salesforce

DEVELOPERS

UNIT-3

By
Mrs. P.SHYAMALA

UNIT–III

Apex Testing: Get Started with Apex Unit Tests, Test Apex Triggers, Create Test Data for Apex Tests

Developer Console Basics: Get Started with the Developer Console, Navigate and Edit Source Code, Generate and Analyze Logs, Inspect Objects at Checkpoints.

Apex Testing - Get Started with Apex Unit Tests

Apex Testing :

- Apex testing is the **process of writing and running test code** to ensure that **Apex code components work as expected.**

- It helps you to **create relevant test data in the test classes and run them in Salesforce**.

As a part of Apex testing, you need to **test for:**

- **Positive behaviour:** Test Apex code to verify that it works as per specification when appropriate input is provided to the code.

- **Negative behaviour:** Test the limitations of the system if an unexpected value or input is provided to the Apex code.

- **Restricted user:** Check that a user whose access to certain objects or records is restricted is barred from accessing them.

Apex Testing - Get Started with Apex Unit Tests

Apex Unit Tests:

- Apex provides a **testing framework** that allows you to
  - **write unit tests,**
  - **run your tests,**
  - **check test results, and**
  - **have code coverage results for your Apex classes and triggers** on the Lightning Platform.
- Apex unit tests **ensure high quality for your Apex code** and let you meet requirements for deploying Apex.
- Testing is the key to **successful long-term development** and is a critical component of the development process.
- The Apex testing framework makes it easy to test your Apex code.

Note: In Salesforce, if the **Apex code is not properly tested, it can't be deployed into the Production environment.**

Apex Testing - Get Started with Apex Unit Tests

These are the benefits of Apex unit tests:

1. **Ensuring** that your **Apex classes and triggers work as expected**
2. 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**
3. **Meeting the code coverage requirements** for deploying Apex to production or distributing Apex to customers via packages
4. **High-quality apps delivered to the production org**, which makes production users more productive
5. **High-quality apps delivered to package subscribers**, which increase your customers trust

Apex Testing - Get Started with Apex Unit Tests

Code Coverage Requirement for Deployment:

- **At least 75%** of your Apex Code must be covered by unit tests, and all of those tests must complete successfully.
- Every **trigger must have some test coverage**.
- All classes and triggers **must compile successfully.**

There are two ways of testing a salesforce application:

- **Manual -** through the Salesforce user interface, important, but merely testing through the UI will not catch all the use cases for your application.
- **Automated -** to test for bulk functionality through the code if it's invoked using SOAP API or by Visualforce standard set controller.

Apex Testing - Get Started with Apex Unit Tests

How To Write Apex Unit Tests?

- **Unit tests are class methods** that verify whether a particular piece of code is working properly or not.

- Unit test **methods take no arguments, commit no data to the database, send no emails**, and are flagged with the **testMethod** keyword or the **@isTest annotation** in the method definition.

- Also, **test methods must be defined in test classes**, that is, classes annotated with isTest.

- Create **different test methods to test different functionalities.**

- In each test, method writes **different test cases to test your code** whether it is working properly with the different inputs or not.

Apex Testing - Get Started with Apex Unit Tests

Test Method Syntax:

Test methods are defined using the **@isTest** annotation and have the following syntax, or using **testmethod** keyword and have the following syntax,

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

(OR)

```
static testMethod void myTest() {

// code_block
  }
```

- The **visibility of a test method doesn't matter**, so declaring a test method as public or private doesn't make a difference as the testing framework is always able to access test methods.

- For this reason, **the access modifiers are omitted** in the syntax.

- **Test methods must be defined in test classes**, which are **classes annotated with @isTest**.

Apex Testing - Get Started with Apex Unit Tests

This sample class shows a definition of a test class with one test method.

- **Test classes** can be **either private or public.**

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

- If you're using a **test class for unit testing only, declare it as private.**

Note: Test methods aren't allowed in non-test classes

Apex Testing - Get Started with Apex Unit Tests

Unit Test Example 1- Test the add Class:

The following simple example is of a test class with **three test methods**.

The class method that's being tested takes a two positive integer numbers as an input.

It adds the two numbers and returns the summed result.

Let's add the custom class and its test class.

1. In the **Developer Console**, click
   File | New | Apex Class, and enter **Add for the class name**, and then click **OK**.

Apex Testing - Get Started with Apex Unit Tests

2. Write the following coding in the **add class body**.

Apex Class – Add.apxc

```
public class Add
 {
   public static integer addvalue (integer a, integer b)
   {
      if( a<0 || b<0 )
      {
         return -1;
      }
      else{
      integer result = a + b;
      return result;
      }
   }
}
```

3.Create the **AddTest class and write test cases to test class Add**

Apex Class – AddTest.apxc

```
@isTest
public class AddTest {
   @isTest
   private static void addValueTest1(){
      integer  res = Add.addvalue(10,-12);
      system.assertEquals(-1,res);
   }
@isTest
   private static void addValueTest2(){
      integer  res = Add.addvalue(10,20);
      system.assertEquals(30,res);
   }
   @isTest
   private static void addValueTest3(){
      integer  res = Add.addvalue(32,20);
      system.assertEquals(10,res);
   }
}
```

Apex Testing - Get Started with Apex Unit Tests

- The **AddTest** test class verifies that the method works as expected by calling it with different inputs.
- Each test method verifies one type of input:
    - Given input is **Positive or Negative,**
    - Result by adding two number has **Positive Behaviour,**
    - Result by adding two number has **Negative Behaviour,**

The verifications are done by calling the **System.assertEquals()** method, which takes two parameters:

1. Expected value, and

2. Actual value.

Apex Testing - Get Started with Apex Unit Tests

Let's run the methods in this class.

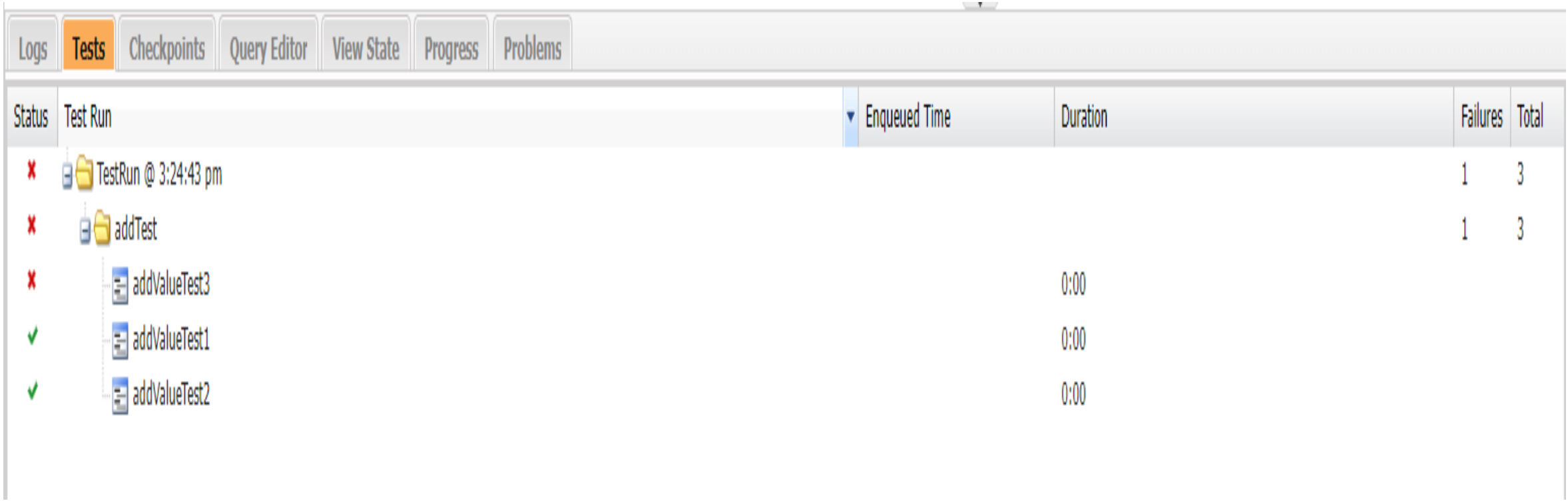
1. Under **Test Classes**, click **RunTest.**

add.apxc | addTest.apxc *

Code Coverage: None | API Version: 61 | Run Test | Go To

```
@isTest
public class addTest {
    @isTest
    private static void addValueTest1(){
        integer res = add.addvalue(10,-12);
        system.assertEquals(-1,res);
    }
@isTest
    private static void addValueTest2(){
        integer res = add.addvalue(10,20);
        system.assertEquals(30,res);
    }
    @isTest
    private static void addValueTest3(){
        integer res = add.addvalue(32,20);
        system.assertEquals(10,res);
    }
}
```

2. In the **Tests tab**, you see the **status of your tests as they're running.** Expand the test run, and expand again until you see the list of individual tests that were run.
They all have **green checkmarks.**



Apex Testing - Get Started with Apex Unit Tests

- After you run tests, **code coverage is automatically generated** for the Apex classes and triggers in the org.
- You can **check the code coverage percentage in the Tests tab** of the Developer Console.
- In this example, the class you've tested, the **Add class, has 100% coverage**, as shown in this image.

