



SALESFORCE VIRTUAL INTERNSHIP REPORT

24-04-2022 TO 30-06-2022

Salesforce, Inc.

PREPARED BY
DEGA NIKHITHA
ANNAMACHARYA INSTITUTE OF TECHNOLOGY AND SCIENCES
BACHELOR OF TECHNOLOGY
DEPARTMENT OF CSE
TIRUPATHI

CONTENTS

INTRODUCTION

COMPANY BACKGROUND

SERVICES

ABOUT MY ROLE

LWC

APEX

CONCLUSION

ACKNOWLEDGEMENT

I would like to express my sincere thanks and gratitude to "THE SALESFORCE INC" for letting me work on this Internship. I am very grateful of the "SMARTBRIDGE" for the support and guidance in completing this Internship.

I am thankful to my parents as well. I was able to successfully complete this project with the help of their guidance and support. Finally, I want to thank all my dear friends as well.

DEGA NIKHITHA

INTRODUCTION

COMPANY BACKGROUND

The company was founded in 1999 by former Oracle executive **Marc Benioff**, together with **Parker Harris**, **Dave Moellenhoff**, and **Frank Dominguez** as a software as a service (SaaS) company. Initial funding for the company came from Larry Ellison and Halsey Minor.

In 2003, Salesforce held its first annual Dreamforce conference in San Francisco. In June 2004, the company had its initial public offering on the New York Stock Exchange under the stock symbol CRM and raised US\$110 million.^{[5][6]} In 2006, Salesforce launched IdeaExchange, a platform that allows customers to connect with company product managers.

In 2009, Salesforce passed \$1 billion in annual revenue. Also in 2009, the company launched Service Cloud, an application that helps companies manage service conversations about their products and services.

In 2014, the company released Trailhead, a free online learning platform. In October 2014, Salesforce announced the development of its Customer Success Platform. In September 2016, Salesforce announced the launch of Einstein, an artificial intelligence platform that supports several of Salesforce's cloud services.

In February 2020, co-chief executive officer Keith Block stepped down from his position in the company. **Marc Benioff** remained as chairman and chief executive officer. On December 1, 2020, it was announced that Salesforce would acquire Slack for \$27.7 billion.^[15] The acquisition closed on July 21, 2021.^[16]

In February 2021, Amy Weaver, previously the chief legal officer, became CFO. Former CFO Mark Hawkins announced that he would be retiring in October. In November 2021, Bret Taylor was named vice chair and co-CEO of the company.

In April 2022, Salesforce acquired Phennecs, a privacy sandbox startup.

Also in April 2022, Salesforce announced a five-year partnership with Formula One.

In May 2022, Salesforce announced the acquisition of Slack-bot maker Troops.ai.

SERVICES

Salesforce's products include several customer relationship management (CRM) technologies, including: Sales Cloud, Service Cloud, Marketing Cloud, and Commerce Cloud and Platform. Additional technologies include Slack, MuleSoft, Tableau Analytics, and Trailhead.

Main services

Salesforce's main technologies are tools for customer management. Other products enable customers to create apps, integrate data from other systems, visualize data, and offer training courses.

Salesforce Platform

Salesforce Platform (also known as Force.com) is a platform as a service (PaaS) that allows developers to create add-on applications that integrate into the main Salesforce.com application. These third-party applications are hosted on Salesforce.com's infrastructure.

Force.com applications are built using declarative tools, backed by Lightning and Apex, a proprietary Java-like programming language for Force.com, as well as Visualforce, a framework including an XML syntax typically used to generate HTML. The Force.com platform typically receives three complete releases a year. As the platform is provided as a service to its developers, every single development instance also receives all these updates.

In 2015, a new framework for building user interfaces – Lightning Components – was introduced in beta. Lightning components are built using the open-source Aura Framework but with support for Apex as the server-side language instead of Aura's JavaScript dependency. This has been described as an alternative to, not necessarily a replacement for, Visualforce pages.

As of 2014, the Force.com platform has 1.5 million registered developers.

Lightning Base Components is the component library built on top of Lightning Web Components.^[38]

AppExchange

Launched in 2005, the Salesforce AppExchange is an online application marketplace that connects customers with third-party applications and consulting services. As of 2021, the exchange has over 5,000 apps listed.

Trailhead

Launched in 2014, Trailhead is a free online learning platform with courses focused on Salesforce technologies.

Retired or end-of-life

Desk.com

Desk.com is a SaaS help desk and customer support product that was acquired by Salesforce for \$50 million in 2011. The product focused on connecting small businesses to their customers.

In March 2018, Salesforce announced that Desk.com would be consolidated with other services into Service Cloud Essentials.

Do.com

Do.com was a cloud-based task management system for small groups and businesses, introduced in 2011 and discontinued in 2014

ABOUT MY ROLE

As a Salesforce Developer intern, I will be responsible for designing and developing customized solutions within the Salesforce platform based on the business requirements.

- Work on application full application lifecycle activities such as planning, analysis, design, implementation, testing, deployment and maintenance
- Develop high level solutions using Apex classes, Triggers, Visualforce pages, Lightning Components and Lightning Web Components based on the best practices
- Design automated processes using Process Builders, Workflows, Approval Processes etc.
- The Apex classes and Triggers code which I worked are follows

Get Started with Apex Triggers

```
trigger AccountAddressTrigger on Account (before insert, before  
update) {
```

```
    For(Account accountAddress : Trigger.new)  
        if(accountAddress.BillingAddress != null)  
            accountAddress.Match_Billing_Address__c =  
                accountAddress.BillingAddress__c  
        }  
    }  
}
```

Here the trigger 'AccountAddressTrigger' which is created on Account object. And in this i did called a method from a trigger and used a context variables, Use the `sObjectaddError()` method in a trigger to restrict save operations.

Bulk Apex Triggers

I Wrote triggers that operate on collections of sObjects. And wrote triggers that perform efficient SOQL and DML operations.

Apex triggers are optimized to operate in bulk. We recommend using bulk design patterns for processing records in 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.

```
trigger ClosedOpportunityTrigger on Opportunity (after insert, after update) {
    List<Task> newtsk = new List<Task>();
    if(trigger.IsAfter && (trigger.IsInsert || trigger.IsUpdate)){
        for(Opportunity op:Trigger.New){
            if(op.StageName == 'Closed Won'){
                Task tsk = new Task();
                tsk.Subject = 'Follow Up Test Task';
                tsk.WhatId = op.id;
                newtsk.add(tsk);
            }
        }
    }
    if(newtsk.size()>0){
        insert newtsk;
    }
}
```

GET STARTED WITH APEX TRIGGERS

The main scope of this module is to

- Ensuring that your Apex classes and triggers work as expected
- Having a suite of regression tests that can be rerun every time classes and triggers are updated to ensure that future updates you make to your app don't break existing functionality
- Meeting the code coverage requirements for deploying Apex to production or distributing Apex to customers via packages

- High-quality apps delivered to the production org, which makes production users more productive
- High-quality apps delivered to package subscribers, which increase your customers trust.

@isTest

```
private class TestVerifyDate {
    static testMethod void TestVerifyDate() {
        VerifyDate.CheckDates(System.today(),System.today().addDays(10));
        VerifyDate.CheckDates(System.today(),System.today().addDays(78));
    }
}
```

VerifyDate is a class in the salesforce.com org. The requirement is to write a test class. So the first step is to create a new class name it as TestVerifyDate . Now I wrote @isTest in the very first line of this class and proceed writting the code which calls every and every method of my made class , the basic motive should be to satisfy each and every condition in VerifyDate class.

