ISSUE 18 | MAY - JUNE 2015

DNCMAGAZINE

www.dotnetcurry.com

Discovering High-Trust SharePoint Apps

Asynchronous Programming in ES6

Using ASP.NET 5 and EF7 with Modern Web Dev Tools

SOFTWARE GARDENING SEEDS Object Oriented Programming

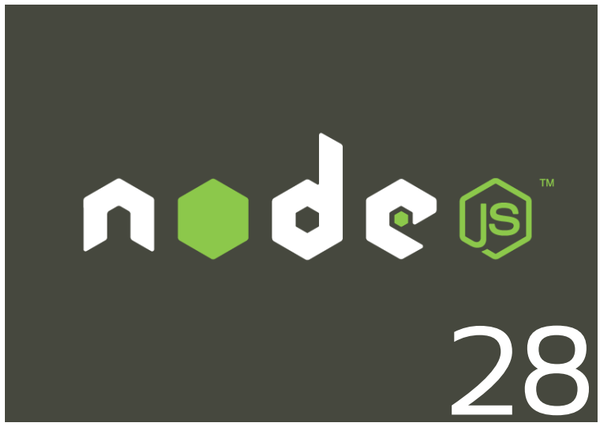
Responsive Image Gallery with Lazyloading

ScreenCapture in Windows Phone 8.1

Building .NET applications with Node.JS

What's New in ASP.NET Web Forms 4.6

CONTENTS



FROM THE EDITOR

Suprotim Agarwal

Editor in Chief

Microsoft's Build and Ignite Conferences (a.k.a TechEd) are around the corner and are one of the best conferences to attend. Craig in one of his Software Gardening columns, had once mentioned that attending conferences is one of the many ways of keeping your skills and knowledge up to date. I couldn't agree more.

When I am not attending an event in person, I make sure to enjoy live streaming at the comfort of my home/office and learn about all the emerging tools and technologies that are shaping development in the Microsoft ecosystem.

One area that we are particularly excited about is Microsoft's renewed commitment to Open source technologies. In this edition, we bring two exclusive articles, one by Ravi which showcases how to use ASP.NET 5 with modern Open Source Web Dev tools like AngularJS & Grunt; and the other by Mahesh which shows how to develop REST APIs using Node and consume it in an ASP.NET MVC application.

In our Software Gardening column, Craig challenges the way many of you think about and use OOP. Vikram shows how to build a simple screen capturing functionality for your Windows Phone, while Ravi shows how Promises and Generators come in handy to design and use asynchronous APIs in ES6.

We welcome Edin Kaplin, Irvin Dominin and Rion Williams as our newest contributors. Edin explains why Sharepoint high trust apps is worth your time, while Irvin shows how to create responsive jQuery Flickr image gallery with Lazyloading. Rion in his awesome article discusses the role of ASP.NET Web Forms within Microsoft's new development ecosystem and debunks the myth that WebForms is dying.

As a developer, there are plenty of opportunities around you to learn and grow. We at DotNetCurry are glad to be part of your learning process!

Do you have any feedback for us? E-mail me your views at suprotimagarwal@dotnetcurry.com

Editor In Chief Suprotim Agarwal
suprotimagarwal@a2zknowledgevisuals.com

Art Director Minal Agarwal
minalagarwal@a2zknowledgevisuals.com

Contributing Writers Craig Berntson, Edin Kapić, Irvin Dominin, Mahesh Sabnis, Ravi Kiran, Rion Williams, Vikram Pendse

Reviewers Mahesh Sabnis, Ravi Kiran, Suprotim Agarwal

Next Edition 1st July 2015
www.dotnetcurry.com/magazine

POWERED BY
a2z | Knowledge Visuals

Discovering High-Trust SharePoint Apps

SharePoint 2013 introduced the concept of SharePoint apps, pushing custom code to remote servers and using the client-side object model. However, not many SharePoint developers are aware that apps come in two distinct "flavours": low-trust and high-trust apps. In this article I will introduce the less known app model: high-trust or S2S apps.

Low-Trust SharePoint Apps

It is probable that you have never heard of low-trust SharePoint apps before. The reason for this is that a vast majority of SharePoint 2013 app examples and documentation is centred on it, so the "low trust" moniker is unnecessary. Let's recap how SharePoint cloud apps (the low-trust ones) work.

In a SharePoint provider-hosted app, there are two separate entities involved: the SharePoint server (either on-premises or Office 365) and the remote server where the app is running. Once we have the app code running outside SharePoint, we need some way of secured communication between the remote server and SharePoint server. This is achieved with the concept of *app identity*.

In addition to users, in SharePoint 2013, apps also have an identity of their own. They are identified by a long GUID-like identifier. Because they have identity, they can be assigned permissions, just like the users. These permissions are the permissions that the app is going to have on the SharePoint site, site collection or tenant where the app is running. Similar to how users are checked against their permissions for every operation in SharePoint, apps are also checked for permission when they use the client-side object model (CSOM) or REST endpoint to communicate with SharePoint.

However, there is one piece missing in this puzzle. How does SharePoint know that the app request is legitimate and not some rogue code that crafts a fake request message? In low-trust apps - the kind of apps you can install from SharePoint store, this is achieved by delegating authentication to a third-party service called *Access Control Service* or ACS.

ACS is a service running in Microsoft cloud and its purpose is to issue and validate identity tokens. When you launch an app from SharePoint, there is a HTTP POST to the app URL. In the parameters that are posted to the server, there is a small encoded payload known as *context token*. Prior to redirecting to the app, SharePoint contacts ACS to get a context token for the app. The context token is a JSON token that contains information about the app identity, the SharePoint farm or tenant ID and the expiration dates. For more details about the context token contents you should refer to this post by Kirk Evans from Microsoft.

The low-trust app gets the context token from the initial request. Immediately, the app calls ACS and exchanges the context token for a new JSON token called *access token*. Access token is used to authenticate the app for making SharePoint API call (CSOM or REST). Access token is the only token SharePoint trusts.

So, why all the fuss with ACS? If you pay attention to all the orchestration, the app is pretty much passive about the authentication. Its only duty is to pass the context token, exchange it for an access token and to attach the access token in its API calls. It can't do much more. The authentication (the user ID and the app ID) is baked in the context token that is passed to the app from SharePoint. This is why the app is called "low-trust", as SharePoint doesn't entrust the app with the authentication. The app can't change the user information nor the permissions it has.

Why High-Trust Apps?

If low-trust apps are so good, why are there high-trust apps? A valid question. As nice and simple the low-trust apps are, they have two main drawbacks.

The first drawback is that they depend on having Internet connectivity. If you are running Office 365 this is no problem, but if you run apps on your SharePoint server in your datacentre, the SharePoint servers have to be able to communicate with ACS servers in the cloud. This involves opening the firewall for bidirectional traffic between your farm and ACS. In many organisations, this is a huge security issue.

The second drawback is that there has to be a trust relationship between ACS and your SharePoint. Again, in Office 365 this is automatically available as both services are run by Microsoft. But, if you have on-premises SharePoint, you have to manually establish this trust by adding your SharePoint as a "principal" in Azure Active Directory (AAD). This involves a custom certificate, changing your SharePoint farm token signing process and it requires some fiddling with PowerShell. The whole nitty-gritty process is well explained in this post by Josh Gavant from Microsoft.

What if you want to leverage the app model but don't want to open your datacentre to Internet or fiddle with trust connections? What if you want the app identity and permissions but for your own code that runs inside your own servers? Well, here is where high-trust apps are coming into place.

High-Trust App Authentication

As opposed to low-trust apps, high-trust apps are entrusted by SharePoint to build their own access tokens. Instead of passively accepting their user and SharePoint context and exchanging them for an access token, they are making access tokens and signing them, just as ACS does. In high-trust apps there is no ACS and there is no context token, just the SharePoint server and the remote server where the app is running. This is why high-trust apps are also known as *server-to-server apps* (S2S).

When we build a high-trust app in Visual Studio, we get a *TokenHelper* class that has all the low-level details of how to build and sign the access token and it uses Windows identity of the current user to fill the token.

But, you might ask yourself how SharePoint can trust the access token coming from a high-trust app. Well it uses a custom SSL certificate for the trust: SharePoint knows the public key of the SSL certificate and can use it to check that the token isn't faked. The app has the private key and uses it to sign the token.

Note that the app is fully responsible for building the access token. It means that the app is putting the user ID, the app ID and the context information such as the SharePoint realm. This is why it's called "high-trust", as SharePoint "trusts" the app to make the tokens in good faith. However, the app can for example make a token that corresponds to another SharePoint user instead of the one opening the app. It can impersonate another app by changing the app ID in the token. What does SharePoint do to prevent fake calls?

The first line of defence is the SSL certificate. Only the certificate that has been registered as a trusted issuer can be used to sign access tokens. It is true that apps can share the same certificate for signing

their tokens, we still need the certificate to be in the list that SharePoint trusts. The second line of defence is the app permissions. Even if the app changes the ID of the user to say an administrator, if the app doesn't have the needed permissions granted on SharePoint, the request is going to be refused with an "Access Denied" error message.

Remember, high-trust doesn't mean full-trust. The app is still limited to what it has been given permission to upon app installation. No more, no less.

Prerequisites for High-Trust Apps

There are several prerequisites for building high-trust apps in our SharePoint farm.

The user profile service has to be running and it has to contain the user profiles for the users that will be using the app. Why is that? Remember that the app is the one that puts the user IDs in the access token. How does SharePoint know which user it should check for permissions? It matches the user ID from the access token against the user profile database. If there is a match, and there is one and only one match, the user is "impersonated" and the API request is made on his or her behalf. The user ID is usually the SID (for Windows users) but it also can be the email (for federated authentication users) or plain username (for custom or FBA users). Steve Peschka has an excellent blog post about the "user rehydration" in high-trust apps.

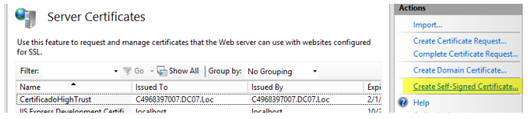
We will need to create a custom SSL certificate for our app. In development we can use a self-issued certificate but in production we need a certificate issued by a trusted authority such as the domain controller or a verified entity such as VeriSign. We will need the PFX certificate with the private key for the app and a CER certificate with the public key for SharePoint. We have to register a "Trusted Root Authority" in SharePoint and also register the "Trusted Issuer" that signs the tokens with the certificate. The trusted issuer is identified by a GUID, and this GUID has to be registered in SharePoint and known to the app at runtime too.

Also, SharePoint Security Token Service (STS) has to be up and running. In addition, the STS application pool identity in IIS needs read permission to the public SSL certificate location in the farm.

Lastly, the web application to which we are installing the app also needs read permission to the public SSL certificate location.

Building "Hello World" High-Trust App

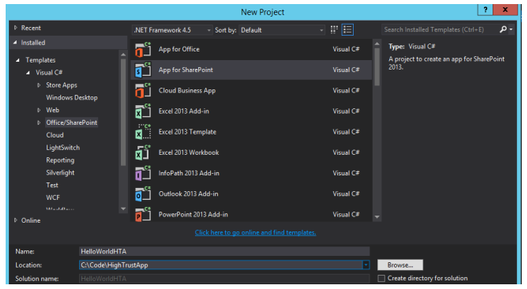
To build a high-trust app, we first need to create a custom certificate. I will be using a self-signed certificate created by IIS in Server Certificates feature. The name of the certificate is 'HighTrustCertificate' and once created, it can be exported in CER or PFX version. For the PFX version, you have to specify a password. In this case, the password will be "password". The certificates are placed in the C:\certificates folder and the SharePoint Secure Token Service and default web application app pool have been given read permissions to the folder.



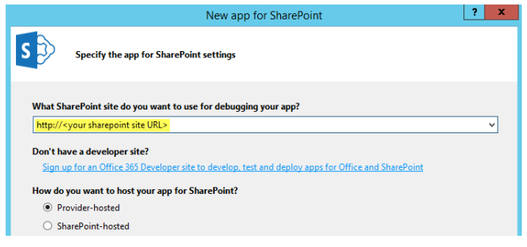
Now we have to register the certificate as a trusted root authority and to register a new issuer based on that certificate. The issuer ID for simplicity sake will be 11111111-1111-1111-1111-111111111111. Open a SharePoint 2013 PowerShell console and execute the following script.

```
# Create root authority
$publicCertPath = "C:\Certificates\
HighTrustCertificate.cer"
$certificate = New-Object System.
Security.Cryptography.X509Certificates.
X509Certificate2($publicCertPath)
New-SPTrustedRootAuthority -Name
"CertificadoHighTrust" -Certificate
$certificate
# Create trusted issuer and IISRESET
$realm = Get-SPAuthenticationRealm
$specificIssuerId = "11111111-1111-1111-
1111-111111111111"
$fullIssuerIdentifier = $specificIssuerId
+ '@' + $realm
New-SPTrustedSecurityTokenIssuer -Name
"HighTrustCertificate" -Certificate
$certificate -RegisteredIssuerName
$fullIssuerIdentifier –IsTrustBroker
iisreset
# Disable HTTPS requirement (for
development)
$serviceConfig = Get-
SPSecurityTokenServiceConfig
$serviceConfig.AllowOAuthOverHttp = $true
$serviceConfig.Update()
```

This script creates the root authority, a trusted issuer and disables the HTTPS for development, so that we don't get SSL errors for self-signed certificates. This must only be done in development environment and never in a production environment. Now we can open Visual Studio and create a new SharePoint 2013 App named 'HelloWorldHTA'.

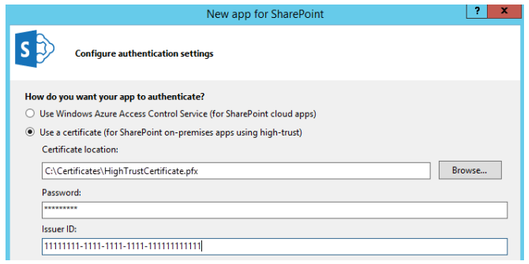


Once the project is created, Visual Studio will ask for more details about the app we are building. First we'll have to specify the SharePoint development URL where the app will be published when it is launched from Visual Studio. Here we'll also specify "Provider-hosted" as the app type.



The next question we'll see is Visual Studio asking us whether to create ASP.NET Web Forms or MVC project for the remote app. In this case for simplicity I will use Web Forms.

The last question to answer is how the app will be authenticated. It asks if we would like to use ACS to create a low-trust app or to use a custom certificate to create a high-trust app. Here we will specify the path to the PFX certificate, providing its password ("password") and trusted issuer ID (11111111-1111-1111-1111-111111111111).



The app itself is very basic. In the Page_Load event handler we can see the following code.

```
protected void Page_Load(object sender,
EventArgs e){
  var spContext =
  SharePointContextProvider.Current.
  GetSharePointContext(Context);

  using (var clientContext = spContext.
  CreateUserClientContextForSPHost()){
    clientContext.Load(clientContext.
    Web, web => web.Title);
    clientContext.ExecuteQuery();
    Response.Write(clientContext.Web.
    Title);
  }
}
```

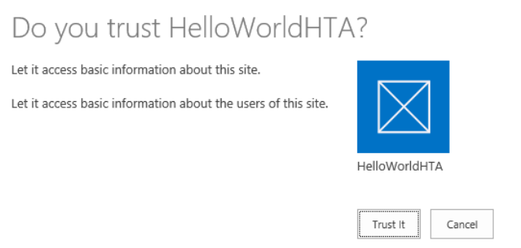
The `SharePointContextProvider` is a helper class contained in the SharePointContext.cs class created alongside `TokenHelper`. It is a static instance of a concrete class `SharePointHighTrustContextProvider` of the abstract `SharePointContextProvider`, the other implementation being `SharePointAcsContextProvider` (for low trust apps). The context provider class is responsible for instantiating a concrete instance of `SharePointContext` class. In our app, this is an instance of a `SharePointHighTrustContext` class.

