## Introduction

We’ll create a full web part from scratch. The scenario is that we work in facilities for a major company, and want to display a list of facilities, their overall status, and a list of issues coming from a SharePoint list.

You can find a full, final version of this web part at <https://github.com/mammerla/spfx-basic-webpart>.

## Prepare your PC

1. Install an Editor: Visual Studio Code

https://code.visualstudio.com/

You’ll want a great editor for your script and code. Any good editor will do – even Notepad – but Visual Studio Code provides a lot of good editing features along with a vibrant add-ons capability.

1. Install NodeJS

https://nodejs.org/

NodeJS provides a lot of functionality, including a whole server-side development framework. In our case, though, we’re really only using NodeJS for the Node Package Manager (NPM).

1. Production Windows Build Tools

npm install --global --production windows-build-tools

1. Install yeoman and gulp

npm i -g yo gulp

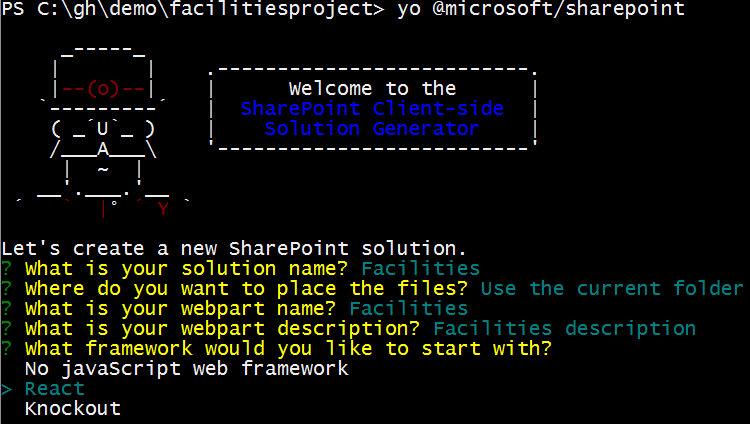
5. Add the Yeoman SharePoint Generator

npm i -g @microsoft/generator-sharepoint

# Create your First Project

6. Create a folder and run Yeoman

md facilitiesproject  
cd facilitiesproject  
yo @microsoft/sharepoint



7. Start Visual Studio Code

code .

Now we’ll start Visual Studio Code, pointed at the root folder of your project. This will open up the entire folder that you can

8. Run your first project

gulp serve

This will run the Gulp in a continuous ‘streaming’ model, so as changes are made, they will be picked up, re-compiled, and re-opened in your default browser.

# Start Retrieving Data

9. Add a State Object and Use it in Your Part

export interface IFacilitiesState  
{  
 items?: any[];  
}

Also, update the following line to replace { } with IFacilitiesState. Change this:

export default class Facilities extends React.Component<IFacilitiesProps, {}> {

to:

export default class Facilities extends React.Component<IFacilitiesProps, IFacilitiesState> {

10 . Add a Data Retrieval Constructor

constructor(props: { description : string })  
 {  
 super(props);  
 this.state = { items: new Array() };  
 let self = this;  
 fetch("https://spawesome.blob.core.windows.net/facilities/facilities.json",   
 { "credentials": "omit" } )

.then((response) => response.json())

.then((responseData) => {   
 self.setState( {  
 items: responseData,  
 });  
 });  
 }

11. Add the following render markup

return (  
 <div className={styles.facilities}>  
 <div>This is the <b>{this.props.description}</b> webpart.</div>  
 <table>  
 <tbody>  
 {

this.state.items.map(function(object, i)  
 {  
 return <tr><td><b>{object.name}</b></td><td>{object.status}</td></tr>;  
 })  
 }  
 </tbody>  
 </table>  
 </div>  
 );

Now, let gulp serve recompile and run the project. You should see a very basic list of facilities, along with a text description of their status.

