Configuration Settings for Angular Applications – Part 2

In the previous blog post on creating a configuration settings system for Angular, you learned to create a configuration settings service to retrieve default settings for your application. You first learned to hard-code a settings class with values, then how to read those same settings from a JSON file.

In this blog post you take those settings from a JSON file, and a Web API call, and store them into local storage. You then learn to modify and delete those values in local storage. If you delete the values in local storage, you can re-read from the JSON file, or make another call to the Web API, to revert back to the original settings values.

Store Settings in Local Storage

All modern browsers allow you to store key/pair values into local storage that persists across browser sessions. This storage is ideal for small amounts of data that you might need across sessions. Global setting values that you wish to modify are an ideal candidate for storing within local storage since items in JSON files cannot be modified programmatically.

If you don't have the code from the previous blog post, and you wish to follow along and create the sample in this blog post, then perform the following. Go to www.pdsa.com/downloads, choose "PDSA Blogs" from the Category, then select "Configuration Settings for Angular Applications". Once you have this sample downloaded you can follow along with this blog.

Saving Data into Local Storage

Open the appsettings.service.ts file, located in the \src\app\shared folder and add a constant at the top of this file called SETTINGS_KEY. This constant is the key used when retrieving or storing values in local storage.

```
const SETTINGS_KEY = "configuration";
```

Create a saveSettings() method which accepts an instance of an AppSettings class. Call the setItem() method on the localStorage object. You pass the key value contained in the SETTINGS_KEY constant, and then you stringify the AppSettings object to store it into local storage.

```
saveSettings(settings: AppSettings) {
  localStorage.setItem(SETTINGS_KEY,
                       JSON.stringify(settings));
}
```

To test out this method, add a new button to the product-detail.component.html page in the sample.

```
<button (click)="saveDefaults()">Save Defaults</button>
```

Open the product-detail.component.ts file and add the saveDefaults() method.

```
saveDefaults(): void {
  this.settings.defaultPrice = this.product.price;
  this.settings.defaultUrl = this.product.url;

  this.appSettingsService.saveSettings(this.settings);
}
```

In the saveDefaults() method you take the bound product properties and move them into the appropriate properties in the settings property of the ProductDetailComponent class. You then call the saveSettings method of the appSettingsService class that was injected by Angular into your ProductDetailComponent class.

Retrieve Settings and Store into Local Storage

Open the appsettings.service.ts file again and modify the getSettings() method to look like the following:

```
getSettings(): Observable<AppSettings> {
  let settings = localStorage.getItem(SETTINGS_KEY);

  if (settings) {
    return Observable.of(JSON.parse(settings));
  }
  else {
    return this.http.get(SETTINGS_LOCATION)
      .map(response => {
        let settings = response.json() || {};
        if (settings) {
          this.saveSettings(settings);
        }

        return settings;
      })
      .catch(this.handleErrors);
  }
}
```

The getSettings() method attempts to get the settings object from local storage by passing the SETTINGS_KEY value to the getItem() method. If the variable named *settings* returns a value, then you create an AppSettings object using JSON.parse() and returning an Observable of the AppSettings object.

If nothing is found in local storage, then retrieve the values from the file using the http.get() method. If the values are found in the file, save them into local storage by calling the saveSettings() method. You can see that by writing this method in this manner, after the first time you are always going to be retrieving your settings from local storage. Only the first time do you get the default values from the JSON file.

Handling Exceptions

If the JSON settings file cannot be found, or if some other exception happens, you call the handleErrors() method you see in the catch method. The handleErrrors() method logs errors to the console window, but returns an instance of an AppSettings class. You don’t want your application to fail just because you can’t get some specific global settings. So, return an instance of AppSettings with appropriate defaults set.

```
private handleErrors(error: any): Observable<AppSettings> {
  // Just log error
  switch (error.status) {
    case 404:
      console.error("Can't find file: " + SETTINGS_LOCATION);
      break;
    default:
      console.error(error);
      break;
  }

  // Return default configuration values
  return Observable.of<AppSettings>(new AppSettings());
}
```

Delete Settings

In many applications the user can reset back to “factory defaults”. To accomplish the same thing in your Angular application, you just need to delete the values stored in local storage. If you delete all the values, then the next time the getSettings() method is called, the original values from the JSON file are read. Add a deleteSettings() method to the AppSettingsService class.

```
deleteSettings(): void {
  localStorage.removeItem(SETTINGS_KEY);
}
```

Since the complete AppSettings object is stored within the one key in local storage, call the removeItem() method, passing in the SETTINGS_KEY constant and all of the settings are erased.

To test out this method, add a new button to the product-detail.component.html page in the sample.

```
<button (click)="deleteDefaults()">Delete Defaults</button>
```

Open the product-detail.component.ts file and add the deleteDefaults() method. In this method, call the deleteSettings() method on the appSettingsService object that was injected into this component.

```
deleteDefaults(): void {
  this.appSettingsService.deleteSettings();
}
```

Create Web API for Configuration Settings

Create a new Web API project in Visual Studio named ConfigWebAPI. There are a few steps to get the configuration Web API to work.