Overall Code Coverage

Class	Percent	Lines
Overall	**15%**	
AccountDeletion	0%	0/4
add	100%	5/5
Person	0%	0/2
TeacherClass	0%	0/8
TeacherTrigger	0%	0/5
Trigger1	0%	0/9

Apex Testing - Get Started with Apex Unit Tests

Unit Test Example 2 – To Test new branch Inserted in the custom object Branch_c :

Example:

1. Write a Apex Class BranchInset for inserting new branch in the custom object Branch_c using addBranch() method

```
public class BranchInsert {
  public static void addBranch(String BranchName) {
        Branch__c br = new Branch__c ( BName__c = BranchName);
        insert br;
  }
}
```

2. Create an Apex Test Class to test and verify the insertion operation in the addBranch() method inside the BranchInsert Class.

```
@isTest
public class BranchInsertTest
{
   @isTest
  static void testInsertBranch() {
      String testBranchName = 'IOT';
      BranchInsert.addBranch(testBranchName);
Branch__c resultBranch = [SELECT Id, BName__c FROM Branch__c   WHERE BName__c =:testBranchName LIMIT 1];

System.assertNotEquals(null, resultBranch);
System.assertEquals(testBranchName, resultBranch.BName__c);    }
 }
```

Unit Test Example-3

Create a test class for the following Apex class that converts a list of strings to uppercase:

```
public class StringUtils {
public List<String> convertToUpper(List<String> inputList) {
for (Integer i = 0; i < inputList.size(); i++)
{
inputList[i] = inputList[i].toUpperCase();
}
return inputList;
 }
}
```

Write all tests that handle empty lists and null values.

```
@isTest public class StringUtilstest {
@isTest static void test1()
{
   List<String> Exp=new List<String>{'STRING'};
   List<String> l1=new List<String>{'string'};
   List<String> res=StringUtils.convertToUpper(l1);
   System.assertEquals(Exp,res,'Not equal');
}
```

```
@isTest static void test2()
{
   List<String> Exp=new List<String>{'STRING'};
   List<String> l1=null;            //null value
   List<String> res=StringUtils.convertToUpper(l1);
   System.assertEquals(Exp,res,'Not equal');
}
@isTest static void test3()
{
   List<String> Exp=new List<String>{'STRING'};
   List<String> l1=new List<String>();  //empty List
   List<String> res=StringUtils.convertToUpper(l1);
   System.assertEquals(Exp,res,'Not equal');
}   }
```

Apex Testing - System-Defined Methods

System-Defined Methods:

The common system-defined unit test methods are:

startTest: startTest method marks the point in your test code **when the test actually begins**.

stopTest: stopTest method comes after the startTest method and **marks the end point of an actual test code.**

Purpose:

- Any code that executes after the call to start test() and before the stop test () is **assigned a new set of governor limits.**

- Any code that executes **after the call to stop test() is arranged with the original limits that were in effect before the start test() was called.**

Apex Testing - System-Defined Methods

In Salesforce, it is the **Governor Limits** that **control how much data or how many records you can store in the shared databases. There are limits on the amount of CPU time, memory, and other resources that can be used by Apex code**.

Because Salesforce is based on the concept of **multi-tenant architecture i.e** Salesforce uses a **single database to store the data of multiple clients/ customers.**

Governor limits are **reset** when the Test.startTest appears and the **code between Test.startTest and Test.stopTest executes in fresh set of governor limits** (Context changes).
Also Test.stopTest appears, the context is again moved back to the original code.

For example,

if your class makes 98 SOQL queries before it **calls startTest**, and the first significant statement after startTest is a DML statement, the program **can now make an additional 100 queries.**
Once stopTest is called, however, the program **goes back into the original context**, and can only make 2 additional SOQL queries before reaching the limit of 100.

Apex Testing - System-Defined Methods

Example:

```
@isTest
private class myClass {
    static testMethod void myTest() {
        // Create test data
        ..................Test.startTest();
        // Actual apex code testing
        .........Test.stopTest();
    }}
```

Apex Testing - System-Defined Methods

Example:

```
public class BranchInsert {
                    public static void insertBranch(String BranchName) {
                              Branch__c br = new Branch__c ( BName__c = BranchName);
                              insert br;          }          }
```

```
@isTest   public class BranchInsertTest {
@isTest static void testInsertBranch() {
 String testBranchName = 'IOT';
        Test.startTest();                    // Start test context
          BranchInsert.insertBranch(testBranchName);
        Test.stopTest();                     // Stop test context
Branch__c resultBranch = [SELECT Id, BName__c FROM Branch__c    WHERE BName__c = :testBranchName LIMIT 1];
 System.assertNotEquals(null, resultBranch);
 System.assertEquals(testBranchName, resultBranch.BName__c);    }   }
```

Apex Testing - Test Apex Triggers

Test Apex Triggers:

- Before deploying a trigger, write unit tests to perform the actions that fire the trigger and verify expected results.
- Let's test a trigger that we worked with in the Apex Triggers module.

Here are some steps for adding a test method to verify an Apex trigger:

1. In the Developer Console, click **File | New | Apex Class**
2. Enter the **class name**
3. Replace the **default class body**
4. **Set up a test**
5. Do the **custom actions**
6. **Verify** that the trigger performed the relevant custom action.

Apex Testing - Test Apex Triggers

Apex triggers are special types of classes that execute custom actions before or after changes to Salesforce records, such as insertions, updates, or deletions.

Example:

- Let's say you have a custom object called **Teacher__c** with the following fields:
    - **Teacher_Name__c (Text)**
    - **Experience__c (Number)**

- You have a trigger on **Teacher__c** that automatically updates the **Experience__c** as **5** whenever a **Teacher__c record is inserted with Teacher_Name__c = 'Teacher5' otherwise** updates the **Experience__c** as **0 .**

Apex Testing - Test Apex Triggers

Apex Trigger- TeacherRecord.apxt

```
trigger TeacherRecord on Teacher__c (before insert) {
    for(Teacher__c a:Trigger.new) {
    if(a.Teacher_Name__c=='Teacher5')
        {
            a.Experience__c = 5;   }
        else
        {
            a.Experience__c = 0;
        }
    }
}
```

Apex Test Class- OrderTriggerTest.apxc

```
@isTest   public class TeacherRecordTest {
@isTest    static void testteacherTrigger1()
   {  Teacher__c testteacher = new Teacher__c(Teacher_Name__c = 'Teacher5');
      insert testteacher;
      Teacher__c insertedteacher = [SELECT Experience__c FROM Teacher__c WHERE Id
                                                                    =:testteacher.Id];
      System.assertEquals(5, insertedteacher.Experience__c);    }

@isTest    static void testteacherTrigger2()
   {  Teacher__c testteacher = new Teacher__c(Teacher_Name__c = 'Teacher');
      insert testteacher;
     Teacher__c insertedteacher = [SELECT Experience__c FROM Teacher__c WHERE Id
                                                                    =:testteacher.Id];
      System.assertEquals(0, insertedteacher.Experience__c);    }

   @isTest    static void testteacherTrigger3()
   {  Teacher__c testteacher = new Teacher__c(Teacher_Name__c = 'Teacher');
      insert testteacher;
      Teacher__c insertedteacher = [SELECT Experience__c FROM Teacher__c WHERE Id
                                                                    =:testteacher.Id];
      System.assertEquals(5, insertedteacher.Experience__c);    } }
```

Explanation of the Test Class:

1.@isTest Annotation: Marks the class and the method as a test class and method, which are used only for testing purposes.

2.Setup: Create an instance of **Teacher__c** with **Teacher_Name__c** set to **'Teacher5'.**

This is the data you'll be using to test the trigger.

3.Act: Insert the **Teacher__c** record. This action fires the **TeacherRecord** trigger, which should automatically set the **Experience__c** field based on the trigger's logic.

4.Assert: After inserting the record, query it from the database and verify:

- The **Teacher_Name__c** field is as expected (**Teacher5**).
- The **Experience__c** field is updated with **5.**

Apex Testing - Create Test Data for Apex Tests

Create Apex Test Data:

- The **data from Apex tests is only temporary** and is **not saved in the database**.
- Salesforce **records that are created in test methods aren't committed to the database**.
- They're **rolled back when the test finishes execution**.
- This rollback behavior is handy for testing because you **don't have to clean up your test data** after the test executes.
- It is recommended that the **test utility classes be created to add reusable methods** for test data setup.