## Improve the Appearance

12. Run the following command from the prompt to add Office UI Fabric Libraries to your Project:

npm i office-ui-fabric-react --save

13. Add Imports for Office UI Fabric into Facilities.tsx, beneath the existing import statements (around line #7).

import  
{  
 DetailsList  
} from 'office-ui-fabric-react';

Add the following line to IFacilitiesState to support tracking a selected item:

selectedItem?: any;

14. Add the following render Function into Facilities.tsx:

public render(): JSX.Element {

return (

<div className={styles.facilities}>

<div className="ms-font-su"> { this.props.description }</div>

<div className="ms-Grid">

<div className="ms-Grid-row">

<div className="ms-Grid-col ms-u-sm6 ms-u-md4 ms-u-lg6">

<DetailsList items={ this.state.items }

onItemInvoked={ (item, index) => this.setState( { selectedItem: item } ) }

onRenderItemColumn={ \_renderItemColumn }

columns={

[

{

key: "status",

name: "Status",

fieldName: "status",

minWidth: 60

},

{

key: "name",

name: "Name",

fieldName: "name",

minWidth: 300

}

] } />

</div>

</div>

</div>

</div>

);

}

}

function \_renderItemColumn(item, index, column)

{

const fieldContent = item[column.fieldName];

switch (column.key)

{

case 'status':

return <div style={ { backgroundColor: fieldContent, borderRadius: "16px", width: "16px", marginLeft: "6px" } }>&nbsp;</div>;

default:

return <span>{ fieldContent }</span>;

}

}

Now, let the project run and compile via Gulp Serve. You should be able to see a much nicer list of facilities.

## Now, let’s add a new React Component

15. Add a new React component called “facility.tsx” to the “facilities” folder.

import \* as React from 'react';

import {   
  EnvironmentType   
} from ['@microsoft/sp-client-base'](mailto:'@microsoft/sp-client-base');

import styles from '../Facilities.module.scss';

import {   
  IWebPartContext   
} from ['@microsoft/sp-client-preview'](mailto:'@microsoft/sp-client-preview');

import   
{   
  DocumentCard,   
  DocumentCardPreview,   
  DocumentCardActivity,   
  DocumentCardTitle   
} from 'office-ui-fabric-react';

export interface IFacilityState {   
}

export interface IFacilityProps {   
  context?: IWebPartContext;   
  item?: any;   
}

export default class Facility extends React.Component<IFacilityProps, IFacilityState> {   
  constructor(props: { context : IWebPartContext })   
  {   
    super(props);

  }

  public render(): JSX.Element {

    return (   
      <div>

        <DocumentCard>   
          <DocumentCardTitle title={ this.props.item ? this.props.item.name : '' } />   
          <DocumentCardPreview previewImages={ [   
            this.props.item ?   
            {   
              previewImageSrc: "<https://spawesome.blob.core.windows.net/facilities/>" + this.props.item.name.toLowerCase() + ".jpg"   
            } : ''   
          ]}/>   
          <DocumentCardActivity   
            activity='Facility Manager'   
            people={   
                   this.props.item ?   
                    [   
                      {   
                        name: this.props.item.facilitiesManagerName,   
                        profileImageSrc: 'https://spawesome.blob.core.windows.net/resources/avatar-' + this.props.item.facilitiesManagerAlias + '.png'   
                      }   
                    ] : null   
                  }   
                />   
        </DocumentCard>   
      </div>   
    );   
  }   
}

16. We need to plumb through some web part contextual selection, along with a list throughout our components. Add the following to facilities.tsx, beneath the div that added the DetailsList:

            <div className="ms-Grid-col ms-u-sm6 ms-u-md8 ms-u-lg6">   
              <Facility context={this.props.context} item={this.state.selectedItem}  />   
            </div>

Add the following line to IFacilitiesWebPartProps, in IFacilitiesWebPartProps.ts:

list?: string;

Add the following to IFacilitiesProps, in Facilities.tsx:

context: IWebPartContext;

Add the following to the top of Facilities.tsx:

import Facility from './Facility';

import {

IWebPartContext

} from '@microsoft/sp-client-preview';

Add the following to the constructor:

, list : string, context : IWebPartContext

So the line will go from reading:

constructor(props: { description : string })

to

constructor(props: { description : string, list : string, context : IWebPartContext })

Add the line to the element creator in FacilitiesWebPart.ts:

It’ll go from:

const element: React.ReactElement<IFacilitiesProps> = React.createElement(Facilities, {

description: this.properties.description,

});

to:

const element: React.ReactElement<IFacilitiesProps> = React.createElement(Facilities, {

description: this.properties.description,

context: this.context

});

## Let users pick a list:

We want to let users pick a list to use as the source of Issues for our Facilities. To do this, we want to add a list picker – and to do this, we’d need a List of Lists in a SharePoint site.

