

Getting Started with SharePoint Framework Development

Lab Time: 60 minutes

Lab Folder: C:\Student\Modules\04_SharePointFramework\Lab

Lab Overview: In this lab, you will begin by creating a simple SharePoint Framework project using the Yeoman generator and by editing the code for a simple webpart using Visual Studio Code. In the next exercise, you will move through the steps of testing your webpart in the local SharePoint Workbench and the Chrome Debugger extension for Visual Studio Code. The lab will also teach you how to add custom properties to a webpart and to program a SPFx webpart against the SharePoint REST API. In the final exercise, you will create an application extension that adds a page header and page footer to all the modern pages in the current site.

Lab Prerequisite: This lab assumes you've already installed Node.JS and Visual Studio Code as described in [setup.docx](#).

Exercise 1: Create an SPFX Project using the Yeoman Generator

In this exercise, you will install a few Node.JS packages required for SharePoint Framework development including the gulp task runner utility and the Yeoman template generator. After that, you will create a simple SharePoint Framework project containing a single webpart and begin editing the project's source files with Visual Studio Code.

1. Install the Node.JS packages required for working with SharePoint Framework.
 - a) Launch the Node.JS command prompt.
 - b) Run the following **npm** command to globally install the packages for **gulp** and the Yeoman Generator (**yo**).

```
npm install -g gulp yo
```

- c) Execute the following **npm** command to globally install the yo template for creating SharePoint Framework projects.

```
npm install -g @microsoft/generator-sharepoint
```

- d) Execute the following command if you have an earlier version of these tools and you want to update them to the latest version.

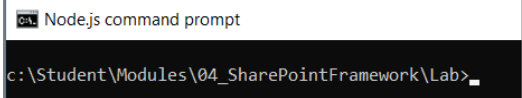
```
npm install -g @microsoft/generator-sharepoint@latest
```

Now you can create a new SPFx project by creating a new empty folder and running the Yeoman generator to create the starter files.

2. Create a new SPFx project named **spfx-lab**.
 - a) From the Node.JS command prompt, run the following command to set your current folder to the folder for this lab.

```
cd C:\Student\Modules\04_SharePointFramework\Lab
```

- b) The current directory for the console should now be at the folder for this lab inside the **Student** folder.



```
Node.js command prompt
```

```
c:\Student\Modules\04_SharePointFramework\Lab>
```

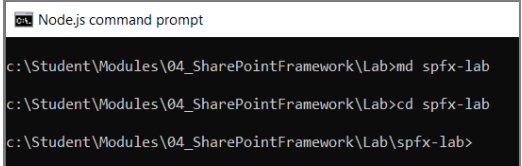
- c) Type the following command and execute it by pressing **Enter** to create a new folder for your project.

```
md spfx-lab
```

- d) Type the following command and execute it by pressing **Enter** to move to the current directory into the new folder.

```
cd spfx-lab
```

- e) The current directory for the console should now be located at the new folder you just created named **spfx-lab**.



```
Node.js command prompt
```

```
c:\Student\Modules\04_SharePointFramework\Lab>md spfx-lab
```

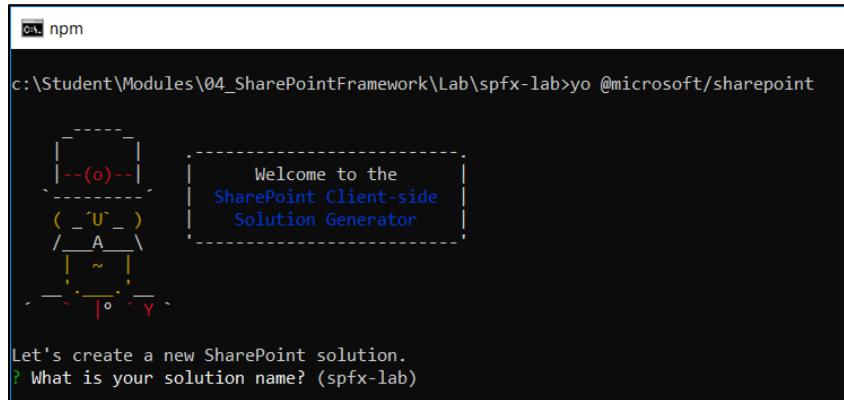
```
c:\Student\Modules\04_SharePointFramework\Lab>cd spfx-lab
```

```
c:\Student\Modules\04_SharePointFramework\Lab\spfx-lab>
```

- f) Type the following command and execute it to launch the Yeoman generator with the SharePoint Framework project template.

```
yo @microsoft/sharepoint
```

- g) When prompted with **What is your solution name?**, press **Enter** to accept the default value which is the name of the folder.



```
C:\> npm
c:\Student\Modules\04_SharePointFramework\Lab\spfx-lab> yo @microsoft/sharepoint

  Welcome to the
  SharePoint Client-side
  Solution Generator

Let's create a new SharePoint solution.
? What is your solution name? (spfx-lab)
```

- h) When prompted with **Which baseline packages do you want to target for your component(s)?**, press **Enter** to accept the default value of **SharePoint Online only (latest)**.

```
Let's create a new SharePoint solution.
? What is your solution name? spfx-lab
? Which baseline packages do you want to target for your component(s)? (Use arrow keys)
> SharePoint Online only (latest)
  SharePoint 2016 onwards, including SharePoint Online
```

- i) When prompted **Where do you want to place the files?**, press **Enter** to accept the default value of **Use the current folder**.

```
? Which baseline packages do you want to target for your component(s)? SharePoint Online only (latest)
? Where do you want to place the files? (Use arrow keys)
> Use the current folder
  Create a subfolder with solution name
```

- j) When prompted **Do you want to allow the tenant admin the choice of being able to deploy to all sites immediately without running any feature deployment or adding apps in sites (y/N)?**, type "y" and press **Enter** to accept the option.

```
? Where do you want to place the files? Use the current folder
Found npm version 5.6.0
? Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without
running any feature deployment or adding apps in sites? (y/N)
```

- k) When prompted with **Which type of client-side component to create?**, press **Enter** to accept the default value of **webpart**.

```
? Which type of client-side component to create? (Use arrow keys)
> WebPart
  Extension
```

- l) When prompted with **What is your Web part name?**, type **WalmartGreeter** and press **Enter** to submit your value.

```
? Which type of client-side component to create? WebPart
Add new Web part to solution spfx-lab.
? What is your Web part name? WalmartGreeter_
```

- m) When prompted with **What is your Web part description?**, type in a short description and press **Enter**.

```
Add new Web part to solution spfx-lab.  
? What is your Web part name? WalmartGreeter  
? What is your Web part description? My first SPFX webpart
```

- n) When prompted with **Which framework would you like to use?**, press **Enter** to accept **No JavaScript Framework**.

```
Add new Web part to solution spfx-lab.  
? What is your Web part name? WalmartGreeter  
? What is your Web part description? My first SPFX webpart  
? Which framework would you like to use? (Use arrow keys)  
> No JavaScript framework  
  React  
  Knockout
```

Once you have answered all the questions, the Yeoman generator will run and add the starter files to your project folder.

- o) Wait until the Yeoman generator completes its work and display a message indicating the new solution has been created..

```
added 1851 packages in 63.063s  
  
_+#####!  
#####|  
###/  (##|(@) |  
### ##### | |  
###/  /###| (@) |  
##### ##| / |  
###   /##| (@) |  
#####| |  
*_*+#####!  
  
c:\Student\Modules\04_SharePointFramework\Lab\spfx-lab>
```

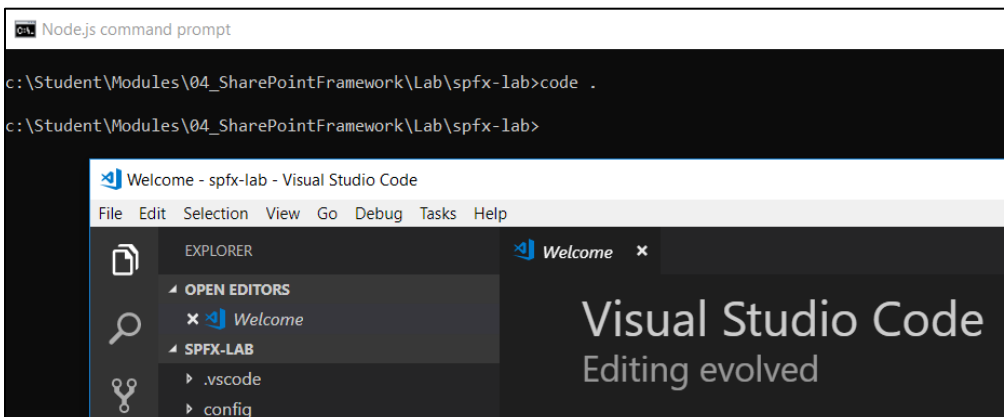
Congratulations!
Solution **spfx-lab** is created.
Run **gulp serve** to play with it!

3. Open the project with Visual Studio Code

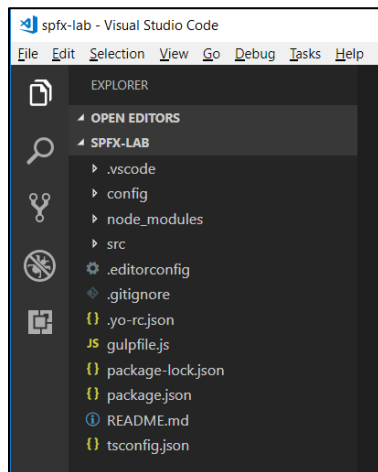
- a) Type the following command and execute it by pressing **Enter** to open your new project in Visual Studio Code.

```
code .
```

- b) As the command executes, it should open your new project folder with Visual Studio Code.



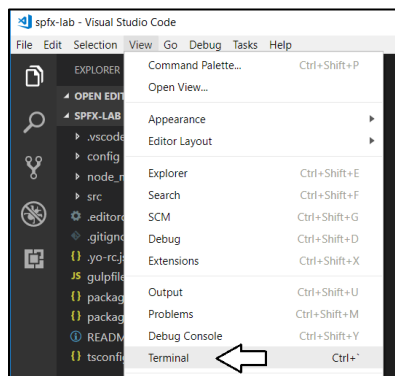
- c) Take a moment to familiarize yourself with the files and folders at the root of the **spfx-lab** project.