Apex Testing - Create Test Data for Apex Tests

- By default, **Apex tests don’t have access to pre-existing data** in the org, except for access to setup and metadata objects, such as the User or Profile objects.
- **Creating test data makes** your **tests more robust** and **prevents failures** that are **caused by missing or changed data in the org.**
- You can **create test data directly in your test method**, or **by using a utility test class**.
- The **primary purpose of an Apex utility class** is **To encapsulate reusable methods that can be called from other classes.**

Apex Testing - Get Started with Apex Unit Tests

Example :

```
@isTest public class BranchInsert {
        public static List< Branch__c > insertBranch(Integer num) {
         List< Branch__c > br = new List< Branch__c >();
        for(Integer i=1;i<=num;i++) {
        Branch__c a = new Branch__c (BranchName ='TestBranch' + i);
        br.add(a);
        return br;    }      }      }
```

```
@isTest   public class BranchInsertTest {
@isTest static void testInsertBranch() {
List< Branch__c > branchlist=BranchInsert.insertBranch(2);
Insert branchlist;
Branch__c resultBranch = [SELECT  BName__c FROM Branch__c WHERE BName__c= TestBranch1
LIMIT 1];

System.assertNotEquals(null, resultBranch);
System.assertEquals(TestBranch1, resultBranch.BName__c);    }  }
```

Developer Console Basics - Get Started with the Developer Console

The Developer Console is an **integrated development environment** (**IDE**) where you can **create, debug, and test apps** in your org.

It's your one-stop solution for a variety of development tasks:

- **Navigate, open, create, and edit** Apex classes and triggers, Aura components, and Visualforce pages and components.
- **Browse packages** that you've created in your org.
- **Generate logs** for **debugging and analyze** them using different perspectives.
- **Test your Apex code** to ensure that it's error free.
- **Identify and resolve errors** by setting checkpoints in your Apex code.
- **Write and execute SOQL and SOSL queries** to find, create, and update the records in your org.

Developer Console Basics - Get Started with the Developer Console

When Do You Use the Developer Console?

- The Developer Console is connected to one org and is browser-based.
- If you want your changes to be effective immediately and you don’t want to install anything on your computer.
- If you want to connect to multiple orgs, compare or synchronize files, or use version control, the Salesforce Extensions for Visual Studio Code is the best option.

Developer Console Basics - Get Started with the Developer Console

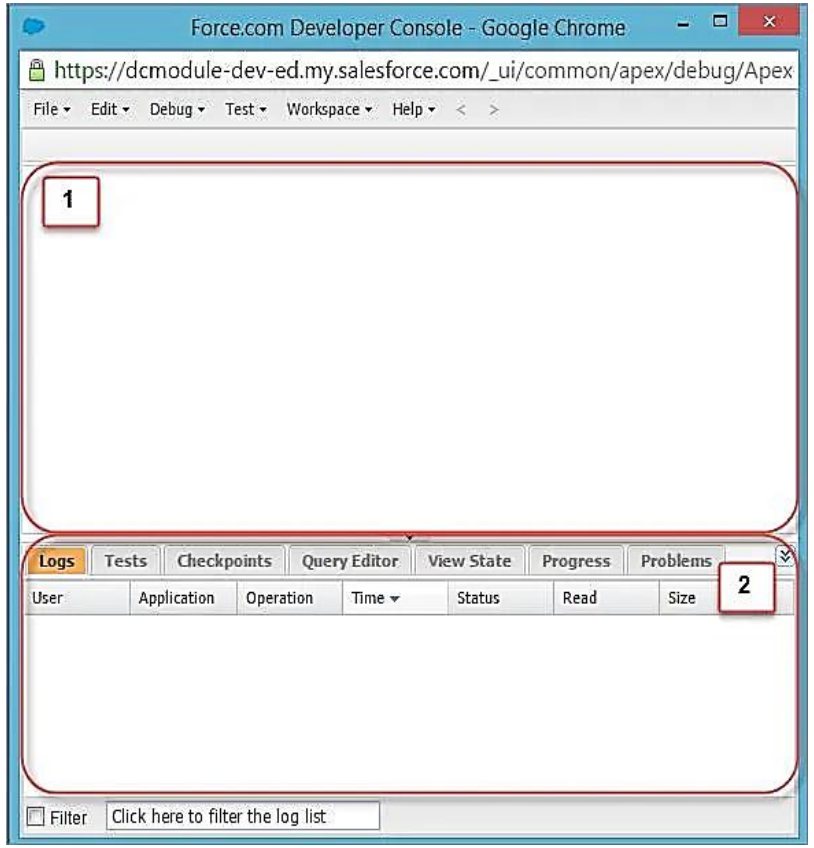
Accessing the Developer Console:

- After logging in to your org, click **Developer Console** under the quick access menu.

Developer Console Basics - Get Started with the Developer Console

Accessing the Developer Console:

- When you open the **Developer Console for the first time**, you see something like this.
- The main pane (1) is the source code editor, where you can write, view, and modify your code.
- The tabs pane (2) is where you can view logs, errors, and other information, and write queries to interact with the records in your org.



Developer Console Basics - Get Started with the Developer Console

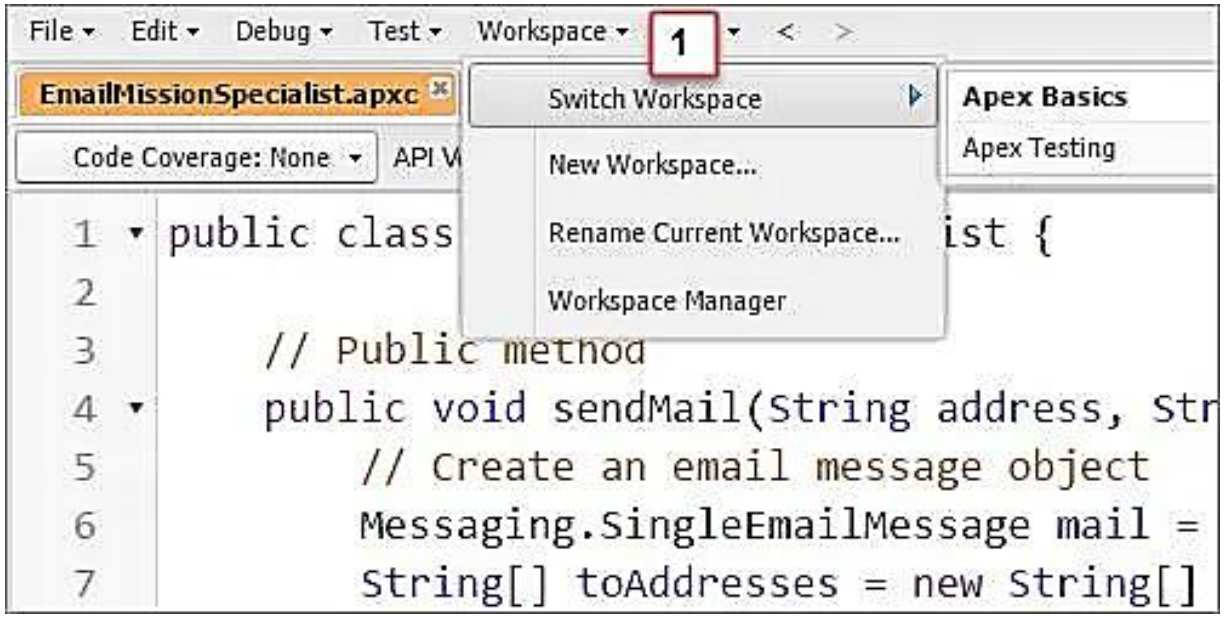
What Is a Workspace?

- 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.
- If you're working on two different projects, **you can have the related code, tests, and logs open simultaneously in separate workspaces**.
- For instance, say you're writing code to update some records for your engineering team, but you also want to check the system details for your navigation team.
- You can **create two workspaces**, each of which **contains only the resources relevant to the project**.
- Workspaces **reduce clutter and make it easier to navigate between different resources**.
- When you use the **Developer Console for the first time**, you **see the default workspace**.

Developer Console Basics - Get Started with the Developer Console

Set Up Your Own Workspace:

- Select **Workspace | New Workspace** and give your workspace a name.
- In your new workspace, you can **create Apex classes, Aura components, Visualforce pages, and more.**
- You can **switch between your workspaces** by selecting `
- In this way, you can **work with code and analyze logs for each project** just by opening a different workspace.



User Interface experiences

Salesforce Classic is the original tab-based user interface (UI) of the Salesforce, introduced in 1999.

framework

Visualforce is a **powerful framework within Salesforce** that allows developers **to build custom user interfaces for Salesforce applications**.
At the beginning of the Salesforce Development times, developers used Visual Force, which is an **HTML Tag-based mark-up language** to **develop their Visual Force web pages and Apex to control the internal logic** that simplify the process of accessing Salesforce data and functionality.
But this HTML based Visual Force standards **were not compatible to build large scale enterprise solutions and complex applications.**

In 2014, Salesforce announced new platform technologies—Lightning App Builder and Lightning Components

framework

Salesforce Lightning Component also known as **Aura Component** is a user interface (UI) based framework for developing **single-page mobile/desktop/web-based applications.**
This Aura Framework which used a component-driven model that was brilliant in developing **large-scale enterprise applications.**

The **Lightning web component (LWC)** introduced in **Feb 2019** is Salesforce's new programming model built on **modern browser improvements or web standards** which results in faster load times and more efficient rendering.
This can be especially important when building complex components with a lot of data or logic.