17. Add a new data type definition file, underneath the <facilities> folder. Call it ISPListList.ts. Add the following content:

// Define List Models   
export interface ISPListList {   
  value: ISPList[];   
}

export interface ISPList  {   
  Title: string;   
  Description: string;   
}

18. Add a new file for a Mock Data Http Client, at <tests>/MockListHttpClient.ts:

// Setup mock Http client   
import { ISPList } from '../ISPListList';

export default class MockListListHttpClient {

    private static \_items: ISPList[] = [{ Title: 'Mock Issue List', Description: '1' }];

    public static get(restUrl: string, options?: any): Promise<ISPList[]> {   
      return new Promise<ISPList[]>((resolve) => {   
            resolve(MockListListHttpClient.\_items);   
        });   
    }   
}

19. Add some data retrieval code to FacilitiesWebPart.ts, right above the Render() function:

// Setup the Web Part Property Pane Dropdown options   
  private \_dropdownOptions: IPropertyPaneDropdownOption[] = [];

  public onInit<T>(): Promise<T> {   
      this.\_getLists()   
        .then((response) => {   
          this.\_dropdownOptions = response.value.map((list: ISPList) => {   
            return {   
              key: list.Title,   
              text: list.Title   
          };   
        });   
      });

    return Promise.resolve();   
  }

  // Retrieve Lists from SharePoint   
  private \_getLists(): Promise<ISPListList> {   
    if (this.context.environment.type === EnvironmentType.Local) {   
      return MockListListHttpClient.get(this.context.pageContext.web.absoluteUrl)   
      .then((response) => {   
         const listData: ISPListList = {   
            value:   
            [   
                { Title: 'Mock List 1', Description: '1' },   
                { Title: 'Mock List 2', Description: '2' },   
                { Title: 'Mock List 3', Description: '3' },   
                { Title: 'Mock List 4', Description: '4' }   
            ]   
            };   
        return listData;   
      });   
    }   
    else   
    {   
      return this.context.httpClient.get(this.context.pageContext.web.absoluteUrl + `/\_api/web/lists`)   
        .then((response: Response) => {   
        return response.json();   
      });   
    }   
  } 

20. Add the following to FacilitiesWebPart.ts, down near the bottom in propertyPaneSettings(). Make sure you add a comma to separate the two properties.

     , PropertyPaneDropdown('list', {   
                  label: 'List',   
                  options: this.\_dropdownOptions   
                })

21. Add the following to the top of FacilitiesWebPart.ts;

import { ISPListList, ISPList } from './ISPListList';

import {  
 EnvironmentType

} from '@microsoft/sp-client-base';

import MockListListHttpClient from './tests/MockListListHttpClient';

22. Add the following:

IPropertyPaneDropdownOption,

PropertyPaneDropdown

to the list of imports in FacilitiesWebpart.ts

23. Ensure that the rendering function in FacilitiesWebPart.ts looks like:

    const element: React.ReactElement<IFacilitiesProps> = React.createElement(Facilities, {   
      description: this.properties.description,   
      list: this.properties.list,   
      context: this.context   
    });

## Retrieve and Display Lists

Add a list retrieval function to your Facility display, to show a list of issues.

25. Create MockIssueListHttpClient.ts underneath tests/ and add the following to it:

// Setup mock Http client   
import { ISPIssue } from '../ISPIssueList';

export default class MockIssueListHttpClient {

    private static \_items: ISPIssue[] = [{ Title: 'Mock Issue List', Description: '1' }];

    public static get(restUrl: string, options?: any): Promise<ISPIssue[]> {   
      return new Promise<ISPIssue[]>((resolve) => {   
            resolve(MockIssueListHttpClient.\_items);   
        });   
    }   
}

25. Add the following to ISPIssueList.ts, underneath Facilities:

// Define Issue Models   
export interface ISPIssueList {   
  value: ISPIssue[];   
}

export interface ISPIssue {   
  Title: string;   
  Description: string;   
}

26. Update the top of facility.tsx, replacing the existing IFacilityState/IFacilityProp:

import MockIssueListHttpClient from '../tests/MockIssueListHttpClient';   
import { ISPIssueList } from '../ISPIssueList';

export interface IFacilityState {   
  issues?: ISPIssueList;   
}

export interface IFacilityProps {   
  context?: IWebPartContext;   
  item?: any;   
  list?: string;   
}