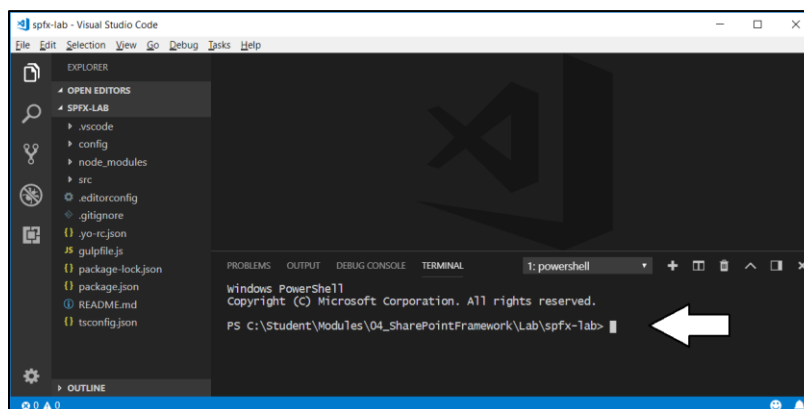
Several of these project files such as **package.json**, **tsconfig.json** and **gulpfile.js** should already be familiar to you from the work you did in the previous lab exercises of this training course.

4. Open the console window from the Integrated Terminal.

- a) Use the **View > Terminal** menu command in Visual Studio Code to display the Integrated Terminal.



- b) The Integrated Terminal should provide a console with its current directory located at your project folder.



Now you have the ability to run **npm** commands and **gulp** commands from within Visual Studio Code.

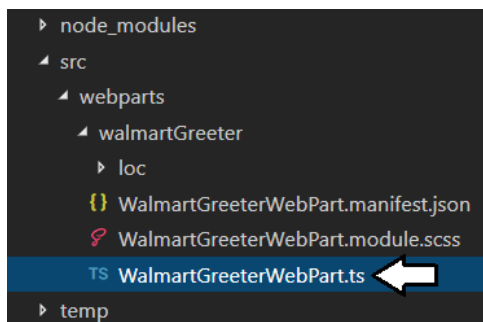
5. Run the gulp task named **trust-dev-cert** to configure your project with an SSL certificate for testing the project at <https://localhost>.
 - a) Type and execute the following command to execute the gulp task named **trust-dev-cert** that is provided by SPFx.

```
gulp trust-dev-cert
```

- b) Verify that the **trust-dev-cert** gulp task executes successfully.

```
PS C:\Student\Modules\04_SharePointFramework\Lab\spfx-lab> gulp trust-dev-cert
Build target: DEBUG
[07:51:33] Using gulpfile C:\Student\Modules\04_SharePointFramework\Lab\spfx-lab\gulpfile.js
[07:51:33] Starting gulp
[07:51:33] Starting 'trust-dev-cert'...
[07:51:33] Starting subtask 'configure-sp-build-rig'...
[07:51:33] Finished subtask 'configure-sp-build-rig' after 4.71 ms
[07:51:33] Starting subtask 'trust-cert'...
[07:51:33] Finished subtask 'trust-cert' after 67 ms
[07:51:33] Finished 'trust-dev-cert' after 73 ms
[07:51:33] =====[ Finished ]=====
[07:51:34] Project spfx-lab version:0.0.1
[07:51:34] Build tools version:3.7.4
[07:51:34] Node version:v8.11.4
[07:51:34] Total duration:3.28 s
PS C:\Student\Modules\04_SharePointFramework\Lab\spfx-lab>
```

6. Update the starter TypeScript code for the webpart class definition inside **WalmartGreeterWebPart.ts**.
 - a) Inside the **src/webparts/walmartGreeter** folder, locate and open the TypeScript file named **WalmartGreeterWebPart.ts**.



- b) You should see a TypeScript definition for a class named **WalmartGreeterWebPart**.
 - c) Inside the **WalmartGreeterWebPart** class, locate the implementation of **render** method.

```
TS WalmartGreeterWebPart.ts x
import { Version } from '@microsoft/sp-core-library';
import {
  BaseClientSideWebPart,
  IPropertyPaneConfiguration,
  PropertyPaneTextField
} from '@microsoft/sp-webpart-base';
import { escape } from '@microsoft/sp-lodash-subset';

import styles from './WalmartGreeterWebPart.module.scss';
import * as strings from 'WalmartGreeterWebPartStrings';

export interface IWalmartGreeterWebPartProps {
  description: string;
}

export default class WalmartGreeterWebPart extends BaseClientSideWebPart<IWalmartGreeterWebPartProps> {
  public render(): void {
    this.domElement.innerHTML = `
      <div class="${styles.walmartGreeter}">
        <div class="${styles.container}">
```

- d) Replace the existing **render** method implementation using the following code.

```
public render(): void {
  this.domElement.innerHTML =
    `<div class="${styles.walmartGreeter}">
      <h1>Hello world</h1>
    </div>`;
}
```

- e) The **render** method of the **WalmartGreeterWebPart** class should now match the following screenshot.

```
export default class WalmartGreeterWebPart extends BaseClientSideWebPart<IWalmartGreeterWebPartProps> {

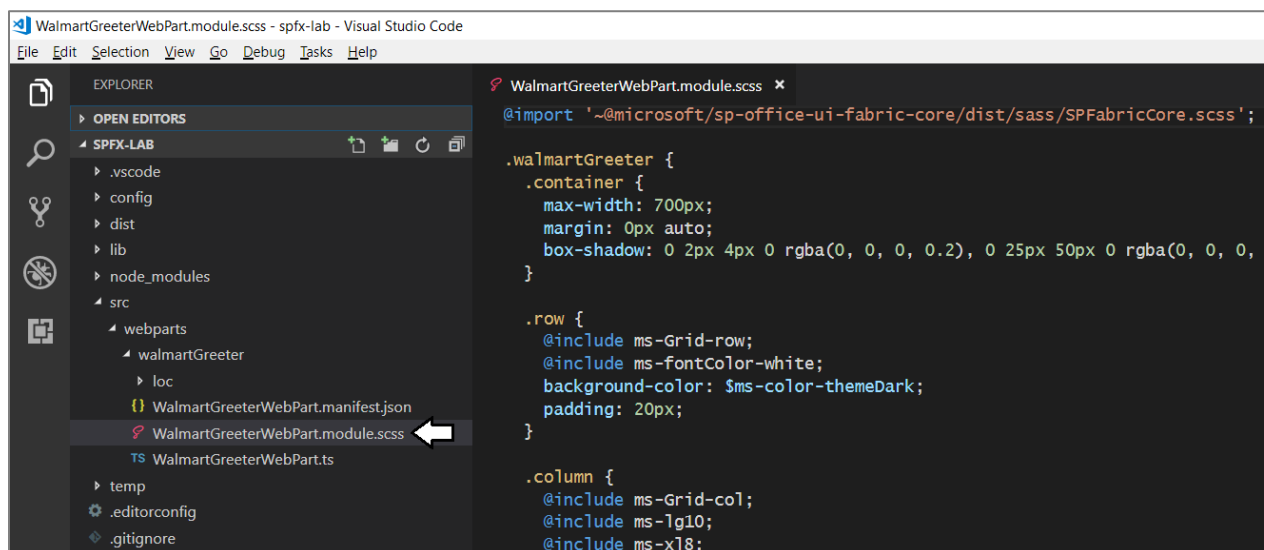
  public render(): void {
    this.domElement.innerHTML =
      `<div class="${styles.walmartGreeter}">
        <h1>Hello World</h1>
      </div>`;
  }
}
```

- f) Save your changes to **WalmartGreeterWebPart.ts** and leave this file open.

Now that you have added some minimal HTML, you will now add a little CSS styling using CSS modules.

7. Modify the CSS styles in the SCSS file named **WalmartGreeter.module.scss**.

- a) Inside the **src/webparts/walmartGreeter** folder, locate and open the SCSS file named **WalmartGreeter.module.scss**.



- b) Delete all the existing content inside **WalmartGreeter.module.scss**.
c) Add the following CSS code to **WalmartGreeter.module.scss**.

```
.walmartGreeter {
  max-width: 700px;
  border: 2px solid black;
  border-radius: 12px;
  background-color: lightyellow;
  padding: 12px;
}
```

So far, you have just added styles that are valid in any CSS file. The real advantage to using stylistically awesome style sheets (SASS) such as to **WalmartGreeter.module.scss** is that they provide syntactic features not available in standard CSS files such as the use of variables and nested classes which improve productivity and maintainability. You will now update **WalmartGreeter.module.scss** using special SASS syntax that is not allowed in a standard CSS file.

- d) Add a nested class inside the **walmartGreeter** class to style **h1** elements as shown in the following code listing.

```
.walmartGreeter {  
  max-width: 700px;  
  border: 2px solid black;  
  border-radius: 12px;  
  background-color: lightyellow;  
  padding: 12px;  
  
  h1{  
    color: darkblue;  
    font-size: 2.5em;  
  }  
}
```

- e) Add two new variables to the top of **WalmartGreeter.module.scss** named **\$background-color** and **\$font-color**.

```
$background-color: lightyellow;  
$font-color: darkblue;
```

- f) Update the **background-color** property of the **walmartGreeter** class to use the variable named **\$background-color**.

```
background-color: $background-color;
```

- g) Update the **color** property of the **h1** class to use the variable named **\$font-color**.

```
h1{  
  color: $font-color;  
  font-size: 2.5em;  
}
```

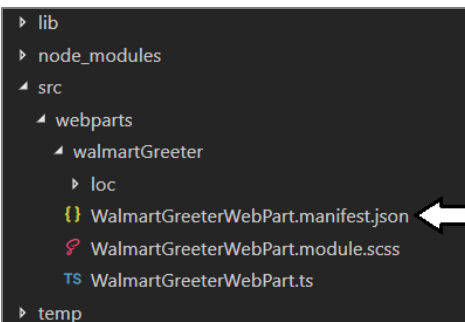
- h) At this point, the contents of **WalmartGreeter.module.scss** should match the following code listing.

```
$background-color: lightblue;  
$font-color: darkred;  
  
.walmartGreeter {  
  max-width: 700px;  
  border: 2px solid black;  
  border-radius: 12px;  
  background-color: $background-color;  
  padding: 12px;  
  
  h1{  
    color: $font-color;  
    font-size: 2.5em;  
  }  
}
```

- i) Save your changes to **WalmartGreeter.module.scss** and leave this file open.

