:outputField value="{! Opportunity.CloseDate }"/>
            <apex:outputField value="{! Opportunity.Account.Name }"/>
        </apex:pageBlockSection>
    </apex:pageBlock>

</apex:page>
```

Input Data Using Forms

Create a Basic Form

- Use `<apex:form>` and `<apex:inputField>` to create a page to edit data. Combine `<apex:commandButton>` with the save action built into the standard controller to create a new record, or save changes to an existing one.

```
<apex:page standardController="Account">

    <h1>Edit Account</h1>

    <apex:form>

        <apex:inputField value="{! Account.Name }"/>

        <apex:commandButton action="{! save }" value="Save" />

    </apex:form>

</apex:page>
```

Add Field Labels and Platform Styling

- When you use input components within `<apex:pageBlock>` and `<apex:pageBlockSection>` tags they adopt the platform visual styling.
- The `<apex:inputField>` component renders the appropriate input widget based on a standard or custom object field's type. For example, if you use an `<apex:inputField>` tag to display a date field, a calendar widget displays on the form. If you use an `<apex:inputField>` tag to display a picklist field, as we did here for the industry field, a drop-down list displays instead.

```
<apex:page standardController="Account">
    <apex:form>

    <apex:pageBlock title="Edit Account">
                <apex:pageBlockSection columns="1">
            <apex:inputField value="{! Account.Name }"/>
            <apex:inputField value="{! Account.Phone }"/>
            <apex:inputField value="{! Account.Industry }"/>
            <apex:inputField value="{! Account.AnnualRevenue }"/>
        </apex:pageBlockSection>
        <apex:pageBlockButtons>
            <apex:commandButton action="{! save }" value="Save" />
        </apex:pageBlockButtons>
    </apex:pageBlock>

    </apex:form>
</apex:page>
```

Display Form Errors and Messages

- Use <apex:pageMessages> to display any form handling errors or messages.

```
<apex:page standardController="Account">
    <apex:form>

    <apex:pageBlock title="Edit Account">
        <apex:pageMessages/>
                <apex:pageBlockSection columns="1">
            <apex:inputField value="{! Account.Name }"/>
            <apex:inputField value="{! Account.Phone }"/>
            <apex:inputField value="{! Account.Industry }"/>
            <apex:inputField value="{! Account.AnnualRevenue }"/>
        </apex:pageBlockSection>
        <apex:pageBlockButtons>
            <apex:commandButton action="{! save }" value="Save" />
        </apex:pageBlockButtons>
    </apex:pageBlock>

    </apex:form>
</apex:page>
```

Edit Related Records

- While you can't save changes to multiple object types in one request with the standard controller, you can make it easier to navigate to related records by offering links that perform actions such as edit or delete on those records.

```
<apex:page standardController="Account">
    <apex:form>

    <apex:pageBlock title="Edit Account">
        <apex:pageMessages/>
                <apex:pageBlockSection columns="1">
            <apex:inputField value="{! Account.Name }"/>
            <apex:inputField value="{! Account.Phone }"/>
            <apex:inputField value="{! Account.Industry }"/>
            <apex:inputField value="{! Account.AnnualRevenue }"/>
        </apex:pageBlockSection>
        <apex:pageBlockButtons>
            <apex:commandButton action="{! save }" value="Save" />
        </apex:pageBlockButtons>
    </apex:pageBlock>
    <apex:pageBlock title="Contacts">
        <apex:pageBlockTable value="{!Account.contacts}" var="contact">
            <apex:column>
                <apex:outputLink
                    value="{! URLFOR($Action.Contact.Edit, contact.Id) }">
                    Edit
                </apex:outputLink>
                 
                <apex:outputLink
                    value="{! URLFOR($Action.Contact.Delete, contact.Id) }">
                    Del
```

```
                </apex:outputLink>
            </apex:column>
            <apex:column value="{!contact.Name}"/>
            <apex:column value="{!contact.Title}"/>
            <apex:column value="{!contact.Phone}"/>
        </apex:pageBlockTable>
    </apex:pageBlock>
    </apex:form>
</apex:page>
```

Solution to Contact Create challenge

```
<apex:page standardController="Contact">
    <apex:form>
        <apex:pageBlock title="Create Contact">
            <apex:pageMessages/>
            <apex:pageBlockSection columns="1">
                <apex:inputField value="{! Contact.FirstName }" />
                <apex:inputField value="{! Contact.LastName }" />
                <apex:inputField value="{! Contact.Email }" />
            </apex:pageBlockSection>
            <apex:pageBlockButtons>
                <apex:commandButton action="{! save }" value="Save" />
            </apex:pageBlockButtons>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

Standard List Controller

- The standard list controller allows you to create Visualforce pages that can display or act on a set of records.