TEST APEX TRIGGERS

@isTest

```
private class TestRestrictContactByName {

    @isTest static void testInvalidName() {
        //try inserting a Contact with INVALIDNAME
        Contact myConact = new Contact(LastName='INVALIDNAME');
        insert myConact;

        // Perform test
        Test.startTest();
        Database.SaveResult result = Database.insert(myConact, false);
        Test.stopTest();
        // Verify
        // In this case the creation 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 create contact with invalid last name.',
                            result.getErrors()[0].getMessage());
    }
}
```

```
}
```

CREATE TEST DATA FOR APEX

```
//@isTest
public static List<Contact> generateRandomContacts(Integer numContactsToGenerate,
FName) {

    List<Contact> contactList = new List<Contact>();

    for(Integer i=0;i<numContactsToGenerate;i++) {

        Contact c = new Contact(FirstName=FName + ' ' + i, LastName = 'Contact '+i);

        contactList.add(c);

        System.debug(c);

    }

    System.debug(contactList.size());

    return contactList;

}

}
```

But it will not complete the challenge. It needs a random FirstName and a provided LastName.
//This will work

```
public class RandomContactFactory{
public static List generateRandomContacts(Integer noOfContacts, String conList){
    List ContactList = new List();
    for(Integer i = 1; i <= noOfContacts; i++){
        Contact con = new Contact(firstName = "+i, LastName = 'Test');
        ContactList.add(con);
    }

    return ContactList;
}
```

```
}
```

Now the module got complet ,now i am going another module:ASYNCHRONOUS APEX

which containg following sub modules like

Asynchronous Processing Basics

- I learnt that
- The difference between synchronous and asynchronous processing.
- Choose which kind of asynchronous Apex to use in various scenarios.

Use Future Methods

After completing this module I came to learn that

- When to use future methods.
- The limitations of using future methods.
- How to use future methods for callouts.
- Future method best practices.

Future methods must be static methods, and can only return a void type. The specified parameters must be primitive data types, arrays of primitive data types, or collections of primitive data types. Notably, future methods can't take standard or custom objects as arguments. A common pattern is to pass the method a List of record IDs that you want to process asynchronously.

```
public class SomeClass {  
    @future  
    public static void someFutureMethod(List<Id> recordIds) {  
        List<Account> accounts = [Select Id, Name from Account Where Id IN  
:recordIds];  
        // process account records to do awesome stuff  
    }  
}
```

To make a Web service callout to an external service or API, you create an Apex class with a future method that is marked with (callout=true). The class below has methods for making the callout both synchronously and asynchronously where callouts are not permitted. We insert a record into a custom log object to track the status of the callout

simply because logging is always fun to do!

```
public class SMSUtils {
    // Call async from triggers, etc, where callouts are not
    permitted.
    @future(callout=true)
    public static void sendSMSAsync(String fromNbr, String toNbr,
String m) {
        String results = sendSMS(fromNbr, toNbr, m);
        System.debug(results);
    }
    // Call from controllers, etc, for immediate processing
    public static String sendSMS(String fromNbr, String toNbr, String
m) {
        // Calling 'send' will result in a callout
        String results = SmsMessage.send(fromNbr, toNbr, m);
        insert new SMS_Log__c(to__c=toNbr, from__c=fromNbr,
msg__c=results);
        return results;
    }
}
```

TEST CLASSES

```
@isTest
public class SMSCalloutMock implements HttpCalloutMock {
    public HttpResponse respond(HttpRequest req) {
        // Create a fake response
        HttpResponse res = new HttpResponse();
        res.setHeader('Content-Type', 'application/json');
        res.setBody('{"status":"success"}');
        res.setStatusCode(200);
        return res;
    }
}
=====
```

```

@IsTest
private class Test_SMSUtils {
    @IsTest
    private static void testSendSms() {
        Test.setMock(HttpCalloutMock.class, new SMSCalloutMock());
        Test.startTest();
        SMSUtils.sendSMSAsync('111', '222', 'Greetings!');
        Test.stopTest();
        // runs callout and check results
        List<SMS_Log__c> logs = [select msg__c from SMS_Log__c];
        System.assertEquals(1, logs.size());
        System.assertEquals('success', logs[0].msg__c);
    }
}

```

AccountProceesor.cls

```

public class
AccountProcessor {

    @future
    public static void countContacts(List<Id> accountId_lst) {

        Map<Id,Integer> account_cno = new Map<Id,Integer>();
        List<account> account_lst_all = new List<account>();
        for(account a:accountId_lst) {
            account_cno.put(a.id,a.contacts.size());
        }

        List<account> account_lst = new List<account>();

        for(Id accountId : accountId_lst) {
            if(account_cno.containsKey(accountId)) {
                account acc = new account();
                acc.Id = accountId;
                acc.Number_of_Contacts__c = account_cno.get(accountId);
                account_lst.add(acc);
            }
        }
        upsert account_lst;
    }
}

```

```
}
```

```
}
```

AccountProcessorTest.cls

```
@isTest
public class AccountProcessorTest {

    @isTest
    public static void testFunc() {
        account acc = new account();
        acc.name = 'MATW INC';
        insert acc;

        contact con = new contact();
        con.lastname = 'Mann1';
        con.AccountId = acc.Id;
        insert con;
        contact con1 = new contact();
        con1.lastname = 'Mann2';
        con1.AccountId = acc.Id;
        insert con1;

        List<Id> acc_list = new List<Id>();
        acc_list.add(acc.Id);
        Test.startTest();
        AccountProcessor.countContacts(acc_list);
        Test.stopTest();
        List<account> acc1 = new List<account>([select Number_of_Contacts__c f
:acc.id]);
        system.assertEquals(2, acc1[0].Number_of_Contacts__c);
    }
}
```

Use Batch Apex

Create an Apex class that implements the Database.Batchable interface to update all Lead records in the org with a specific LeadSource. Write unit tests that achieve 100%

code coverage for the class.

- Create an Apex class called 'LeadProcessor' that uses the Database.Batchable interface.
- Use a QueryLocator in the start method to collect all Lead records in the org.
- The execute method must update all Lead records in the org with the LeadSource value of 'Dreamforce'.
- Create an Apex test class called 'LeadProcessorTest'.
- In the test class, insert 200 Lead records, execute the 'LeadProcessor' Batch class and test that all Lead records were updated correctly.
- The unit tests must cover all lines of code included in the LeadProcessor class, resulting in 100% code coverage.
- Run your test class at least once (via 'Run All' tests the Developer Console) before attempting to verify this challenge.