Salesforce Development Overview

Development Overview:

Versions	Client Side (UI)	Server Side (DB Interaction)
Version 1	Visualforce Page	Apex Class
Version 2	Lightning Aura Component	Apex Class
Version 3	Lightning Web Component	Apex Class

Developer Console Basics - Navigate and Edit Source Code

Create Visualforce Pages and Components:

- Visualforce is a **web development framework** for building sophisticated **user interfaces** for **mobile** and **desktop apps** in **Classic Experience**.
- These interfaces can also be **hosted on the force.com Platform**. Your user interface can look like the standard Salesforce interface, or you can customize it.
- It **uses a tag-based markup language** similar to HTML.

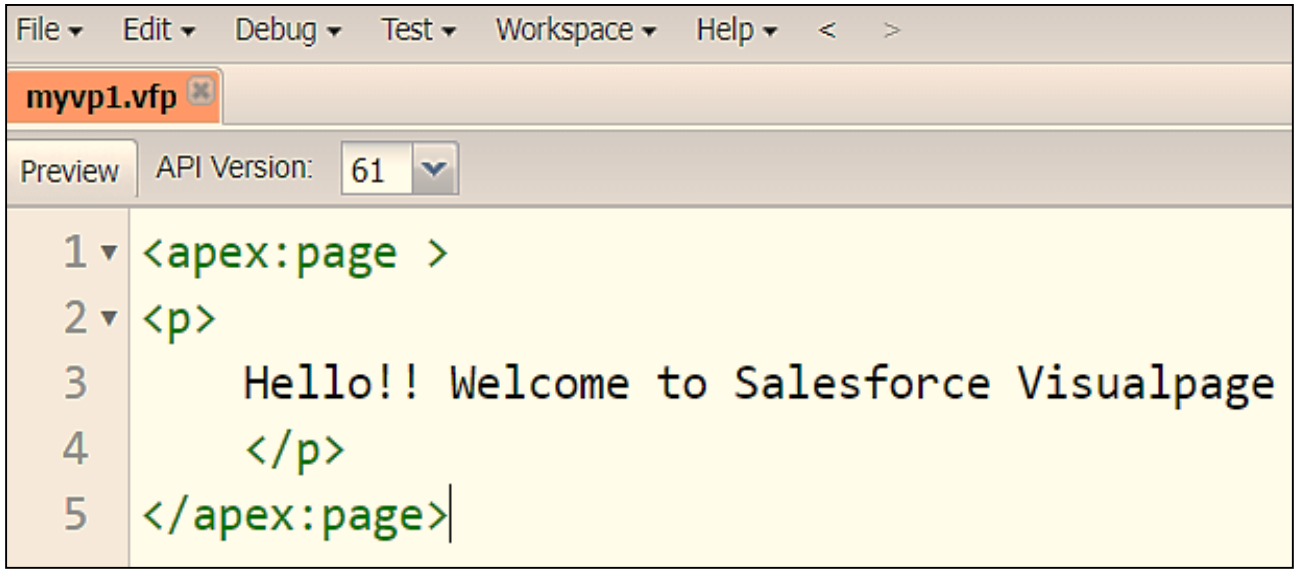
Advantages of Visualforce:

- It is a **Model View Controller** (MVC) development.
- It has **huge number of components** and is **flexible and customizable** with **web technologies**.
- It can be **integrated with HTML, CSS, Ajax, Jquery**.

Developer Console Basics - Navigate and Edit Source Code

Let's create a Visualforce page by using the following steps:

- Select **File | New | Visualforce Page**.
- **Name** your page.
- In the **text editor**, **type the code.**
- Select **File | Save.**
- In the top left corner, **click Preview**.
- Your **browser** opens a **preview of your Visualforce page**.



Developer Console Basics - Navigate and Edit Source Code

Example 2: Let's create a Visualforce page for a Flight Systems Checklist that your Control Engineers update every 2 hours, when they perform their engine and fuel tank safety checks.

- A functional Flight Systems Checklist page needs to interact with objects that store the values entered by the Control Engineers. But, for now, let's focus on creating the UI. You can have your underlings create those custom objects.
- Follow the same steps mentioned above to create a visualforce page and **copy** the below code in the **text editor** and **save the file**.
- Click on **Preview** to view the page.

Developer Console Basics - Navigate and Edit Source Code

```
<apex:page sidebar="false">
<!--Flight Systems Checklist Visualforce Page-->
  <h1>Checklist</h1>
  <apex:form id="engineReadinessChecklist">
    <apex:pageBlock title="Flight Systems Checklist">
      <!--First Section-->
      <apex:pageBlockSection title="Engines">
        <!--Adding Checkboxes-->
        <apex:inputCheckbox immediate="true"/>Engine 1
        <apex:inputCheckbox immediate="true"/>Engine 2
        <apex:inputCheckbox immediate="true"/>Engine 3
        <apex:inputCheckbox immediate="true"/>Engine 4
        <apex:inputCheckbox immediate="true"/>Engine 5
        <apex:inputCheckbox immediate="true"/>Engine 6
      </apex:pageBlockSection>
<apex:pageBlockButtons>
        <!--Adding Save Button-->
        <apex:commandButton value="Save" action="{!save}"/>
      </apex:pageBlockButtons>
    </apex:pageBlock>
  </apex:form>
</apex:page>
```

Developer Console Basics - Navigate and Edit Source Code

- Your **browser** opens a **preview of your Visualforce page**.
- The Visualforce markup in your pageform, pageBlock, inputCheckbox, and so on is rendered in the preview.

Developer Console Basics - Navigate and Edit Source Code

Let’s see how you can open a saved Visualforce page.

- Select **File | Open.**
- Under **Entity Type**, click **Pages**.
- Under **Entities**, **double-click the page** you want to open.
- You can **create, edit, and customize applications** for your org using any or all these methods in the Developer Console.

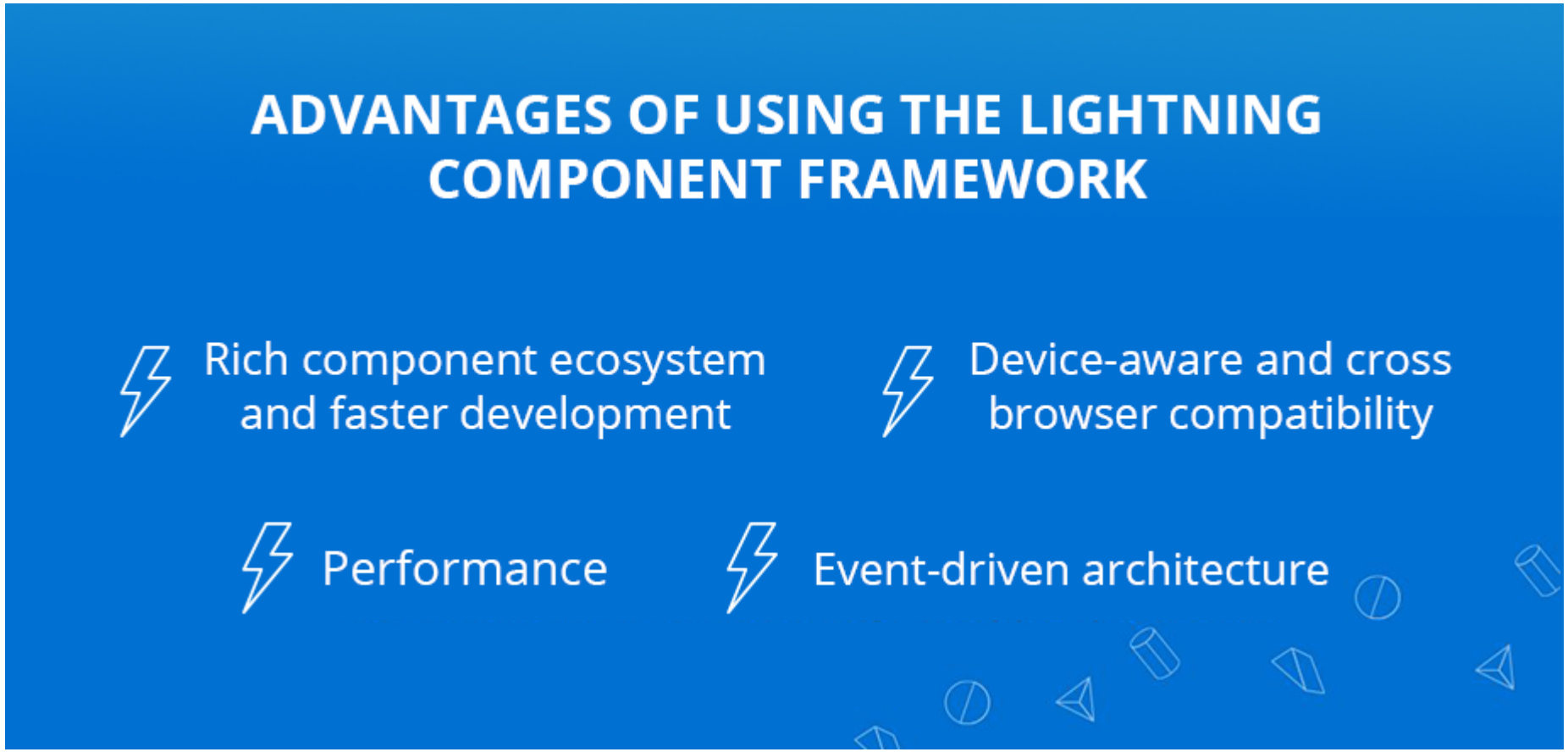
Developer Console Basics - Navigate and Edit Source Code

Lightning Experience:

- **Lightning Experience** is the **new** Salesforce, **reimagined with a modern user interface**, a suite of **new features and tools**, and even **more advanced technology**.
- It **empowers sales representatives** and **service representatives** to sell more faster and support customers more efficiently.
- It uses an **App-centric model** made up of several components.
- These components are called **Lightning Components**.
- Lightning Experience **supports** both **Visualforce** as well as **Lightning Components**.