Display a List of Records

- Use the standard list controller and an iteration component, such as `<apex:pageBlockTable>`, to display a list of records.

```
<apex:page standardController="Contact" recordSetVar="contacts">
    <apex:pageBlock title="Contacts List">

        <!-- Contacts List -->
        <apex:pageBlockTable value="{! contacts }" var="ct">
            <apex:column value="{! ct.FirstName }"/>
            <apex:column value="{! ct.LastName }"/>
            <apex:column value="{! ct.Email }"/>
            <apex:column value="{! ct.Account.Name }"/>
        </apex:pageBlockTable>

    </apex:pageBlock>
```

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

- `recordSetVar` by convention is usually named the plural of the object name.
- `<apex:pageBlockTable>` is an iteration component that generates a table of data, complete with platform styling. Here's what's going on in the table markup.
- There are other iteration components such as `<apex:dataList>` and `<apex:repeat>`.

Add List View Filtering to the List

- Use `{! listViewOptions }` to get a list of list view filters available for an object.
- Use `{! filterId }` to set the list view filter to use for a standard list controller's results.

```
<apex:page standardController="Contact" recordSetVar="contacts">
    <apex:form>
        <apex:pageBlock title="Contacts List" id="contacts_list">

            Filter:
            <apex:selectList value="{! filterId }" size="1">
                <apex:selectOptions value="{! listViewOptions }"/>
                <apex:actionSupport event="onchange" reRender="contacts_list"/>
            </apex:selectList>
            <!-- Contacts List -->
            <apex:pageBlockTable value="{! contacts }" var="ct">
                <apex:column value="{! ct.FirstName }"/>
                <apex:column value="{! ct.LastName }"/>
                <apex:column value="{! ct.Email }"/>
                <apex:column value="{! ct.Account.Name }"/>
            </apex:pageBlockTable>

        </apex:pageBlock>
    </apex:form>
</apex:page>
```

- `<apex:actionSupport event="onchange" reRender="contacts_list"/>` reloads the `contacts_list` page block without reloading the entire page.

Add Pagination to the List

- Use the pagination features of the standard list controller to allow users to look at long lists of records one "page" at a time.

```
<apex:page standardController="Contact" recordSetVar="contacts">
    <apex:form>
        <apex:pageBlock title="Contacts List" id="contacts_list">

            Filter:
            <apex:selectList value="{! filterId }" size="1">
                <apex:selectOptions value="{! listViewOptions }"/>
                <apex:actionSupport event="onchange" reRender="contacts_list"/>
            </apex:selectList>
            <!-- Contacts List -->
```

```
            <apex:pageBlockTable value="{! contacts }" var="ct">
                <apex:column value="{! ct.FirstName }"/>
                <apex:column value="{! ct.LastName }"/>
                <apex:column value="{! ct.Email }"/>
                <apex:column value="{! ct.Account.Name }"/>
            </apex:pageBlockTable>

            <!-- Pagination -->
            <table style="width: 100%"><tr>
                <td>
                    <!-- Page X of Y -->
                    Page: <apex:outputText
                        value=" {!PageNumber} of {! CEILING(ResultSize / PageSize)
}"/>
                </td>
                <td align="center">
                    <!-- Previous page -->
                    <!-- active -->
                    <apex:commandLink action="{! Previous }" value="« Previous"
                         rendered="{! HasPrevious }"/>
                    <!-- inactive (no earlier pages) -->
                    <apex:outputText style="color: #ccc;" value="« Previous"
                         rendered="{! NOT(HasPrevious) }"/>

                      
                    <!-- Next page -->
                    <!-- active -->
                    <apex:commandLink action="{! Next }" value="Next »"
                         rendered="{! HasNext }"/>
                    <!-- inactive (no more pages) -->
                    <apex:outputText style="color: #ccc;" value="Next »"
                         rendered="{! NOT(HasNext) }"/>
                </td>

                <td align="right">
                    <!-- Records per page -->
                    Records per page:
                    <apex:selectList value="{! PageSize }" size="1">
                        <apex:selectOption itemValue="5" itemLabel="5"/>
                        <apex:selectOption itemValue="20" itemLabel="20"/>
                        <apex:actionSupport event="onchange"
reRender="contacts_list"/>
                    </apex:selectList>
                </td>
            </tr></table>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

- In the progress indicator, three properties are used to indicate how many pages there are: `PageNumber`, `ResultSize`, and `PageSize`.
- The page markup is referencing Boolean properties provided by the standard list controller, `HasPrevious` and `HasNext`, which let you know if there are more records in a given direction or not.
- The records per page selection menu uses an `<apex:selectList>`.
- Aside from `Previous` and `Next`, there are also `First` and `Last` actions for pagination.

Solution to Account Record Detail challenge