Apex Class

```
global class LeadProcessor implements
Database.Batchable<sObject>, Database.Stateful {

    // instance member to retain state across transactions
    global Integer recordsProcessed = 0;

    global Database.QueryLocator start(Database.BatchableContext bc) {
        return Database.getQueryLocator('SELECT Id, LeadSource FROM
Lead');
    }

    global void execute(Database.BatchableContext bc, List<Lead> scope){
        // process each batch of records
        List<Lead> leads = new List<Lead>();
        for (Lead lead : scope) {

            lead.LeadSource = 'Dreamforce';
            // increment the instance member counter
```

```

        recordsProcessed = recordsProcessed + 1;

    }
    update leads;
}

global void finish(Database.BatchableContext bc){
    System.debug(recordsProcessed + ' records processed. Shazam!');
}
}

```

Apex Test Class

```

@isTest
public class LeadProcessorTest {
    @testSetup
    static void setup() {
        List<Lead> leads = new List<Lead>();
        // insert 200 leads
        for (Integer i=0;i<200;i++) {
            leads.add(new Lead(LastName='Lead '+i,
                               Company='Lead', Status='Open - Not Contacted'));
        }
        insert leads;
    }

    static testmethod void test() {
        Test.startTest();
        LeadProcessor lp = new LeadProcessor();
        Id batchId = Database.executeBatch(lp, 200);
        Test.stopTest();

        // after the testing stops, assert records were updated properly
    }
}

```



```
        System.assertEquals(200, [select count() from lead where
LeadSource = 'Dreamforce']);
    }
}
```

Control Processes with Queueable Apex

AddPrimarycontact.cls

```
public class AddPrimaryContact implements Queueable {

    public contact c;

    public String state;

    public AddPrimaryContact(Contact c, String state) {

        this.c = c;

        this.state = state;

    }

    public void execute(QueueableContext qc) {

        system.debug('this.c = '+this.c+' this.state = '+this.state);

        List<Account> acc_lst = new List<account>([select id, name, BillingState from
account where account.BillingState = :this.state limit 200]);

        List<contact> c_lst = new List<contact>();

        for(account a: acc_lst) {

            contact c = new contact();

            c = this.c.clone(false, false, false, false);

            c.AccountId = a.Id;

            c_lst.add(c);

        }

        insert c_lst;

    }

}
```

```
}
```

AddPrimaryContactTest.cls

```
@IsTest
```

```
public class AddPrimaryContactTest {
```

```
    @IsTest
```

```
    public static void testing() {
```

```
        List<account> acc_lst = new List<account>();
```

```
        for (Integer i=0; i<50;i++) {
```

```
            account a = new account(name=string.valueOf(i),billingstate='NY');
```

```
            system.debug('account a = '+a);
```

```
            acc_lst.add(a);
```

```
        }
```

```
        for (Integer i=0; i<50;i++) {
```

```
            account a = new account(name=string.valueOf(50+i),billingstate='CA');
```

```
            system.debug('account a = '+a);
```

```
            acc_lst.add(a);
```

```
        }
```

```
        insert acc_lst;
```

```
        Test.startTest();
```

```
        contact c = new contact(lastname='alex');
```

```
        AddPrimaryContact apc = new AddPrimaryContact(c, 'CA');
```

```
        system.debug('apc = '+apc);
```

```
        System.enqueueJob(apc);
```

```
        Test.stopTest();
```

```
        List<contact> c_lst = new List<contact>([select id from contact]);
```

```
        Integer size = c_lst.size();
```

```
system.assertEquals(50, size);  
  
}  
  
}
```

Schedule Jobs Using the Apex Scheduler

Create an Apex class that uses Scheduled Apex to update Lead records.

Create an Apex class that implements the Schedulable interface to update Lead records with a specific LeadSource. Write unit tests that achieve 100% code coverage for the class. This is very similar to what you did for Batch Apex.

- Create an Apex class called 'DailyLeadProcessor' that uses the Schedulable interface.
- The execute method must find the first 200 Leads with a blank LeadSource field and update them with the LeadSource value of 'Dreamforce'.
- Create an Apex test class called 'DailyLeadProcessorTest'.
- In the test class, insert 200 Lead records, schedule the DailyLeadProcessor class to run and test that all Lead records were updated correctly.
- The unit tests must cover all lines of code included in the DailyLeadProcessor class, resulting in 100% code coverage.
- Run your test class at least once (via 'Run All' tests the Developer Console) before attempting to verify this challenge.

Apex Class

```
global class DailyLeadProcessor implements Schedulable{  
    global void execute(SchedulableContext ctx){  
        List<Lead> leads = [SELECT Id, LeadSource FROM Lead WHERE  
LeadSource = ''];
```

```

        if(leads.size() > 0){
            List<Lead> newLeads = new List<Lead>();

            for(Lead lead : leads){
                lead.LeadSource = 'DreamForce';
                newLeads.add(lead);
            }

            update newLeads;
        }
    }
}

```

Apex Test Class

```

@Test
private class DailyLeadProcessorTest{
    //Seconds Minutes Hours Day_of_month Month Day_of_week optional_year
    public static String CRON_EXP = '0 0 0 2 6 ? 2022';

    static testmethod void testScheduledJob(){
        List<Lead> leads = new List<Lead>();

        for(Integer i = 0; i < 200; i++){
            Lead lead = new Lead(LastName = 'Test ' + i, LeadSource = '',
Company = 'Test Company ' + i, Status = 'Open - Not Contacted');
            leads.add(lead);
        }

        insert leads;

        Test.startTest();
        // Schedule the test job
    }
}

```

```
String jobId = System.schedule('Update LeadSource to DreamForce',  
    CRON_EXP, new DailyLeadProcessor());  
  
    // Stopping the test will run the job synchronously  
    Test.stopTest();  
}  
}
```

Monitor Asynchronous Apex

After completing this module, I knew:

- How to monitor the different types of jobs.
- How to use the flex queue.

Create Lightning Web Components

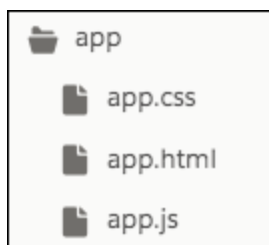
After completing this unit, I'll be able to:

- Describe the contents of each component file.
- Create JavaScript methods for a Lightning web component.
- Use Lifecycle Hooks in component JavaScript.

More Play Time

Say you want to build a data display element, independent of a specific object in Salesforce. A good example is the [productCard component](#) in the ebikes sample repo. Let's examine that card component, and build our own version from scratch so you can see how it evolves into a full-fledged Lightning web component. You'll quickly get the basics when you build up the parts of a component and explore more samples.

A component simply needs a folder and its files with the same name. They're automatically linked by name and location.



All Lightning web components have a namespace that's separated from the folder name by a hyphen. For example, the markup for the Lightning web component with the folder name `app` in the default namespace `c` is `<c-app>`.

However, the Salesforce platform doesn't allow hyphens in the component folder or file names. What if a component's name has more than one word, like "mycomponent"? You can't name the folder and files `my-component`, but we do have a handy solution.

Use camel case to name your component `myComponent`. Camel case component folder names map to kebab-case in markup. In markup, to reference a component with the folder name `myComponent`, use `<c-my-component>`.

For example, the LWC Samples repo has the [viewSource folder](#) containing the files for the `viewSource` component. When the `hello` component [references the viewSource component in HTML](#), it uses `c-view-source`.

A Look Into the HTML File

Lightning web component HTML files all include the `template` tag. The `template` tag contains the HTML that defines the structure of your component. Let's look at the HTML for a simplified version of the `productCard` component from the `ebikes` repo.

Follow along by pasting these examples in [WebComponents.dev](#).