27. Replace the interior, starting from the bottom of the constructor:

    this.state = { issues: null };   
  }

  private lastList : string  = null;   
  private lastItem : string = null;

  // Define and retrieve mock List data   
  private \_getMockListData(): Promise<ISPIssueList> {   
    return MockIssueListHttpClient.get(this.props.context.pageContext.web.absoluteUrl).then(() => {   
        const listData: ISPIssueList = {   
            value:   
            [   
                { Title: 'Mock Issue 1', Description: '1' },   
                { Title: 'Mock Issue 2', Description: '2' },   
                { Title: 'Mock Issue 3', Description: '3' },   
                { Title: 'Mock Issue 4', Description: '4' }   
            ]   
            };   
        return listData;   
    }) as Promise<ISPIssueList>;   
  }

  // Retrieve List data from SharePoint   
  private \_getListData(): Promise<ISPIssueList> {   
    return this.props.context.httpClient.get(this.props.context.pageContext.web.absoluteUrl + `/\_api/web/lists/GetByTitle('` + this.props.list + `')/items ?$filter=Facility eq '` + this.props.item.name + `'`)   
      .then((response: Response) => {   
      return response.json();   
      });   
  }

  // Call methods for List data retrieval   
  private \_retrieveListAsync(): void   
  {   
    const self = this;

    this.lastItem = this.props.item;   
    this.lastList = this.props.list;

    // Mock List data   
    if (this.props.context.environment.type === EnvironmentType.Local) {   
      this.\_getMockListData().then((response) => {   
        self.setState( {   
          issues: response,   
        });   
      });   
    }   
    // Get Full List data   
    else {   
      this.\_getListData()   
        .then((response) => {   
          self.setState( {   
            issues: response,   
          });   
      });   
    }   
  }

  public render(): JSX.Element {

    if (this.props.item != null && this.props.list != null && this.props.context != null   
        && (this.props.item != this.lastItem || this.props.list != this.lastList))   
    {   
      this.\_retrieveListAsync();   
    } 

28. Add a simple renderer to the bottom of facility.tsx:

    <div>{ this.props.list ? this.props.list : '(no list was selected.)' }</div>   
        <table>   
          <tbody>   
            {   
              this.state.issues ?   
              this.state.issues.value.map(function(object, i) {   
                return <tr><td><b>{object.Title}</b></td><td>{object.Description}</td></tr>;   
              }) : ''   
            }   
          </tbody>   
        </table> 