```
<apex:page standardController="Account" recordSetVar="accounts">
    <apex:form>
        <apex:pageBlock title="Accounts List" id="accounts_list">
            Filter:
            <apex:selectList value="{! filterId }" size="1">
                <apex:selectOptions value="{! listViewOptions }"/>
                <apex:actionSupport event="onchange" reRender="accounts_list"/>
            </apex:selectList>
            <apex:repeat value="{! accounts }" var="a">
                <li>
                    <apex:outputLink value="/{!a.id}">
                        {! a.name }
                    </apex:outputLink>
                </li>
            </apex:repeat>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

Static Resources

- Static resources allow you to upload content that you can reference in a Visualforce page. Resources can be archives (such as .zip and .jar files), images, stylesheets, JavaScript, and other files.
- Static resources are managed and distributed by Lightning Platform, which acts as a content distribution network (CDN) for the files. Caching and distribution are handled automatically.
- Static resources are referenced using the `$Resource` global variable, which can be used directly by Visualforce, or used as a parameter to functions such as `URLFOR()`.

Create and Upload a Simple Static Resource

- Setup | Static Resources | New
- e.g. Name as `JQuery`
- Cache Control should be Public

Add a Static Resource to a Visualforce Page

```
<apex:page>

    <!-- Add the static resource to page's <head> -->
```

```
    <apex:includeScript value="{! $Resource.jQuery }"/>

    <!-- A short bit of jQuery to test it's there -->
    <script type="text/javascript">
        jQuery.noConflict();
        jQuery(document).ready(function() {
            jQuery("#message").html("Hello from jQuery!");
        });
    </script>

    <!-- Where the jQuery message will appear -->
    <h1 id="message"></h1>

</apex:page>
```

Create and Upload a Zipped Static Resource

- Create zipped static resources to group together related files that are usually updated together.
- Setup | Static Resources | New
- e.g. Name as `jQueryMobile`
- Cache Control should be Public
- Zip file limit is 5MB (i.e. for jQueryMobile, unzip and remove the demos folder then zip it again)

Add Zipped Static Resources to a Visualforce Page

```
<apex:page showHeader="false" sidebar="false" standardStylesheets="false">

    <!-- Add static resources to page's <head> -->
    <apex:stylesheet value="{!
          URLFOR($Resource.jQueryMobile,'jquery.mobile-1.4.5/jquery.mobile-
1.4.5.css')}"/>
    <apex:includeScript value="{! $Resource.jQuery }"/>
    <apex:includeScript value="{!
         URLFOR($Resource.jQueryMobile,'jquery.mobile-1.4.5/jquery.mobile-
1.4.5.js')}"/>
    <div style="margin-left: auto; margin-right: auto; width: 50%">
        <!-- Display images directly referenced in a static resource -->
        <h3>Images</h3>
        <p>A hidden message:
            <apex:image alt="eye" title="eye"
                 url="{!URLFOR($Resource.jQueryMobile, 'jquery.mobile-
1.4.5/images/icons-png/eye-black.png')}"/>
            <apex:image alt="heart" title="heart"
                 url="{!URLFOR($Resource.jQueryMobile, 'jquery.mobile-
1.4.5/images/icons-png/heart-black.png')}"/>
            <apex:image alt="cloud" title="cloud"
                 url="{!URLFOR($Resource.jQueryMobile, 'jquery.mobile-
1.4.5/images/icons-png/cloud-black.png')}"/>
        </p>
    <!-- Display images referenced by CSS styles,
         all from a static resource. -->
```

```
    <h3>Background Images on Buttons</h3>
    <button class="ui-btn ui-shadow ui-corner-all
         ui-btn-icon-left ui-icon-action">action</button>
    <button class="ui-btn ui-shadow ui-corner-all
         ui-btn-icon-left ui-icon-star">star</button>
    </div>
</apex:page>
```