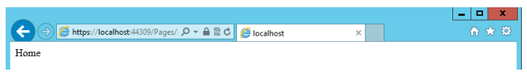
The line in the code that is responsible for instantiating the SharePoint access token when called in the CreateUserClientContextForSPHost method, is this one.

```
public override string
UserAccessTokenForSPHost
{
 get {
  return GetAccessTokenString(ref
  this.userAccessTokenForSPHost,
  ()=>TokenHelper.
  GetS2SAccessTokenWithWindowsIdentity
  (this.SPHostUrl,
  this.LogonUserIdentity));
 }
}
```

Here `GetS2SAccessTokenWithWindowsIdentity` method from `TokenHelper` is called to fill the access token with the Windows user identity. It is also given a SharePoint host web URL to scope the token to a specific SharePoint site, the one where the app is launched from. There is another method called `CreateAppOnlyClientContextForSPHost` that returns app-only access token, without user identity in it. This is used in the app when the current user has less permissions than the app, so that the app can do things in SharePoint that the current user cannot.

Let's run the app from Visual Studio by pressing F5. It will launch the app, ask for permissions in SharePoint and after that it will show the current site title ("Home" in our example).





Tricks and Caveats

As high-trust apps have a simpler mechanism for authentication, we can leverage it to our advantage. The low-trust apps are launched by SharePoint, and SharePoint injects the context token in the app request. We can launch our high-trust apps outside SharePoint, as long as we change the TokenHelper slightly to inject the required parameters (SPHostUrl and similar).

We don't have to stop here. We can provision non-web applications as high-trust apps. In essence, any process that can write JSON tokens and sign them with a SSL certificate can be exposed as high-trust apps. You could write "timer jobs" as console applications, for example. The app model allows you for simpler authentication and app permissions, as opposed to having hard-coded credentials or using Windows authentication only. However, high-trust apps are highly dependent on the validity of its two basic pillars: user profile and signing certificate. Any missing information such as incomplete profiles, multiple profiles for any single user or untrusted certificate will give the dreaded "401 Access Denied" error. Luckily for us, the ULS diagnostics logs are particularly helpful and loaded with details for OAuth app authorization.

Summary

High-trust apps (HTA) are on-premises alternative for cloud SharePoint apps. They leverage the same overall mechanism but their authentication process is simpler. This allows for using the recommended app model for your on-premises SharePoint development but still keeping all the pieces in your datacentre.

If you are still running your own SharePoint, high-trust apps may be worth a try ■

Download the entire source code from GitHub at bit.ly/dncm18-sharepoint-hightrust

About the Author

Edin Kapić is a SharePoint Server MVP since 2013 and works as a SharePoint Architect in Barcelona, Spain. He speaks about SharePoint and Office 365 regularly and maintains a blog www.edinkapic.com and Twitter @ekapic.

Using ASP.NET 5 and EF7 with Modern Web Dev Tools like AngularJS and Grunt

In this article, we will see how to build a simple Virtual Tutorial Tracker application using ASP.NET 5 and AngularJS. The list of technologies used in this article are:

- SQL Server
- ASP.NET 5
- Entity Framework
- AngularJS
- Grunt

The web development ecosystem is changing rapidly. The good thing is, existing frameworks are catching up on the latest and greatest trends and making them first class citizens in their frameworks. In recent times, the Microsoft's web development stack, ASP.NET, has started making several changes to itself including:

- Moving from XML to JSON for storing configurations
- Out-of-the-box Dependency Injection
- Leveraging Node.js environment for packaging client side dependencies
- Reducing the start time of application by removing all dependencies
- Providing an easier way to build and plug middleware
- Unifying ASP.NET MVC, Web API and Web Pages, into a single programming model called MVC 6
- Build and run cross-platform web apps on OS X and Linux with the Mono runtime
- Ability to host on IIS or self-host in your own process

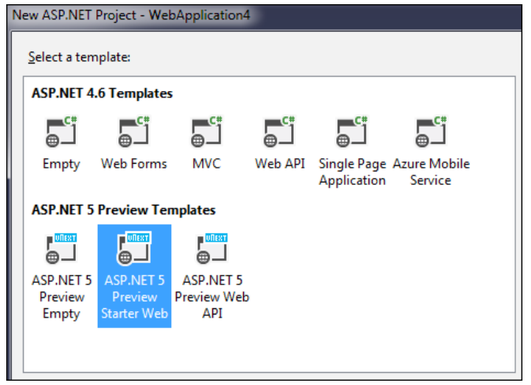
Because of these changes, ASP.NET is becoming an easier ecosystem for building and running rich front-end based applications on Windows as well OS X and Linux. Because of availability of task runners like Grunt and Gulp and the ability to use the most frequently updated package sources like NPM and Bower, we need not wait for the package to be available or, updated on NuGet. So it is a win-win situation for framework and the developers using the framework.

In this article, we will see how to build a simple Virtual Tutorial Tracker application using ASP.NET 5 and Angular JS. The list of technologies used in this article are:

- SQL Server
- ASP.NET 5
- Entity Framework
- AngularJS
- Grunt

Setting up the Project

For this article, I have used Visual Studio 2015 CTP. You can also use the Free Visual Studio 2013 Community Edition to build the sample. Open Visual Studio 2013 or, 2015 and select File -> New Project. From the new Project dialog box, choose the option of ASP.NET Web Application and from the ASP.NET project dialog, choose ASP.NET Preview Starter Web and hit OK.



This template creates an ASP.NET 5 application with most of the required hooks. It contains a Bootstrap-based ASP.NET application with Identity and a *DbContext* to create database to store Identity information. It uses Bower to bring in jQuery and bootstrap and includes a *gruntfile* with a few tasks. The *package.json* file contains the NPM tasks required for *Grunt*. We will add more content to these files.

Adding Bower components and Grunt tasks

As already mentioned, we will be using AngularJS in this application. Let's add an entry of the required Angular packages to the *bower.json* file. Open the *bower.json* file and modify the list of dependencies to the following:

```
"dependencies": {
  "bootstrap": "3.0.0",
  "jquery": "1.10.2",
  "jquery-validation": "1.11.1",
  "jquery-validation-unobtrusive":
  "3.2.2",
  "hammer.js": "2.0.4",
  "bootstrap-touch-carousel": "0.8.0",
  "angular": "1.3.15",
  "angular-route": "1.3.15"
}
```

We need to add a couple of grunt tasks to copy the static files to their target locations and minify them. Let's add these packages to the *package.json* file. Add the following packages to the *devDependencies* section in *package.json* file:

```
"devDependencies": {
  "grunt": "0.4.5",
  "grunt-bower-task": "0.4.0",
  "grunt-bowercopy": "^1.2.0",
  "grunt-contrib-copy": "^0.8.0",
  "grunt-contrib-concat": "^0.5.1",
  "grunt-html2js": "^0.3.1"
}
```

Add a new folder to the project and change the name to *App*. Add two folders named *Scripts* and *Templates* to this folder. We will add our JavaScript and HTML template files to these respective folders. The ASP.NET Starter application contains a folder named wwwroot; this folder is used by the project to store the static files that have to be deployed. So we need to copy all required libraries and custom scripts into this folder. Which means, this is the target folder for the Grunt tasks.

Open the file gruntfile.js and replace the code in it with the following:

```
module.exports = function (grunt) {
 grunt.initConfig({
```

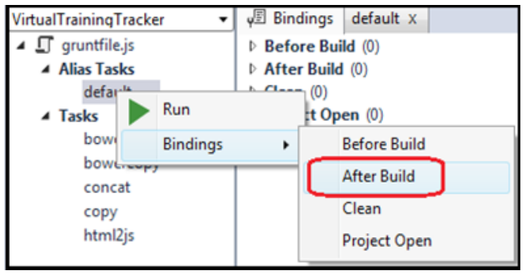
```
bowercopy: {
 options: {
  runBower: true,
  destPrefix: 'wwwroot/lib'
 },
 libs: {
  files: {
   'angular': 'angular',
   'angular-route': 'angular-route',
   'jquery': 'jquery',
   'bootstrap': 'bootstrap/dist/css'
  }
 }
},
copy: {
 main: {
  expand: true,
  flatten: true,
  filter: 'isFile',
  src: ['App/Scripts/*.js'],
  dest: 'wwwroot/dist/'
 }
},
concat: {
 options: {
  separator: ';'
 },
 dist: {
  src: ['wwwroot/dist/*.js'],
  dest: 'wwwroot/dist/appCombined.js'
 }
},
html2js: {
 options: {
  module: 'virtualTrainingTemplates'
 },
 main: {
  src: ['App/Templates/*.html'],
  dest: 'wwwroot/dist/templates.js'
 }
}
});

/* This command registers the default
task which will install bower packages
into wwwroot/lib */
grunt.registerTask("default",
["bowercopy:libs", "html2js",
"copy:main", "concat"]);

/* The following line loads the grunt
plugins.
This line needs to be at the end of
this  file. */
grunt.loadNpmTasks("grunt-bower-task");
grunt.loadNpmTasks("grunt-bowercopy");
grunt.loadNpmTasks("grunt-contrib-
copy");
grunt.loadNpmTasks("grunt-contrib-
 concat");
 grunt.loadNpmTasks("grunt-html2js");
};
```

There are 4 tasks configured in this file:

- **bowercopy:** To install the front-end libraries and copy them into the wwwroot folder
- **copy:** To copy custom scripts into a folder called *dist* inside the wwwroot folder
- **html2js:** To convert HTML templates to AngularJS modules
- **concat:** To concatenate all JavaScript files into a single file
- **default:** It is a custom alias task that runs the above tasks in an order

Visual Studio has built-in tools to install the NPM packages and run grunt tasks. Check my article on Using Grunt, Gulp and Bower in Visual Studio 2013 and 2015 to learn more about it.

Open the Task Runner Explorer from View -> Other Windows -> Task Runner Explorer menu option. The widget reads gruntfile and lists all tasks to run them easily. We can set the option to run the Grunt tasks on every build. To do so, you need to right click on the task and choose bindings and then your favorite option from the menu. Following screenshot sets grunt to run after every build:



Setting up Database Using Entity Framework

The project already includes a beta version of Entity Framework 7 and uses it to create a database to hold the Identity tables. The *DbContext* class of the project is *ApplicationDbContext* and it can be found in the IdentityModels.cs file inside the Models folder of the project. We will add the *DbSet* properties of the new tables to the same class. This context is already added to the services in *ConfigureServices* method of the *Startup* class, so we don't need to register the *DbContext* class.

In the Models folder, add a new file and name it Course.cs and add the following code to it:

```
public class Course
{
  public int CourseID { get; set; }
  public string Title { get; set; }
  public int Duration { get; set; }
  public string Author { get; set; }
  public virtual
ICollection<CourseTracker> Trackers {
get; set; }
}
```

Add another file and name it CourseTracker.cs and add the following code to it:

```
public class CourseTracker
{
  public int CourseTrackerID { get; set;
}
  public int CourseID { get; set; }
  public string ApplicationUserId { get;
  set; }
  public bool IsCompleted { get; set; }
  public virtual Course Course { get;
  set; }
  public virtual ApplicationUser
ApplicationUser { get; set; }
}
```

Now we need to add the *DbSet* entries to the *ApplicationDbContext* class.

```
public DbSet<Course> Courses { get; set;
}
public DbSet<CourseTracker>
CourseTrackers { get; set; }
```

The project comes with pre-configured EF Code First migrations. But as we added new classes to the DbContext, let's create a new migration. Delete the existing Migrations folder. Open Package Manager Console and move to the folder where the project.json file is located. In this folder, run the following commands in the command prompt:

```
> k ef migration add init
> k ef migration apply
```

Once these commands are executed, you would see the database with the tables to store Identity data and the two tables corresponding to the entities we added.

As of now, Entity Framework 7 doesn't support seeding data. The only way to seed data is using the *Configure* method of the *Startup* class. This method is executed at the beginning of the application, so we can insert the initial data in this method.

The arguments passed to this method are injected by the default Dependency Injection system that comes with ASP.NET 5. So we can specify a parameter for the *DbContext* object and it will be available to us. Following snippet adds some rows to the *Courses* table.

```
public void Configure(IApplicationBuilder
app, IHostingEnvironment env,
ILoggerFactory loggerfactory,
ApplicationDbContext context)
{
 //Rest of logic inside the method

 if (context.Courses.FirstOrDefault()
 == null)
 {
  context.Courses.Add(new Course() {
   Title = "Basics of ASP.NET 5", Author
   = "Mahesh", Duration = 4 });
   context.Courses.Add(new Course() {
   Title = "Getting started with Grunt",
   Author = "Ravi", Duration = 3 });
   context.Courses.Add(new Course() {
   Title = "What's new in Angular 1.4",
   Author = "Ravi", Duration = 4 });
   context.Courses.Add(new Course() {
   Title = "Imagining JavaScript in
   ES6", Author = "Suprotim", Duration =
   4 });
   context.Courses.Add(new Course() {
```

```
    Title = "C# Best Practices", Author =
    "Craig", Duration = 3 });

    context.SaveChanges();
   }
}
```

If you run the application now and then check the database, you will see the new data inserted into the database.

Designing REST APIs using Web API

Unlike previous versions of ASP.NET, the base controllers for both MVC controller and Web API controller are now the same. But we should take the responsibility of not combining both behaviors in one controller. It is always advised to keep the controller serving views, and the controller serving JSON data; in separate classes.

The *Configure* method of *Startup* class adds MVC to the services, the same service can serve Web API as well. Default settings of the Web API return JSON response in Pascal case notation. As we prefer Camel-case notation in JavaScript, let's change the output formatter to serialize JSON data into Camel-case notation. Following code does this:

```
services.AddMvc().
Configure<MvcOptions>(options =>
{
 int position = options.
 OutputFormatters.FindIndex(f =>
 f.Instance is JsonOutputFormatter);

 var settings = new
 JsonSerializerSettings()
 {
  ContractResolver = new
  CamelCasePropertyNamesContractResolver
  ()
 };
 var formatter = new
 JsonOutputFormatter();
 formatter.SerializerSettings =
 settings;

 options.OutputFormatters.
 Insert(position, formatter);
});
```

REST API for Courses

Let's write a REST API to serve data in the Courses table. The controller will have a single *GET* endpoint to serve list of courses. I want this method to respond based on user's identity. Following are the two kinds of responses:

1. If the user is logged in, return courses along with a course track object

2. If the request is by an anonymous user, return data in the courses table with the course track object set to *null*

As the schema of the data to be returned from this API doesn't match with the *Course* class, we need a Data Transfer Object (DTO). Following is the DTO class:

```
public class CourseDTO
{
  public string Author { get; internal
  set; }
  public int CourseId { get; internal
  set; }
  public int Duration { get; internal
  set; }
  public string Title { get; internal
  set; }
  public CourseTracker Tracked { get;
  internal set; }
}
```

As the data conversion needs some logic, it is better to have a repository class for data conversion. Add a new class to the Models folder and name it *Courserepository*. Following is the logic inside this class:

```
public class CourseRepository
{
 ApplicationDbContext _context;

 public
 CourseRepository(ApplicationDbContext
 context)
 {
  this._context = context;
 }

 public IEnumerable<CourseDTO>
 GetCourses(ClaimsIdentity identity)
{
 if (identity.IsAuthenticated)
 {
  var userId = identity.Claims.
  FirstOrDefault().Value;

  var userTrackers =
  _context.CourseTrackers.Where(ct =>
  ct.ApplicationUserId == userId);

  var withTrackStatus = this._context.
  Courses.Select(c => new CourseDTO()
  {
   Tracked = userTrackers.
   FirstOrDefault(t => t.CourseID ==
   c.CourseID),
   CourseId = c.CourseID,
   Author = c.Author,
   Title = c.Title,
   Duration = c.Duration
  });

  return withTrackStatus.ToList();
 }

 return this._context.Courses.Select(c
 => new CourseDTO()
 {
  Tracked = null,
   CourseId = c.CourseID,
   Author = c.Author,
   Title = c.Title,
   Duration = c.Duration
  }).ToList();
 }
}
```

To be able to inject the repository class to the controller, we need to add one statement to the *ConfigureService* method of *Startup* class.

```
services.
AddTransient(typeof(CourseRepository),
typeof(CourseRepository));
```

Now add a new MVC Controller to the Controllers folder and name it *CourseController*. Write the following code in it:

```
[Route("api/[controller]")]
public class CoursesController :
Controller {
  CourseRepository _repository;

  public CoursesController
  (CourseRepository repository)
  {
    this._repository = repository;
  }

  [HttpGet]
  public IEnumerable<CourseDTO>
  GetAllCourses()
  {
    return _repository.
    GetCourses((ClaimsIdentity)User.
    Identity);
  }
}
```

Run the application, open a browser and change the URL to http://localhost:<port-no>/api/courses. You will see the list of courses in JSON format in the browser.