29. Add a

list? : string

property to IFacilityProps, and IFacilitiesWebPartProps; add

list={this.item.props}

to Facilities.tsx

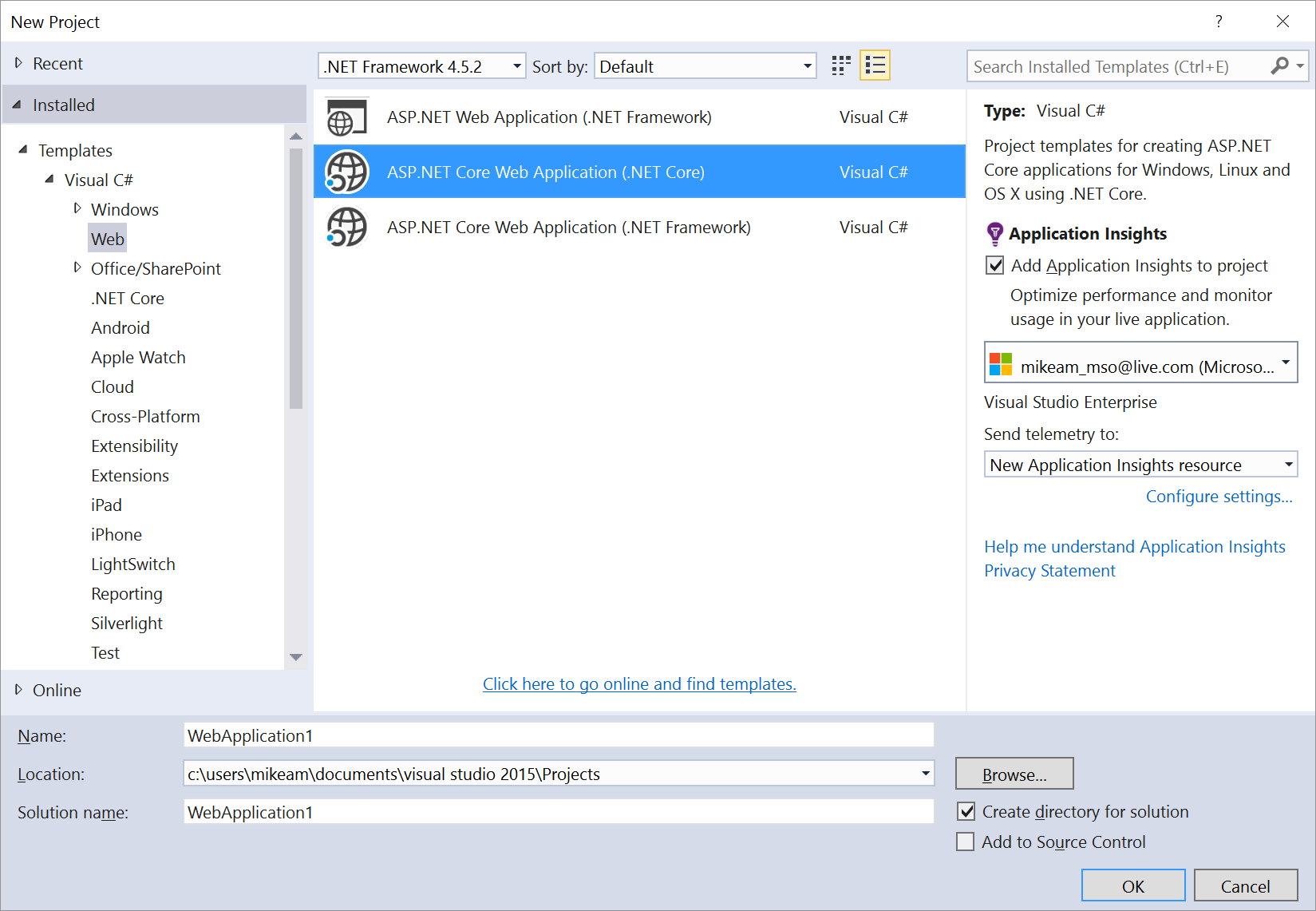
and a

list:this.properties.list

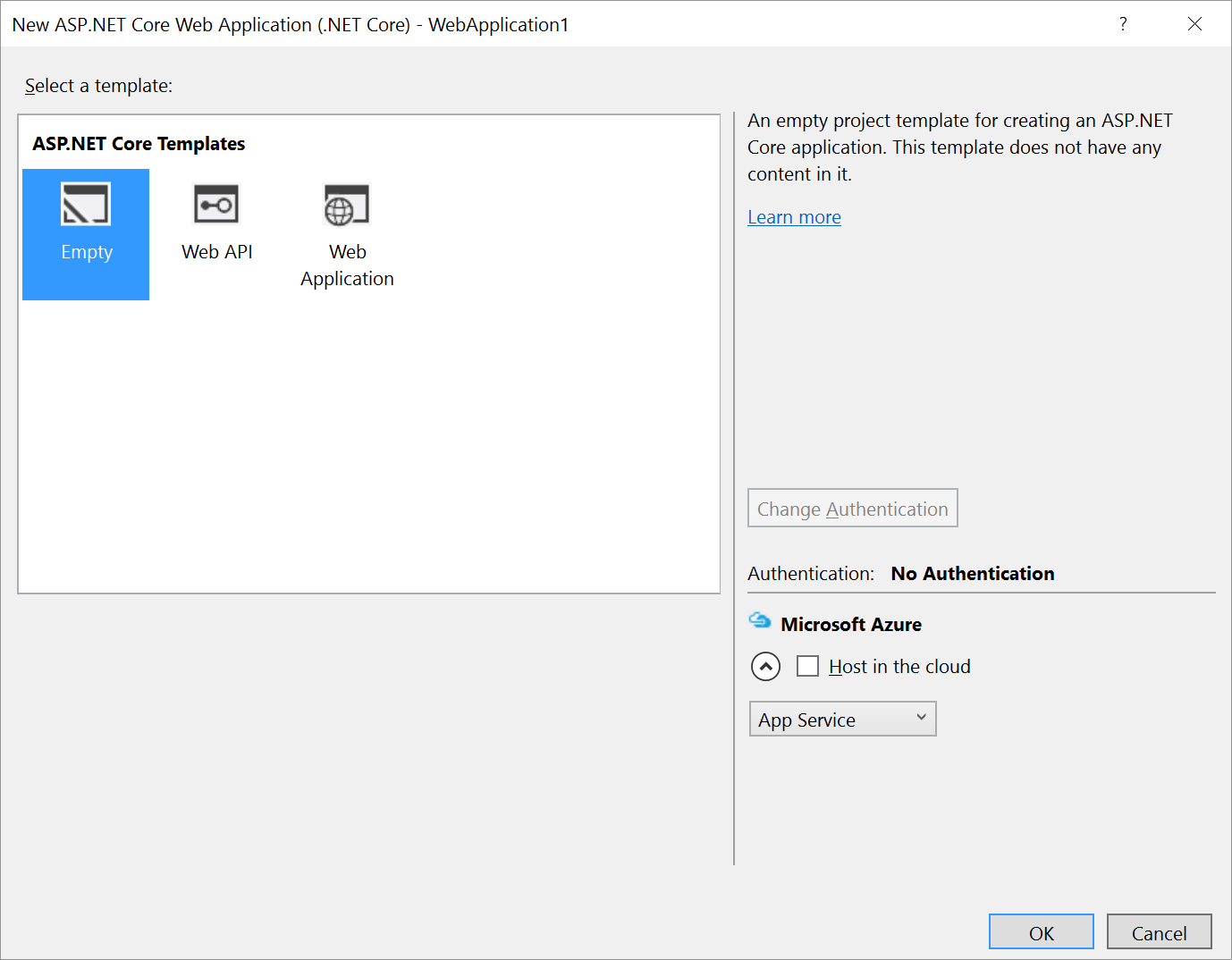