- Use the `$Resource` global variable and the `URLFOR()` function to reference items within a zipped static resource.
- The `URLFOR()` function can combine a reference to a zipped static resource and a relative path to an item within it to create a URL that can be used with Visualforce components that reference static assets. For example, `URLFOR($Resource.jQueryMobile, 'images/icons-png/cloud-black.png')` returns a URL to a specific graphic asset within the zipped static resource, which can be used by the `<apex:image>` component. You can construct similar URLs for JavaScript and stylesheet files for the `<apex:includeScript>` and `<apex:stylesheet>` components.

Solution to static resource challenge

```
<apex:page >
    <apex:image url="{!URLFOR($Resource.vfimagetest, 'cats/kitten1.jpg')}" />
</apex:page>
```

Custom Controllers

- Custom controllers contain custom logic and data manipulation that can be used by a Visualforce page. For example, a custom controller can retrieve a list of items to be displayed, make a callout to an external web service, validate and insert data, and more—and all of these operations will be available to the Visualforce page that uses it as a controller.

- Used when you want to override existing functionality, customize the navigation through an application, use callouts or Web services, or if you need finer control for how information is accessed for your page.

Create a Custom Controller Apex Class

```
public class ContactsListController {
    // Controller code goes here
}
```

Create a Visualforce Page that Uses a Custom Controller

```
<apex:page controller="ContactsListController">
    <apex:form>
        <apex:pageBlock title="Contacts List" id="contacts_list">

            <!-- Contacts List goes here -->
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

Add a Method to Retrieve Records

- Apex Class

```
public class ContactsListController {
        private String sortOrder = 'LastName';

    public List<Contact> getContacts() {

        List<Contact> results = Database.query(
            'SELECT Id, FirstName, LastName, Title, Email ' +
            'FROM Contact ' +
            'ORDER BY ' + sortOrder + ' ASC ' +
            'LIMIT 10'
        );
        return results;
    }
}
```

- Visualforce Page

```
<apex:page controller="ContactsListController">
    <apex:form>
        <apex:pageBlock title="Contacts List" id="contacts_list">

            <!-- Contacts List -->
            <apex:pageBlockTable value="{! contacts }" var="ct">
                <apex:column value="{! ct.FirstName }"/>
                <apex:column value="{! ct.LastName }"/>
                <apex:column value="{! ct.Title }"/>
                <apex:column value="{! ct.Email }"/>

            </apex:pageBlockTable>
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

- Any getter method (i.e. `getSomething()`) is translated to a `{! something }` variable.
- As such, when happens when the `{! contacts }` expression is evaluated, that expression calls the `getContacts()` method. That method returns a List of contact records, which is exactly what the `<apex:pageBlockTable>` is expecting.

Add a New Action Method

- Create action methods in your custom controller to respond to user input on the page.

- Apex Class

```
public class ContactsListController {
        private String sortOrder = 'LastName';

    public List<Contact> getContacts() {

        List<Contact> results = Database.query(
            'SELECT Id, FirstName, LastName, Title, Email ' +
            'FROM Contact ' +
            'ORDER BY ' + sortOrder + ' ASC ' +
            'LIMIT 10'
        );
        return results;
    }

    public void sortByLastName() {
        this.sortOrder = 'LastName';
    }

    public void sortByFirstName() {
        this.sortOrder = 'FirstName';
    }
}
```