REST API for CourseTrackers

The course trackers API will provide APIs to list, add or update tracker based on the currently logged in user. So all APIs in this controller would be restricted and hence we need to decorate them with the *Authorize* attribute.

Let's create a repository class to perform the required operations on CourseTrackers table. The repository class would contain the following three methods:

- A method to get details of trackers by user. It accepts user ID of the logged in user and returns a list of Data Transfer Object containing tracker ID, course title, author name and if the tracker is completed

- A method to add a new track

- A method to mark an existing track completed

Let's create the DTO class. As described, it will have four properties that will be shown on the screen. Add a new class to the Models folder and name it *CourseTrackerDTO*. Following is the code in this class:

```
public class CourseTrackerDTO
{
  public int CourseTrackerId { get; set;
}
  public string CourseTitle { get; set;
```

```
}
  public string Author { get; set; }
  public bool IsCompleted { get; set; }
}
```

Add another class to the Models folder and name it CourseTrackerRepository. Replace code in this file with the following:

```
public class CourseTrackerRepository
{
 ApplicationDbContext _context;

 public CourseTrackerRepository
 (ApplicationDbContextcontext)
 {
  this._context = context;
 }
 public IEnumerable<CourseTrackerDTO>
 GetCourseTrackers(string userId)
 {
  var userTrackers =
  context.CourseTrackers.Where(ct =>
  ct.ApplicationUserId == userId);
  var trackerDtoList =
  new List<CourseTrackerDTO>();

  foreach (var tracker in userTrackers)
  {
   var course = _context.Courses.
   FirstOrDefault(c =>
   c.CourseID == tracker.CourseID);

   trackerDtoList.Add(new
   CourseTrackerDTO()
   {
    CourseTrackerId = tracker.
    CourseTrackerID,
    CourseTitle = course.Title,
    Author = course.Author,
    IsCompleted = tracker.IsCompleted
   });
  }

  return trackerDtoList;
}

public CourseTracker UpdateCourseTracker
(CourseTracker updatedTracker)
{
 var tracker = _context.CourseTrackers.
 FirstOrDefault(ct => ct.CourseTrackerID
 == updatedTracker.CourseTrackerID);

 if (tracker != null)
 {
  tracker.IsCompleted =
  updatedTracker.IsCompleted;
  _context.Entry(tracker).SetState
  (Microsoft.Data.Entity.EntityState.
  Modified);

  _context.SaveChanges();
  return tracker;
 }

 return null;
}

public CourseTracker
AddTracker(CourseTracker tracker)
{
 tracker.IsCompleted = false;

 if (_context.CourseTrackers.
 FirstOrDefault(ct => ct.CourseID ==
 tracker.CourseID
 && tracker.ApplicationUserId
 == tracker.ApplicationUserId) ==
 null)
 {
  var addedTracker = _context.
  Add(tracker);
  _context.SaveChanges();
  return tracker;
  }
  return null;
 }
}
```

The controller has to simply call the methods of the repository class. Before we do that, we need to register the repository class to the IoC container of the environment. Add the following statement to the *ConfigureServices* method of the Startup class: services.

```
AddTransient(typeof
(CourseTrackerRepository),
typeof(CourseTrackerRepository));
```

Now we can inject this repository inside the controller and use it. Add a new MVC Controller to the Controllers folder and name it *CourseTrackersController*. Following is the controller with the Get method. Add and update methods are straight forward, you can refer to them from the sample code.

```
[Route("api/[controller]")]
public class CourseTrackersController :
Controller
{
 CourseTrackerRepository _repository;

 public CourseTrackersController
 (CourseTrackerRepository repository)
 {
  this._repository = repository;
 }

 [HttpGet]
 [Authorize]
 public IEnumerable<CourseTrackerDTO>
 GetCoursesOfUser()
 {
  var userId = ((ClaimsIdentity)
  User.Identity).Claims.
  FirstOrDefault().Value;
  var trackers = _repository.
  GetCourseTrackers(userId);
  return trackers;
 }
}
```

Building Views and Binding Data to them using AngularJS

Let's start modifying the index.cshtml page present in Home folder under Views and make it behave as a Single Page Application. Delete all of the existing mark-up inside the Index.cshtml file. Before we start adding AngularJS content to this file, let's add the required scripts to the _Layout.cshtml page. We need to add the scripts files of Angular.js, Angular routes and the concatenated script file generated from the Grunt process. We also need to add a section that will be used in the index page to set a global variable with user's ID. Following are the script tags:

```
<script src="~/lib/angular/angular.js">
</script>
<script src="~/lib/angular-route/
angular-route.js"></script>

@RenderSection("userScript", required:
false)
<script src="~/dist/appCombined.js">
</script>
```

At the end of the index.cshtml page, let's add the userScript section to store the user ID in a global variable. We will use this variable later in the Angular code. Following is the script:

```
@section userScript{
<script>
 window.userName = "@ViewBag.UserName";
</script>
}
```

We need to set value of the UserName to ViewBag object inside the Home controller. To do so, modify the Index action method as follows:

```
public IActionResult Index()
{
  if (User.Identity.IsAuthenticated)
  {
    ViewBag.UserName = User.Identity.
    GetUserName();
  }
return View();
}
```

Defining Views and Angular Routes

The application will have just two pages: One to list all of the courses and the second view to list all tracked courses. To define a module and routes, add a new JavaScript file to the Scripts folder under App folder and name it app.js. Add following code to it:

```
(function(){
 angular.module('virtualTrainingApp',
 ['ngRoute',
 'virtualTrainingTemplates'])
 .config(['$routeProvider',
 function($routeProvider){
 //TODO: Configure routes
 $routeProvider.when('/courseList', {
  templateUrl: 'App/Templates/
  courseList.html',
  controller: 'courseListCtrl',
  controllerAs: 'vm'
 })
 .when('/courseTracks', {
  templateUrl: 'App/Templates/
  courseTracks.html',
  controller: 'courseTrackCtrl',
  controllerAs: 'vm'
 })
 .when('/courseTracks2', {
 templateUrl: 'App/Templates/
 courseTracks.html',
 controller: 'courseTrackCtrl',
 controllerAs: 'vm'
```

```
 })

 .otherwise({redirectTo: '/courseList'});
 }]);
}());
```

I am using "controller as" syntax in the views. This feature is added in Angular 1.2 and it is the recommended way of hooking controllers with views these days. You can read my blog post on this topic to get an overview of it.

The HTML template inside the file index.html has to load this module, have a navigation bar and it has to define a placeholder, where the views will be loaded. The navigation bar has to be shown to only logged-in users, as the anonymous users can't set the courses in watching state. Following is the mark-up inside the view:

```
<div class="row" ng-
app="virtualTrainingApp">
 <div class="col-md-2" ng-
 controller="navigationCtrl as nav" ng-
 show="nav.isAuthenticated">
  <ul class="nav nav-list">
   <li ng-repeat="navigationItem in
    nav.navigationList">
    <a ng-href="{{navigationItem.
    url}}">{{navigationItem.text}}
   </a></li>
  </ul>
 </div>
<div class="col-md-10">
  <div ng-view></div>
  </div>
</div>
```

Add a new HTML file to the templates folder and name it courseList.html. This view performs the following functionality:

- Shows the list of courses available
- Allows a logged-in user to start watching a course
- If a logged-in user has already started watching a course or has finished watching, the corresponding status has to be shown
- The option of starting to watch a course or, the status of course completion shouldn't be shown to an anonymous user

Following is the mark-up on courseList.html:

```
<div class="col-md-3 rowmargin"
style="height: 150px; margin-top: 5px;
margin-bottom: 15px;" ng-repeat="course
in vm.courses">
 <div class="thumbnail">
  <div class="caption"><strong>{{course.
  title}}</strong></div>

  <div style="text-align:center;">by
  {{course.author}}</div>
  <br />

  <span class="right" style="margin-
  right: 9px;">{{course.duration}} hr</
  span>

  <div class="left" ng-show="vm.
  isAuthenticated">
  <button class="left btn"
  style="padding: 3px;" ng-click="vm.
  addTrack(course.courseId)" ng-
  show="!course.tracked">Watch Course</
  button>

  <span class="left" ng-show="course.
  tracked && !course.tracked.
  isCompleted">Watching...</span>
  <span class="left" ng-show="course.
  tracked && course.tracked.
  isCompleted">Watched</span>
  </div>
 </div>
</div>
```

The second and last view of the application is, the courseTrack view. Add a new HTML file to the Templates folder and name it courseTrackers.html. This view performs the following functionality:

- Lists the courses in progress and completed courses in a table
- The user can mark the course as completed using this screen

Following is the mark-up of the page:

```
<div>
  <h3>Status of Courses Tracked</h3>
  <table class="table">
    <tr>
      <th>Course Title</th>
      <th>Author</th>
      <th>Completed?</th>
      </tr>
    <tr ng-repeat="track in vm.courses">
      <td>{{track.courseTitle}}</td>
      <td>{{track.author}}</td>
      <td>
        <span ng-show="track.
        isCompleted">Completed</span>
        <button class="btn btn-
        link" style="padding: 0;"
        ng-show="!track.isCompleted"
        ng-click="vm.completeCourse
        (track.courseTrackerId)">Mark
        Completed</button>
      </td>
    </tr>
  </table>
</div>
```

Building Services

The application needs two services: One to store logged-in status of the user and the other is to call the HTTP APIs.

The service to store User information looks for the global variable that we added to the index.cshtml page and then stores the user name in a variable and also stores a flag indicating if a user is currently logged-in. It exposes two functions that can be used to get this information from the service.

Add a new JavaScript file to the Scripts folder and name it userInfo.js. Following is the code to be placed in the service:

```
(function (module) {
  module.service('userInfoSvc',
  ['$window', function ($window) {
    var isUserAuthenticated = false,
    userName;

    if ($window.userName) {
      isUserAuthenticated = true;
      userName = $window.userName;
    }

    this.getUserName = function () {
      return userName;
    };

    this.isAuthenticated = function () {
      return isUserAuthenticated;
    };
  }]);
}(angular.
module('virtualTrainingApp')));
```

The second service abstracts the HTTP calls. The controllers would be using this service to get data from the Web API services that we created earlier. Logic of the service is straight-forward, as there are no decisions to be taken in it.

Add a new JavaScript file to the Scripts folder and name it dataSvc.js. Add the following code to it:

```
(function (module) {
 module.factory('coursesDataSvc',
 ['$http', function ($http) {
  function getAllCourses() {
   return $http.get('/api/courses').
   then(function (result) {
    return result.data;
   }, function (error) {
       return error;
   });
  }

  function getCourseTracks() {
   return $http.get('/api/
   courseTrackers').then
   (function (result) {
    return result.data;
   }, function (error) {
       return error;
   });
  }

  function addCourseToTrack(newCourse) {
   return $http.post('/api/
   courseTrackers', {
    courseId: newCourse.courseId
    }).then(function (result) {
     return result;
    }, function (error) {
        return error;
    });
   };

   function modifyCourseTrack
   (courseTrackDetails) {
    return $http.put('/api/
    courseTrackers/' +
    courseTrackDetails.courseTrackerId,
    {
     courseTrackerId:
     courseTrackDetails.courseTrackerId,
     isCompleted: true
    }).then(function (result) {
```

```
    return result;
   }, function (error) {
      return error;
   });
  };

  return {
   getAllCourses: getAllCourses,
   getCourseTracks: getCourseTracks,
   addCourseToTrack: addCourseToTrack,
   modifyCourseTrack: modifyCourseTrack
  };
 }]);
}(angular.
module('virtualTrainingApp')));
```

Creating Controllers

The application needs three controllers, one for each of the views and the other one for the navigation menu.

As already stated, the navigation menu will be visible only to the logged-in users and it will have two menu options. The controller does the following to control behavior of the navigation menu:

- Gets user information from the userInfoSvc service and sets a Boolean flag to show or, hide the navigation menu

- Stores navigation links and text to be displayed on the links in a collection

This controller contains the following code:

```
(function (module) {
 module.controller('navigationCtrl',
 ['userInfoSvc', function (userInfoSvc){
  var nav = this;
  nav.navigationList = [{ url: '#/
  courseList', text: 'Course List' },
  { url: '#/courseTracks', text: 'Course
  Track' }];
  nav.isAuthenticated = userInfoSvc.
  isAuthenticated();
 }]);
}(angular.
module('virtualTrainingApp')));
```

As you see, I didn't inject *$scope* to the above controller and the data to be exposed to the view has been added to the instance of the controller.

Angular internally adds the instance of the controller to the $scope that gets created behind the scenes and hence this object is available to the view.

The controller for the courseList view does the following:

- Fetches the list of courses and stores them in the controller's instance

- Checks if the user is authenticated and if yes, it provides a method to add a course to the tracker of the user

This controller contains the following code:

```
(function (module) {
  module.controller('courseListCtrl',
  ['userInfoSvc', 'coursesDataSvc',
  function (userInfoSvc, coursesDataSvc)
  {
    var vm = this;

    function init() {
    coursesDataSvc.getAllCourses().
    then(function (result) {
      vm.courses = result.data;
    }, function (error) {
        console.log(error);
    });
  }

  vm.isAuthenticated = userInfoSvc.
  isAuthenticated();

  if (vm.isAuthenticated) {
    vm.addTrack = function (courseId) {
      coursesDataSvc.addCourseToTrack({
        courseId: courseId
      }).then(function (result) {
        init();
      }, function (error) {
          console.log(error);
      });
    };
  }

  init();
 }]);
}(angular.
module('virtualTrainingApp')));
```

The third and last controller of the application is, courseTrackCtrl. This controller does the following:

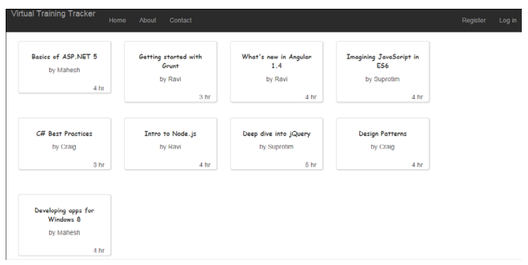
- Gets the list of courses that the user is currently watching or, already watched

- Provides a method using which the user can mark the course as completed

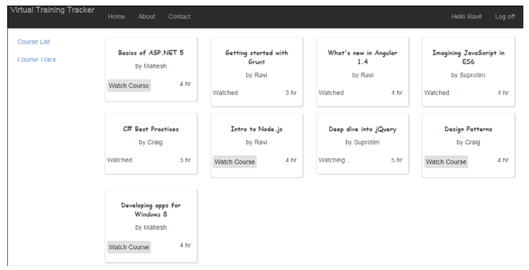
The courseTrackCtrl controller contains the following code:

```
(function (module) {
 module.controller('courseTrackCtrl',
 ['userInfoSvc', 'coursesDataSvc',
 function (userInfoSvc, coursesDataSvc)
 {
  var vm = this;

  function init() {
   coursesDataSvc.getCourseTracks().
   then(function (result) {
    vm.courses = result.data;
   });
  }

  vm.completeCourse = function
  (courseTrackerId) {
   coursesDataSvc.modifyCourseTrack({
    courseTrackerId: courseTrackerId })
    .then(function (result) {
     init();
    }, function (error) {
       console.log(error);
    });
   };

   init();
 }]);
}(angular.
module('virtualTrainingApp')));
```