8. Update the manifest file for the Walmart Greeter webpart.

- a) Inside **src/webparts/walmartGreeter**, open the webpart manifest file named **WalmartGreeterWebPart.manifest.json**.



You can see there's a strange issue where comments inside **WalmartGreeterWebPart.manifest.json** are displayed with a red underline indicating an error whenever the file is open in an editor window. This is not a problem when building the project, but it is a bit confusing when you have the file open because it seems as though there are errors inside it. In the next step you will remove all the comments from **WalmartGreeterWebPart.manifest.json** until all the red underlining goes away.

- b) When you examine **WalmartGreeterWebPart.manifest.json**, you can see how the comments are underlined in red.

```
WalmartGreeterWebPart.manifest.json
{
  "$schema": "https://developer.microsoft.com/json-schemas/spfx/client-side-web-part-manifest.schema.json",
  "id": "ab9ec10-54cf-4b02-a1db-66031e669c99",
  "alias": "WalmartGreeterWebPart",
  "componentType": "WebPart",

  // The "x" signifies that the version should be taken from the package.json
  "version": "x",
  "manifestVersion": 2,

  // If true, the component can only be installed on sites where Custom Script is allowed.
  // Components that allow authors to embed arbitrary script code should set this to true.
  // https://support.office.com/en-us/article/turn-scripting-capabilities-on-or-off-if2c515f-5d7e-448a-9fd7-835da935584f
  "requiresCustomScript": false,

  "preconfiguredEntries": [{
    "groupId": "5c03119e-3074-46fd-976b-c60198311f70", // other
    "group": { "default": "other" },
  ]
}
```

- c) Remove all the comments from **WalmartGreeterWebPart.manifest.json** until the red underlining is gone.
d) At the bottom of file named **WalmartGreeterWebPart.manifest.json**, locate the **preconfiguredEntries** section.

```
"preconfiguredEntries": [{
  "groupId": "8aee034e-29cc-4c44-8490-fc96a9175734",
  "group": { "default": "Under Development" },
  "title": { "default": "WalmartGreeter" },
  "description": { "default": "My First SPFx WebPart" },
  "officeFabricIconFontName": "Page",
  "properties": {
    "description": "WalmartGreeter"
  }
}]
```

- e) Inside the **preconfiguredEntries** section, modify the **default** value of **title** from **WalmartGreeter** to **Walmart Greeter**.

```
"title": { "default": "Walmart Greeter" },
```

- f) Modify the value of **officeFabricIconFontName** from **Page** to **Emoji2**.

```
"officeFabricIconFontName": "Emoji2",
```

- g) Your edits should match what is shown in the following screenshot.

```
"preconfiguredEntries": [{
  "groupId": "5c03119e-3074-46fd-976b-c60198311f70",
  "group": { "default": "Other" },
  "title": { "default": "Walmart Greeter" },
  "description": { "default": "My first SPFx webpart" },
  "officeFabricIconFontName": "Emoji2",
  "properties": {
    "description": "WalmartGreeter"
  }
}]
}
```

- h) Save your changes to **WalmartGreeterWebPart.manifest.json** and leave this file open. Note that Visual Studio Code will likely reformat the JSON code inside **WalmartGreeterWebPart.manifest.json** when you save the file.

Now you have done enough initial work on the project to test it for the first time.

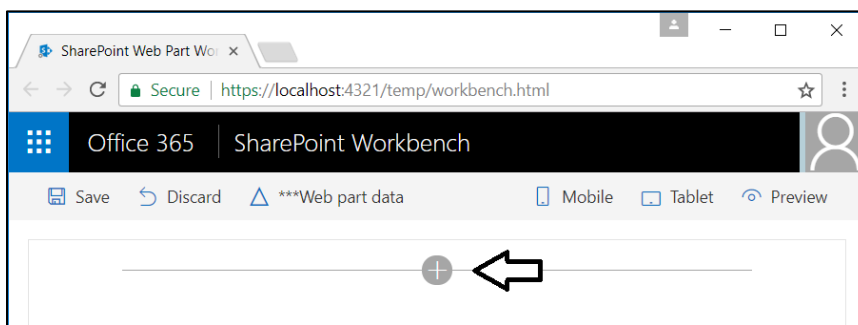
Exercise 2: Test and Debug a Webpart in the Local SharePoint Workbench

In the previous lab exercise, you created a new SharePoint Framework project and you modified its source files to prepare it for testing. In this exercise, you will learn how to run your project by serving it up through a local web server and testing your webpart in the local SharePoint Workbench. Along the way, you will also learn how to configure client-side debugging support for your project using the Chrome Debugger extension for Visual Studio Code.

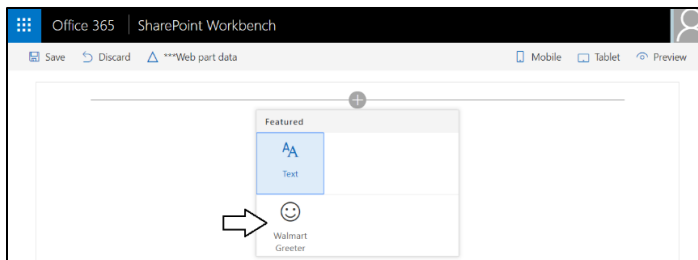
1. Test out the **spfx-lab** project by running it in the local SharePoint workbench
 - a) Navigate to the Terminal console.
 - b) Execute the **gulp serve** command to start up the project and test it out using the local workbench.

```
gulp serve
```

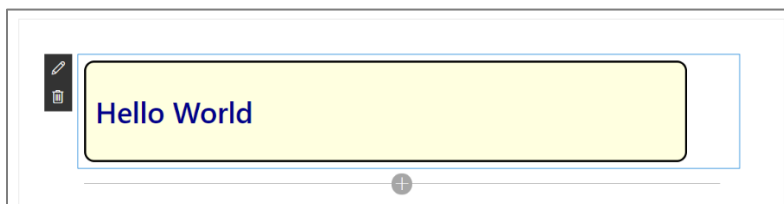
- c) The browser should launch and display a page for adding modern webparts like the one shown in the following screenshot. Click on the button with the **+** sign in the middle of the page to add your webpart to the page so you can test it.



- d) Select the **Walmart Greeter** to add it to the page as a new SPFx webpart.



- e) The webpart should display the text "Hello World".



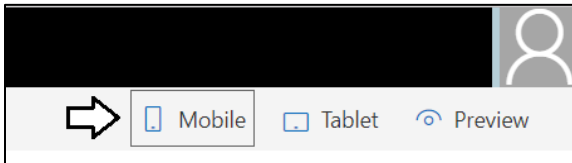
- f) Click the **Preview** button to transition the page from edit mode to preview mode.



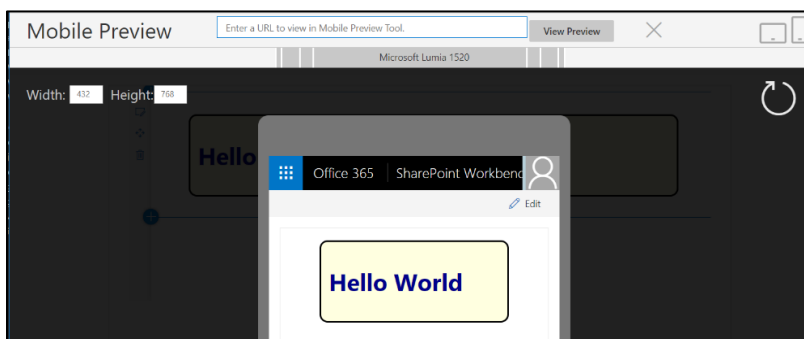
- g) Click on the **Edit** button to move the page back into edit mode.



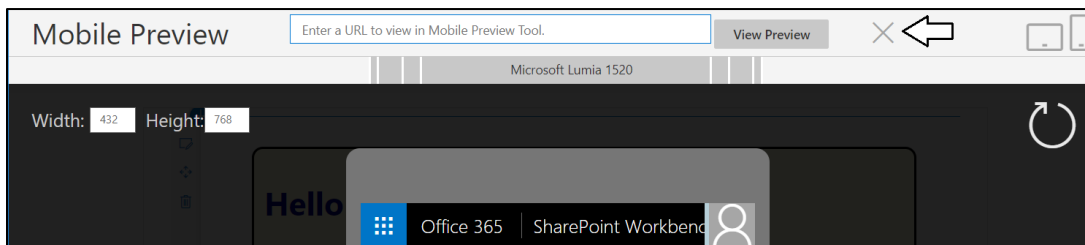
- h) Click on the **Mobile** button to see a simulation of what your webpart looks when viewed using a mobile device.



- i) You should now see a view of the webpart as it will look when viewed with a mobile device such as an iPhone.



- j) Once you have seen the Web Part in the mobile view, click the **X** at the top right of the mobile view dialog to close it.



- k) You should now be looking at the webpart in Edit view.
2. Modify the webpart source files and observe the file watching behavior that automatically updates the webpart in the browser.
- a) Return to Visual Studio Code and the editor window for the source file named **WalmartGreeterWebPart.ts**.
- b) Locate the **render** method.
- c) Modify the text inside the **h1** element from "Hello World" to "Hello World of SPFx WebParts".

```
public render(): void {  
  this.domElement.innerHTML = `  
    <div class="${styles.walmartGreeter}">  
      <h1>Hello world of SPFx webparts</h1>  
    </div>`;  
}
```

- d) Save your changes to **WalmartGreeterWebPart.ts**.