- Visualforce Page

```
<apex:page controller="ContactsListController">
    <apex:form>
        <apex:pageBlock title="Contacts List" id="contacts_list">

            <!-- Contacts List -->
            <apex:pageBlockTable value="{! contacts }" var="ct">
                <apex:column value="{! ct.FirstName }">
                    <apex:facet name="header">
                        <apex:commandLink action="{! sortByFirstName }"
                            reRender="contacts_list">First Name
                        </apex:commandLink>
                    </apex:facet>
                </apex:column>
                <apex:column value="{! ct.LastName }">
                    <apex:facet name="header">
                        <apex:commandLink action="{! sortByLastName }"
                            reRender="contacts_list">Last Name
                        </apex:commandLink>
                    </apex:facet>
                </apex:column>
                <apex:column value="{! ct.Title }"/>
                <apex:column value="{! ct.Email }"/>

            </apex:pageBlockTable>
```

```
        </apex:pageBlock>
    </apex:form>
</apex:page>
```

- `<apex:facet>` lets us set the contents of the column header to whatever we want.
- The link is created using the `<apex:commandLink>` component, with the `action` attribute set to an expression that references the action method in our controller.
- The attribute `reRender="contacts_list"` will reload the `contacts_list` page block.

Field Labels which support Internationalization

- Instead of hard coding the header text for 'First Name' or 'Last Name', use the following syntax:

```
<apex:outputText value="{! $ObjectType.Contact.Fields.FirstName.Label }"/>
```

- This syntax will automatically translate the header text into whichever language the org is using.

Alternative way to declare Apex properties as getters/setters

```
public MyObject__c myVariable { get; set; }
```

Solution to New Case List Controller Challenge

- Apex Class

```
public class NewCaseListController {
    public List<Case> getNewCases() {
        String newStatus = 'New';
        List<Case> results = Database.query(
            'SELECT Id, CaseNumber ' +
            'FROM Case ' +
            'WHERE Status=:newStatus'
        );
        return results;
    }
}
```

- Visualforce Page

```
<apex:page controller="NewCaseListController">
    <apex:repeat value="{! newCases }" var="case">
        <li>
            <apex:outputLink value="/{! case.id }">
                {! case.ID }
            </apex:outputLink>
        </li>
    </apex:repeat>
</apex:page>
```

Search Solution Basics

- Search is the no.1 most used Salesforce feature.
- The `search index` and `tokens` allow the search engine to apply advanced features like spell correction, nicknames, lemmatization, and synonym groups.
- `Lemmatization` identifies and returns variants of the search term, such as `add`, `adding`, and `added`, in search results.)
- The search index also provides the opportunity to introduce relevance ranking into the mix.
- In general, you need a `custom search solution` when your org uses a custom UI instead of the standard Salesforce UI.

Connect to Search with APIs

Two main APIs

- `Salesforce Object Query Language (SOQL)`
- `Salesforce Object Search Language (SOSL)`

Other APIs

- `Suggested records API` - e.g. auto-suggestion, instant results, type-ahead
- `Salesforce Federated Search` - search records stored outside of Salesforce.

SOQL vs SOSL

- Use SOQL when you know in which objects or fields the data resides and you want to:
    i. Retrieve data from a single object or from multiple objects that are related to one another.
    ii. Count the number of records that meet specified criteria.
    iii. Sort results as part of the query.
    iv. Retrieve data from number, date, or checkbox fields.
- Use SOSL when you don't know in which object or field the data resides and you want to:

i. Retrieve data for a specific term that you know exists within a field. Because SOSL can tokenize multiple terms within a field and build a search index from this, SOSL searches are faster and can return more relevant results.
ii. Retrieve multiple objects and fields efficiently, and the objects might or might not be related to one another.
iii. Retrieve data for a particular division in an organization using the divisions feature, and you want to find it in the most efficient way possible.

Send Queries with Protocols

- `Query (REST) and query() (SOAP)` — Executes a SOQL query against the specified object and returns data that matches the specified criteria.
- `Search (REST) and search() (SOAP)` — Executes a SOSL text string search against your org's data.
- REST API Developer Guide

Build Search for Common Use Cases

Search Within a Single Object (SOSL)

- Syntax

```
FIND {term} RETURNING ObjectTypeName
```

- Example

```
FIND {march 2016 email} RETURNING Campaign
```

Search Within Multiple Objects

- Syntax

```
FIND {term} RETURNING ObjectTypeName1, ObjectTypeName2,
ObjectTypeNameYouGetTheIdea
```

- Example

```
FIND {recycled materials} RETURNING Product2, ContentVersion, FeedItem
```

Search Within Custom Objects

- Example

```
FIND {pink hi\-top} RETURNING Merchandise__c
```

SOQL?