Now that we have the entire application ready, run the application. You should be able to see a screen similar to the one shown below:



Now register and create a new user. If you have already created a user, login using the credentials and you will see a slightly different look on the screen. Following is a screenshot of the page after logging in:



Conclusion

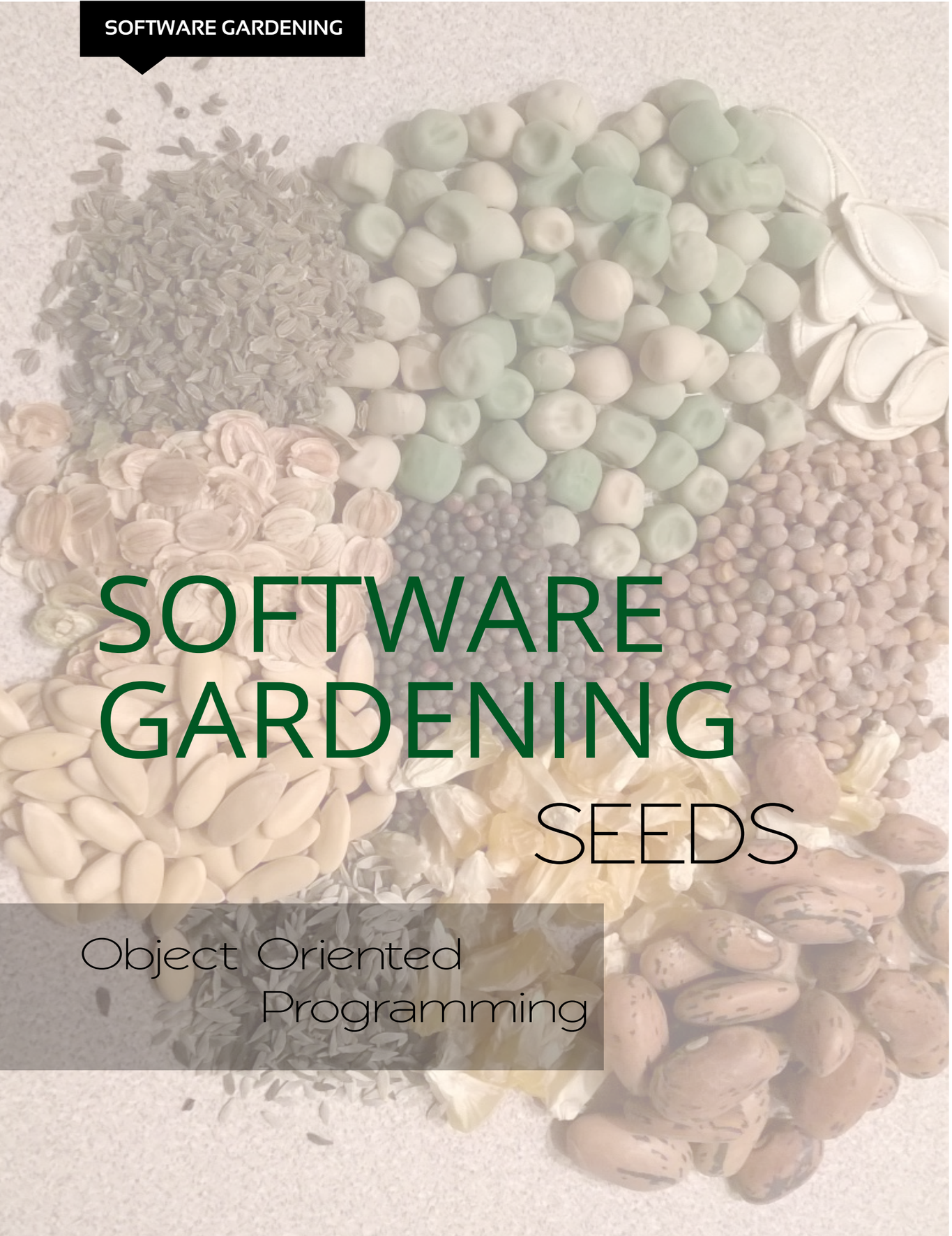
I think, by now you got a good idea of the way ASP.NET 5 is designed and how it welcomes usage of some of the coolest modern tools to make web development enjoyable on the platform. The new architecture of the framework makes it easier to build applications on it and it also seems more open for developing rich applications using client side MVC frameworks like AngularJS.

ASP.NET 5 is still under development and ASP.NET team is putting together a lot of work to make the framework even better by the time it is ready for market. Let's wait and see what more Microsoft has to pour into our buckets ■

Download the entire source code from GitHub at bit.ly/dncm18-aspnet5-angular

About the Author

Ravi Kiran is a developer working on Microsoft Technologies. These days, he spends his time on the front-end JavaScript framework Angular JS and server frameworks like ASP.NET Web API and SignalR. He actively writes what he learns on his blog at sravi-kiran.blogspot.com. He is a DZone MVB. You can follow him on twitter at @sravi_kiran



SOFTWARE GARDENING

SEEDS

Object Oriented Programming

How are you going to grow anything in your garden without seeds? You may argue that you can buy young plants and replant them. This is often done with tomatoes, flowers, and trees. But at some point, those young plants too began as seeds. You also need to plant at the proper time. In the early spring, when the frost can cover the ground in the morning, you should only plant seeds that can withstand the colder mornings. The depth of the hole is important. The seed needs enough moisture and warmth from the sun that it can germinate and take root, yet not be so shallow that cold mornings will kill it completely. Finally, you need to take into account things like the gestation and growing time. Some plants grow quickly and are ready to harvest in early summer, others not until fall.

It is much the same with software development. Do we pick a language that is more dynamic like JavaScript or PHP? What about functional languages such as F#. How about C++, Java, .Net, or other Object Oriented languages? Is there a right time to pick a language for a particular job or do we just go with the language we know best? Is one language better suited for a particular purpose vs. a different language? I'm going to leave some of these questions for you to answer and since this is a .Net magazine, focus on one particular topic: Object Oriented .Net languages. For most of us, even those doing web development with ASP.Net; C# or VB are the seeds that we begin with.

This issue, I begin talking about Object Oriented Programming (OOP), starting with the basics. In the next few issues, we'll get into other OOP topics. Beware that I may challenge the way many of you think about and use OOP.

The first question to ask is what is an *object*? The simple answer is it is a software construct (class) that represents or models something, say a person. More correctly, a class is a definition of the object. The object is the instantiated class. The object contains data (attributes or properties) about the thing being modeled. It also has procedures (methods) that give behavior or functionality to what is being modeled. Coming back to the person example, properties could be things like name, eye color, hair color, height, or weight. Methods could be walk, talk, eat, or snore.

Three pillars of Object Oriented Programming

Of all the topics in OOP, three stand out as the topics that support everything else. These are inheritance, polymorphism, and encapsulation. Let's look at each of these.

Inheritance

Each of us has inherited traits from our parents. Eye color, hair color, and medical conditions are attributes we inherit. We also can inherit behavior such as our sense of humor, extrovert or introvert, and others. Classes can have a parent or parents and inherit attributes or behavior.

A class can have a single class and inherit all the behavior and attributes from the parent class. This is called *implementation inheritance*. In .Net, the child class can change or override the behavior. The child can also call into the parent's method to get the default behavior. In .Net, a class can only do implementation inheritance from a single parent class. In some languages, such as C++, the child can inherit from multiple classes.

Another type of inheritance is *interface inheritance*. In humans, you have two eyes, just like your parents, but your eyes may have a different color than either parents. Or you may behave differently in a similar situation when compared to mom and dad. In OOP languages, when a class supports interface inheritance, it only gets the property names and types and method names, parameter types, and return types for each behavior defined in the parent. It is up to the child class whether or not the particular method will have any behavior. However, the child must define each property or method defined in the parent. In .Net, a child can have one or more parent classes using interface inheritance.

Polymorphism

What an interesting word. Polymorphism comes from two Greek words, *poly* which means many and *morph* which means shapes. So it literally means *many shapes*. It shouldn't be surprising that there

are different types of polymorphism.

One type of polymorphism is when you have multiple method names in the same class. This is commonly called *method overloading*. For example, you may have multiple constructors on a class, each with different parameters.

Generics is a second type of polymorphism. With generics, you can reference different data types with the same class. For example, think of a generic list. You can define List<string> or List<int> or even use a custom type such as List<Customer>.

A third type is called *Subtyping* polymorphism. The best way to understand this is with an example that I have adapted from Wikipedia.

```
class Program
{
  static void Main(string[] args)
  {
    MakeNoise(new Ford());
    MakeNoise(new Beetle());
    Console.ReadLine();
  }

  private static void MakeNoise(Car c)
  {
    Console.WriteLine(c.Honk());
  }
}

class Ford : Car
{
  public override string Honk()
  {
    return “Honk!”;
  }
}

class Beetle : Car
{
  public override string Honk()
  {
    return “Beep! Beep!”;
  }
}

abstract class Car
{
  public abstract string Honk();
}
```

Note the MakeNoise() method. It expects Car as a parameter, yet when we pass a subclass of Car and call the Honk() method, we get the proper sound for that type of Car. The MakeNoise() method uses Subtyping polymorphism.

Encapsulation

The final pillar of OOP is encapsulation. There are two types of encapsulation.

The first is used to restrict access to the object's components. One way to do this is data hiding. You don't let outside components have direct access to the data or you prevent it entirely. Another way to do this is by defining methods as private so they can't be used externally.

Now that we have looked at the three pillars, let's look at a couple of other important concepts we should strive for in defining classes, loose coupling and tight cohesion.

Loose coupling

With loose coupling, one class has little knowledge of another class. Every time we *new* an object, we are tight coupling one class to another. It may be better to define an interface and pass in instances of a class rather than new-ing it where it's used.

The classic example of this is a logging class. It shouldn't matter if we log to a text file, a data, or the event log. If we define the logging type externally, then pass that instance into where it's used, the result will be the type of logging we desire and the code that actually uses the log has no idea where the log is being written to, nor does it care.

Tight cohesion

Cohesion is kind of the opposite of loose coupling. When a class has tight cohesion, its methods and properties are tightly related. Going back to the logging class, if it has a method that updates a customer record, that method would be out of place. It doesn't belong in a logging class.

Using classes

Was this too basic for you? Well, we should all know these basics, but how do you actually determine what classes your application needs? If you have good Soil or Agile methodologies for your team, you will be using something like a Use Case at the beginning of your analysis and design. A Use Case is a very simple description of how the user sees the system as functioning. In other words, you start with how the application should function before you get into details of the data that functionality needs.

In the 2004 book "Object Thinking", Dr David West wrote:

> "Only when you are satisfied with the distribution of responsibilities among your objects are you ready to make a decision about what they need to know to fulfill those responsibilities and which part of that knowledge they need to keep as part of their structure

(Object Thinking, p. 124). In other words, you need to know what an object will do before you know what it needs to do it. Today, this book still stands out as a must read. Safari recently named it one of the most influential books in software.

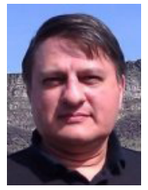
Do you think about what a class should do before you define the data it needs? After talking to many developers, I feel pretty safe to say that you don't. It seems most developers start with the data.
Another thing we often do wrong is define classes and methods, but then use methods procedurally. In other words, we may define a method to perform a complex function then put all the code needed into that single method. This method is doing more than one thing, making it more likely to have bugs and more difficult to unit test.

Thinking about how we define classes has a big impact on maintaining our code and our ability to have fewer bugs. Planting the right seeds at the right time means you are growing the right things. After all, you plant seeds so that the right things will grow and keep your code lush, green, and vibrant.

About Software Gardening

Comparing software development to constructing a building says that software is solid and difficult to change. Instead, we should compare software development to gardening as a garden changes all the time. Software Gardening embraces practices and tools that help you create the best possible garden for your software, allowing it to grow and change with less effort ■

About the Author



Craig Berntson is the Chief Software Gardener at Mojo Software Worx, a consultancy that specializes in helping teams get better. He has spoken at developer events across the US, Canada, and Europe for over 20 years. He is the co-author of "Continuous Integration in .NET" available from Manning. Craig has been a Microsoft MVP since 1996. Email: craig@mojosoftwareworx.com, Blog: www.craigberntson.com/blog, Twitter: @craigber. Craig lives in Salt Lake City, Utah.

.NET Web Development using Node.js

Today's web applications have undergone a paradigm shift from being a stateless request-response implementation where only the client initiates communication; to being asynchronous real-time two-way channel, in which both the client and server can initiate communication. Meanwhile during this shift, JavaScript has quietly become omnipresent, and is used in websites, modern NoSQL databases, complicated mobile apps and even on servers.

Node.js was born somewhere in between the intersection of these trends and provides a core library to build network applications that allows the server to make the best use of the available I/O in a non-blocking fashion. This library fits well in this era of modern apps which consists of a thick client built with modern JavaScript frameworks like Angular and Backbone, and a thin backend layer represented by the REST API.

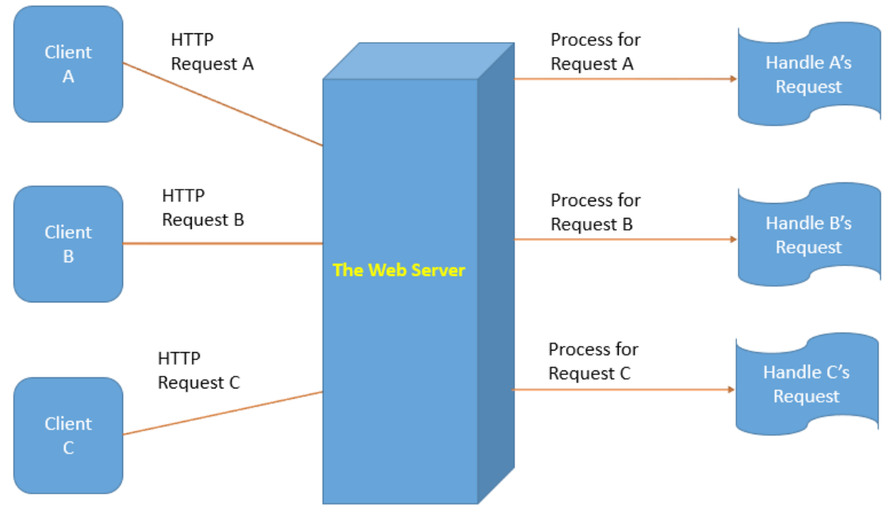
In this article we will learn how to create REST APIs using Node.js, MongoDB and Express and then consume it in an ASP.NET MVC application.

Let us quickly introduce these technologies for those who are new to it.

Node.js

Node is a platform built on Google's V8 JavaScript Runtime for building fast, scalable network applications, with ease. It uses non-blocking I/O and event-driven model. Before using Node.js in our application, it is important to understand some important features of it.

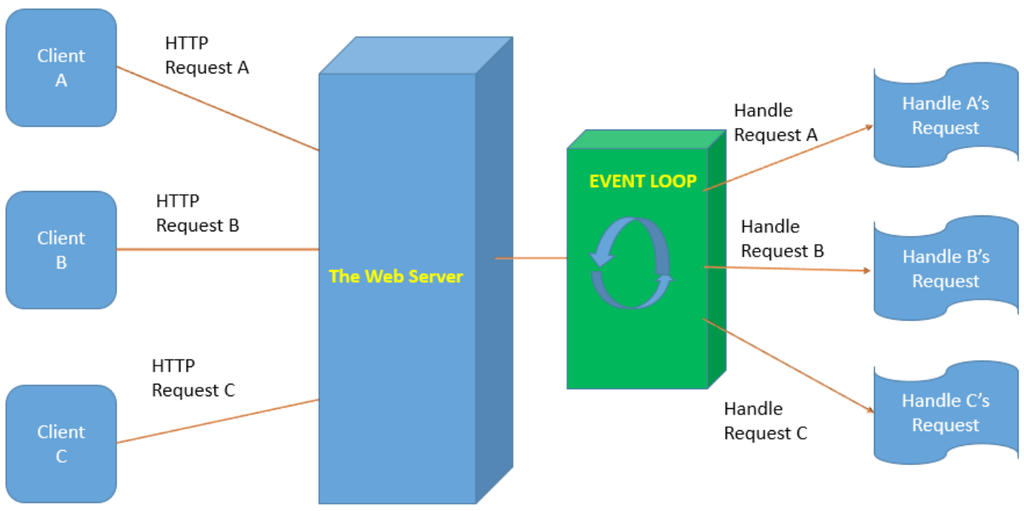
Traditionally web applications hosted on the servers spin up a process for each incoming request. This is an expensive operation both in terms of CPU and memory utilization. The following diagram illustrates the traditional web server behavior.