1. Navigate to [webcomponents.dev](#) and select **LWC**. (Or go directly to [webcomponents.dev/create/lwc](#).)
2. If you aren't already logged in, use your GitHub account to log in to [WebComponents.dev](#).
3. Paste the following into `app.html` (replacing any existing HTML in the file).

```
4. <template>
    <div>
        <div>Name: {name}</div>
        <div>Description: {description}</div>
        <div>Category: {category}</div>
        <div>Material: {material}</div>
```

```

        <div>Price: {price}</div>

        <div><img src={pictureUrl}/></div>

    </div>
</template>

```

5. Copy

6. The identifiers in the curly braces {} are bound to the fields of the same name in the corresponding JavaScript class.

7. Paste the following into `app.js`.

```

8. import { LightningElement } from 'lwc';
export default class App extends LightningElement {
    name = 'Electra X4';
    description = 'A sweet bike built for comfort.';
    category = 'Mountain';
    material = 'Steel';
    price = '$2,700';
    pictureUrl = 'https://s3-us-west-1.amazonaws.com/sfdc-
demo/ebikes/electrax4.jpg';
}

```

9. Copy

10. Click the update icon.

You see a bright shiny bike!

Now let's play with a real-world example. Say you want to display data, but you know it can take some time to load. You don't want the user wondering what's up. You can use `if:false` and `if:true` conditional directives within your template to determine which visual elements are rendered.

1. Paste the following into `app.html`. The content in the "display" `div` tag doesn't appear until the value of `ready` is `true` in the HTML file.

```

2. <template>

    <div id="waiting" if:false={ready}>Loading...</div>

    <div id="display" if:true={ready}>

```

```

        <div>Name: {name}</div>

        <div>Description: {description}</div>

        <div>Category: {category}</div>

        <div>Material: {material}</div>

        <div>Price: {price}</div>

        <div><img src={pictureUrl}/></div>

    </div>
</template>

```

3. Copy

4. Paste the following into app.js. This holds our data values and sets a 3-second timer. After 3 seconds, the content should appear. (Of course, this is only for testing purposes.)

```

5. import { LightningElement } from 'lwc';

export default class App extends LightningElement {
    name = 'Electra X4';
    description = 'A sweet bike built for comfort.';
    category = 'Mountain';
    material = 'Steel';
    price = '$2,700';
    pictureUrl = 'https://s3-us-west-1.amazonaws.com/sfdc-
demo/ebikes/electrax4.jpg';
    ready = false;

    connectedCallback() {
        setTimeout(() => {
            this.ready = true;
        }, 3000);
    }
}

```

6. Copy

Base Lightning Web Components

Now, you don't want to build all your components from the ground up. So let's explore using a base Lightning web component. And of course, there are lots of components, including field types, display controllers, navigation items, and more. All of them are listed in the [Component Reference](#).

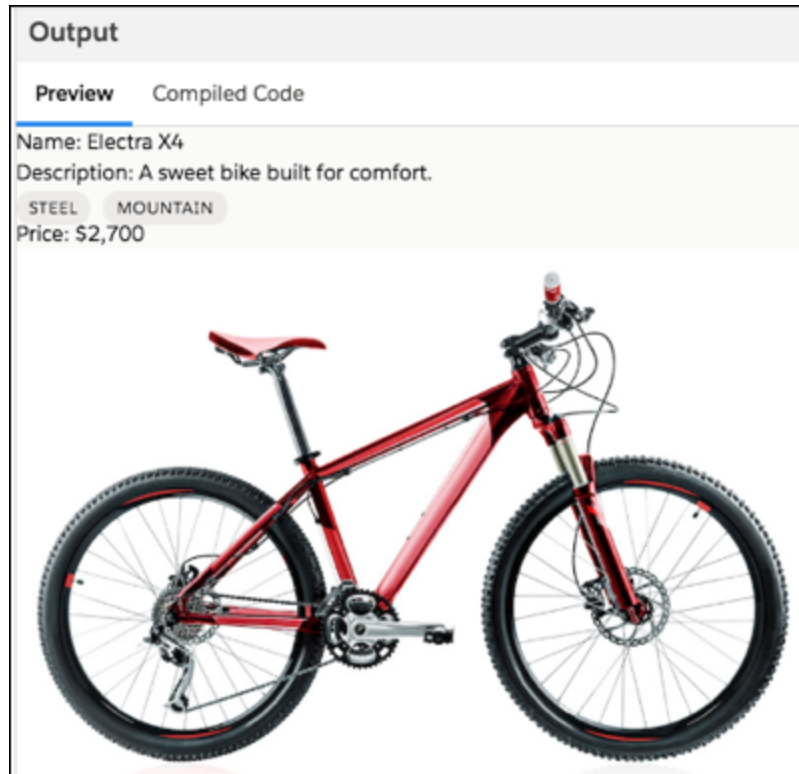
Let's make the details of the bike stand out. In the app.html file, replace the div tags for material and category in the last example with a lightning-badge component. Here's the HTML.

```
<template>

  <div id="waiting" if:false={ready}>Loading...</div>
  <div id="display" if:true={ready}>
    <div>Name: {name}</div>
    <div>Description: {description}</div>
    <lightning-badge label={material}></lightning-badge>
    <lightning-badge label={category}></lightning-badge>
    <div>Price: {price}</div>
    <div><img src={pictureUrl}/></div>
  </div>
</template>
```

Copy

The words **Steel** and **Mountain** appear as badges. It's that simple.



OK. Let's look at the JavaScript.

Working with JavaScript

Here's where you make stuff happen. As you've seen so far, JavaScript methods define what to do with input, data, events, changes to state, and more to make your component work.

The JavaScript file for a Lightning web component must include at least this code, where `MyComponent` is the name you assign your component class.

```
import { LightningElement } from 'lwc';
export default class MyComponent extends LightningElement {
}
```

Copy

The `export` statement defines a class that extends the `LightningElement` class. As a best practice, the name of the class usually matches the file name of the JavaScript class, but it's not a requirement.

The LWC Module

Lightning Web Components uses modules (built-in modules were introduced in ECMAScript 6) to bundle core functionality and make it accessible to the JavaScript in your component file. The core module for Lightning web components is `lwc`.

Begin the module with the `import` statement and specify the functionality of the module that your component uses.

The `import` statement indicates the JavaScript uses the `LightningElement` functionality from the `lwc` module.

```
// import module elements
import { LightningElement } from 'lwc';
// declare class to expose the component
export default class App extends LightningElement {
  ready = false;
  // use lifecycle hook
  connectedCallback() {
    setTimeout(() => {
      this.ready = true;
    }, 3000);
  }
}
```

Copy

- `LightningElement` is the base class for Lightning web components, which allows us to use `connectedCallback()`.
- The `connectedCallback()` method is one of our lifecycle hooks. You'll learn more about lifecycle hooks in the next section. For now, know that the method is triggered when a component is inserted in the document object model (DOM). In this case, it starts the timer.

Deploy Lightning Web Component Files

Learning Objectives

After completing this unit, I'll be able to:

- Configure Lightning web component files for display in an org.
- Deploy your files to an org.
- Verify component behavior in an org environment.

Step up to an Org

In this unit, we develop a Lightning web component using VS Code with the Salesforce extension. We deploy the files to an org and build an app to use your component.

What You Need

As stated in the first unit, you need some familiarity with Salesforce DX to continue. To complete this unit, you need:

- Visual Studio Code with the Salesforce Extension Pack
- Salesforce CLI
- Dev Hub enabled org
- My Domain deployed to users in your Dev Hub enabled org (Playground orgs created within Trailhead have My Domain deployed for you. If you associated a Developer Edition org with your Trailhead account, enable and deploy My Domain.)