- You use SOQL for single object searches, when you know the fields to be searched, when the search term is an exact match for the field (no partial or out-of-order matches), when you need number, date, or checkbox field data, and when you're looking for just a few results

---

Optimize Search Results

Create Efficient Text Searches

- Search queries can be expensive. The more data you're searching through and the more results you're returning, the more you can slow down the entire operation.
- Basic strategies:
  - Limit which data you're searching through
  - Limit which data you're returning

Search by SearchGroup (e.g. EMAIL FIELDS)

- Example

```
FIND {jsmith@cloudkicks.com} IN EMAIL FIELDS RETURNING Contact
```

Specify the object to return.

```
FIND {Cloud Kicks} RETURNING Account
```

Specify the field to return.

```
FIND {Cloud Kicks} RETURNING Account(Name, Industry)
```

Order the results by field in ascending order, which is the default.

```
FIND {Cloud Kicks} RETURNING Account (Name, Industry ORDER BY Name)
```

Set the max number of records returned

```
FIND {Cloud Kicks} RETURNING Account (Name, Industry ORDER BY Name LIMIT 10)
```

Set the starting row offset into the results.

```
FIND {Cloud Kicks} RETURNING Account (Name, Industry ORDER BY Name LIMIT 10 OFFSET
25)
```

WITH DIVISION

```
FIND {Cloud Kicks} RETURNING Account (Name, Industry)
    WITH DIVISION = 'Global'
```

WITH DATA CATEGORY

```
FIND {race} RETURNING KnowledgeArticleVersion
    (Id, Title WHERE PublishStatus='online' and language='en_US')
    WITH DATA CATEGORY Location__c AT America__c
```

WITH NETWORK

```
FIND {first place} RETURNING User (Id, Name),
FeedItem (id, ParentId WHERE CreatedDate = THIS_YEAR Order by CreatedDate DESC)
WITH NETWORK = '00000000000001'
```

WITH PRICEBOOK

```
Find {shoe} RETURNING Product2 WITH PricebookId = '01sxx0000002MffAAE'
```

Display Suggested Results

Go-to REST resources

1. `Search Suggested Records` — Returns a list of suggested records whose names match the user's search string. The suggestions resource provides a shortcut for users to navigate directly to likely relevant records, before performing a full search.
2. `Search Suggested Article Title Matches` — Returns a list of Salesforce Knowledge articles whose titles match the user's search query string. Provides a shortcut to navigate directly to likely relevant articles before the user performs a search.
3. `SObject Suggested Articles for Case` — Returns a list of suggested Salesforce Knowledge articles for a case.

- Syntax for Search Suggested Article Title Matches

```
/vXX.X/search/suggestTitleMatches?q=search string&language=article
language&publishStatus=article publication status
```

- Example for Search Suggested Article Title Matches

```
/vXX.X/search/suggestTitleMatches?q=race+tips&language=en_US&publishStatus=Online
```

- Response JSON

```
{
  "autoSuggestResults" : [ {
    "attributes" : {
    "type" : "KnowledgeArticleVersion",
    "url" :
"/services/data/v30.0/sobjects/KnowledgeArticleVersion/ka0D00000004CcQ"
    },
  "Id" : "ka0D00000004CcQ",
  "UrlName" : "tips-for-your-first-trail-race",
  "Title" : "race tips",
  "KnowledgeArticleId" : "kA0D00000004Cfz",
  "isMasterLanguage" : "1",
  } ],
  "hasMoreResults" : false
}
```

Synonyms

- A search for one term in a synonym group returns results for all terms in the group. For example, a search for USB returns results for all terms in the synonym group, which contains `USB`, `thumb drive`, `flash stick`, and `memory stick`.
- Setup | Quick Find box | Synonyms
- The maximum number of characters per term is 100.
- Your organization can create a maximum of 2,000 promoted terms.