This mechanism makes use of threads from a thread pool to manage incoming requests from each user. The server assigns a separate thread for each incoming request and uses the thread for the entire lifecycle of the request. The thread for each request keeps on utilizing resources of the server. This is an expensive approach, as it involves a lot of processing and resources. Here please note that unless the *async* request processing is not configured, e.g. in case of ASP.NET 4.5 the async Http handler; we cannot expect any improvement in resource utilization.

Node.js on the other hand uses *event loop*, which runs under a single thread. JavaScript runtime has a message queue which stores the list of messages to be processed with their associated callback functions. These messages are actually I/O calls with the callback registered with it. As soon as the I/O operation is completed, the associated callback is pushed in the event loop and executed. So despite of being "single threaded", there are an arbitrary number of operations that Node can handle in the background.

We can see the event loop in action in the following figure:



All the requests to be processed go into a single thread and hence result in less memory utilization, and because of the lack of the thread context switching, the CPU utilization is also less. This makes Node.js effective in data intensive applications.

Node.js Modules

Node.js ships with a number of built-in modules which provides core set of features to develop Node.js based solutions. Node.js modules are always loaded by passing their identifier to the *require()* function. Node has three kinds of module: core modules, file modules and external node modules.

File based modules in Node.js

The File-Based module system of Node.js has the following specifications:

- Each file is its own module
- Each file uses *module* variable to access the module definition
- *module.export* is used to determine the export of the current module.

The require() function of Node.js is the way to import modules into the current file.

e.g. `require('../foldername/file')`

The above example will load the modules exported from the *file* under the *foldername* in Node applications.

Facts about Node.js File-Based Module

- Modules can be loaded conditionally e.g. use *if* condition

```
if(somecondition){
  var mod = require('/moduleFIle')
}
```

In this case the *require* function will load the module when the condition is satisfied.

- The *require* function will block the execution unless the module is not completely loaded. This will make sure that the required functionality from the module will always be available to the caller.

- Once the *require* function loads the module, it will be *cached*. The next time when the call is made to the require function with the same module, it will be loaded from the cache.

- Once the module is loaded they can *share the state* across the context.

- To export content from the module, Node.js uses *module.exports*. This empty object is by default available to all module in Node.js.

e.g. `module.exports={};`

Node.js Core Modules

The Node.js core modules can still be consumed using the *require* function. The only difference here is that unlike the file-based modules, we directly pass the name of the module to the require function. The following is the list of Node.js core modules:

Path - The *path* module provides useful string transformations while working with the file system. This helps to eliminate inconsistencies in handling file system paths.

e.g. `var path=require('path');`

Fs - The *fs* module provides an access to the file system. This has functions for renaming files, reading files, writing files, deleting files, etc.

e.g. `var fs = require('fs');`

OS - The *os* module provides access to some of the operating system functionality e.g. memory usage, free memory etc.

e.g. `var os = require ('os');`

In Node.js, module gets loaded based on its path. If the node module is specified without a path and if this module is not the core module, then Node looks for the module in the *node_modules* folder. If the module is located in this folder, then it will be loaded from here, else Node will search through the parent directory of node_modules.

External Modules

Apart from the Node.js core modules, there are various modules which provides rich functionality and are available to Node via third party sources. There are modules for relational and document databases, testing modules and many more. Developers can create their modules and upload on GitHub. External node modules can be loaded using the command line utility:

```
npm install <modulename>
```
e.g.
```
npm install express
```

This will install the *express* module. Express.js is a middleware module. It is a minimal and flexible Node.js web application framework which provides several features like HTTP utilities for requests/ responses and routing. In our application we will be

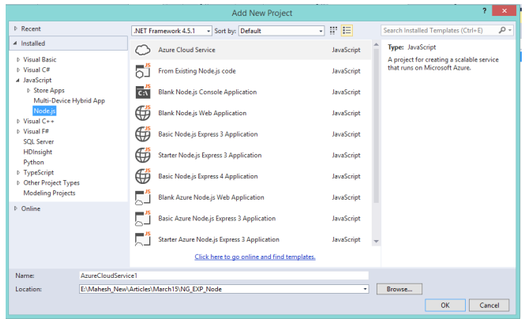
using this framework for making HTTP GET, POST calls. More information about express can be obtained from here.

In the following application, we will be using this module. Once this module is installed, it will be available from the application's local *node_modules* directory.

Now that we have gone through the basics of Node.js modules, it is time for us to create Node.js applications in .NET.

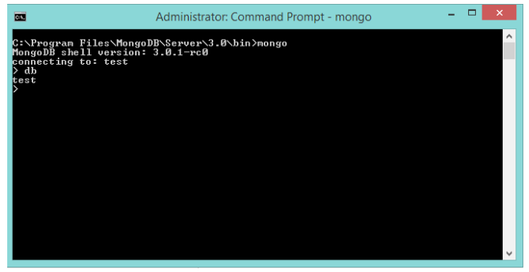
Developing a REST API using Node.js and consuming it in an ASP.NET MVC application

In this section, we will create a REST API using Node.js and then consume this API in an ASP.NET MVC application. To implement this solution, we will use the Free Visual Studio 2013 Community Edition. Node.js templates can be added to Visual Studio 2013 using Node.js tools for Visual Studio (NTVS). Once this extension is installed, Visual Studio will show Node.js templates as shown here:

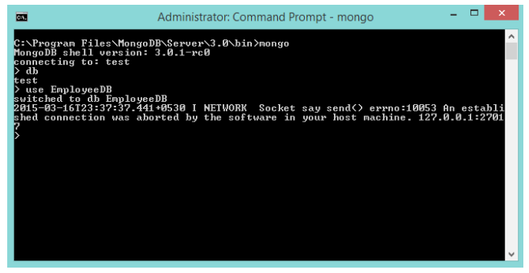


Our application will make use of MongoDB to store data. We can install MongoDB for windows by downloading it from this link. We need to install and configure MongoDB using the steps from the link here. Mongo can be installed in the %Program Files% folder and the *MongoDB* folder inside it. Using *mongod –dbpath* command, a connection with the MongoDB can be started as shown here:

Using *mongo* command from the command prompt, the MongoDB shell can be connected with the default *test* database as shown in the following figure.

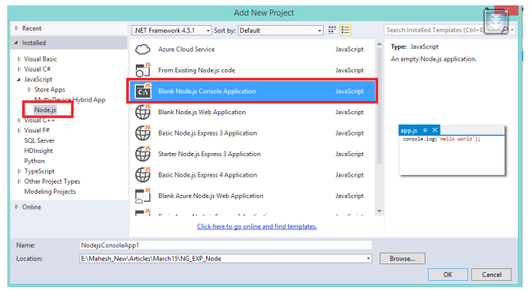


The New database can be created using the command use *<dbname>*



The command *use EmployeeDB* creates a new database named EmployeeDB.

Step 1: Open Visual Studio and create a new blank Node.js console application and name it 'NG_EXP_Node' as shown in the following figure.



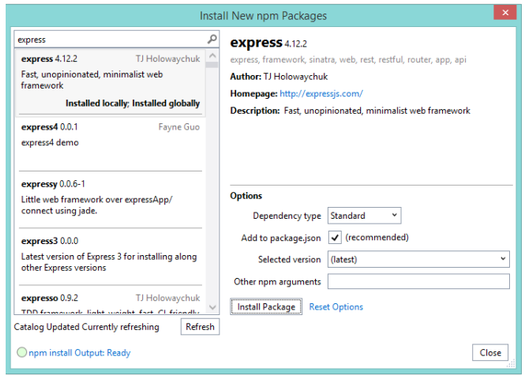
This step will create an app.js file by default in the project.

Step 2: Since we will be using MongoDB, Express and Mongoose in the project, we need to install these packages first.

The Mongoose npm package allows to connect to MongoDB and helps to define the schema for storing data in MongoDB. This package also provides functions for performing CRUD operations against MongoDB.

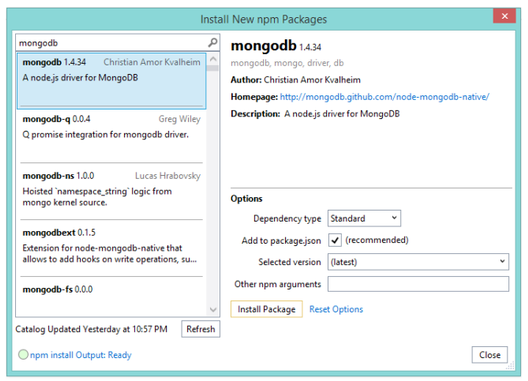
Right-click on the project and select the *Install new npm packages*....option which brings up the package window. Note that if you are doing this for the first time, it will first list all packages. Luckily the window has a *Search for package* text box using which the necessary packages can be searched. Add the packages as shown in the following figure:

Express



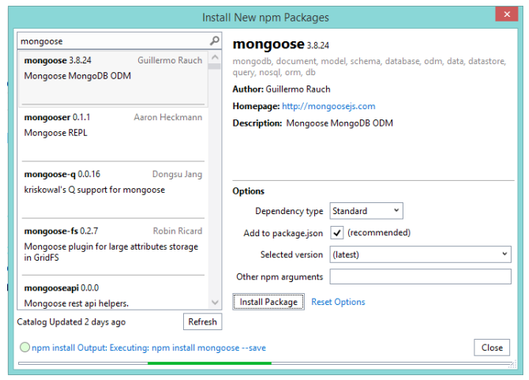
Click on Express and click on 'Install Package' to install the Express package.

MongoDB



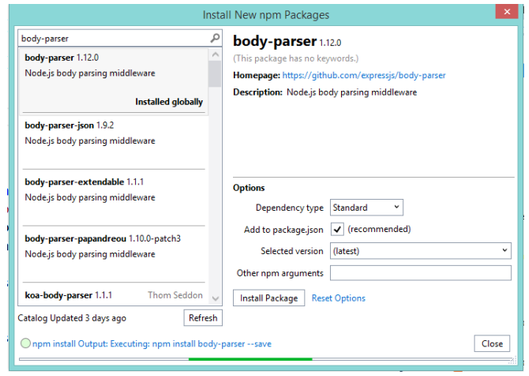
This is the driver for the MongoDB database access.

Mongoose



This will install the Mongoose npm package.

Body Parser



Body parser is a tool that gives you easy access to values submitted using a form. After these packages are added, the package.json file in the project will be updated with the following entries:

```
{
  "name": "NG_EXP_Node",
  "version": "0.0.0",
  "description": "NG_EXP_Node",
  "main": "app.js",
  "author": {
    "name": "Mahesh",
    "email": ""
  },
  "dependencies": {
    "body-parser": "^1.12.1",
    "bson": "^0.2.19",
    "express": "^4.12.2",
    "mongodb": "^1.4.34",
    "mongoose": "^3.8.25"
  }
}
```

Step 3: In the project, add a new JavaScript file named ReadWrite.js with the following code:

```
var mongooseDrv = require('mongoose');
/* Get the Mongoose Driver
connection with MongoDB */
mongooseDrv.connect('mongodb://
localhost/EmployeeDB');
var db = mongooseDrv.Connection; //The
Connection

if (db == 'undefined') {
  console.log("The Connecion issues");
}

//The Schema for the Data to be Stored
var EmployeeInfoSchema = mongooseDrv.
Schema({
  EmpNo: String,
  EmpName: String,
  Salary: String,
  DeptName: String,
  Designation: String
});

var EmployeeInfoModel =
mongooseDrv.model('EmployeeInfo',
EmployeeInfoSchema);

//retrieve all records from the database
exports.get = function (req, resp) {
  EmployeeInfoModel.find().exec(function
  (error, res) {
    if (error) {
      resp.send(500, {error:error});
    } else {
      resp.send(res);
    }

  });
};

//Add a new Record in the Employee Model
exports.add = function (request,
response) {
  var newEmp = { EmpNo: request.
  body.EmpNo,EmpName: request.body.
  EmpName,Salary: request.body. Salary,
  DeptName: request.body.
  DeptName,Designation: request.body.
  Designation};
  EmployeeInfoModel.create(newEmp,
  function (addError, addedEmp) {
    if (addError) {
      response.send(500, { error:
    addError });
    }
    else {
      response.send({ success: true,
      emp: addedEmp });
    }
  });
};
```

This JavaScript code uses the *Mongoose* driver to connect to the EmployeeDB database created in MongoDB. This also defines the *EmployeeInfoSchema* object containing the schema for storing Employee information. The methods *get()* and *add()* are used to read and write data received from the client application respectively.

Step 4: In the project, add the following code in app.js to load the necessary modules for Http utilities and REST APIs.

```
var body_Parser = require('body-
parser'); //Load Modules for bodyparser
var path = require('path'); //Load
'path' the File base module
var express = require('express'); //Load
express module
var http = require('http'); //Load Http
module for Http operation

var app = express(); //The Express
object

app.use(body_Parser());

var rwOperation = require('./ReadWrite.
js'); //Load the File

var communicationPort = 8080; //The
Communication port

//The REST API

app.get('/EmployeeList/api/employees',
rwOperation.get);

app.post('/EmployeeList/api/employees',
rwOperation.add);

app.listen(communicationPort);
```

This code loads *express* module for creating REST APIs. The get() and post() methods on the express application object are used to expose REST APIs to perform HTTP GET and POST methods respectively.

Step 5: Open the command prompt with Admin (Run as Administrator) rights. Navigate to the project folder and run the following command.

```
<Project Path>:>node app.js
```

This will start the Node.js environment.

Step 6: In the same solution, add a new empty ASP.NET MVC application with the name Node_MVCClient. In this project, add JavaScript references for jQuery, Bootstrap and Angular.js using NuGet.

Step 7: In the Controllers folder, add a new empty MVC controller of the name NodeAppController. Scaffold an empty Index view from the Index method of this controller.

Step 8: In the Scripts folder add a new folder of name MyScripts. In this folder, add the following JavaScript files:

module.js

```
var app;
(function () {
  app = angular.
module('mynodemodule',[]);
})();
```

service.js

```
app.service('mynodeservice', function
($http) {
  this.get = function(){
    var res = $http.get("http://
    localhost:8080/EmployeeList/api/
    employees");
    return res;
  }

  this.post = function (emp) {
    var res = $http.post("http://
    localhost:8080/EmployeeList/api/
    employees",emp);
    return res;
  }
});
```

The above code defines methods for making a call to the REST APIs using Node.js application.