When you save your changes, you will notice activity in the Terminal console as the SharePoint Framework tools rebuild your project.

- e) Return to the browser and verify that your webpart has been automatically updated with the new text for the **h1** element.



- f) Return to Visual Studio Code and the editor window with the source file named **WalmartGreeter.module.scss**.
g) Modify the two new variables named **\$background-color** and **\$font-color** to use different colors.

```
$background-color: lightblue;  
$font-color: darkred;
```

- h) Save your changes to **WalmartGreeter.module.scss**.
i) Return to the browser and verify that the webpart has been automatically updated with the new colors.



The point of these last few steps is to show that you can edit any of the TypeScript or CSS in your project and simply save the edited files to automatically trigger rebuilding your project and refreshing the browser.

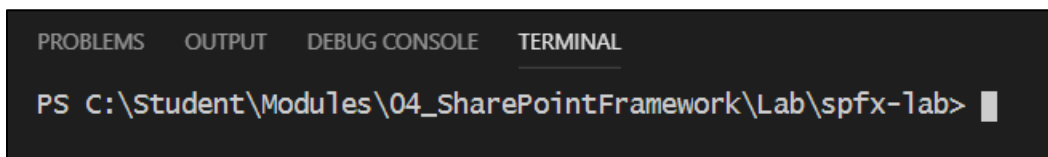
3. Stop the web server process for the current debugging session.
a) Return to the console in the Integrated Terminal.
b) Make sure the console is the active window
c) Press the **Ctrl + C** keyboard combination to stop the web server from running.

```
[08:34:35] Server stopped  
About to exit with code: 0  
Process terminated before summary could be written, possible error in async code not continuing!  
Trying to exit with exit code 1  
Terminate batch job (Y/N)?
```

- d) When prompted to **Terminate the batch job (Y/N)**, type **Y** and press **Enter**.

```
Terminate batch job (Y/N)? y  
PS C:\Student\Modules\04_SharePointFramework\Lab\spfx-lab>
```

- e) Type **cls** and then press **Enter** to clear to console window.

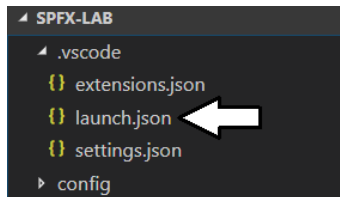


This lab assumes you have already installed the Chrome Debugger extension for Visual Studio Code. If you did not complete the previous lab on NPM and have not installed this extension, follow the steps at the following URL:

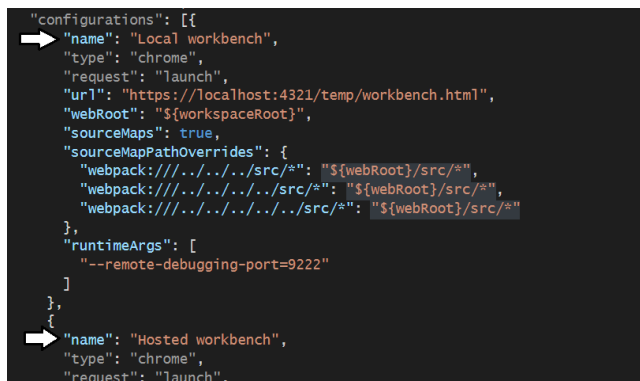
<https://github.com/SharePoint/sp-dev-docs/blob/master/docs/spfx/debug-in-vscode.md>

4. Examine the two debug configurations that have been added to the **launch.json**.

- a) Open the **launch.json** file in the **.vscode** folder and examine its contents.



- b) You can see this file contains the JSON data with two configurations named **Local workbench** and **Hosted workbench**.



- c) Close **launch.json** without saving any changes.

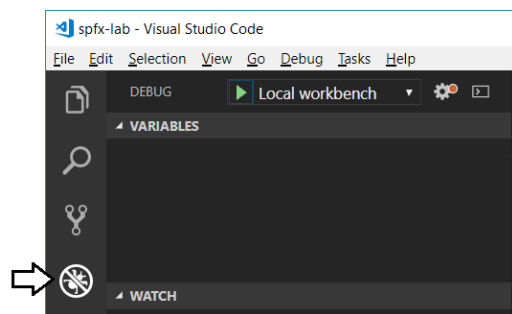
You will make an update to the **launch.json** file in a later lab exercise. For now you can just use the version of **launch.json** that was created with your project to get the project up and running the **Local workbench** debug configuration.

5. Start the web server process and launch a debugging session using the Chrome Debugger extension in Visual Studio Code.

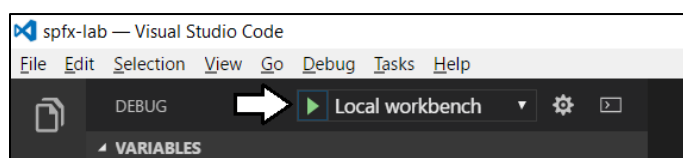
- a) Return to the console in the Integrated Terminal.
b) Execute the **gulp serve** task using the **--nobrowser** argument to start the web server without launching the browser.

```
gulp serve --nobrowser
```

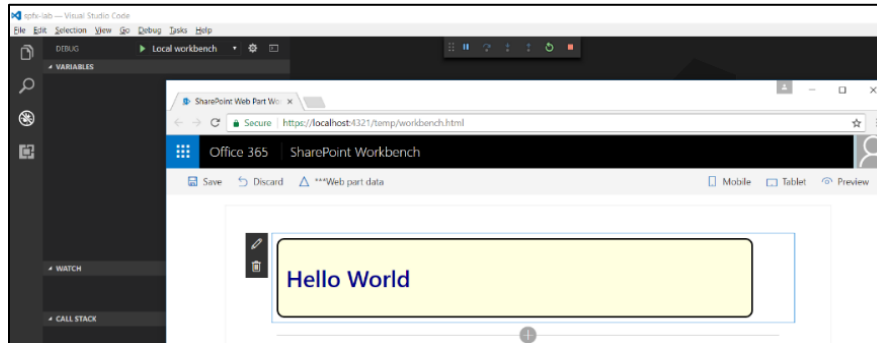
- c) Click the **Debug** tab in the left navigation.



- d) Click the button with the green arrow to begin a debugging session in Visual Studio Code.

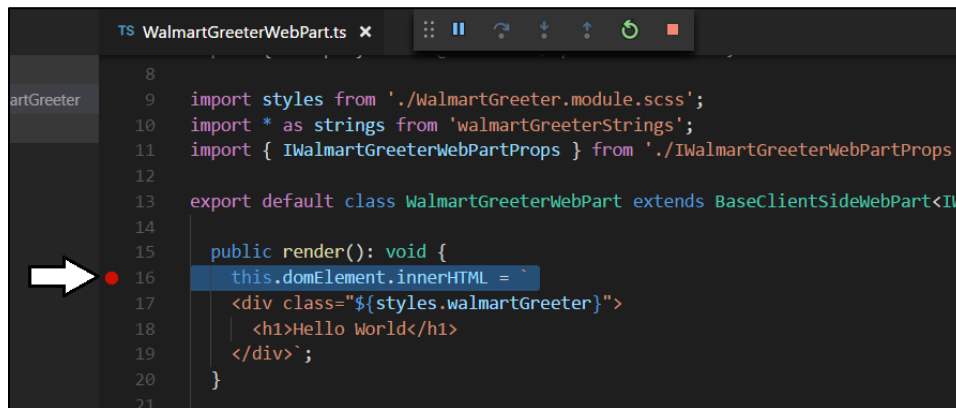


- e) The local SharePoint Workbench should launch in the browser.
- f) Add the Walmart Greeter webpart to the page as you did in previous steps of this exercise.
- g) Once you see your webpart, you should also be able to see the debugging toolbar appear in Visual Studio Code.

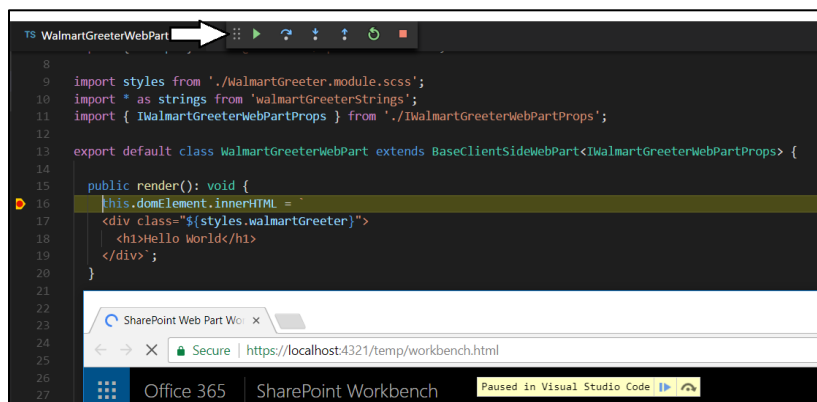


Now you are going to set a breakpoint to test see if you can single step through your code using the Visual Studio Code debugger.

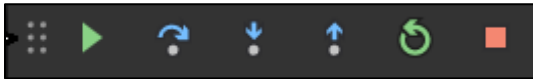
- h) Return to Visual Studio Code.
- i) Navigate to the code editor window for the TypeScript file named **WalmartGreeterWebPart.ts**.
- j) Select the first line of code in the **render** method and set a breakpoint by pressing the **{F9}** key.



- k) Return to the browser window with the local SharePoint Workbench that is displaying the browser.
- l) Refresh the page displaying the webpart.
- m) Return to Visual Studio Code and you should see that code execution has suspended at the breakpoint you set inside **render**.



- n) Experiment with the button on the debugging toolbar which let you step into and step over code while debugging



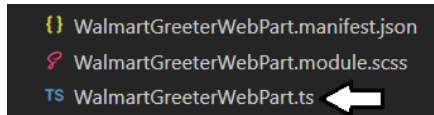
As you can see, it's not overly difficult to set up debugging so you can single step through the TypeScript code you are writing.