IFacilitiesWebPart.ts as appropriate.

You’re done!

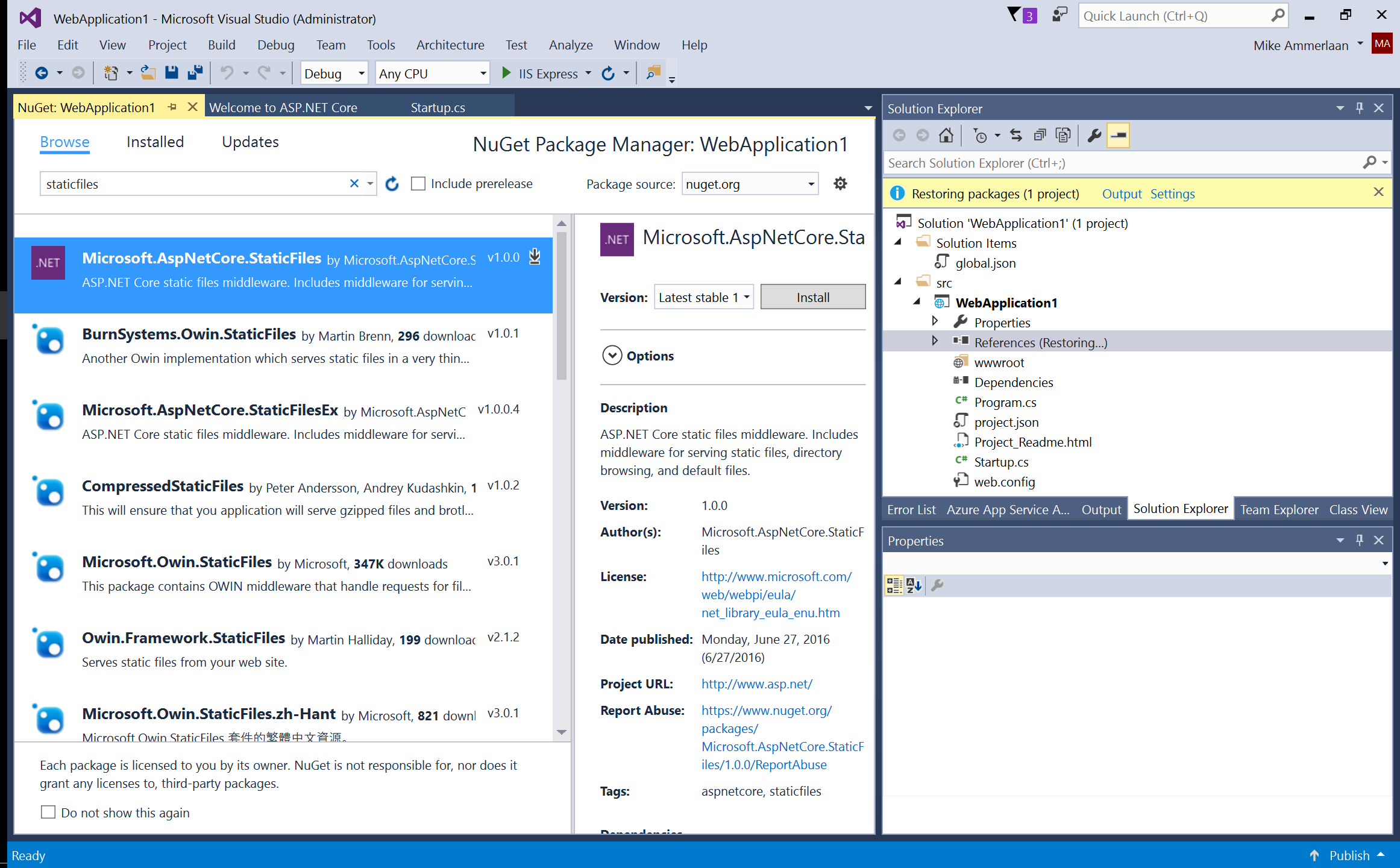
Create a new ASP.NET Core Project. You may need to install ASP.NET Core Tools for Visual Studio (<https://docs.microsoft.com/en-us/dotnet/articles/core/windows-prerequisites>)