Developer Console Basics - Navigate and Edit Source Code

Benefits of Lightning Framework:



What Are 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.
- You can **use the Developer Console to create Aura components**.
- Using the Developer Console, you can **create a 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**.

Developer Console Basics - Navigate and Edit Source Code

Create an Aura Component:

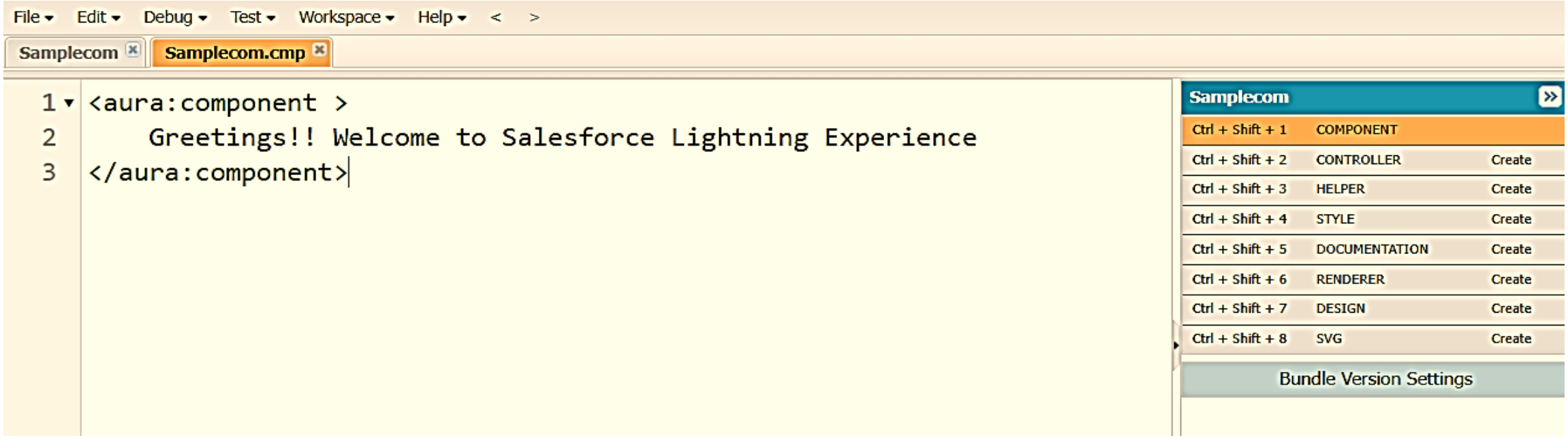
You're bound to encounter other humans as you evade asteroid 2014 QO441, and we want them to know you're friendly so they don't open fire on you. Fortunately, your ship has large display panels on the sides that are running the Salesforce app. (You knew those would come in handy someday!) Let's create a component that greets any other spaceship nearby.

1. Select **File | New | Lightning Component**. The window that pops up prompts you for a name and description.
2. **Name your component** and **click Submit**. The window also has options to configure your **app's tab, page, record page, and communities page**. For now, we're only focusing on writing basic Aura component code.
3. **Two tabs are created**. Click the one labeled **".cmp"**. This file **contains the opening and closing tags for Aura components.**

Developer Console Basics - Navigate and Edit Source Code

4. Between the opening and closing **<aura:component>** and **</aura:component>tags**, **copy** and **paste** this line,

Greetings!! Welcome to Salesforce Lightning Experience



5. To save the component, select **File | Save**.

Developer Console Basics - Navigate and Edit Source Code

The **right-hand side of the window** includes all the **resources in a component bundle** that you can use to build your component. If you **click any item in the right sidebar**, a **corresponding resource opens**. You can **write code in the new resource** to build the different parts of the component bundle.

Developer Console Basics - Navigate and Edit Source Code

How to execute and see the output of Lightning Components?

As, Lightning Experience is **App-Centric** , Lightning Components are **not stand-alone** and **cannot be executed directly**.

So, create a Lightning app and execute the components.

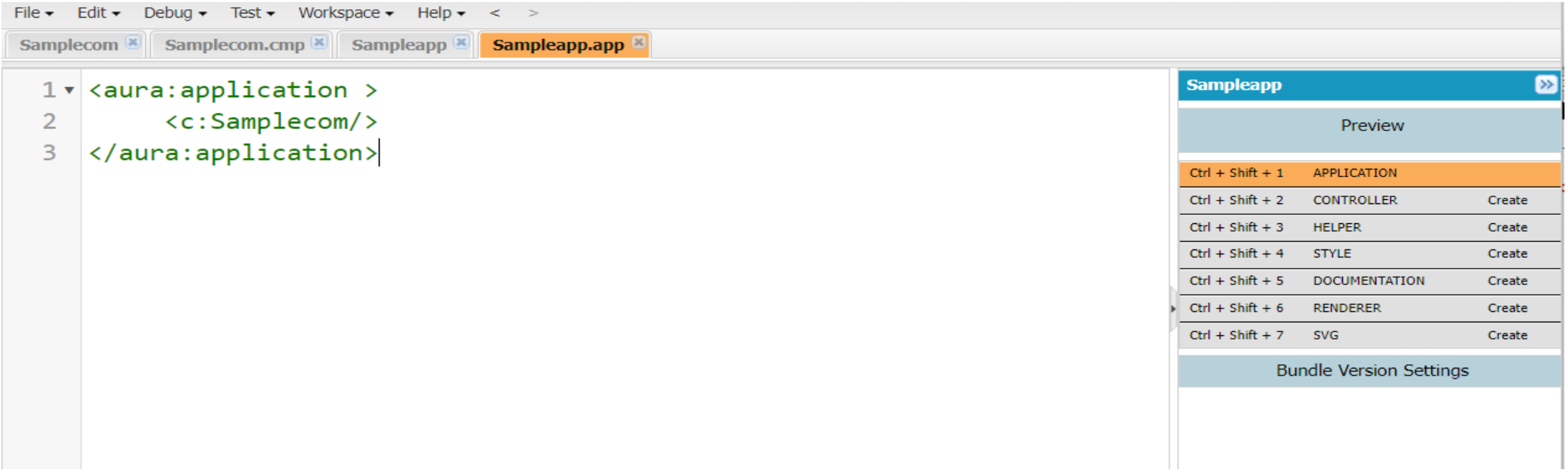
How to create a Lightning App?

1. Select **File | New | Lightning Application**. The window that pops up prompts you for a name and description.
2. **Name your Application** and **click Submit**. The window also has options to configure your **app's tab, page, record page, and communities page**. For now, we're only focusing on executing basic Aura component code.
3. **Two tabs are created**. Click the one labeled **".app"**. This file **contains the opening and closing tags for Aura application.**

Developer Console Basics - Navigate and Edit Source Code

4. Between the opening and closing **<aura:application>** and **</aura:application>tags**, **copy** and **paste** this line,