Controller.js

```
app.controller('mynodecontroller',
function ($scope, mynodeservice) {

 //The Employee Object used to Post
 $scope.Employee = {
  _id:"",
  EmpNo: 0,
  EmpName: "",
  Salary: 0,
  DeptName: "",
  Designation:""
 }

 loaddata();
 //Function to load all records
 function loaddata() {
  var promise = mynodeservice.get();

  promise.then(function (resp) {
   $scope.Employees = resp.data;
   $scope.message = "Call is Successful
   ";
  }, function (err) {
   $scope.message = "Error in Call" +
   err.status
  });
 };

 //Function to POST Employee
 $scope.save = function () {
  var promisePost = mynodeservice.
  post($scope.Employee);

  promisePost.then(function (resp) {
   $scope.message = "Call is
   Successful";
   loaddata();
  }, function (err) {
   $scope.message = "Error in Call" +
   err.status
  });
 };

 $scope.clear = function () {
  $scope.Employee.EmpNo = 0;
  $scope.Employee.EmpName = "";
  $scope.Employee.Salary = 0;
  $scope.Employee.DeptName = "";
  $scope.Employee.Designation = "";
 };
});
```

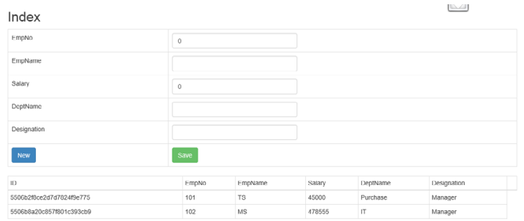
The above code contains functions for reading all data using REST APIs and performing POST operations.

Step 9: In the Index.cshtml add the following markup:

```
<html ng-app=”mynodemodule”>
<head title=””>
 <script src=”~/Scripts/jquery-
 2.1.3.min.js”></script>

 <script src=”~/Scripts/bootstrap.min.
 js”></script>

 <script src=”~/Scripts/angular.min.
 js”></script>

 <script src=”~/Scripts/MyScripts/
 module.js”></script>

 <script src=”~/Scripts/MyScripts/
 service.js”></script>

 <script src=”~/Scripts/MyScripts/
 controller.js”></script>
</head>

<body ng-controller=”mynodecontroller”>
 <h2>Index</h2>
  <table class=”table table-bordered”>
   <tr>
     <td>EmpNo</td>
   <td><input type=”text” class=”form-
    control” ng-model=”Employee.
    EmpNo”/>
   </td>
</tr>

<tr>
 <td>EmpName</td>
 <td><input type=”text” class=”form-
  control” ng-model=”Employee.
  EmpName” />
 </td>
</tr>

<tr>
 <td>Salary</td>
 <td>
  <input type=”text” class=”form-
  control” ng-model=”Employee.Salary”
  />
 </td>
</tr>

<tr>
 <td>DeptName</td>
 <td>
  <input type=”text” class=”form-
  control” ng-model=”Employee.
  DeptName” />
 </td>
</tr>

<tr>
 <td>Designation</td>
 <td><input type=”text” class=”form-
  control” ng-model=”Employee.
  Designation” />
 </td>
</tr>

<tr>
 <td><input type=”button” value=”New”
  class=”btn btn-primary”
  ng-click=”clear()”/>
 </td>
 <td>
  <input type=”button” value=”Save”
  class=”btn btn-success”
  ng-click=”save()” />
 </td>
</tr>
</table>

<table class=”table table-bordered
table-condensed”>
 <thead>
  <tr>
   <td>ID</td>
   <td>EmpNo</td>
   <td>EmpName</td>
   <td>Salary</td>
   <td>DeptName</td>
   <td>Designation</td>
   <td></td>
  </tr>
 </thead>

 <tbody ng-repeat=”Emp in Employees”>
  <tr>
   <td>{{Emp._id}}</td>
   <td>{{Emp.EmpNo}}</td>
   <td>{{Emp.EmpName}}</td>
   <td>{{Emp.Salary}}</td>
   <td>{{Emp.DeptName}}</td>
   <td>{{Emp.Designation}}</td>
  </tr>
 </tbody>
</table>

 <div>
  {{Message}}
 </div>
</body>
</html>
```

This markup contains Angularjs databinding using various directives.

Run the application and you should get the following result:



Enter values in the TextBoxes and click on the *Save* button. The table will display the newly added record.

CONCLUSION

This article demonstrated how Node.js provides server-side JavaScript features for REST APIs and how it can be consumed by any client application. This article aimed at helping .NET developers learn the essentials of Node.js web development ■

Download the entire source code from GitHub at bit.ly/dncm18-nodejsdotnet

About the Author

Mahesh Sabnis is a Microsoft MVP in .NET. He is also a Microsoft Certified Trainer (MCT) since 2005 and has conducted various Corporate Training programs for .NET Technologies (all versions). Follow him on twitter @maheshdotnet. Mahesh blogs regularly on .NET Server-side & other client-side Technologies at bit.ly/HsS2on

Create a Responsive jQuery Flickr Image Gallery with Lazyloading

Flickr is a popular photo-sharing site and provides a couple of ways to retrieve photos and related information. The Flickr Feed Service allows you to get a listing of the 20 most recent photos from a particular person or a particular group. To access photos from user accounts, you can use http://api.flickr.com/services/feeds/photos_public.gne . Once you know which type of photo feed you'd like, you can search for tags on that Flickr's public photo feed . A *tag* is a word or short phrase that the photo owner uses to describe an element of the photo. In the url, add the *tag* keyword to the URL to specify a tag; for example: `&tags=nature`

The Flickr photo feed service can return photo information in many different formats like RSS, Atom, CSV, and JSON. We want data in JSON format, so we will add `&format=json` to the query string.

Finally, in order to successfully request JSON data from another website, we will add one last bit to the URL: `&jsoncallback=?`. For security reasons you can't send an XMLHTTP request to a different domain. To get around that problem, we use the `&jsoncallback=?` piece to notify the external wesbite that we want to receive JSONP data. Internally jQuery's `$.getJSON()` function treats this request as if the web browser was requesting an external JavaScript file and thus makes the call successful.

So our request url will be similar to the following:

```
'http://api.flickr.com/services/feeds/photos_public.
gne?tags=nature&tagmode=any&format=
json&jsoncallback=?'
```

With this information under our belt, let's request a collection of photos from Flickr and display them on a page. We will use Twitter Bootstrap to make our page Responsive Design enabled.

We will also be using *Lazy Loading*. Lazy Loading is delaying the loading of images, particularly in long web pages that has a lot of images. Images outside of the viewport are not loaded, until the user scrolls to them. This makes the page load faster and also reduces the load on the server, by not requesting all the images at one go.

Create a new file 'FlickrLazyload.html' and an 'Images' folder. To begin with, we will need a simple `<div>` as follows:

```
<body>
  <div class="flickr" />
</body>
```

For layout purpose, this example uses css kept in `Images/css/flickr.css`

Consider the following piece of code. I have added some comments in the code to help you understand what's going on:

```
$(function () {

  /* When the page is scrolled call the
lazyLoadImage function */
  $(window).scroll(lazyLoadImage);

  var imageList='';

  // Get the photo feed from flickr
    $.getJSON('http://api.flickr.com/
    services/feeds/photos_public.
    gne?tags=nature&tagmode=any&format
    =json&jsoncallback=?', function(data)
    {
      /* Loop through the flickr json
      response */
      $(data.items).each(function
      (i, item) {
        /* Create a new figure element
        with an img element with:
         - class "lazy"
         - the current src as a loding
        gif
         - the data-src with the real
        flickr image path
         Create a figcaption element
        with the image title */
        imageList+='<div class=
        "col-md-4"><figure>
        <div class="imgWrap">
        <img class="lazy" src="images/
        ajax-loader_b.gif"
        data-src="'+item.media.m+'"/>
        </div> <figcaption>'+item.title+'
        </figcaption></figure></div>';
                              });

if (imageList=='') {
  $(".flickr").html("Error");
  return
}

/* Append the newly created element
list */
$(".flickr").append(imageList);
/* Wrap every three elements in a
div row for scaffolding purposes */
var $images = $(".flickr .col-md-4");
for(var i = 0; i < $images.length;
i+=3) {
  $images.slice(i, i+3).wrapAll(
  "<div class='row'></div>");
}
  /* After the flickr response call the
  lazyLoadImage function */
          lazyLoadImage();
  });

});
```

The `lazyLoadImage()` function which we will define shortly, is called on the page scroll.

```
$(window).scroll(lazyLoadImage);
```

We use `$.getJSON` to fetch data from the flickr service.

```
$.getJSON('http://api.flickr.com/
services/feeds/photos_public.
tags=nature&tagmode=
any&format=json&jsoncallback=?',
function(data) { }
```

We then loop through the response and create a new `<figure>` for each element.

```
$(data.items).each(function(i, item) {
  /* Create a new figure element with an
  img element with:
   - class "lazy"
   - the current src as a loding gif
   - the data-src with the real flickr
  image path
  Create a figcaption element with the
  image title */

  imageList+='<div class="col-md-
  4"><figure><div class="imgWrap"><img
  class="lazy" src="images/ajax-
  loader_b.gif" data-src="'+item.media.
  m+'"/></div> <figcaption>'+item.
  title+'</figcaption></figure></div>';

});
```

Each figure element is structured with:

1. "lazy" class
2. the current src attribute as a loading gif image
3. the data-src attribute with the *real* flickr image path

and a `<figcaption>` element with the image title displayed below the respective image.

The next step is to check if `imageList` is empty. If it is, display an error message to the user. Finally, we append the images to the flickr div *outside* the loop.

```
if (imageList==’’) {
  $(“.flickr”).html(“Error while loading
images”);
  return
}

// Append the newly created element list

$(“.flickr”).append(imageList);
```

Note: *Observe that we are only constructing the string inside the loop and not appending it to the imagelist. A common mistake made by developers is to call* `append` *inside every loop. We are using* `append()` *outside the loop to improve performance.*

For scaffolding purposes, we are using the Twitter Bootstrap plugin, so that the images are organized three per row using col-md-4 and row classes.

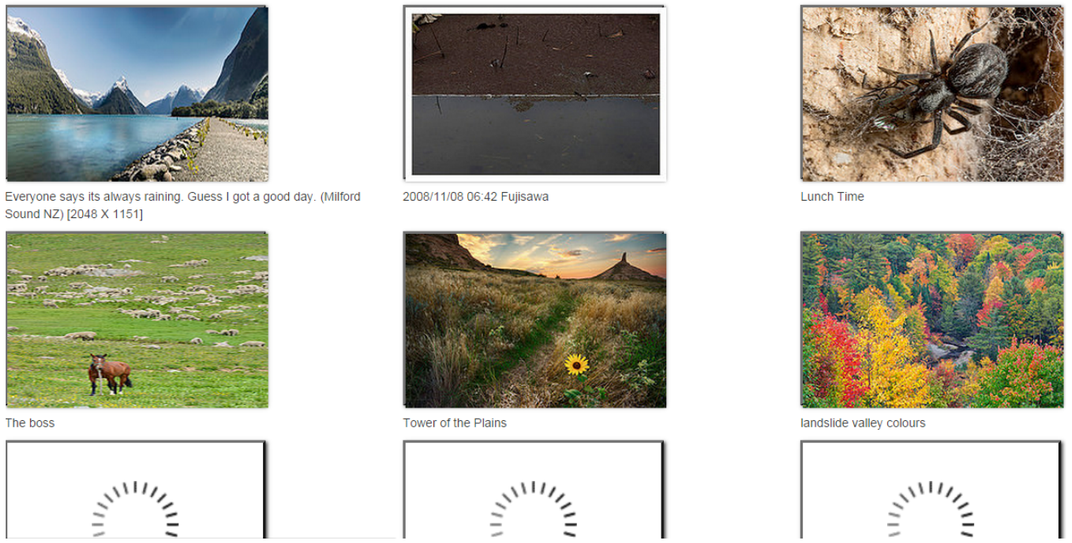
```
var $images = $(“.flickr .col-md-4”);
for(var i = 0; i < $images.length; i+=3)
{
  $images.slice(i, i+3).wrapAll(“<div
  class=’row’></div>”);
}

lazyLoadImage();
```

And finally, with the flickr div filled with our structure, we call the `isVisible` function and `lazyLoadImage` function which are defined as:

```
/* Check if the requested element is
  COMPLETELY visible in the user
viewport */
function isVisible($element)
{
  // Get the current page visible area
  in the user screen
  var topView = $(window).scrollTop();
  var botView = topView + $(window).
  height();
  var topElement = $element.offset().
  top;
  var botElement = topElement +
  $element.height();

  return ((botElement <= botView) &&
  (topElement >= topView));
}

function lazyLoadImage(){
  /* Loop through the images not already
  loaded, with “lazy” class */
  $(‘.flickr img.lazy’).each(function(){
  /* Check if the element is visible in
  the viewport */

  if (isVisible($(this).closest(‘.
  imgWrap’))) {
    /* If the element is visibile,
    set the img src as its data-src
    attribute,
    remove the lazy class
    and remove the data-src attribute */

    $(this).one(“load”, function() {
      // image loaded here
      $(this).removeClass(‘lazy’);
    }).attr(‘src’, $(this).attr(‘data-
    src’)).removeAttr(‘data-src’);

    }
  });
};
```

In our example, we are loading 20 images which are divided across 7 rows, with 3 images each in a row. While scrolling the page, in the `lazyLoadImage()` function, we loop through the images which are not already loaded (recognized by the “lazy” class). We then use the `isVisible()` function defined by us to check if the requested element is *completely* visible in the user viewport i.e. if the user has scrolled down further. If the element is visible in the user viewport that means it’s time to display the image to the user; so we switch the *data-src* attribute with the *src* attribute. Remember the *data-src* contains the gif preloading image, while the *src* attribute contains the path to the actual flickr image. This switch causes the browser to download and display the real image.

Save and browse ‘FlickrLazyload.html’ in your browser and you will see a Flickr image gallery powered with Lazy Loading. The images will load as you scroll down the page.



This code was authored by Irvin Dominin and explained by Suprotim Agarwal

Live Demo: http://www.jquerycookbook.com/demos/S6-Images/52-FlickrLazyload.html ■

Download the entire source code from GitHub at bit.ly/dncm18-jqueryflickr

About the Author

Irvin Dominin is currently working as lead of a technical team in SISTEMI S.p.A. (Turin, Italy) on .NET, jQuery and windows Projects. He is an active member on StackOverflow. You can reach him at: irvin.dominin@gmail.com or on LinkedIn

Asynchronous Programming in ES6

Using Generators and Promises

Asynchronous programming has started gaining a lot of attention over the past couple of years. In JavaScript-heavy applications, there are a number of cases (viz., making an AJAX call or executing a piece of logic after certain time) where we need to deal with asynchronous logic. Some of the JavaScript APIs that were added to browsers as part of HTML5 (viz., IndexedDb, Geo locations and Web sockets) are asynchronous. Despite of such heavy usage of asynchrony, JavaScript as a language doesn't support it well. Callbacks are the only way one can handle asynchronous APIs in JavaScript; but the approach fails to scale. For example, when you have to deal with more than 3 levels of nested callbacks, the code starts forming a nice looking tree that is hard to read and understand.

There are a number of libraries like Q, RSVP, Bluebird, and APIs in frameworks like jQuery's deferred API and Angular's $q API which addresses this issue. Most of these APIs are designed based on Promise/A specification. Though these APIs do a good job of avoiding to deal with callbacks, they add one more entry into the developer's list of learning. And each of them are different. So if you have to switch from one library to the other, you need to unlearn and relearn. Although unlearning and relearning is not new to developers, it would be certainly better if JavaScript itself had async APIs.

This article is a continuation of the ECMA Script 6 (ES6) articles published earlier in the DNC Magazine.

The first part covered the improvements to the JavaScript's language features and second part covered the new data structures and new APIs added to the existing objects.

This article will focus on improvements that make asynchronous programming better in ES6.

ES6 adds the support of promise API to the language. It is also possible to design some asynchronous APIs using generators. In this article, we will see how to use these features individually and together, to deal with asynchrony in JavaScript.

Generators

Simple Generator Functions

As I stated in my first article on ES6 (ES6: New Language Improvements in JavaScript), generators are used to make objects of classes iterable. They can also be used to create functions or methods that return iterable objects.