6. Close down the current debugging session.
 - a) Close the browser which is displaying the webpart and return to Visual Studio Code.
 - b) Navigate to the console of the Integrated Terminal.
 - c) Make sure the console is the active window
 - d) Press the **Ctrl + C** keyboard combination to stop the web server from running.
 - e) When prompted to **Terminate the batch job (Y/N)**, type **Y** and press **Enter**.
 - f) Type **cls** and then press **Enter** to clear to console window.

Exercise 3: Extend an SPFX Webpart with Custom Properties

In this exercise, you will extend the **WalmartGreeterWebPart** with a set of custom properties. You will accomplish this by designing an interface that defines a set of custom properties and then you will work through the steps to integrate the interface with your webpart class. You will also create a customized user experience for editing these webpart properties specific types of user input elements to the property pane that can be seen in webpart edit mode.

1. Redesign the interface definition inside the webpart source files named **IWalmartGreeterWebPartProps.ts**.
 - a) Open the TypeScript source file named **WalmartGreeterWebPart.ts**.



- b) The source file contains an interface named **IWalmartGreeterWebPartProps** with a single property named **description**.

```
export interface IWalmartGreeterWebPartProps {  
  description: string;  
}
```

- c) Modify the **IWalmartGreeterWebPartProps** interface by removing the **description** property and adding four new properties named **greeting**, **fontBold**, **fontSize** and **fontType** as shown in the following code listing.

```
export interface IWalmartGreeterWebPartProps {  
  greeting: string;  
  fontBold: boolean;  
  fontSize: number;  
  fontType: string;  
}
```

- d) The interface inside **WalmartGreeterWebPart.ts** should now match the following interface definition.

```
export interface IWalmartGreeterWebPartProps {  
  greeting: string;  
  fontBold: boolean;  
  fontSize: number;  
  fontType: string;  
}
```

- e) Save your changes and close **WalmartGreeterWebPart.ts**.
2. Set the default values for the four webpart properties in the webpart manifest.
 - a) Open the webpart manifest file named **WalmartGreeterWebPart.manifest.json** in a code editor window.

- b) Locate the **properties** section inside the **preconfiguredEntries** section.
- c) Update the **properties** section to match the following code listing.

```
"properties": {
  "greeting": "Welcome to Walmart" ,
  "fontBold": false,
  "fontType": "Arial",
  "fontSize": 36
}
```

- d) Save and close **WalmartGreeterWebPart.manifest.json**.
3. Modify the **render** method of the **WalmartGreeterWebPart** class
- a) Return to the code editor window for the TypeScript file named **WalmartGreeterWebPart.ts**.
 - b) Replace the current implementation of the **render** method with the following implementation.

```
public render(): void {

  var fontStyle = `font-weight:${this.properties.fontBold ? "bold" : "normal"};
                  font-family:${this.properties.fontType};
                  font-size:${this.properties.fontSize}px`;

  this.domElement.innerHTML = `
  <div class="${styles.walmartGreeter}" >
    <h1 style="${fontStyle}" >${this.properties.greeting}</h1>
  </div>`;
}
```

This new implementation of **render** reads the current value of all four custom webpart properties and uses them to control how its output to the page is displayed. Remember that each time one of these properties is updated, the webpart will automatically execute the **render** method to keep its view in sync with its underlying property values.

4. Customize the property pane editing experience for each of the four custom properties.
- a) Move up in **WalmartGreeterWebPart.ts** and locate the **import** statement for **@microsoft/sp-webpart-base**.

```
import {
  BaseClientSideWebPart,
  IPropertyPaneConfiguration,
  PropertyPaneTextField
} from '@microsoft/sp-webpart-base';
```

- b) Extend this import statement with the types **PropertyPaneToggle**, **PropertyPaneDropdown** and **PropertyPaneSlider**.

```
import {
  BaseClientSideWebPart,
  IPropertyPaneConfiguration,
  PropertyPaneTextField,
  PropertyPaneToggle,
  PropertyPaneDropdown,
  PropertyPaneSlider
} from '@microsoft/sp-webpart-base';
```

- c) Move down in **WalmartGreeterWebPart.ts** and locate the implement of **getPropertyPaneConfiguration**.
- d) Replace the current implementation of **getPropertyPaneConfiguration** with the following starter implementation.

```
protected getPropertyPaneConfiguration(): IPropertyPaneConfiguration {
  return {
    pages: [ {
      header: { description: "Greeter Web Part" },
      groups: []
    } ]
  };
}
```

- e) Inside the **groups** section, add two groups named **General Properties** and **Cosmetic Properties** using the following code.

```
groups: [
{
  groupName: "General Properties",
  groupFields: []
},
{
  groupName: "Cosmetic Properties",
  groupFields: []
}
```

- f) In the **groupFields** section of the **General Properties** group, add a single text field for **greeting** using the following code.

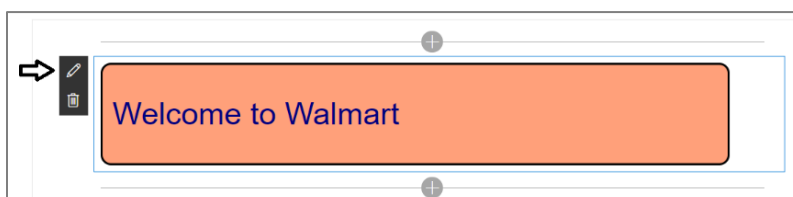
```
{
  groupName: "General Properties",
  groupFields: [
    PropertyPaneTextField('greeting', { label: 'Greeting' }),
  ]
}
```

- g) In the **groupFields** section of the **Cosmetic Properties** group, add the user interface elements for a toggle, a dropdown menu and a slider as shown in the following code listing.

```
{
  groupName: "Cosmetic Properties",
  groupFields: [
    PropertyPaneToggle('fontBold', {
      label: 'Font Bold',
      onText: 'On',
      offText: 'off'
    }),
    PropertyPaneDropdown('fontType', {
      label: 'Font Type',
      options: [
        { key: 'Arial', text: 'Arial' },
        { key: 'Times New Roman', text: 'Times New Roman' },
        { key: 'Courier,', text: 'Courier' },
        { key: 'Verdana', text: 'Verdana' }
      ]
    }),
    PropertyPaneSlider("fontSize", {
      label: "Font Size",
      min: 24,
      max: 64
    })
  ]
}
```

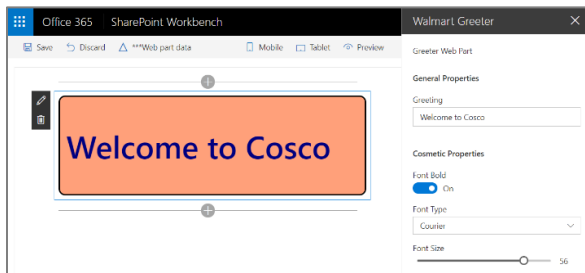
If you would rather just copy-and-paste the completed implementation of the **getPropertyPaneConfiguration** method, you can find it inside the **StarterFiles** folder in a file named **getPropertyPaneConfiguration.ts.txt**.

- h) Save your changes to **WalmartGreeterWebPart.ts**.
5. Run the webpart to test out the custom properties.
- a) Return to the Terminal console and execute the **gulp serve** command.
 - b) When the local SharePoint Workbench launches, add the Walmart Greeter webpart as you have done in previous steps.
 - c) When the webpart displays, click the Edit button to display the properties pane.

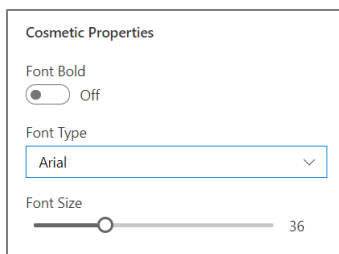


Remember that you are still in the local SharePoint workbench and there is not yet any connection to a SharePoint Online site. However, it is still possible for you to develop, test and debug a webpart with custom properties.

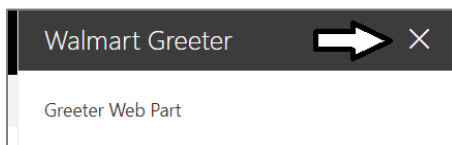
- d) At this point, you should see the webpart property pane on the right-hand side of the page.
- e) Modify the text for the **Greeting** property and see how your changes are instantly reflected in the webpart.



- f) Experiment by updating properties in the **Cosmetic Properties** group and seeing how it affects the webpart display.

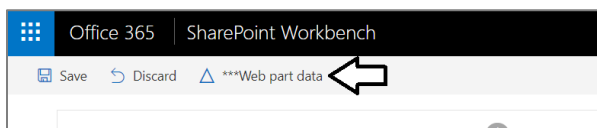


- g) When you are done, close the property pane for the **Walmart Greeter** webpart by clicking the X in the upper right corner.



The SharePoint Workbench provides you with a viewer which makes it possible to see how webpart property values are serialized for storage in SharePoint Online. Keep in mind that the SharePoint Framework defines its own new serialization format which is used for webpart instances on modern pages. For backwards compatibility with classic pages, the SharePoint Framework also supports serializing webpart instances using the classic webpart format.

- h) On the toolbar of the SharePoint Workbench, click on the **Web part data** button to view the webpart in a serialized format.



- i) Note that the Web Part Data viewer has one view for modern pages and a second for classic pages.



Exercise 4: Test a SharePoint Framework Webpart in SharePoint Online

In this exercise, you will extend the webpart with code that cannot be fully tested in the local SharePoint Workbench. Therefore, you will now learn the steps required to run and test the webpart in a hosted version of the SharePoint Workbench running inside the SharePoint Online environment.

1. Modify the **render** method of the **WalmartGreeterWebPart** class
 - a) Return to the code editor window for the TypeScript file named **WalmartGreeterWebPart.ts**.
 - b) Locate the **render** method.
 - c) Add the following line of code to the top of the **render** method before any other code.

```
var userName: string = this.context.pageContext.user.displayName;
```

- d) Update the code that generates the HTML to add **Hello \${userName}** as shown in the following code.

```
this.domElement.innerHTML = `  
<div class="${styles.walmartGreeter}">  
  <h1 style="${fontStyle}" >Hello ${userName}, ${this.properties.greeting}</h1>  
</div>`;
```

- e) At this point, the completed implementation of **render** should match the following code listing.