Choose an empty project:



In the project, Right Click on References, select Manage Nuget Packages, and select Browse. Search for static files. Add Microsoft.AspNetCore.StaticFiles to your project.



Double click on Startup.cs.

Add the following code to the top:

using Microsoft.AspNetCore.StaticFiles;

Add the following code as well, within Configure():

app.UseDefaultFiles();

app.UseStaticFiles();

Add a file under wwwroot called index.html. and add the following content; paste the following start up code that will authenticate a user.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>SharePoint Info Data Display</title>

<link rel="stylesheet" href="https://appsforoffice.microsoft.com/fabric/fabric-core/4.0.0/fabric.min.css">

<link rel="stylesheet" href="//spawesome.blob.core.windows.net/resources/simpleuserinfo.css">

<script src="http://code.jquery.com/jquery-2.2.4.min.js"

integrity="sha256-BbhdlvQf/xTY9gja0Dq3HiwQF8LaCRTXxZKRutelT44="

crossorigin="anonymous"></script>

<script src="//secure.aadcdn.microsoftonline-p.com/lib/1.0.0/js/adal.min.js"></script>

</head>

<body>

<div id="content-header">

<div class="ms-font-xl header">Simple SharePoint Info Display</div>

<div class="contentArea">

<div id="signedInAs">

<span class="ms-font-m userLabel">Signed in as:</span>

<strong><span class='app-userDisplay ms-font-m'></span></strong>

<button href="javascript:;" class="app-signOut ms-font-m">Logout</button>

<button href="javascript:;" class="app-signIn ms-font-m">Login</button>

</div>

<div>

<span class="ms-font-m listsLabel">Lists:</span>

<select id="app-listPicker"></select>

<div id="results" class="ms-font-m">

</div>

<div id="items" class="ms-font-m">

</div>

</div>

</div>

</div>

<script>

window.authConfig = {

tenant: '<YOUR TENANCY>.onmicrosoft.com',

clientId: '<YOUR CLIENT ID>',

postLogoutRedirectUri: window.location.origin,

endpoints: {

officeGraph: 'https://graph.microsoft.com',

},

cacheLocation: 'localStorage'

};

var authContext = new AuthenticationContext(window.authConfig);

var $userDisplay = $(".app-userDisplay");

var $signInButton = $(".app-signIn");

var $signOutButton = $(".app-signOut");

var listPicker = document.getElementById("app-listPicker");