To meet these requirements, complete the [Quick Start: Lightning Web Components](#) project. Enable Dev Hub and deploy My Domain from the Setup menu in your org, if they're not already configured.

Set Up Lightning Web Component Files for Use in an Org

You're going to create the bike component discussed in the Create Lightning Component unit and push it to your org.

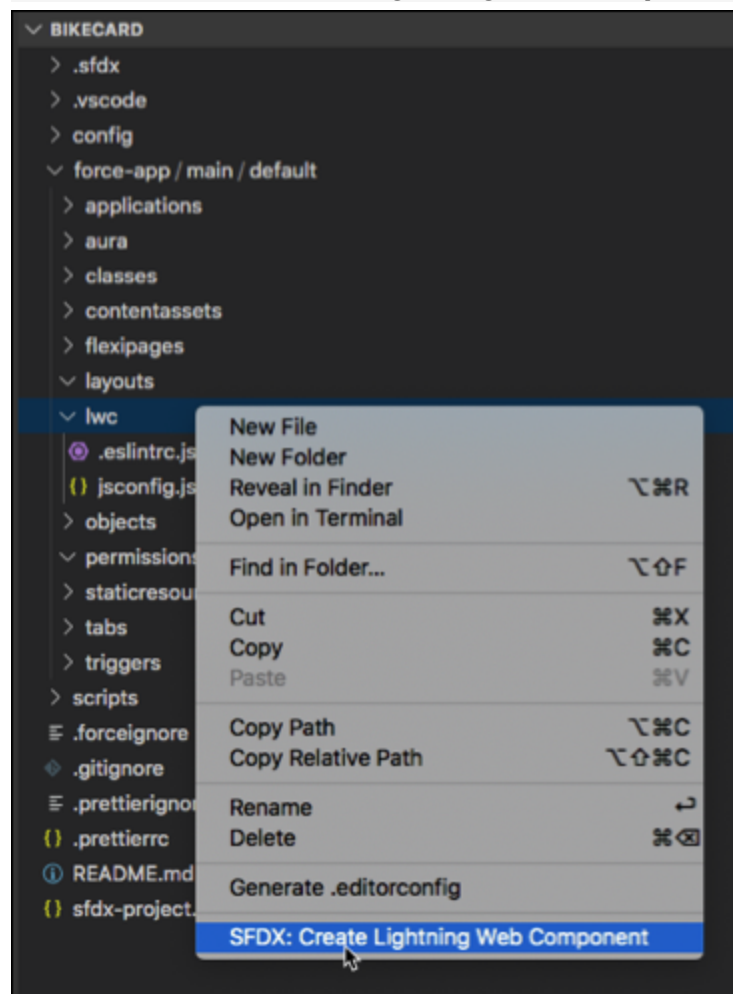


Note

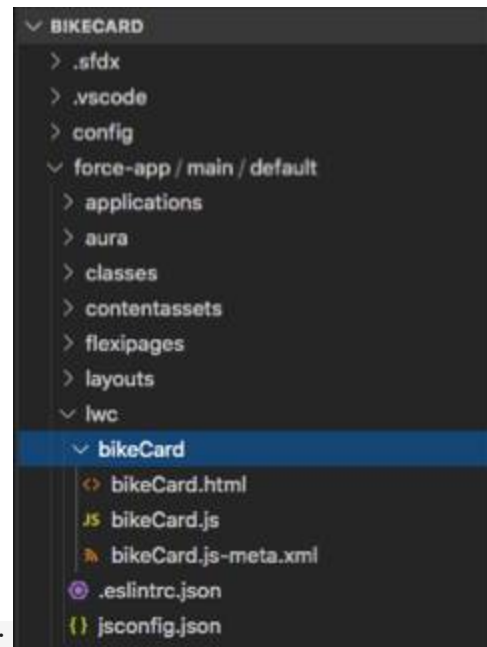
We're not defining any styling of our own, so we don't need a CSS file.

The files you need to push a component to an org:

- bikeCard.html
 - bikeCard.js
 - bikeCard.js-meta.xml
1. Create a project by selecting **SFDX: Create Project** from the Command Palette in VS Code. Accept the standard template and give it the project name **bikeCard**.
 2. Create a **bikeCard** component folder and files by right-clicking on the **lwc** folder and select **SFDX: Create Lightning Web Component**.



3. Enter the filename **bikeCard**.
4. Save the contents of the files under bikeCard\force-app\main\default\lwc so



you see the following in VS Code: Lightning web components follow web standards. The HTML standard recommends that custom element names contain a hyphen. However, the Salesforce platform doesn't allow hyphens in the component folder or file names. So we use camelCase naming conventions here.

5. Copy the contents for the bikeCard.html, bikeCard.js, and bikeCard.js-meta.xml files.

6. **bikeCard.html**

```
7. <template>
    <div>
        <div>Name: {name}</div>
        <div>Description: {description}</div>
        <lightning-badge label={material}></lightning-badge>
        <lightning-badge label={category}></lightning-badge>
        <div>Price: {price}</div>
        <div><img src={pictureUrl}/></div>
    </div>
</template>
```

8. Copy

9. **bikeCard.js**

```
10. import { LightningElement } from 'lwc';

    export default class BikeCard extends LightningElement {
        name = 'Electra X4';
        description = 'A sweet bike built for comfort.';
        category = 'Mountain';
        material = 'Steel';
        price = '$2,700';
        pictureUrl = 'https://s3-us-west-1.amazonaws.com/sfdc-
demo/ebikes/electrax4.jpg';
    }
```

11. Copy

12. bikeCard.js-meta.xml

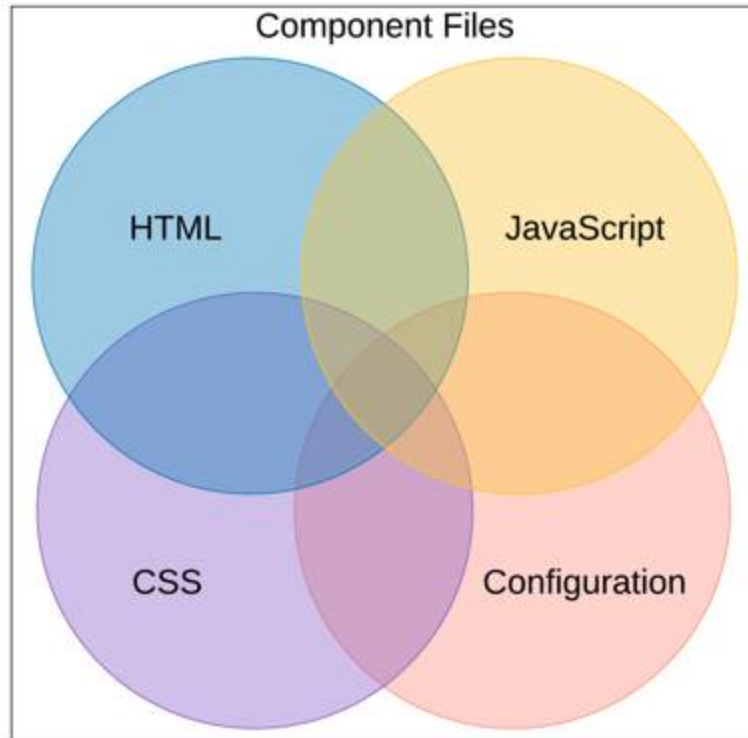
```
13. <?xml version="1.0" encoding="UTF-8"?>
    <LightningComponentBundle
xmlns="http://soap.sforce.com/2006/04/metadata">
        <!-- The apiVersion may need to be increased for the current
release -->
        <apiVersion>52.0</apiVersion>
        <isExposed>true</isExposed>
        <masterLabel>Product Card</masterLabel>
        <targets>
            <target>lightning__AppPage</target>
            <target>lightning__RecordPage</target>
            <target>lightning__HomePage</target>
        </targets>
    </LightningComponentBundle>
```

14. Copy

15. Save your files.

The Component Configuration File

The file we haven't covered yet is the component configuration file with the extension `.js-meta.xml`. This file provides metadata for Salesforce, including the design configuration for components intended for use in Lightning App Builder.