<c: Samplecom/>

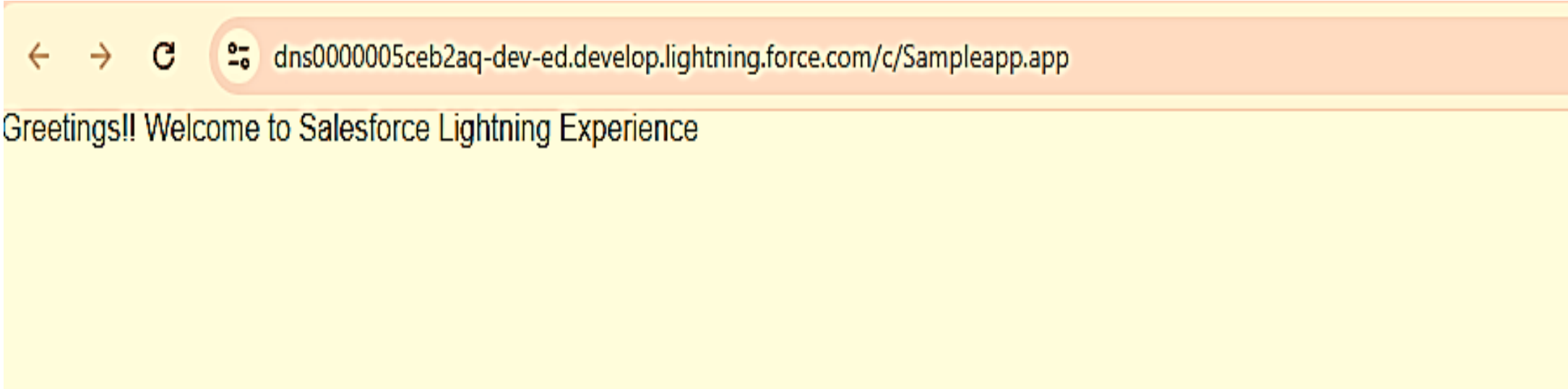


5. To save the component, select **File | Save**.

Developer Console Basics - Navigate and Edit Source Code

6. Click on **Preview** to view the page.

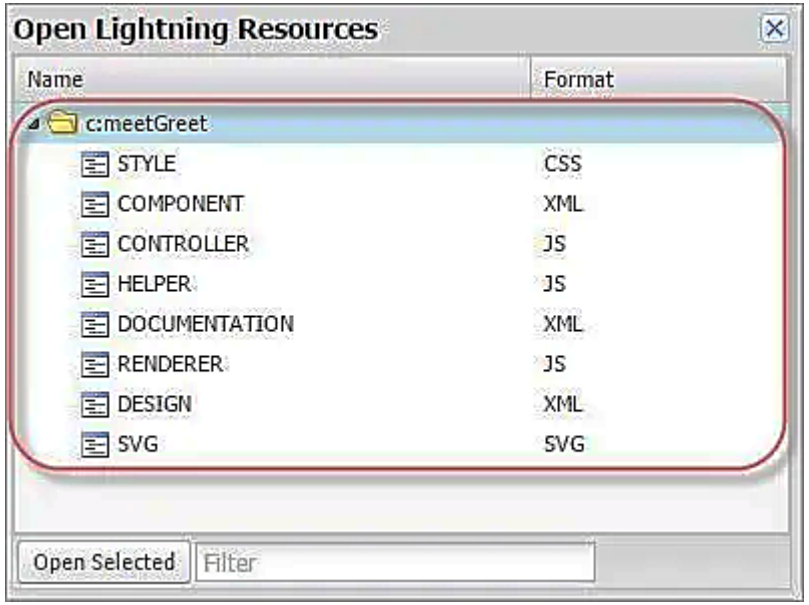
7. Your **browser** opens a **preview of your Lightning page**.



Developer Console Basics - Navigate and Edit Source Code

Here's how you can open a saved Aura component or any of these resources.

1. Select **File | Open Lightning Resources**.
2. Type your **component's name** in the search box to find your bundle, or select its folder from the list.
3. To see the bundle's resources, **click the arrow next to the folder**.
4. **Select the resource** you want to work on, and then click **Open Selected**.



Developer Console Basics - Navigate and Edit Source Code

Difference between Visualforce and Lightning components:

	Visualforce	Lightning
UI Generation	Server Side	Client Side
Flow	1. User requests a page 2. The server executes the page's underlying code and sends the resulting HTML to the browser 3. The browser displays the HTML 4. When the user interacts with the page, return to step one.	1. The user requests an application or a component 2. The application or component bundle is returned to the client 3. The browser loads the bundle 4. The JavaScript application generates the UI 5. When the user interacts with the page, the JavaScript application modifies the user interface as needed (return to previous step)
Model	Page Centric	App Centric
Framework	MVC Framework	Component based Framework

Developer Console Basics - Generate and Analyze Logs

View Debug Logs:

You review the system logs to check that everything is in working order. 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.

- View Logs in the Text Editor
- Viewing a debug log is simple. To generate a log, let’s execute the Apex class that you created earlier.

You can view your log in two ways.

- **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.

Developer Console Basics - Generate and Analyze Logs

- The execution log that you see probably seems like a confusing jumble of numbers and words, so let's talk about how you can read and understand log data.

EmailMissionSpecialist.apxc | Log executeAnonymous @7/22/2016, 5:49:20 PM

Execution Log

Timestamp	Event	Details
17:49:20:053	HEAP_ALLOCATE	[1]|Bytes:4
17:49:20:053	HEAP_ALLOCATE	[50]|Bytes:5
17:49:20:053	HEAP_ALLOCATE	[56]|Bytes:5
17:49:20:053	HEAP_ALLOCATE	[64]|Bytes:7
17:49:20:053	SYSTEM_MODE_...	false
17:49:20:053	HEAP_ALLOCATE	[1]|Bytes:5
17:49:20:053	CONSTRUCTOR...	[1]|01p50000001H3vF|<init>()
17:49:20:053	VARIABLE_SCO...	[1]|this|EmailMissionSpecialist|true|false
17:49:20:054	VARIABLE_ASSI...	[1]|this|{}|0x722ad411
17:49:20:054	HEAP_ALLOCATE	[EXTERNAL]|Bytes:6
17:49:20:054	STATEMENT_EX...	[1]|
17:49:20:054	CONSTRUCTOR...	[1]|01p50000001H3vF|<init>()

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

Logs | Tests | Checkpoints | Query Editor | View State | Progress | Problems

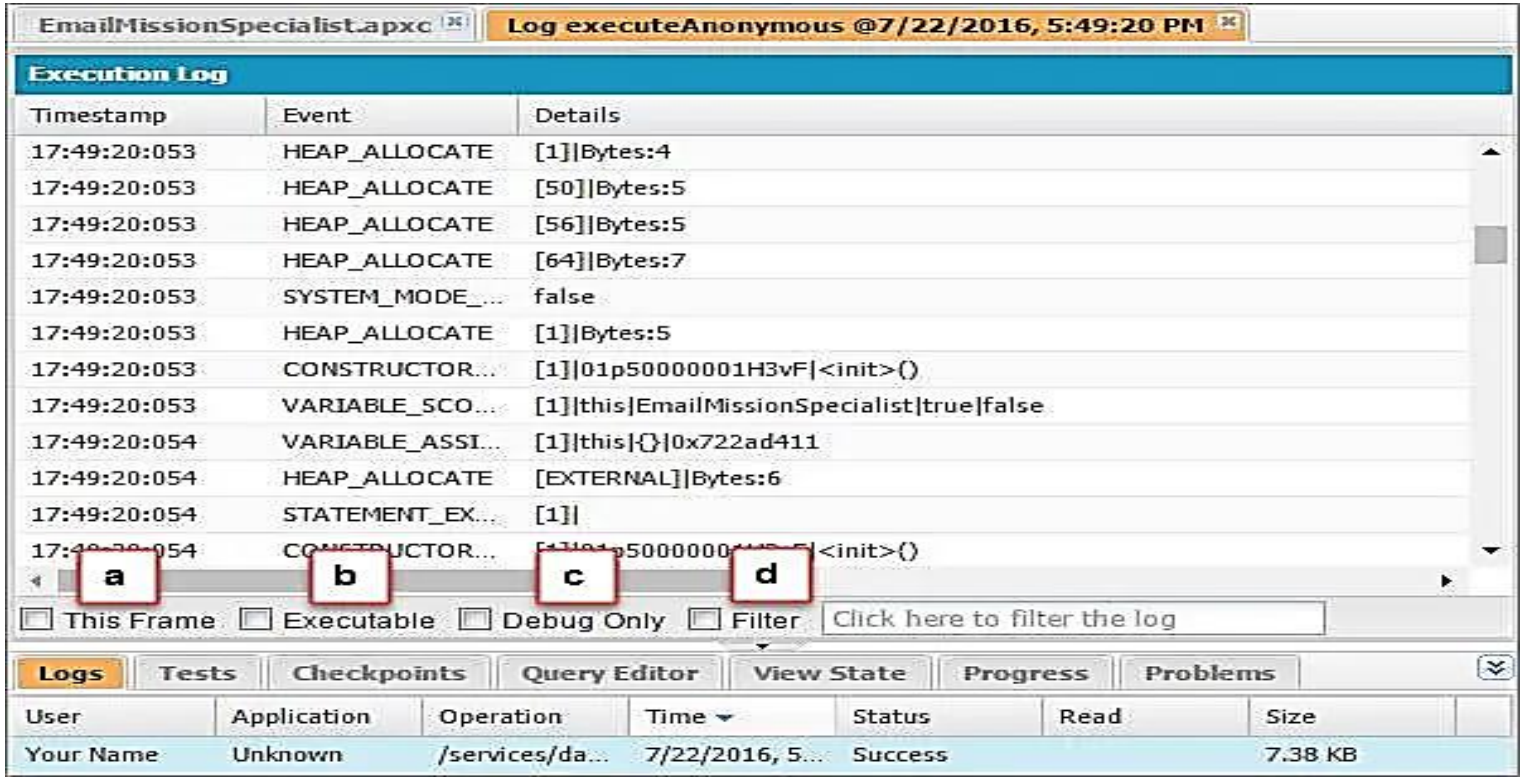
User	Application	Operation	Time	Status	Read	Size
Your Name	Unknown	/services/da...	7/22/2016, 5...	Success		7.38 KB

Developer Console Basics - Generate and Analyze Logs

- Read Your Log Data
- You can read a debug log by identifying what each column represents.
- **Timestamp -** The time when the event occurred. The timestamp is always in the user's time zone and in **HH:mm:ss:SSS** format.
- **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.
- **Details -** Details about the line of code and the method name where the code was executed.