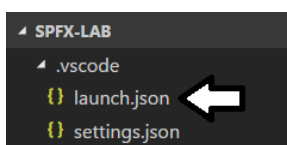
```
public render(): void {  
  var userName: string = this.context.pageContext.user.displayName;  
  
  var fontStyle = `font-weight:${this.properties.fontBold ? "bold" : "normal"};  
                  font-family:${this.properties.fontType};  
                  font-size:${this.properties.fontSize}px;`;   
  
  this.domElement.innerHTML = `  
    <div class="${styles.walmartGreeter}">  
      <h1 style="${fontStyle}" >Hello ${userName}, ${this.properties.greeting}</h1>  
    </div>`;  
}
```

- f) Save your changes to **WalmartGreeterWebPart.ts**.
2. Start a debugging session with the SharePoint Workbench.
 - a) Return to the Terminal console and execute the **gulp serve** command to launch the local SharePoint Workbench.
 - b) Add the Walmart Greeter as you have done in previous steps.
 - c) When the webpart displays, you should see the webpart displays **User 1** for the user display name.



The local SharePoint Workbench is not connected to any real SharePoint environment. Therefore, it cannot provide any information about users who have authenticated with Office 365 and SharePoint Online.

3. Add support for testing and debugging in the SharePoint Online environment.
 - a) Open the **launch.json** file in the **.vscode** folder.



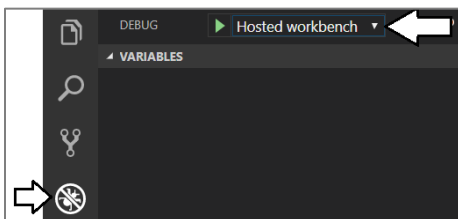
- b) Inside **launch.json**, you should see that there are two configurations named **Local workbench** and **Hosted workbench**.

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Local workbench",
      "type": "chrome",
      "request": "launch",
      "url": "https://localhost:4321/temp/workbench.html",
      "webRoot": "${workspaceRoot}",
      "sourceMaps": true,
      "sourceMapPathOverrides": {
        "webpack:///./../../src/*": "${webRoot}/src/*",
        "webpack:///./../../src/*": "${webRoot}/src/*",
        "webpack:///./../../src/*": "${webRoot}/src/*"
      },
      "runtimeArgs": [
        "--remote-debugging-port=9222"
      ]
    },
    {
      "name": "Hosted workbench",
      "type": "chrome",
      "request": "launch",
      "url": "https://enter-your-SharePoint-site/_layouts/workbench.aspx",
      "webRoot": "${workspaceRoot}",
      "sourceMaps": true,
      "sourceMapPathOverrides": {
        "webpack:///./../../src/*": "${webRoot}/src/*",
        "webpack:///./../../src/*": "${webRoot}/src/*",
        "webpack:///./../../src/*": "${webRoot}/src/*"
      },
      "runtimeArgs": [
        "--remote-debugging-port=9222",
        "--incognito"
      ]
    }
  ]
}
```

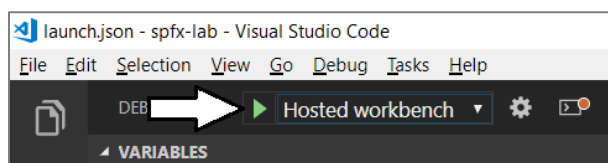
- c) Locate the line with **enter-your-SharePoint-site** and replace it with the name of the SharePoint Online development site.

```
"url": "https://msd0910.sharepoint.com/_layouts/workbench.aspx",
```

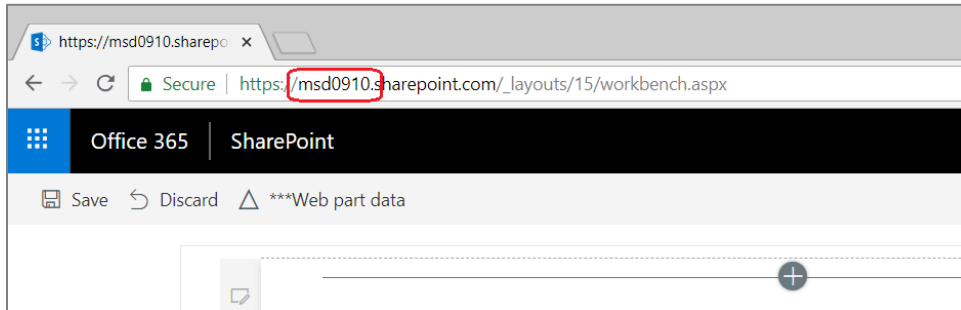
- d) In the **runtimeArgs** property array of the **Hosted workbench** configuration, remove the **--incognito** parameter.
e) Save your changes and close **launch.json**.
f) Navigate to the **Debug** tab and then select **Hosted workbench** in the dropdown configuration menu



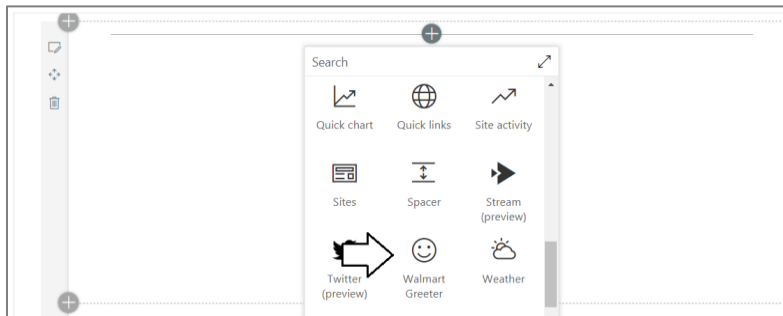
- g) Click on the button with the green arrow or press **{F5}** to start a new debugging session.



- h) The Chrome browser should launch using a URL inside the SharePoint Online environment.
- i) If you are prompted to sign in, enter the credentials of your Office 365 developer account and sign in to continue.
- j) After you are authenticated, you should see hosted page in SharePoint Online running the SharePoint Workbench.

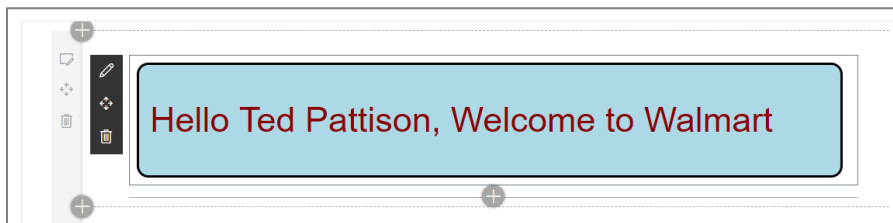


- k) Add the Walmart Greeter webpart as you have done in previous steps.



Note that quite a few other standard webparts are available once you are running inside your own tenancy in SharePoint Online.

- l) The webpart should now display the actual display name for the user account you have used to sign in.



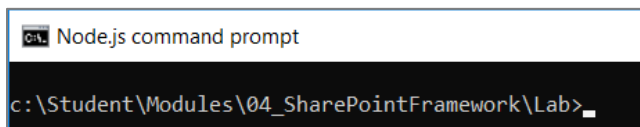
Exercise 5: Create an Application Customizer using SPFX

In this exercise, you will create a new SPFx project with an application extension that adds a page header and page footer.

1. Create a new SharePoint Framework project named **spfx-extension-lab**.
 - a) From the Node.JS command prompt, run the following command to set your current folder to the folder for this lab.

```
cd c:\Student\Modules\04_SharePointFramework\Lab
```

- b) The current directory for the console should now be at the folder for this lab inside the **Student** folder.



- c) Type the following command and execute it by pressing **Enter** to create a new folder for your project.

```
md spfx-extension-lab
```

- d) Type the following command and execute it by pressing **Enter** to move to the current directory into the new folder.

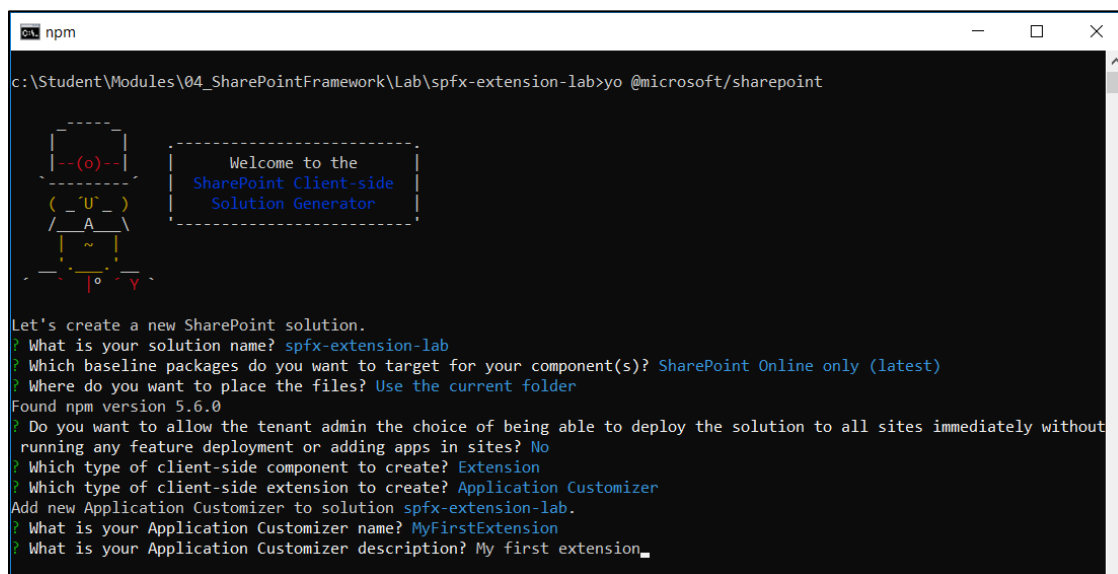
```
cd spfx-extension-lab
```

- e) The current directory for the console should now be at the new folder you just created named **spfx-extension-lab**.

- f) Type the following command and execute it to launch the Yeoman generator with the SPFx project template.