We haven't covered configuration files yet, because we've been playing in WebComponents.dev. Now that you're going to start using the content within an org, you must include a configuration file.

Notice that the ebikes repo components all have this configuration file. Here's [an example from the ebikes repo](#):

```
<?xml version="1.0" encoding="UTF-8" ?>
<LightningComponentBundle
  xmlns="http://soap.sforce.com/2006/04/metadata">
  <apiVersion>52.0</apiVersion>
  <isExposed>true</isExposed>
```



```

<masterLabel>Product Card</masterLabel>
<targets>
  <target>lightning__AppPage</target>
  <target>lightning__RecordPage</target>
  <target>lightning__HomePage</target>
  <target>lightningCommunity__Page</target>
</targets>
<targetConfigs>
  <targetConfig targets="lightning__RecordPage">
    <objects>
      <object>Product__c</object>
    </objects>
  </targetConfig>
</targetConfigs>
</LightningComponentBundle>

```

Copy

Required *apiVersion* binds the component to a Salesforce API version.

isExposed (*true* or *false*) makes the component available from other namespaces. Only set this to true to make a Lightning component usable in these specific cases:

- From a managed package in Aura
- From Lightning App Builder in another org

Optional *targets* specify which types of Lightning pages the component can be added to in the Lightning App Builder.

targetConfigs let you specify behavior specific to each type of Lightning page, including things like which objects support the component.

See [the documentation](#) for the full list of supported syntax.

Displaying a Component in an Org

You have two options for displaying a Lightning web component in the UI.

1. Set the component to support various flexipage types (home, record home, and

so on) then add it to a flexipage using the Lightning App Builder. This is the simplest approach and the one you follow in this unit.

2. You can also create a tab which points to an Aura component containing your Lightning web component. You can see the required pieces in the repo.
 - [Wrapper Components](#)
 - [Tabs](#)
 - [Visibility Settings](#)
 - [Default application configuration file](#)

Deploy Your Files

Now, you need to deploy the component files to your Dev Hub enabled org.


1. Authenticate with your Dev Hub org using **SFDX: Authorize an Org** from the Command Palette in VS Code. When prompted, accept the Project Default and press **Enter** to accept the default alias. If prompted to allow access, click **Allow**.
2. Deploy the project files using **SFDX: Deploy this Source to Org** from the Command Palette in VS Code.

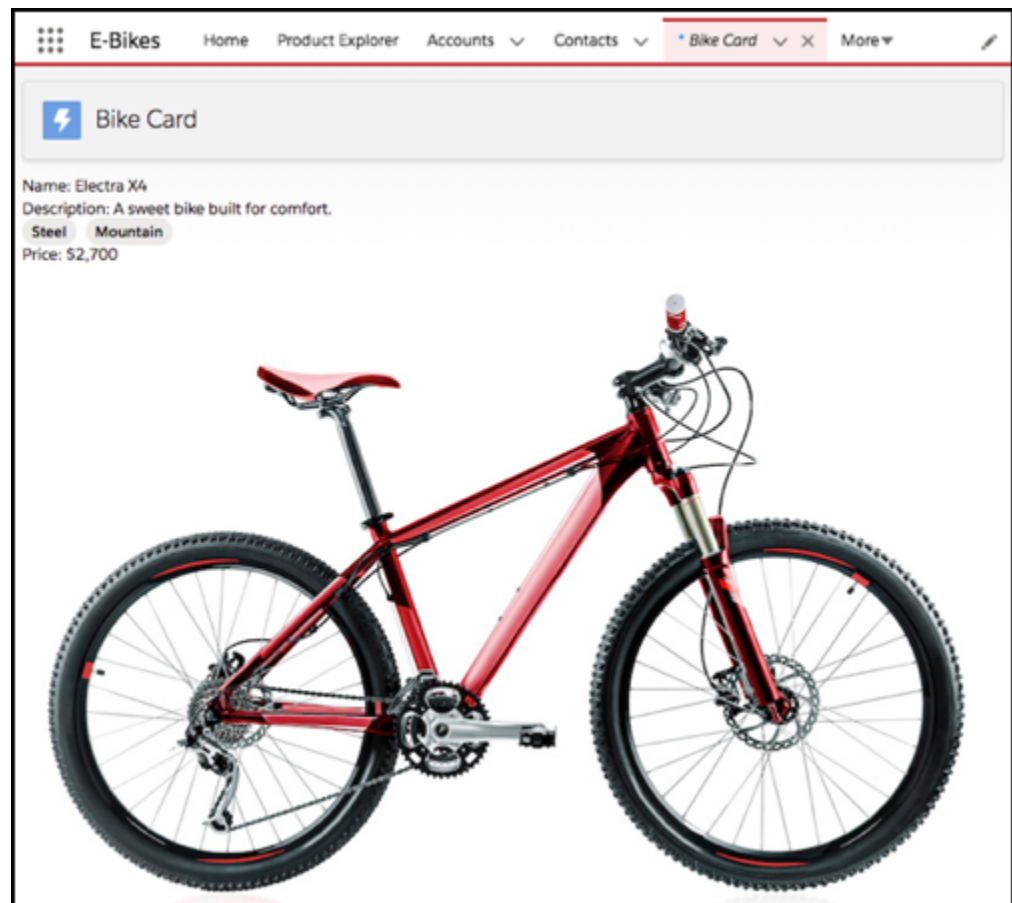
Create a New Page for Your Component

Since we set up our component configuration file to enable the use of the component in Lightning App Builder, use the UI to create an app and add your component to it.

1. To open your org, use **SFDX: Open Default Org** from the Command Palette in VS Code.
2. In Setup, enter Lightning App Builder in the Quick Find box and then select **Lightning App Builder**.
3. Click **New**.
4. Select **App Page** and Click **Next**.
5. Give it the label Bike Card and Click **Next**.
6. Select **One Region** and click **Finish**.
7. In Lightning App Builder, scroll down the Lightning components list on the left side until you see your Product Card component.

Now you can drag it onto the page. Save the page, Activate it, and the Product Card component shows up on the assigned page.

1. Drag your Product Card component to the top of the page layout until the bike appears.
2. Click **Save**.
3. Click **Activate**.
4. Keep **Activate for all users** selected. And, optionally, change the name or icon for your app.
5. Click **Save**. You're asked to add your page to navigation menus, but you don't need to. You can still get to your page in this environment.
6. Click **Finish**.
7. Click **Back** in the upper left corner to exit the Lightning App Builder.
8. From the App Launcher (), find and select **Bike Card**.
9. Open it and see your component working in the UI.