For example, let's write a simple function that returns a specified number of even numbers.

```
function *firstNEvenNumbers(count){
  var index = 1;
  while(index <= count){
    yield 2*index;
    index++;
  }
}
```

Now we can call this function with different values of count to produce different number of even numbers on each call. The following snippet shows two instances of calling the above function:

```
for(let n of firstNEvenNumbers(3)){
  console.log(n);
}

for(let n of firstNEvenNumbers(5)){
  console.log(n);
}
```

The important thing to notice here is that the execution of the Generator function is paused after *yielding* a result until the next iteration is invoked. Here the *for...of* loop implicitly invokes the next iteration. We can manually invoke the next iteration using generator object returned by the function.

The following snippet shows the same:

```
var generator = firstNEvenNumbers(4);
var next = generator.next();
while(!next.done){
  console.log(next.value);
  next=generator.next();
}
```

Asynchronous Functions using Generators

As we saw, execution of a generator is paused till the next iteration is invoked. This behavior of pausing and continuing can be used to perform a set of asynchronous tasks in which the tasks to be executed depend on result of already finished tasks. We can design a generator function like the one in the following snippet, without using a loop:

```
function *generatorFn(){
  //first task
  yield firstValue;

  //second task
  yield secondValue;

  //third task
  yield thirdValue;
}
```

We can pause execution of the above function in between, by calling *setTimeout* in the statement that uses *yield* keyword. This causes the statements below the yield keyword to wait till the control is executing the timeout. The following function demonstrates this:

```
function *simpleAsyncGenerator(){
  console.log("starting...");
  yield pauseExecution(2000);
  console.log("Came back after first
  pause!");

  yield pauseExecution(5000);
  console.log("Came back after second
  pause!");

  yield pauseExecution(3000);
}
```

The *pauseExecution* function in the above snippet calls *setTimeout* internally and passes the control back to the generator function by calling the *next()* method on the generator object. So it needs a reference of the generator object. Following snippet

gets a generator object of the above function and implements the *pauseExecution* method:

```
var generatorObj =
simpleAsyncGenerator();
generatorObj.next();

function pauseExecution(time){
  console.log(`Pausing for ${time}ms`);
  return setTimeout(function () {
    generatorObj.next();
  }, time);
}
```

The *pauseExecution* method defined above is not generic; it directly depends on generator object of the *simpleAsyncGenerator* function. To make it generic, we need to create an object or, array to hold all generator objects and pass name of the property of the generator object wherever it is needed. Here's a snippet that demonstrates this:

```
function pauseExecution(time,
generatorName){
  console.log(`Pausing for ${time}ms`);
  return setTimeout(function () {
  generators[generatorName].next();
  }, time);
}

function *simpleAsyncGenerator
(generatorName){
  console.log("starting...");
  yield pauseExecution(2000,
  generatorName);
  console.log("Came back!");

  yield pauseExecution(5000,
  generatorName);
  console.log("Came back!");

  yield pauseExecution(3000,
  generatorName);
}

var generators={},
generatorName="simpleAsyncGenerator";
generators[generatorName] =
simpleAsyncGenerator(generatorName);
generators[generatorName].next();
```

Promises

What is promise?

The traditional callback approach of handling asynchronous tasks is good for a limited number of nested levels and they can be used in very few layers. Usually, a nested callback model consisting of 3 or more async operations becomes unmanageable. Also, if the application is divided into multiple layers, keeping track of the callback becomes a challenge after certain depth. Promises define a different way of dealing with asynchronous operations and they attempt to reduce these difficulties.

A promise is an object wrapper around an asynchronous operation that gets back with the result (either success or failure) once execution of the operation is finished. At a given point of time, a promise would be in one of the following states:

- Pending: When execution of the operation is still in progress
- Success: Once execution of the operation is successfully completed
- Failure: Once execution of the operation is failed

The promise object accepts callbacks to be executed upon completion of the asynchronous operation. These callbacks in turn return promises; which makes it easier to chain multiple promises.

Now that you got some idea about promises, let's look at Promises in ES6.

ES6 Promise API

ES6 adds support of Promise to the JavaScript language. To create a promise, we need to call the constructor of the *Promise* class. Following snippet shows the syntax of a promise handling an asynchronous operation:

```
var promise = new Promise(function
(resolve, reject) {
  //Statements in the function

  if(allIsGood) {
    resolve(result.data); //All is well,
    so resolve the promise
  }
  else {
    reject(result.error); //Something is
    wrong, reject the promise
    }
  });
});
```

Here, the imaginary function *anAsynchronousFn* accepts a callback that will be called after the asynchronous operation is completed.

The parameters *resolve* and *reject* passed into the promise function are the callbacks that will be called once the promise succeeds or fails respectively.

We can get result of the above promise by calling the *then* method on the above promise object.

```
promise.then(function (data) {
  //Use data
}, function (error) {
  //Handle the error
});
```

Let's take an example to understand the above snippets better. Consider the following function. It uses *setTimeout* to make its behavior asynchronous and uses *Promise* to emit the result once the operation is completed:

```
function asyncFunctionUsingPromise(pass)
{
  return new Promise(function (resolve,
  reject) {
    setTimeout(function () {
      if(pass){
        resolve(100);
      }
      else{
        reject({error: "Some error
        occured!"});
      }
    }, 2000);
  });
}
```

Here the first parameter pass is a *Boolean* parameter to force the async operation to either pass or fail. Second parameter *onSuccess* is a callback that will be called once the operation succeeds and the third parameter *onFailure* is another callback that will be called if the operation fails.

Now let's consume this function and test the cases of success and failure.

```
function onSuccess(data){
  console.log("Promise Passed with
  result: " + data);
  return data;
}

function onFailure(error){
  console.log("Promise Failed!");
  console.error(error);
}

asyncFunctionUsingPromise(true).
then(onSuccess, onFailure);

asyncFunctionUsingPromise(false).
then(onSuccess, onFailure);
```

It is not mandatory to pass both callbacks to the *then* method. We can omit the one that is not going to be used. In most of the cases, we omit the failure callback.

Now let's see some more API functions that are available through *Promise* object:

- **Promise.resolve:** Used to create and resolve a promise immediately with a fixed value

```
Promise.resolve({city: "Hyderabad"}).
then(function (data) {
  console.log(data);
});
```

- **Promise.reject:** Used to create and reject a promise immediately with a fixed error message

```
Promise.reject({error: "Some error
occured!"}).then(null, function (error)
{
  console.log(error);
});
```

- **Promise.prototype.catch:** If it is known that a promise is going to fail, or to avoid handling of success case of a promise, the *catch* method is used

```
Promise.reject({error: “Some error
occured! (Using catch)”}).catch(function
(error) {
  console.log(error);
});
```

- **Promise.all:** Used to combine an array of promises into one promise. The resulting promise will succeed if all promises are succeeded and it will fail

```
if any of the promises fail
var array = [Promise.resolve(100),
Promise.resolve({name:”Ravi”}), Promise.
resolve(true)];
Promise.all(array).then(function (data)
{
  console.log(data);
});
```

- **Promise.race:** Accepts an array of promises and results with result of first promise

```
Promise.race(array).then(function (data)
{
  console.log(data);
});
```

Chaining Promises

One of the advantages of using promises is it is easy to chain them. As the *then* method returns a promise, we can wire up another asynchronous operation inside the *then* method and handle its callbacks using a chained *then* method. Consider the following asynchronous functions:

```
function getEmployee(empId){
  var employees = [
    {employeeId: ‘E0001’, name: ‘Alex’,
    department: ‘D002’},
    {employeeId: ‘E0002’, name: ‘Jil’,
    department: ‘D001’}
  ];
  var emp = employees.find(e =>
  e.employeeId === empId);

  if(emp)
    return Promise.resolve(emp);
  else
    return Promise.
    reject({error:’Invalid employee
    id’});
  }

function getDepartment(deptId){
  var departments = [
    {departmentId: ‘D001’, name:
    ‘Accounts’},
    {departmentId: ‘D002’, name: ‘IT’}
  ];

  var dept = departments.find(d =>
  d.departmentId === deptId);

  if(dept)
    return Promise.resolve(dept);
  else
    return Promise.
  reject({error:’Invalid department
  id’});
}
```

Now let’s write a function that would get details of an employee and then get details of the department. So the logic of getting department depends on result of the promise returned by the *getEmployee* function. Here’s the snippet that does this operation:

```
getEmployee(employeeId).then(function
(employee) {
    console.log(employee);
    return getDepartment(employee.
department);
}, function(error){
    return error;
})
.then(function (department) {
    console.log(department);
}, function(error){
    console.log(error);
});
```

Though the above snippet has an asynchronous task depending on another, it still looks sequential and hence easily manageable. Similarly, if there is a need to perform another asynchronous operation using department object, it can be chained with promise returned from the second *then* method.

Combining Generators and Promises

Till now we saw how promises and generators can be used separately to compose asynchronous operations. We can use the powers of both of these features together to build some advanced and sophisticated asynchronous APIs.

One of the use cases that I can think of where promises and generators play well together is, *polling*. Polling is a technique using which a client keeps pinging the server for data. The server responds with data when it is available.

We can use a generator to respond with data each time we receive a response from the server. Here as we don’t have a server setup, let’s use *Promise.resolve* to respond with different results on every poll. The polling function would return promises on every iteration.

Following is a sample polling function that runs five times:

```
function *polling(){
  var count = 0, val={};
  do{
    count++;
    if(count !== 4){
      yield Promise.resolve({});
    }
    else{
      yield Promise.resolve({value:
      “Some value”});
    }
  }while(count<5);
}
```

Now let’s write a recursive function that runs the above function and uses its values. As the polling function returns a generator, we will call the next iteration after every one second.

Following is the *runPolling* function and its first call:

```
function runPolling(pollingIterator){
  if(!pollingIterator){
    pollingIterator = polling();
  }
  setTimeout(function(){
    let result = pollingIterator.next();
    if(result.value) {
      result.value.then(function (data)
      {
        console.log(data);
      });
    }
    if(!result.done) {
      runPolling(pollingIterator);
    }
  }, 1000);
}

runPolling();
```

If you run this code, it would keep printing results on the console with a gap of one second each.

Generators and promises can also be combined when we need multiple and dependent asynchronous operations to be performed by one function and result of each operation has to be returned to the calling function.

Conclusion

As we saw, Promises and Generators come in handy to design and use asynchronous APIs and still make the code look sequential and synchronous. These features help in interacting with any asynchronous browser APIs or any library APIs. Down the line, we may see some of the browser APIs using these features ■

Download the entire source code from GitHub at bit.ly/dncm18-es6async

About the Author

Ravi Kiran is a developer working on Microsoft Technologies. These days, he spends his time on the front-end JavaScript framework Angular JS and server frameworks like ASP.NET Web API and SignalR. He actively writes what he learns on his blog at sravi-kiran.blogspot.com. He is a DZone MVB. You can follow him on twitter at @sravi_kiran

SCREENCAPTURE IN WINDOWS PHONE 8.1

Introduction

Microsoft has released a wide range of APIs for Windows and Windows Phone. Some of these APIs can be used on both Platforms using the Universal App Development Model. However there are some features that are limited to an individual platform only, one of them being the ScreenCapture feature which is currently available only on Windows Phone 8.1.

With these APIs, developers across the world are building some unique Windows Store apps every day. Being app developers, although we've a variety of devices at our disposal and some fully functional Windows and Windows Phone emulators to test our apps; we still share a Beta version of our apps to a limited set of people, to get feedback. When it comes to Games and other critical enterprise apps, it is important to capture the behavior of the end user. It is important to capture how they interact with the app, how they navigate and which options they choose frequently on a screen. If you have used Microsoft Test Management Studio, it actually records the screen which helps the QA team to file an exact bug by using this recording. This is also helpful for developers to check the recording and follow the steps to reproduce the bugs and fix the same.

Picture Courtesy: Enterely

For Windows Phone apps, if you want to test and observe how users are interacting with your apps, then recording the screen is a good option here. You may have come across various desktop recording tools like Camtasia Studio and Expression encoder, but there aren't many tools available out-of-the-box from Microsoft for Windows Phone.

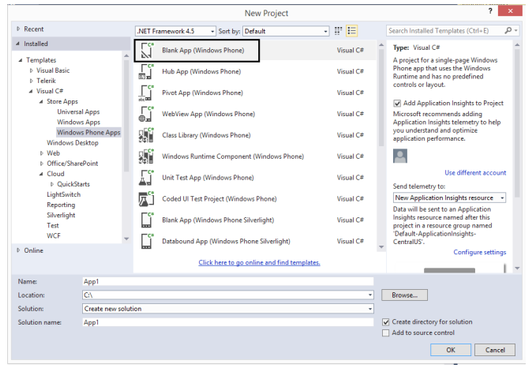
Luckily we have the ScreenCapture API that allows us to build such functionality in our apps. Though the API (Namespace) we are going to use for our example, is available in Desktop application development as well; but the classes and methods we will talk about in this article, are currently only available in Windows Phone 8.1. Let's explore this API in detail.

Namespace used for ScreenCapture class

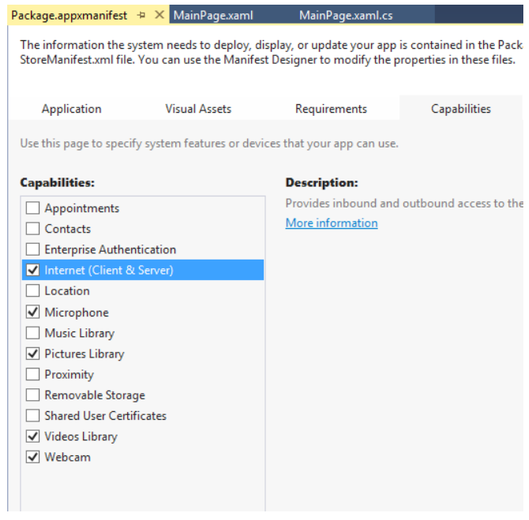
The *ScreenCapture* class falls under the namespace "Windows.Media.Capture". Although this namespace is also available for Windows Store Applications to capture photos and perform other image related tasks; the ScreenCapture class is only made available for Windows Phone 8.1. To be more specific, it is available on Windows Phone Silverlight 8.1 and Windows Runtime (WinRT) apps as well.

Screen Recording Application for Windows Phone 8.1 (WinRT)

I have already mentioned that the ScreenCapture feature is available for both Windows Phone 8.1 WinRT and Silverlight Project. However here we will take WinRT as our base project. For this, you must have Visual Studio 2013 Update 3 and above which consists of the tools and APIs required in the Visual Studio update package.



Once you create a new Project, It is important to make the following changes in the Capabilities tab in Package.appxmanifest :



Now lets do our XAML design. In this example, we are building a simple RTO (Regional Transport Office) Vehicle Registration Screen with a couple of Textboxes and a DatePicker. I have used a simple layout, but you can design the XAML according to your business scenario.