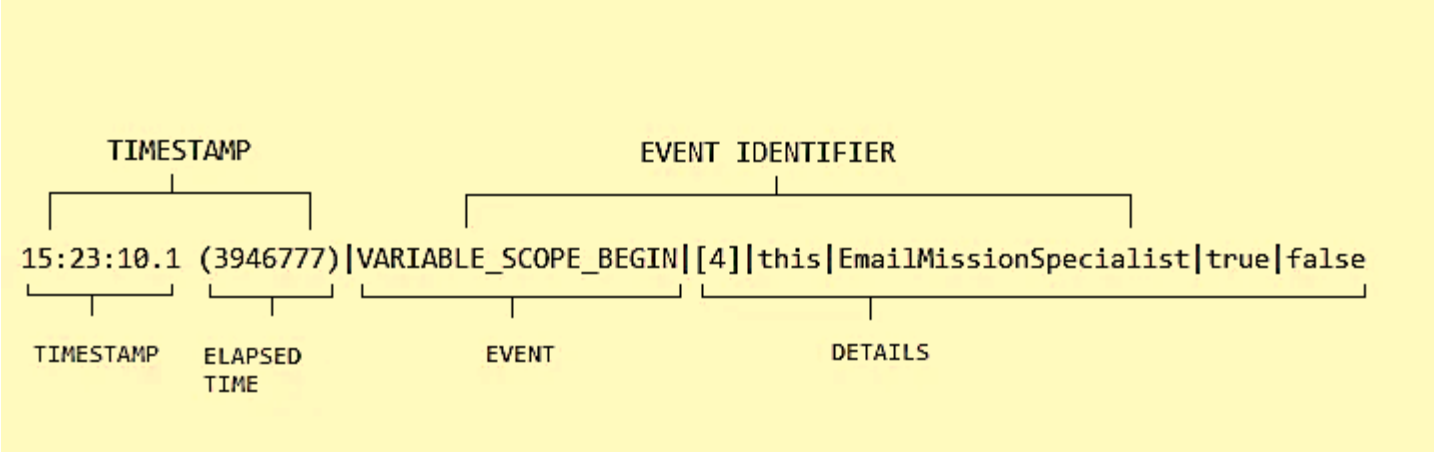
Developer Console Basics - Generate and Analyze Logs

- You can change what you see in the Execution Log by selecting This **Frame (a)**, **Executable (b)**, or **Debug Only (c).** Selecting these options shows you only certain types of events. For instance, Debug Only shows USER_DEBUG events. You can also filter different parts of the log using **Filter (d)**. Enter a method name, or any other text you are specifically looking for, and the log filters your results.



Developer Console Basics - Generate and Analyze Logs

- You can also view the debug log as a raw log, which shows you more information.
- Select **File | Open Raw Log**. The timestamp in a raw log shows the time elapsed in nanoseconds (in parentheses) since the start of the event.



- This combination of the timestamp, event, and details provides valuable insights into how your code works and the errors that occur.

Developer Console Basics - Generate and Analyze Logs

What if you want to quickly look for certain values in the debug log?

After all, you have many other responsibilities as commander. An excellent way to do so is to use the System.debug() method in Apex.

- The great thing about System.debug() is that you can add it anywhere in your code to track values, helping you debug your own code.
- Here is the syntax for System.debug().
- To display a message:

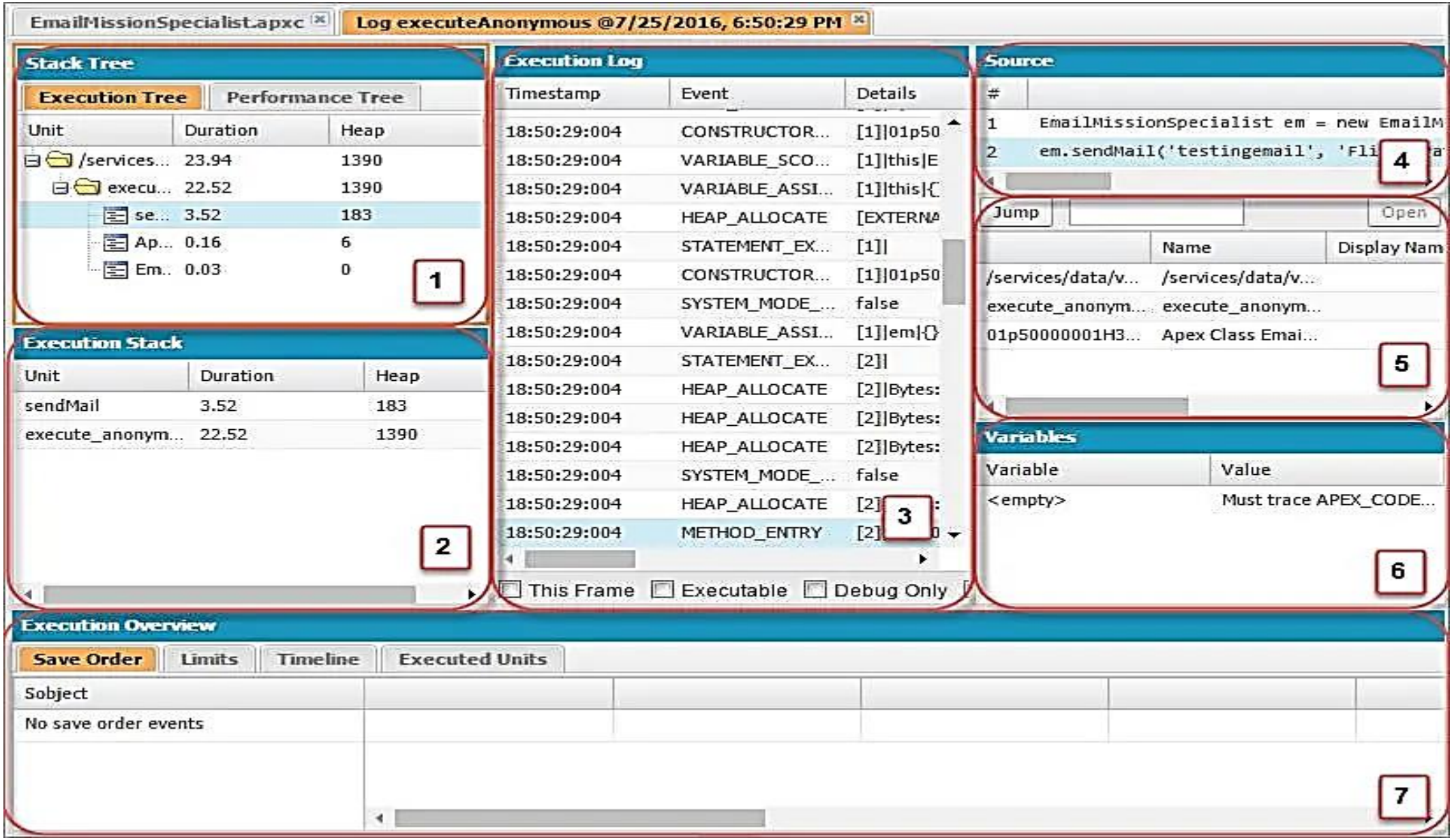
```
System.debug('Your Message');
```

Developer Console Basics - Generate and Analyze Logs

Use the 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. Check it out by selecting **Debug | View Log Panels.**
- **Log panels change the structure of the log**, to give other helpful information about the context for the code being executed. For example, different panels show the source, execution times, heap size, and calling hierarchy.
- These **log panels interact with each other to help you debug your own code**. For instance, when you click a log entry in the Execution Log or Stack Tree, the other panels (Source, Source List, Variables, and Execution Stack) refresh to show related information.

Developer Console Basics - Generate and Analyze Logs



Developer Console Basics - Generate and Analyze Logs

- These panels are available in the Log Inspector.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.
- **Execution Stack -** Displays a bottom-up view of the selected item. It displays the log entry, followed by the operation that called it.
- **Execution Log -** Displays every action that occurred during the execution of your code.
- **Source -** Displays the contents of the source file, indicating the line of code being run when the selected log entry was generated.
- **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.
- **Variables -** Displays the variables and their assigned values that were in scope when the code that generated the selected log entry was run.
- **Execution Overview -** Displays statistics for the code being executed, including the execution time and heap size.