There you go, a shiny new bike. You've pushed a component to an org, seen it on the page, and can verify it in the UI.

In the next unit, you build an interactive component with event handling and deploy it to your org for testing.

Lightning Web Components provides methods that allow you to “hook” your code up to critical events in a component’s lifecycle. These events include when a component is:

- Created
- Added to the DOM
- Rendered in the browser
- Encountering errors
- Removed from the DOM

Respond to any of these lifecycle events using callback methods. For example, the `connectedCallback()` is invoked when a component is inserted into the DOM. The `disconnectedCallback()` is invoked when a component is removed from the DOM.

In the JavaScript file we used to test our conditional rendering, we used the `connectedCallback()` method to automatically execute code when the component is inserted into the DOM. The code waits 3 seconds, then sets `ready` to `true`.

```
import { LightningElement } from 'lwc';
export default class App extends LightningElement {
  ready = false;
  connectedCallback() {
    setTimeout(() => {
      this.ready = true;
    }, 3000);
  }
}
```

Handle Events in Lightning Web Components

Lets take one example for “**Declarative via html markup**”

Step 1) Create one child component component from where we will raise a

event

Create child html file to get value from user

childComp.html

```
<template>
  <lightning-card title="Child Component">
    <div class="slds-m-around_medium">
      <lightning-input name="textVal" label="Enter Text"
onchange={handleChange}></lightning-input>
    </div>
  </lightning-card>
</template>
```

Step 2) Now update Child Comp javascript file to raise a CustomEvent with text value

childComp.js

```
import { LightningElement } from 'lwc';

export default class ChildComp extends LightningElement {

  handleChange(event) {
    event.preventDefault();
    const name = event.target.value;
    const selectEvent = new CustomEvent('mycustomevent', {
      detail: name
    });
    this.dispatchEvent(selectEvent);
  }
}
```

Step 3) Create one Parent component where we will handle the event

Now create parent component. We need to add prefix as “on” before the

custom event name and in parent component we need to invoke the event listener as `handleCustomEvent` using `onmycustomevent` attribute. we recommend that you conform with the DOM event standard.

- No uppercase letters
- No spaces
- Use underscores to separate words

ParentComponent.html

```
<template>
  <div class="slds-m-around_medium">
    Value From Child : {msg}
    <c-child-comp onmycustomevent={handleCustomEvent}></c-child-comp>
  </div>
</template>
```

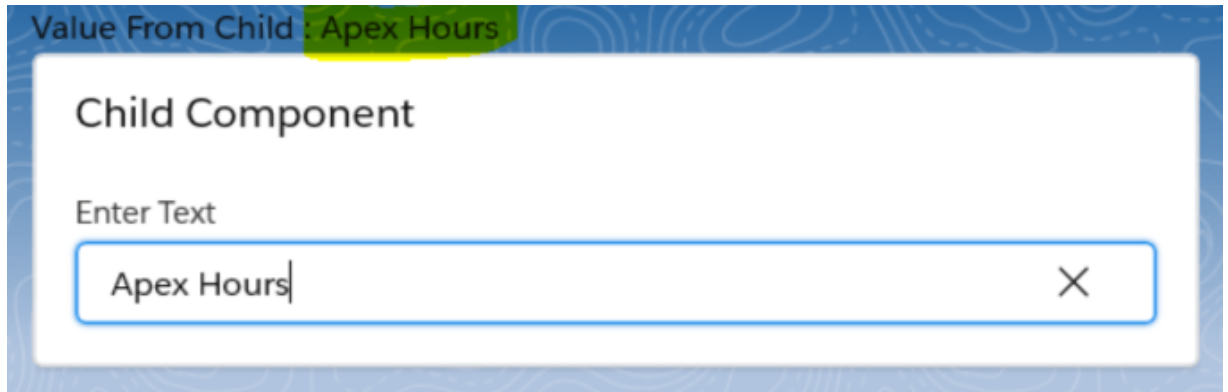
Now update parent component javascript file and add `handleCustomEvent` method

ParentComponent.js

```
import { LightningElement , track } from 'lwc';

export default class ParentComponent extends LightningElement {
  @track msg;
  handleCustomEvent(event) {
    const textVal = event.detail;
    this.msg = textVal;
  }
}
```

Output :-



Lets see how we can use “**JavaScript using addEventListener method | Attaching event Listener Programmatically**”

Step 1) We can use the same above sample and do below change in parent component

Update parent component JavaScript like below

ParentComponent.js

```
import { LightningElement , track } from 'lwc';

export default class ParentComponent extends LightningElement {
  @track msg;

  constructor() {
    super();
    this.template.addEventListener('mycustomevent',
this.handleCustomEvent.bind(this));
  }

  handleCustomEvent(event) {
    const textVal = event.detail;
    this.msg = textVal;
  }
}
```

Remove onmycustomevent attribute from child component tag. like below

ParentComponent.html

```
<template>
  <div class="slds-m-around_medium">
    Value From Child : {msg}
    <c-child-comp ></c-child-comp>
  </div>
</template>
```

Step 3) Set the bubbles: true while raising the event like below

childComp.js

```
import { LightningElement } from 'lwc';

export default class ChildComp extends LightningElement {

  handleChange(event) {
    event.preventDefault();
    const name = event.target.value;
    const selectEvent = new CustomEvent('mycustomevent', {
      detail: name ,bubbles: true
    });
    this.dispatchEvent(selectEvent);
  }
}
```

Event Propagation: When an event is fired the event is propagated up to the DOM. Event propagation typically involves two phases event bubbling and event capturing. The most commonly used event propagation phase for handling events is event bubbling. In this case the event is triggered at the child level and propagates up to the DOM. Where as event capturing phases moves from top to bottom of the DOM. This phase is rarely used for event handling.

The below picture shows the event phases both in capture and bubbles phase. In LWC we have two flags which determines the behavior of event in event bubbling phase.

1. **bubbles** A Boolean value indicating whether the event bubbles up through the DOM or not. Defaults to false.
2. **composed** A Boolean value indicating whether the event can pass through the shadow boundary. Defaults to false.

====APEX REST CALLOUTS

--AnimalLocator.cls

```
public class AnimalLocator {
    public class cls_animal {
        public Integer id;
        public String name;
        public String eats;
        public String says;
    }
    public class JSONOutput{
        public cls_animal animal;

        //public JSONOutput parse(String json){
        //return (JSONOutput) System.JSON.deserialize(json,
JSONOutput.class);
        //}
    }

    public static String getAnimalNameById (Integer id) {
        Http http = new Http();
        HttpRequest request = new HttpRequest();
        request.setEndpoint('https://th-apex-http-
callout.herokuapp.com/animals/' + id);
        //request.setHeader('id', String.valueOf(id)); -- cannot be
used in this challenge :)
        request.setMethod('GET');
        HttpResponse response = http.send(request);
        system.debug('response: ' + response.getBody());
        //Map<String, Object> map_results = (Map<String, Object>)
```

```

JSON.deserializeUntyped(response.getBody());
    jsonOutput results = (jsonOutput)
JSON.deserialize(response.getBody(), jsonOutput.class);
    //Object results = (Object) map_results.get('animal');
    system.debug('results= ' + results.animal.name);
    return(results.animal.name);
}

}