```
yo @microsoft/sharepoint
```

- g) When prompted with **What is your solution name?**, press **Enter** to accept the default value which is the name of the folder.
- h) When prompted with **Which baseline packages do you want to target for your component(s)?**, press **Enter** to accept the default value of **SharePoint Online only (latest)**.
- i) When prompted **Where do you want to place the files?**, press **Enter** to accept the default value of **Use the current folder**.
- j) When prompted **Do you want to allow the tenant admin the choice of being able to deploy to all sites immediately without running any feature deployment or adding apps in sites (y/N)?**, type "N" and press **Enter** to accept the option.
- k) When prompted with **Which type of client-side component to create?**, select **Extension** and press **Enter**.
- l) When prompted with **Which type of client-side extension to create?**, select **Application Customizer** and press **Enter**.
- m) When prompted with **What is your Application Customizer name?**, type **MyFirstExtension** and press **Enter**.
- n) When prompted with **What is your Application Customizer description?**, type in a short description and press **Enter**.



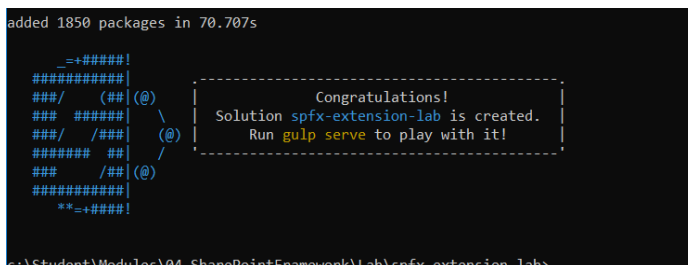
```
c:\Student\Modules\04_SharePointFramework\Lab\spfx-extension-lab>yo @microsoft/sharepoint

Welcome to the
SharePoint Client-side
Solution Generator

Let's create a new SharePoint solution.
? What is your solution name? spfx-extension-lab
? Which baseline packages do you want to target for your component(s)? SharePoint Online only (latest)
? Where do you want to place the files? Use the current folder
Found npm version 5.6.0
? Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without
running any feature deployment or adding apps in sites? No
? Which type of client-side component to create? Extension
? Which type of client-side extension to create? Application Customizer
Add new Application Customizer to solution spfx-extension-lab.
? What is your Application Customizer name? MyFirstExtension
? What is your Application Customizer description? My first extension_
```

Once you have answered all the questions, the Yeoman generator will run and add the starter files to your project folder.

- o) Wait until the Yeoman generator completes its work and display a message indicating the new solution has been created..



```
added 1850 packages in 70.707s

#####
##/ (##) (@)
##/ ##### \
##/ /### ( @)
##### ## /
##/ /## ( @)
#####
**=+#####

Congratulations!
Solution spfx-extension-lab is created.
Run gulp serve to play with it!

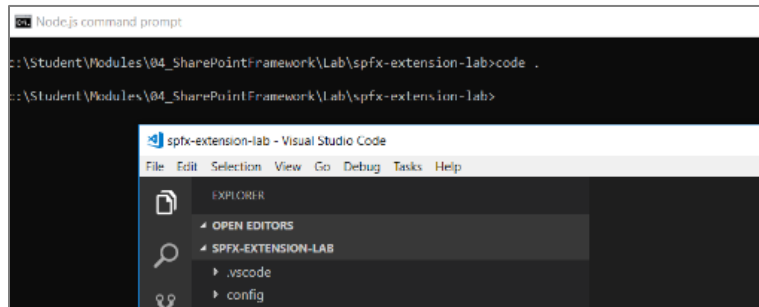
c:\Student\Modules\04_SharePointFramework\Lab\spfx-extension-lab>
```

2. Open the project with Visual Studio Code

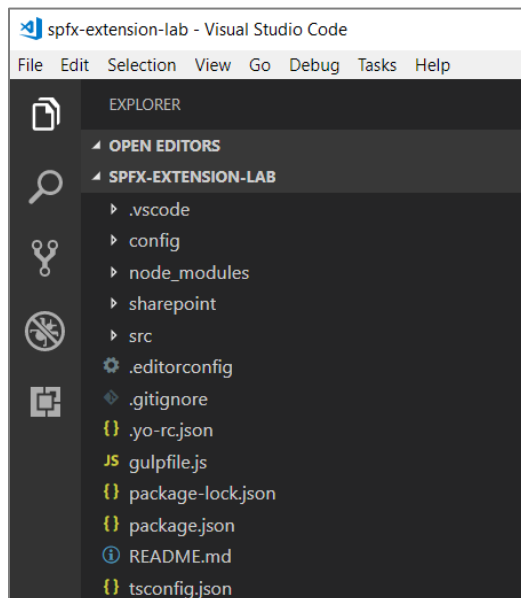
- a) Type the following command and execute it by pressing **Enter** to open your new project in Visual Studio Code.

```
code .
```

- b) As the command execute, it should open your new project folder with Visual Studio Code.

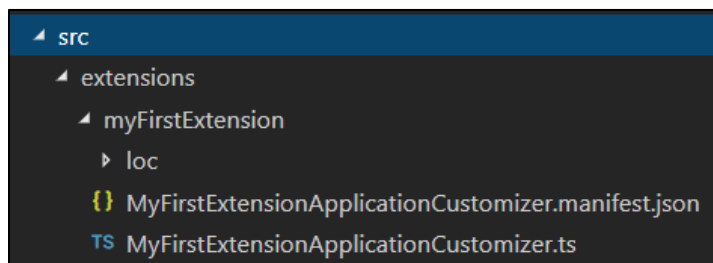


- c) Take a moment to familiarize yourself with the files and folders at the root of the **spfx-extension-lab** project.



There really isn't any difference between an application customizer project and a webpart project until you look inside the **src** folder.

- d) Expand the **src/extensions/myFirstExtension** folder.
e) There is an application customizer manifest file named **MyFirstExtensionApplicationCustomizer.manifest.json**.
f) There is an application customizer implementation file named **MyFirstExtensionApplicationCustomizer.ts**.

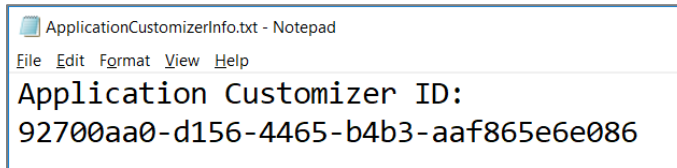


The two files in the **src/extension/myFirstExtension** folder are the primary files you work with to develop an application customizer.

3. Inspect the application customizer manifest file.
 - a) Open **MyFirstExtensionApplicationCustomizer.manifest.json**.
 - b) Remove all the comments from **MyFirstExtensionApplicationCustomizer.manifest.json**.
 - c) At this point, the application customizer manifest file should appear as the one shown in the following screenshot.

```
{
  "id": "92700aa0-d156-4465-b4b3-aaf865e6e086",
  "alias": "MyFirstExtensionApplicationCustomizer",
  "componentType": "Extension",
  "extensionType": "ApplicationCustomizer",
  "version": "*",
  "manifestVersion": 2,
  "requiresCustomScript": false
}
```

- d) Save your changes to **MyFirstExtensionApplicationCustomizer.manifest.json**
 - e) Locate the application customizer ID in the manifest file and copy it into a new document in Notepad.



You will need the GUID for this ID later in this lab when it's time to test and debug your application extension. More specifically, you will copy-and-paste this ID into a URL you will use when testing the application customizer in the browser.

- f) Close **MyFirstExtensionApplicationCustomizer.manifest.json**.
4. Inspect the application customizer implementation file.
 - a) Open **MyFirstExtensionApplicationCustomizer.ts** in an editor window.
 - b) Inspect the **render** method implementation that was added when you created by the project with the Yeoman generator.

```
@override
public onInit(): Promise<void> {
  Log.info(LOG_SOURCE, `Initialized ${strings.Title}`);

  let message: string = this.properties.testMessage;
  if (!message) {
    message = '(No properties were provided)';
  }

  Dialog.alert(`Hello from ${strings.Title}: \n\n${message}`);

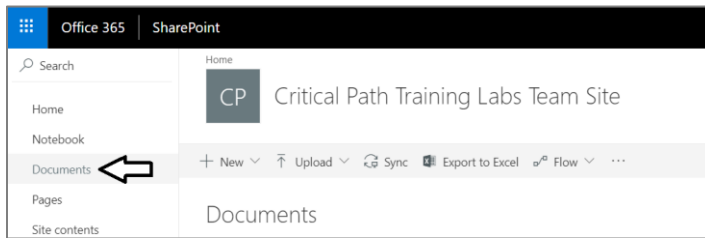
  return Promise.resolve();
}
```

While the code in **render** only displays a 'hello world' dialog, it provides enough to test the application customizer in the browser.

5. Test the application customizer in a SharePoint Online site.
 - a) Return to the console in the Integrated Terminal.
 - b) Execute the **gulp serve** task using the **--nobrowser** argument to start the web server without launching the browser.

```
gulp serve --nobrowser
```

- c) In the Chrome browser, log on to the SharePoint Online team site where you will do your testing.
- d) Once you get the SharePoint Online site, navigate to the **Documents** document library.



- e) Look at the browser address bar. You should see that the URL ends with **Shared%20Documents/Forms/AllItems.aspx**.

https://YOUR_TENANT_NAME.sharepoint.com/Shared%20Documents/Forms/AllItems.aspx

The next step is to parse together a set of query string parameters that you will paste to the end of this URL to trigger the loading of your application extension.

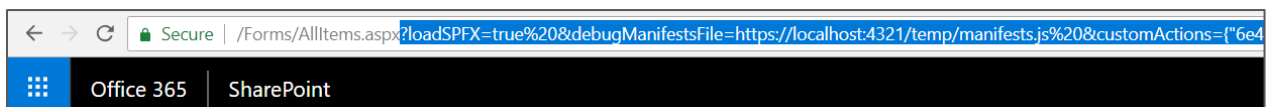
- f) Paste the following text into Notepad below down below where you pasted the GUID in an earlier step.