// Check For & Handle Redirect From AAD After Login

var isCallback = authContext.isCallback(window.location.hash);

authContext.handleWindowCallback();

if (isCallback && !authContext.getLoginError()) {

window.location = authContext.\_getItem(authContext.CONSTANTS.STORAGE.LOGIN\_REQUEST);

}

// Check Login Status, Update UI

var user = authContext.getCachedUser();

// is user signed in?

if (user) {

$userDisplay.html(user.userName);

// retrieveLists();

// getDataFromSelection();

$userDisplay.show();

$signInButton.hide();

$signOutButton.show();

} else {

$userDisplay.empty();

$userDisplay.hide();

$signInButton.show();

$signOutButton.hide();

}

// Register NavBar Click Handlers

$signOutButton.click(function () {

authContext.logOut();

});

$signInButton.click(function () {

authContext.login();

});

</script>

</body>

</html>

Now we’ve created a basic page that will sign you in and out.

But we want to go one step further and offer people a way to work with SharePoint data. To do this, add the following markup:

function getDataFromSelection() {

authContext.acquireToken("https://graph.microsoft.com", function (error, token) {

if (error || !token) {

$("#results").html("Error/no token: " + error);

return;

}

var id = listPicker.value;

var url = "https://graph.microsoft.com/beta/" + window.authConfig.tenant + "/me/sharepoint/site/lists/" + id ;

var html = "";

// request the site data

$.ajax({

beforeSend: function (request) {

request.setRequestHeader("Accept", "application/json");

},

type: "GET",

url: url,

dataType: "json",

headers: {

'Authorization': 'Bearer ' + token,

}

}).done(function (response) {

html += "<table>"

html += getPropertyHtml("Description", response.description);

html += getPropertyHtml("Created Date Time", response.createdDateTime);

html += "</table>";

$("#results").html(html);

}).fail(function (response) {

$("#results").html("Web Request Failed: " + response.responseText);

});

url = "https://graph.microsoft.com/beta/" + window.authConfig.tenant + "/me/sharepoint/site/lists/" + id + "/items";

html = "";

// request the site data

$.ajax({

beforeSend: function (request) {

request.setRequestHeader("Accept", "application/json");

},

type: "GET",

url: url,

dataType: "json",

headers: {

'Authorization': 'Bearer ' + token,

}

}).done(function (response) {

var listItems = response.value;

for (var i = 0; i < listItems.length; i++) {

html += getItemHtml(listItems[i]);

}

$("#items").html(html);

}).fail(function (response) {

$("#results").html("Web Request Failed: " + response.responseText);

});

});

}

function retrieveLists() {

authContext.acquireToken("https://graph.microsoft.com", function (error, token) {

if (error || !token) {

$("#results").html("Error/no token: " + error);

return;

}

var url = "https://graph.microsoft.com/beta/" + window.authConfig.tenant + "/me/sharepoint/site/lists";

var html = "";

// request the list data

$.ajax({

beforeSend: function (request) {

request.setRequestHeader("Accept", "application/json");

},

type: "GET",

url: url,

dataType: "json",

headers: {

'Authorization': 'Bearer ' + token,

}

}).done(function (response) {

var lists = response.value;

for (var i = 0; i < lists.length; i++) {

ensureOption(lists[i].id, lists[i].name);

}

});

});

}

listPicker.addEventListener("change", getDataFromSelection);

function getPropertyHtml(key, value) {

var propHtml = "<tr><td><strong>" + key + "</strong></td><td>"

if (value != null) {

propHtml += value;

}

return propHtml + "</td></tr>";

}

function ensureOption(value, displayName) {

for (var opt in listPicker.options) {

if (opt.value == value)

return;

}

var newOpt = document.createElement("option");

newOpt.value = value;

newOpt.innerHTML = displayName;

if (listPicker.length < 1) {

newOpt.selected = true;

}

listPicker.appendChild(newOpt);

}

function getItemHtml(value) {

var itemHtml = "<table style='margin-top:20px; border:solid 1px #C0C0C0'>";

itemHtml += getPropertyHtml("Url", value.webUrl);

itemHtml += getPropertyHtml("Item Id", value.listItemId);

itemHtml += getPropertyHtml("Created Date Time", value.createdDateTime);

itemHtml += "</table>";

return itemHtml;

}

Also, uncomment these lines in the original markup we pasted in:

retrieveLists();

getDataFromSelection();