Developer Console Basics - Generate and Analyze Logs

Log Categories:

- A log category is the type of information that is being logged.

Here are two common 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 (something we get into later).

Developer Console Basics - Inspect Objects at Checkpoints

Set Checkpoints in Your Apex Code:

- When your **Apex code is causing errors**, has **performance issues**, or **isn't producing the desired results**, your first step is to **identify the problem using your debug log**.
- **Combing line by line** through the entire log is **a tedious task**.
- That's where checkpoints come in handy!
- **Checkpoints show you snapshots** of what's happening in your Apex code **at particular points during execution**.
- You can **set up to five checkpoints** in your Apex code.
- Checkpoints **aren't available for Visualforce markup**.

Developer Console Basics - Inspect Objects at Checkpoints

Let’s set a checkpoint in the EmailMissionSpecialistclass that we created earlier.

1. Select **File | Open**, and open the class.
2. Select **Debug | Change Log Levels**.
3. In the General **Trace Settings** for You section, click **Add/Change**.
4. Set the **ApexCode log level** to **FINEST**.

   Note: To set checkpoints, you need the View All Data user permission. To generate results using checkpoints, run code using execute anonymous, or set a DEVELOPER_LOG trace flag on yourself. The trace flag must have a log level for Apex of INFO or higher.
5. To **save your changes**, click **Done**.
6. To **exit the Change Log Levels** dialog box, click **Done**.

Developer Console Basics - Inspect Objects at Checkpoints

- When your code is displayed in the source code editor, you can see line numbers on the left side. **Click the line number** for inspectResults(results);. A **red dot (1) appears**, indicating that a checkpoint has been created.

EmailMissionSpecialist.apxc

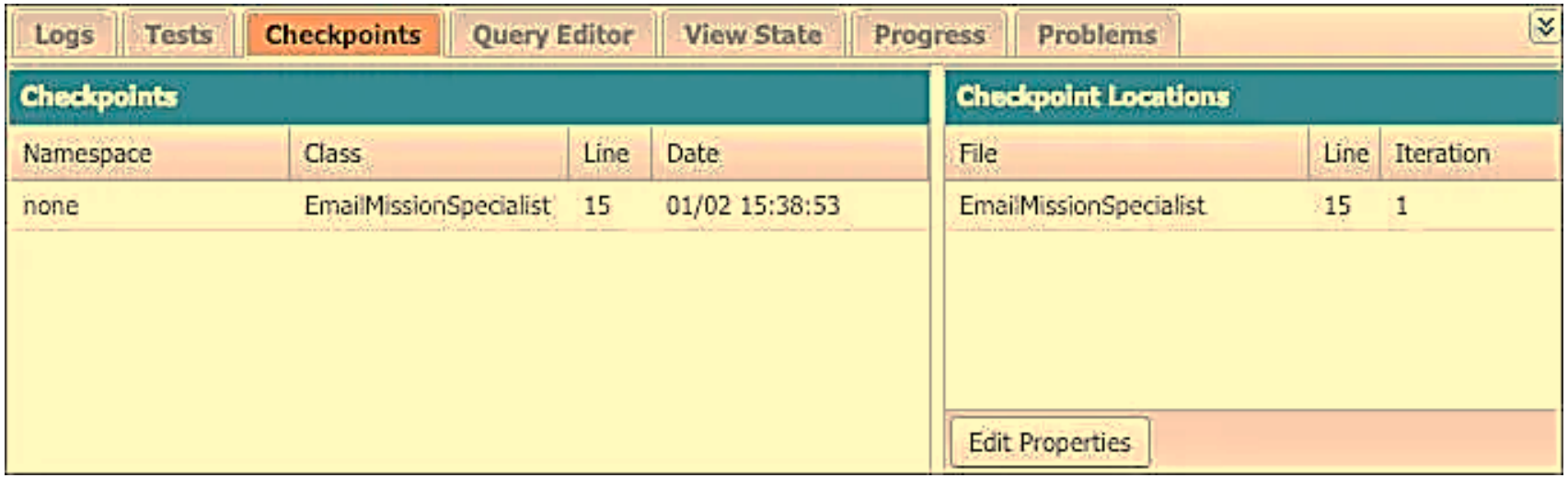
Code Coverage: None | API Version: 44 | Go To

```
1  public class EmailMissionSpecialist {
2      // Public method
3      public void sendMail(String address, String subject, String body) {
4          // Create an email message object
5          Messaging.SingleEmailMessage mail = new Messaging.SingleEmailMessage();
6          String[] toAddresses = new String[] {address};
7          mail.setToAddresses(toAddresses);
8          mail.setSubject(subject);
9          mail.setPlainTextBody(body);
10         // Pass this email message to the built-in sendEmail method
11         // of the Messaging class
12         Messaging.SendEmailResult[] results = Messaging.sendEmail(
13                                     new Messaging.SingleEmailMessage[] { mail });
14         // Call a helper method to inspect the returned results
•15 1      inspectResults(results);
16     }
17     // Helper method
18     private static Boolean inspectResults(Messaging.SendEmailResult[] results) {↔}
35 }
```

Developer Console Basics - Inspect Objects at Checkpoints

Checkpoints Tab:

- You can view exactly where your code's execution is going wrong, and what the values of the objects are at that point, using the Checkpoints tab.
- After you run the Apex code successfully, open your debug log and click the Checkpoints tab to see the results.



Developer Console Basics - Inspect Objects at Checkpoints

- The Checkpoints table displays the namespace, class, and line number of each checkpoint. It also shows you the date and time when each checkpoint was created.
- The Checkpoint Locations table displays the file name, line number, and iterations captured by the selected checkpoint.
- Double-click a checkpoint in the Checkpoints table to see the captured results in the Checkpoint Inspector. Now the fun begins!
- Checkpoint Inspector
- The Checkpoint Inspector has two tabs: Heap and Symbols.
- Heap Displays all objects present in memory at the line of code where your checkpoint was executed.
- Symbols—Displays all symbols in memory in tree view.

Developer Console Basics - Inspect Objects at Checkpoints

Heap Tab:

- The Heap tab includes some great panels for debugging, like the Types panel. This panel shows how many objects were instantiated and the memory they consumed in bytes. Let’s look at the details captured by the checkpoint you set.Under Types, click Messaging.SingleEmailMessage.
- Under Instances, click any instance of this object type.
- Under State, view the object’s fields and their values.

Symbols Tab:

- The Symbols tab is a quick and simple way to review the states of various objects at any checkpoint. Symbols are unique names that reference particular objects. The tab displays all symbols in memory using a tree view.

Developer Console Basics - Inspect Objects at Checkpoints

File Edit Debug Test Workspace Help < >

EmailMissionSpecialist.apxc | EmailMissionSpecialist:15@01/02 15:38:53

Heap | Symbols

Types

Type	Count	Total Size
EmailMissionSpecialist	1	4
List<Messaging.SendEmailRes...	1	8
List<String>	1	8
Messaging.SendEmailResult	1	4
Messaging.SingleEmailMessage	1	16
String	3	138

1

Instances

Address	Size
0x6e831aeb	16

2

State

3

Field	Value
plaintextbody	Mission Control 123: Yo...
emailpriority	Normal
usesignature	true
senderdisplayname	
subject	Flight Path Change
bccsender	false

References | Search

Inbound References

Field	Type

Referencing Instances

Address	Size

Logs | Tests | Checkpoints | Query Editor | View State | Progress | Problems

Checkpoints

Namespace	Class	Line	Date
none	EmailMissionSpecialist	15	01/02 15:38:53

Checkpoint Locations

File	Line	Iteration
EmailMissionSpecialist	15	1

Edit Properties

Developer Console Basics - Inspect Objects at Checkpoints

File Edit Debug Test Workspace Help < >

EmailMissionSpecialist.apxc | EmailMissionSpecialist:15@01/02 15:38:53

Heap | Symbols

Symbol	Key	Value
Key		
this	this	Type EmailMissionSpecialist (4 bytes)
anon.em	anon.em	Type EmailMissionSpecialist (4 bytes)
results	results	Type List<Messaging.SendEmailResult> (8 bytes)
toAddresses	toAddresses	Type List<String> (8 bytes)
mail	mail	Type Messaging.SingleEmailMessage (16 bytes)
address	address	Type String (21 bytes)
subject	subject	Type String (18 bytes)
value	value	Flight Path Change
body	body	Type String (99 bytes)
value	value	Mission Control 123: Your flight path has been changed to avoid collision with asteroid 2014 QO441.

Logs | Tests | Checkpoints | Query Editor | View State | Progress | Problems

Checkpoints

Namespace	Class	Line	Date
none	EmailMissionSpecialist	15	01/02 15:38:53

Checkpoint Locations

File	Line	Iteration
EmailMissionSpecialist	15	1

Edit Properties