```
?loadSPFX=true
&debugManifestsFile=https://localhost:4321/temp/manifests.js
&customActions={"YOUR_GUID_HERE":
  {"location":"ClientSideExtension.ApplicationCustomizer",
   "properties":{"testMessage":"Hello as property!"}}
```

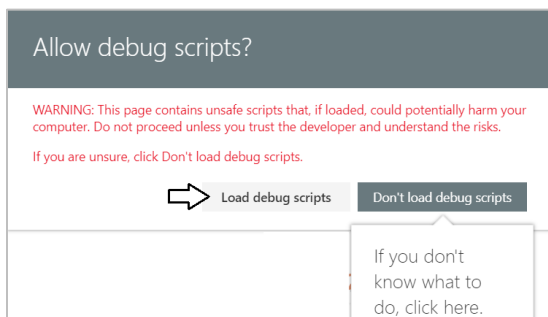
- g) Locate the **YOUR_GUID_HERE** text and replace it with your Application Customizer ID.



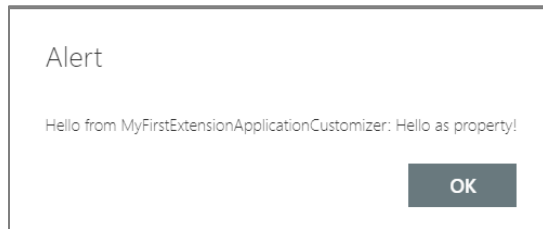
- h) Copy the text with the query string parameters for debugging the Application Customizer and paste it into the address bar rights after the URL which ends with **AllItems.aspx** and then press ENTER to trigger a new HTTP GET request.



- i) As the page refreshes, you will be prompted by the **Allow debug scripts?** dialog. Click **Load debug scripts**.



- j) When the Application Customer runs, you will be prompted with the Alert dialog shown in the following screenshot.



- k) Click OK to dismiss the Alert dialog.
l) Leave the browser window open where you tested the Application Customizer because you will return in just a minute.

Over the next few steps, you will make a change to the Application Customer and retest it. Note that when you make a change to the source file named **MyFirstExtensionApplicationCustomizer.ts** and save the file in Visual Studio Code, the project will automatically rebuild and the updated version of the Application Customizer will instantly be available for testing.

6. Make a change to the Application Customer.

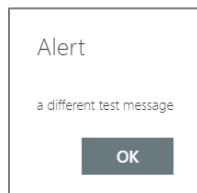
- a) Return to the Application Customizer project in Visual Studio Code.
b) Make sure you have **MyFirstExtensionApplicationCustomizer.ts** open in an editor window.
c) Locate the implementation of the **onInit** method.
d) Inside the **onInit** method, find the line of code that matches the following listing.

```
dialog.alert(`Hello from ${strings.Title}: \n\n${message}`);
```

- e) Replace that line of code with the following line.

```
dialog.alert("a different test message");
```

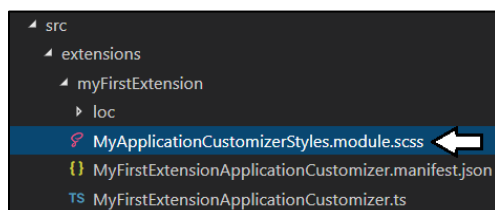
- f) Save your changes have **MyFirstExtensionApplicationCustomizer.ts**
g) Return to the page in SharePoint Online where you tested the Application Customer and refresh the page.
h) The page URL should still contain the query string parameters you pasted in earlier.
i) You should now see an Alert dialog with the updated message.



- j) Leave the browser window open where you tested the Application Customizer because you will return in just a minute.
k) Return to Visual Studio Code and stop the current debugging session.

7. Create a CSS module for your project.

- a) Return to the Application Customizer project in Visual Studio Code
b) Inside the **myFirstExtension** folder, create a new file named **MyApplicationCustomizerStyles.module.scss**.



- c) Add the following SCSS code into **MyApplicationCustomizerStyles.module.scss**.

```
$my-background-color: darkblue;
$my-font-color: yellow;

.app {
  .top {
    height: 32px;
    display: flex;
    align-items: center;
    justify-content: center;
    background-color: $my-background-color;
    color: $my-font-color;
  }

  .bottom {
    height: 32px;
    display: flex;
    align-items: center;
    justify-content: center;
    background-color: $my-background-color;
    color: $my-font-color;
  }
}
```

- d) Save and close **MyApplicationCustomizerStyles.module.scss**.
- e) Run the **gulp build** command from the Terminal console to build the project which will build out the CSS module code.
8. Update the Application Customizer to add a page header and a page footer.
- a) Open **MyFirstExtensionApplicationCustomizer.ts** in an editor window.
- b) Delete all the code this is currently inside **MyFirstExtensionApplicationCustomizer.ts**.
- c) Add the following **import** statements to the top of **MyFirstExtensionApplicationCustomizer.ts**.

```
import { override } from '@microsoft/decorators';

import {
  BaseApplicationCustomizer,
  PlaceholderContent,
  PlaceholderName
} from '@microsoft/sp-application-base';

import styles from './MyApplicationCustomizerStyles.module.scss'
```

- d) Underneath the **import** statements, add the following starter code for the Application Customizer class.

```
export default class MyFirstExtensionApplicationCustomizer
  extends BaseApplicationCustomizer<any> {

  private PageHeader: PlaceholderContent | undefined;
  private PageFooter: PlaceholderContent | undefined;

  @override
  public onInit(): Promise<void> {
  }

  private RenderPlaceHolders(): void {
  }
}
```

- e) Implement the **onInit** method to call the **RenderPlaceHolders** method using the following code

```
@override
public onInit(): Promise<void> {
  this.context.placeholderProvider.changedEvent.add(this, this.RenderPlaceHolders);
  this.RenderPlaceHolders();
  return Promise.resolve<void>();
}
```

- f) Implement the **RenderPlaceHolders** using the following code.

```
private RenderPlaceHolders(): void {  
    if (!this.PageHeader) {  
        this.PageHeader = this.context.placeholderProvider.tryCreateContent(PlaceholderName.Top);  
        if (!this.PageHeader) {  
            console.error('The expected placeholder (Top) was not found.');            return;  
        }  
        this.PageHeader.domElement.innerHTML = `  
        <div class="${styles.app}">  
            <div class="${styles.top}">  
                <div>This is the page header</div>  
            </div>  
        </div>`;  
    }  
  
    if (!this.PageFooter) {  
        this.PageFooter = this.context.placeholderProvider.tryCreateContent(PlaceholderName.Bottom);  
        if (!this.PageFooter) {  
            console.error('The expected placeholder (Bottom) was not found.');            return;  
        }  
        this.PageFooter.domElement.innerHTML = `  
        <div class="${styles.app}">  
            <div class="${styles.bottom}">  
                <div>This is the page footer</div>  
            </div>  
        </div>`;  
    }  
}
```

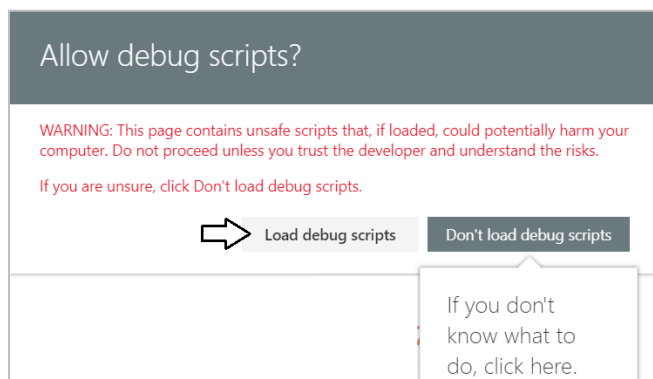
- g) Save our changes to **MyFirstExtensionApplicationCustomizer.ts**.
9. Test the application customizer in a SharePoint Online site.
- a) Return to the console in the Integrated Terminal.
- b) Execute the **gulp serve** task using the **--nobrowser** argument to start the web server without launching the browser.

```
gulp serve --nobrowser
```

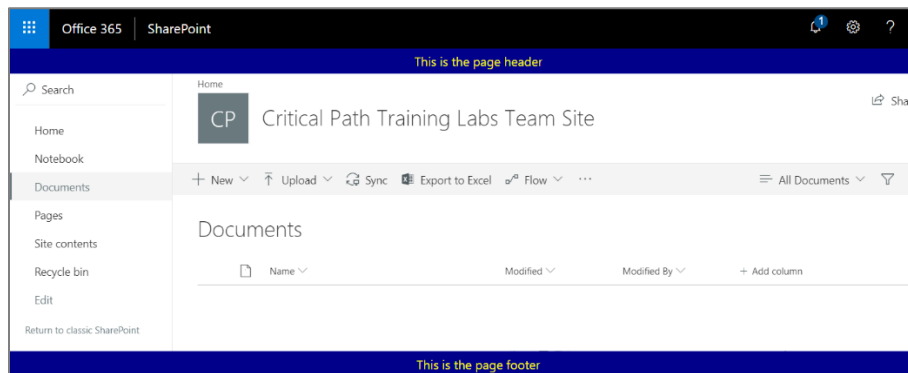
- c) Move back to the **AllItems.aspx** page in SharePoint Online where you tested the Application Customer earlier.
- d) Refresh the page.

The page URL should still contain the query string parameters for debugging an Application Customer that you pasted in earlier from Notepad. If these query string parameters are not still in the address bar following the page name AllItems.aspx, then you will have to copy and paste them again from Notepad as you did earlier.

- e) As the page refreshes, you will be prompted by the **Allow debug scripts?** dialog. Click **Load debug scripts**.



- f) When the Application Customer runs, it should generate a page header and page footer as shown in the following screenshot.



- g) If you have time, use your creativity to design a better-looking and more-useful page header and page footer by modifying the HTML layout generated in the **RenderPlaceHolders** method and the CSS styles defined inside the source file for the CSS module named **MyApplicationCustomizerStyles.module.scss**. At this point, you can simply save your edits in Visual Studio Code and then refresh the browser to see how your changes look.
- h) When you are done with your work, close the browser window, return to Visual Studio Code and stop the debugging session.

Congratulations. You are now done with this lab.