- Add a AppSettings Class
- Add a ConfigController Class
- Enable Cross-Origin Resource Sharing
- Convert C# pascal-case to JSON camel-case

AppSettings Class

Right-mouse click on the Models folder and add a new class named AppSettings. Add the following two properties within this class.

```
public class AppSettings
{
  public string DefaultUrl { get; set; }
  public decimal DefaultPrice { get; set; }
}
```

ConfigController Class

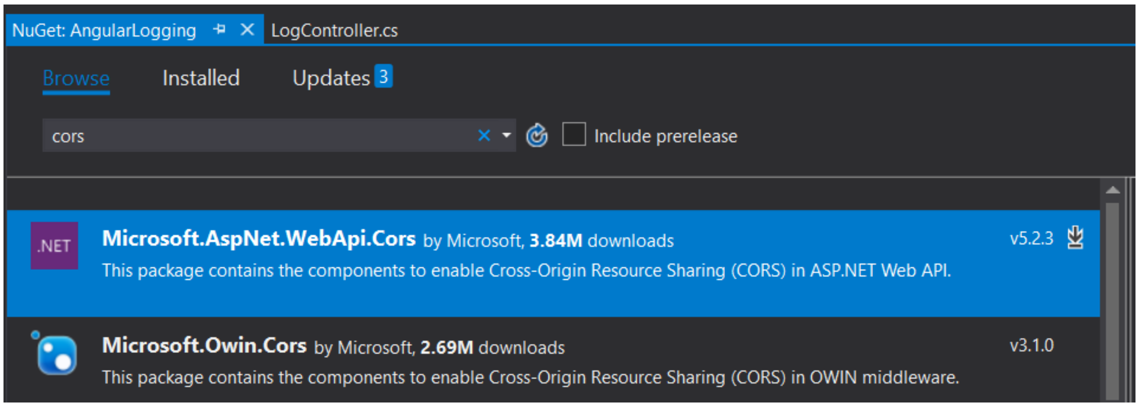
Right-mouse click on the Controllers folder and add a new Web API Controller Class (v2.1). Set the name of this new controller to ConfigController. Wipe out all the methods in this class. Add the following method.

```
public class ConfigController : ApiController
{
  [HttpGet]
  public IHttpActionResult Get()
  {
    IHttpActionResult ret;
    AppSettings settings = new AppSettings();

    // TODO: Write code here to retrieve
    //       settings from XML file or SQL table
    // For now, just hard-code some default values
    settings.DefaultPrice = 25;
    settings.DefaultUrl = "http://www.fairwaytech.com/api";

    ret = Ok(settings);

    return ret;
  }
}
```

The Get() method instantiates a new instance of an AppSettings class. Feel free to write some code to retrieve settings from an XML file or a SQL table. For this blog post, just hard-code a couple of default values that are different from what is in the AppSettings class in Angular. This way you know you are getting the values from the Web API and not the local Angular settings.

Enable Cors

Since you create a new Visual Studio application, this Web API is running in a different domain from your Angular application. In order for your Angular application to call this Web API you must tell the Web API that you are allowing Cross-Origin Resource Sharing (CORS). Right-mouse click on your Web API project and select Manage NuGet Packages… Click on the Browse tab and search for “cors” as shown in Figure 1. Install this package into your project.



Microsoft.AspNet.WebApi.Cors by Microsoft, 3.84M downloads v5.2.3
This package contains the components to enable Cross-Origin Resource Sharing (CORS) in ASP.NET Web API.

Microsoft.Owin.Cors by Microsoft, 2.69M downloads v3.1.0
This package contains the components to enable Cross-Origin Resource Sharing (CORS) in OWIN middleware.

Figure 1: Search for Microsoft CORS

Once you have installed CORS into your project, open the \App_Start\WebApiConfig.cs file. Add the following line of code in the Register() method.

```
public static void Register(HttpConfiguration config)
{
  config.EnableCors();

  ...
}
```

You also need to add an attribute to your ConfigController class. Open the ConfigController.cs file and add the following using statement.

```
using System.Web.Http.Cors;
```

Add the EnableCors() attribute just above your ConfigController class. You can get specific on the origins, headers and methods properties to restrict access to only your Angular application if you want. For the purposes of this blog post, I am just setting them to accept all requests.

```
[EnableCors(origins: "*", headers: "*", methods: "*")]
public class ConfigController : ApiController
{
   ...
}
```

Convert C# Class to JSON

The last thing you need to do is convert the pascal-case C# property names to camel-case property names so they map to the Angular AppSettings class.

This is accomplished by adding the following code to the Application_Start method. Add the following code just below the line: GlobalConfiguration.Configure(WebApiConfig.Register).

```
// Get Global Configuration
HttpConfiguration config =
    GlobalConfiguration.Configuration;

// Handle self-referencing in Entity Framework
config.Formatters.JsonFormatter
  .SerializerSettings.ReferenceLoopHandling =
      Newtonsoft.Json.ReferenceLoopHandling.Ignore;

// Convert to camelCase
var jsonFormatter = config.Formatters
  .OfType<JsonMediaTypeFormatter>()
    .FirstOrDefault();

jsonFormatter.SerializerSettings
  .ContractResolver = new
      CamelCasePropertyNamesContractResolver();
```

Modify AppSettingsService

You are almost ready to try calling your Web API to retrieve your global settings. Retrieve the port number from your Web API project. Open the project properties of your Web API project, click on the Web tab and locate the **Project Url** property. Copy the complete URL from this property to the clipboard.

Switch back to your Angular project. Open the appsettings.service.ts file and locate the constant SETTINGS_LOCATION. Replace the contents of the value with what is in your clipboard. Then, add on “api/config” to the end. Your constant should now look like the following (with a different port number, of course).

```
const SETTINGS_LOCATION = "http://localhost:8314/api/config";
```

Go back to the Web API project and run the project. Go back to the Angular project and run that project. You may need to click on the “Delete Defaults” button and then re-run the Angular project one more time to see the values from the Web API.

Summary

Congratulations! You have now created a configuration system for your Angular applications. This system allows you to retrieve default settings in a JSON file or a Web API call and store those values into local storage. You may modify the values, and delete the values in local storage.

You can get the samples at www.pdsa.com/downloads. Choose “PDSA Blogs” from the Category, then select “Configuration Settings for Angular Applications – Part 2”.