```

----AnimalLocatorMock.cls

```

@IsTest
global class AnimalLocatorMock implements HttpCalloutMock {

    global HTTPResponse respond(HTTPPrerequest request) {
        Httpresponse response = new Httpresponse();
        response.setStatusCode(200);
        //-- directly output the JSON, instead of creating a logic
        //response.setHeader('key, value)
        //Integer id = Integer.valueOf(request.getHeader('id'));
        //Integer id = 1;
        //List<String> lst_body = new List<String> {'majestic badger',
'fluffy bunny'};
        //system.debug('animal return value: ' + lst_body[id]);

        response.setBody('{"animal":{"id":1,"name":"chicken","eats":"chicken
food","says":"cluck cluck"}}');
        return response;
    }
}

```

```
}
```

--AnimalLocatorTest.cls

```
@IsTest
public class AnimalLocatorTest {
    @isTest
    public static void testAnimalLocator() {
        Test.setMock(HttpCalloutMock.class, new AnimalLocatorMock());
        //HttpResponse response = AnimalLocator.getAnimalNameById(1);
        String s = AnimalLocator.getAnimalNameById(1);
        system.debug('string returned: ' + s);
    }
}
```

Apex SOAP Callouts

Generate an Apex class using WSDL2Apex and write a test class.

Generate an Apex class using WSDL2Apex for a SOAP web service, write unit tests that achieve 100% code coverage for the class using a mock response, and run your Apex tests.

- Use WSDL2Apex to generate a class called 'ParkService' in public scope using this WSDL file. After you click the 'Parse WSDL' button don't forget to change the name of the Apex Class Name from 'parksServices' to 'ParkService'.
- Create a class called 'ParkLocator' that has a 'country' method that uses the 'ParkService' class and returns an array of available park names for a particular country passed to the web service. Possible country names that can be passed to the web service include Germany, India, Japan and United States.
- Create a test class named ParkLocatorTest that uses a mock class called ParkServiceMock to mock the callout response.
- The unit tests must cover all lines of code included in the ParkLocator class, resulting in 100% code coverage.

- Run your test class at least once (via 'Run All' tests the Developer Console) before attempting to verify this challenge.

Apex Service

//Generated by wsdl2apex

```
public class ParkService {
    public class byCountryResponse {
        public String[] return_x;
        private String[] return_x_type_info = new
String[]{'return','http://parks.services/',null,'0','-1','false'};
        private String[] apex_schema_type_info = new
String[]{'http://parks.services/','false','false'};
        private String[] field_order_type_info = new String[]{'return_x'};
    }
    public class byCountry {
        public String arg0;
        private String[] arg0_type_info = new
String[]{'arg0','http://parks.services/',null,'0','1','false'};
        private String[] apex_schema_type_info = new
String[]{'http://parks.services/','false','false'};
        private String[] field_order_type_info = new String[]{'arg0'};
    }
    public class ParksImplPort {
        public String endpoint_x = 'https://th-apex-soap-
service.herokuapp.com/service/parks';
        public Map<String,String> inputHttpHeaders_x;
        public Map<String,String> outputHttpHeaders_x;
        public String clientCertName_x;
        public String clientCert_x;
        public String clientCertPasswd_x;
        public Integer timeout_x;
        private String[] ns_map_type_info = new
String[]{'http://parks.services/','ParkService'};
        public String[] byCountry(String arg0) {
            ParkService.byCountry request_x = new ParkService.byCountry();
```

```

        request_x.arg0 = arg0;
        ParkService.byCountryResponse response_x;
        Map<String, ParkService.byCountryResponse> response_map_x = new
Map<String, ParkService.byCountryResponse>();
        response_map_x.put('response_x', response_x);
        WebServiceCallout.invoke(
            this,
            request_x,
            response_map_x,
            new String[]{endpoint_x,
'',
'http://parks.services/',
'byCountry',
'http://parks.services/',
'byCountryResponse',
'ParkService.byCountryResponse'}
        );
        response_x = response_map_x.get('response_x');
        return response_x.return_x;
    }
}
}

```

Apex Class

```

public class ParkLocator {
    public static String[] country(String country){
        ParkService.ParksImplPort parks = new ParkService.ParksImplPort();
        String[] parksname = parks.byCountry(country);
        return parksname;
    }
}

```

Apex Test Class

```
@isTest
private class ParkLocatorTest{
    @isTest
    static void testParkLocator() {
        Test.setMock(WebServiceMock.class, new ParkServiceMock());
        String[] arrayOfParks = ParkLocator.country('India');

        System.assertEquals('Park1', arrayOfParks[0]);
    }
}
```

Apex Mock Test Class

```
@isTest
global class ParkServiceMock implements WebServiceMock {
    global void doInvoke(
        Object stub,
        Object request,
        Map<String, Object> response,
        String endpoint,
        String soapAction,
        String requestName,
        String responseNS,
        String responseName,
        String responseType) {
        ParkService.byCountryResponse response_x = new
ParkService.byCountryResponse();
        List<String> lstOfDummyParks = new List<String>
```

```

{'Park1', 'Park2', 'Park3'};
    response_x.return_x = 1stOfDummyParks;

    response.put('response_x', response_x);
}
}

```

Apex Web Services

Create an Apex REST service that returns an account and it's contacts.

To pass this challenge, create an Apex REST class that is accessible at '/Accounts/<Account_ID>/contacts'. The service will return the account's ID and Name plus the ID and Name of all contacts associated with the account. Write unit tests that achieve 100% code coverage for the class and run your Apex tests.

- The Apex class must be called 'AccountManager'.
- The Apex class must have a method called 'getAccount' that is annotated with @HttpGet and returns an Account object.
- The method must return the ID and Name for the requested record and all associated contacts with their ID and Name.
- The unit tests must be in a separate Apex class called 'AccountManagerTest'.
- The unit tests must cover all lines of code included in the AccountManager class, resulting in 100% code coverage.
- Run your test class at least once (via 'Run All' tests the Developer Console) before attempting to verify this challenge.

Apex Class

```
@RestResource(urlMapping='/Accounts/*/contacts')
```

```

global with sharing class AccountManager{
    @HttpGet
    global static Account getAccount(){
        RestRequest req = RestContext.request;
        String accId = req.requestURI.substringBetween('Accounts/',
'/contacts');
        Account acc = [SELECT Id, Name, (SELECT Id, Name FROM Contacts)
                        FROM Account WHERE Id = :accId];

        return acc;
    }
}

```

Apex Test Class

```

@IsTest
private class AccountManagerTest{
    @isTest static void testAccountManager(){
        Id recordId = getTestAccountId();
        // Set up a test request
        RestRequest request = new RestRequest();
        request.requestUri =
            'https://ap5.salesforce.com/services/apexrest/Accounts/'+
recordId + '/contacts';
        request.httpMethod = 'GET';
        RestContext.request = request;

        // Call the method to test
        Account acc = AccountManager.getAccount();

        // Verify results
        System.assert(acc != null);
    }
}

```



```
private static Id getTestAccountId(){
    Account acc = new Account(Name = 'TestAcc2');
    Insert acc;

    Contact con = new Contact(LastName = 'TestCont2', AccountId =
acc.Id);
    Insert con;

    return acc.Id;
}
}
```

CONCLUSION

Working as an intern the SALESFORCE COMPANY it was so oppurtunitive and motivative. There are many skills which I learnt while doing this internship like APEX which was totally novel to me and I am hoping this experience would give me great oppurtunities. And I would like to Thank those who were guiding me to step into the success of this Internship.