Step 1 : Build XAML

Here is our XAML Design

```
<Grid>
 <StackPanel
 HorizontalAlignment=”Stretch”
 Margin=”24” >
  <TextBlock Text=”RTO – Vehicle
  Details” Margin=”0,0,0,24”
  VerticalAlignment=”Center”
  FontSize=”20” />

  <TextBlock Text=”Customer
  Name”  VerticalAlignment=”Center”
  FontSize=”20”/>
  <TextBox Margin=”0,0,0,12” />
  <TextBlock Text=”Registration
  Number”  VerticalAlignment=”Center”
  FontSize=”20”/>

  <TextBox Margin=”0,0,0,12” />
  <TextBlock Text=”Date of
  Purchase”  VerticalAlignment=”Center”
  FontSize=”20”/>
  <DatePicker Margin=”0,0,0,12”
  Orientation=”Horizontal” />

  <TextBlock Text=”Vehicle
  Type”  VerticalAlignment=”Center”
  FontSize=”20”/>
  <ToggleSwitch OffContent=”Light Motor
  Vehicle” OnContent=”Heavy Motor
  Vehicle” />

  <TextBlock Text=”Vehicle
  Number”  VerticalAlignment=”Center”
  FontSize=”20”/>
  <TextBox Margin=”0,0,0,12” />
 </StackPanel>
</Grid>

<Page.BottomAppBar>
 <CommandBar>
  <AppBarToggleButton x:Name=”btnRecord”
  Icon=”Play” Label=”Capture”
  Click=”btnRecord_Click”/>
 </CommandBar>
</Page.BottomAppBar>
```

Step 2: Understanding the structure of Windows.Media.Capture Namespace

The following namespaces need to be added:

```
using System;
using System.Threading.Tasks;
using Windows.Media.Capture;
using Windows.Media.MediaProperties;
using Windows.Storage;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;
```

Here Windows.Media.Capture is our core namespace which is widely used in Windows Store and Windows Phone apps for capturing Image. However in our example, we are using the *ScreenCapture* class which is part of this namespace. If you open the definition of the ScreenCapture class, you will be able to see the following methods and members details:

```
#region Assembly Windows.winmd, v255.255.255.255
// C:\Program Files (x86)\Windows Phone Kits\8.1\References\CommonConfiguration\Neutral\Windows.winmd
#endregion

using System;
using Windows.Foundation;
using Windows.Foundation.Metadata;
using Windows.Media.Core;

namespace Windows.Media.Capture
{
    // Summary:
    //     Enables an app to capture audio and video of the contents being displayed
    //     on the device.
    [MarshalingBehavior(MarshalingType.Agile)]
    [SupportedOn(100859904, Platform.WindowsPhone)]
    [Version(100859904)]
    public sealed class ScreenCapture
    {
        ...public IMediaSource AudioSource { get; }
        ...public bool IsAudioSuspended { get; }
        ...public bool IsVideoSuspended { get; }
        ...public IMediaSource VideoSource { get; }

        ...public event TypedEventHandler<ScreenCapture, SourceSuspensionChangedEventArgs> SourceSuspensionChanged;

        // Summary:
        //     Gets the ScreenCapture object associated with the app's current view.
        //
        // Returns:
        //     The ScreenCapture object associated with the app's current view.
        [SupportedOn(100859904, Platform.WindowsPhone)]
        public static ScreenCapture GetForCurrentView();
    }
}
```

Let’s build the recording functionality using this ScreenCapture class.

Step 3: Building ScreenCapturing functionality and storing output as Video file

We need to globally define a variable of type *MediaCapture* which will be used across methods.

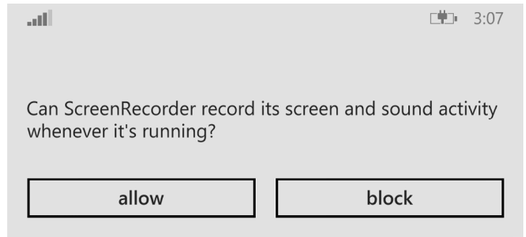
```
private MediaCapture mCap;
```

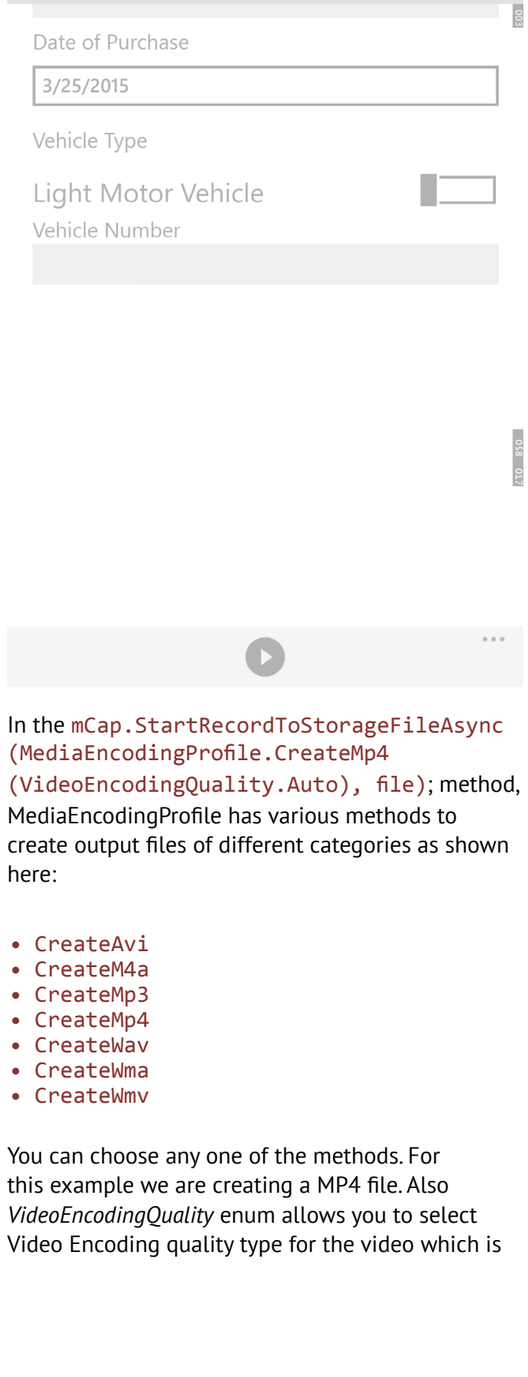
First we will see how we can start the recording.

Check the following code which will be part of the button click placed in the App Bar of the application. This will first initialize the current screen to be captured with ScreenCapture. `GetForCurrentView();` method.

```
//Record the Screen
private async void btnRecord_Click
(object sender,
Windows.UI.Xaml.RoutedEventArgs e)
{
 if (btnRecord.IsChecked.HasValue &&
 btnRecord.IsChecked.Value)
 {
  // Initialization - Set the current
  screen as input
  var scrCaptre =
  ScreenCapture.GetForCurrentView();

  mCap = new MediaCapture();
  await mCap.InitializeAsync(new

  MediaCaptureInitializationSettings
  {
   VideoSource = scrCaptre.VideoSource,
   AudioSource = scrCaptre.AudioSource,
  });

  // Start Recording to a File and set
  the Video Encoding Quality
  var file = await GetScreenRecVdo();
  await mCap.
  StartRecordToStorageFileAsync
  (MediaEncodingProfile.CreateMp4
  (VideoEncodingQuality.Auto), file);
 }

 else
 {
  await StopRecording();
 }
}
```

Note that the capturing functionality has to be triggered based on consent provided by end user. Once you start your app, you can either “allow” or “block” capturing functionality. Once you allow, you cannot then block it again unless you uninstall and reinstall and enforce blocking option. The following screen will help you to understand the consent part once you run the application for the first time.





In the `mCap.StartRecordToStorageFileAsync (MediaEncodingProfile.CreateMp4 (VideoEncodingQuality.Auto), file)`; method, MediaEncodingProfile has various methods to create output files of different categories as shown here:

- `CreateAvi`
- `CreateM4a`
- `CreateMp3`
- `CreateMp4`
- `CreateWav`
- `CreateWma`
- `CreateWmv`

You can choose any one of the methods. For this example we are creating a MP4 file. Also *VideoEncodingQuality* enum allows you to select Video Encoding quality type for the video which is

being captured. Currently it is kept at *Auto*

- `Auto`
- `HD1080p`
- `HD720p`
- `Wvga`
- `Ntsc`
- `Pal`
- `Vga`
- `Qvga`

For stopping the recording, we have the `StopRecordAsync()` method as shown here:

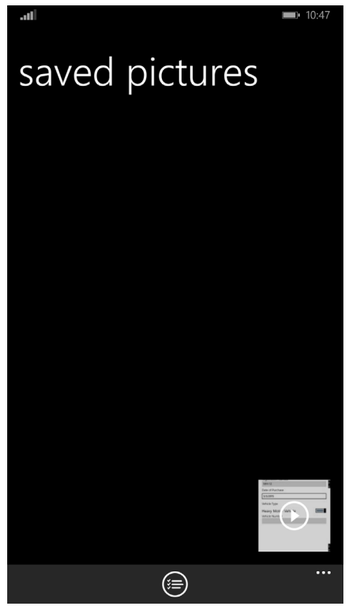
```
private async Task StopRecording()
{
 // Stop recording and dispose resources
 if (mCap != null)
 {
  await mCap.StopRecordAsync ();
  mCap.Dispose();
  mCap = null;
 }
}
protected override async void
OnNavigatedFrom(NavigationEventArgs e)
{
 await StopRecording();
}
```

Step 4: Set the Target Output Folder and Test the Screen Recorder

The following method will help you to set the Video file name and destination folder where you want to save the file. Note that we are not storing Video on an SD card, we are storing it in the Phone storage. To store the video on an SD card, you need to make some code changes and also turn on "Removable Storage" capability from Package.appxmanifest file.

```
//Create File for Recording on Phone/
Storage
private static async Task<StorageFile>
GetScreenRecVdo(CreationCollisionOption
creationCollisionOption =
CreationCollisionOption.ReplaceExisting)
{
  return await KnownFolders.
  SavedPictures.
  CreateFileAsync("VehicleDetails.mp4",
  creationCollisionOption);
}
```

KnownFolders class from Windows.Storage gives you the flexibility to choose the destination location to store Video File. Out-of-the-box, the following locations are present as StorageFolder



Step 5: Watch the Screen Recording

Now you can watch the video from your Phone and see how screen capturing is done. If you do not have a device and still want to test, you can do it on the emulator by storing the file in some folder on the emulator and then accessing it and displaying it in the MediaElement control.

```
<MediaElement x:Name="OutPutScreen"
VerticalAlignment="Bottom"
HorizontalAlignment="Stretch"
MaxHeight="280" Margin="24" />
```

You can then access the video from the SavedPictures once you stop recording. You can put this code right after Stop Recording or in a separate Method.

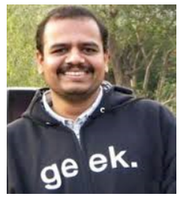
```
var file = await
GetScreenRecVdo(CreationCollisionOption.
OpenIfExists);
OutPutScreen.SetSource(await file.
OpenReadAsync(), file.ContentType);
```

Summary

With the ScreenCapture class, you can build simple Screen Capturing functionality for your application. This functionality can be used to understand nature and behavior of end users of an application. You can also use it to understand bugs in an application and determine enhancements that can be made to fix them. This is specifically useful for critical enterprise applications and games where User interaction is considered as a key aspect for the success of the app and used for benchmarking as well as to record user satisfaction. This also helps to generate analytical data by capturing different types and levels of interaction of users with the app. I hope you will try this unique feature of Screen Capture for your app in the pre-release phase to understand your end audience and how they interact with your app ■

Download the entire source code from GitHub at
bit.ly/dncm18-screencapture

About the Author



Vikram Pendse is currently working in Avanade (Accenture, India) for .NET, Azure & Windows Phone Projects. He is responsible for Providing Estimates, Architecture, Supporting RFPs and Deals. He is a Microsoft MVP since 2008 and currently a Windows Phone MVP. He is a very active member in various Microsoft Communities and participates as a 'Speaker' in many events. You can follow him on Twitter at: @VikramPendse

WHAT'S NEW IN ASP.NET WEB FORMS IN .NET 4.6

With the release of the latest iteration of the .NET Framework edging closer and closer, the Microsoft development community has been buzzing about all of the new features, changes and improvements coming to MVC, but what about Web Forms and its future? Is it going the way of Classic ASP or Silverlight? Should current Web Forms developers be worried?

No, they shouldn't.

Today, we will discuss Web Forms' role within the new development ecosystem, introduce some of the latest features available within Web Forms applications and demonstrate that the future of Web Forms is still a bright one.

Framework Changes and the Role of Web Forms

With the release of .NET 5, the Microsoft development ecosystem is going to experience quite a bit of change. The latest version of the widely popular runtime framework was a complete re-write of .NET from the ground-up. This project was undertaken to accomplish various architectural goals (i.e. cross platform applications, increased modularity etc.), provide additional flexibility to its developers and ultimately provide the best possible technology for building modern web applications.

Many of these changes revolved around the familiar System.Web.dll, which is a core component of Web Forms and an integral part of web development within the .NET framework. The rewrite removed this dependency and introduced all kinds of new revolutionary features for more recent technologies like ASP.NET MVC, but it essentially removed the core component of Web Forms and left it with an uncertain future.

This scenario probably sounds a bit scary for Web Forms developers. Why would Microsoft seemingly abandon its most mature and widely used development technology? The answer is that they aren't. While everyone has been discussing the new changes in .NET 5, Microsoft has also been hard at work developing .NET 4.6, another version of the framework to continue to support and improve Web Forms moving forward.

> ***Note:*** *It should be noted that all of the code and features discussed within this article was written using Preview builds of Visual Studio 2015, Windows 10 and the .NET Framework. As a result of this, syntax, features and functionality may change prior to the publication of this article or the release builds of any of the aforementioned products. Some of the new features might not "just work" at present, but they are expected to work within the final release.*

Web Forms Improvements in .NET 4.6

There are several new features available for Web Forms within .NET 4.6 that will focus on improving performance, development time and efficiency within Web Forms applications. Some of the features that will be focused on within this article include:

- HTTP2 Support
- Asynchronous Model Binding
- Roslyn Code DOM Compilers

Developers can take advantage of some of these features by downloading the latest version of Visual Studio 2015 (currently in Preview), which comes packaged with the latest versions of both branches of the framework: .NET 4.6 and .NET 5.0.

HTTP2 Support to Improve Performance

HTTP is a protocol that has been around for quite a while and needless to say, the landscape of the web has changed significantly during that period. Web sites and applications are far more complex than they were in the 90s and the protocol that handles this information interchange needed a make-over to adapt to these changing demands. This is where HTTP2 comes in with a single goal in mind: to improve web site performance.

Based on Google's SPDY protocol, HTTP2 was designed with improving the performance and load times of sites by reducing latency as much as possible through header compression, leveraging server-based push technologies and parallel loading support for page elements over a single connection. The positive impact from these changes can be immediately seen when using a browser that supports the HTTP2 protocol and the benefits should keep both servers and end-users happy.

While reading about these benefits might sound nice, it's much more favorable to actually see the impact that they can have on a web site or application. In the next section, we will walk through an example of using HTTP2 within a Web Forms application running on IIS in .NET 4.6.

Seeing HTTP2 in Action within Web Forms Applications

In order to take advantage of HTTP2, we will need to build an application that is capable of supporting it. Which means that the following dependencies will be required for this example:

- Windows 10 Preview – At the time of writing, IIS only supports HTTP2 while running on Windows 10.

- Visual Studio 2015 – Visual Studio 2015 is required for building .NET 4.6 applications.

Open Visual Studio 2015 and create a new Web Forms project through the following steps:

1. Open **Visual Studio 2015** on a machine running Windows 10.

2. Create a new Web Forms project through the **File > New Project** dialog.

3. **Ensure you are targeting .NET Framework 4.6** through the available dropdown in the New Project

dialog.

4. **Name and create your new ASP.NET Web Application project** by choosing it from the list of available templates.

5. **Choose the Empty Project template and add the appropriate Web Forms references by clicking the Web Forms checkbox.**

The next step will be to construct a very basic Web Forms page that consists of a collection of images. The actual content will not make much of a difference at all, we merely want to see how the requests differ between HTTP and HTTP2.

An example Web Form to demonstrate this might appear as follows:

```
<%@ Page Language="C#"
AutoEventWireup="true"
CodeBehind="Default.aspx.cs"
Inherits="HTTP2Example.HTTP2Example" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/
xhtml">
<head runat="server">
  <title>HTTP2 Support in .NET 4.6</
title>
</head>
<body>
  <form id="form1" runat="server">
    <!-- Define all of your images below
    -->
    <img src="Images/A.png" />
    <img src="Images/B.png" />
    <img src="Images/C.png" />
    <img src="Images/D.png" />
    <!-- Additional images omitted for
    brevity -->
  </form>
</body>
</html>
```

Next, if you run the application within Visual Studio, you should be presented with it in your default browser as expected. We aren't terribly concerned with the actual appearance of the page at this point, we are more interested in what is going on behind the scenes. To take a glimpse into this, we will take advantage of using the Developer Tools available within most modern browsers.

With the previously created Web Form open, *press F12 within the browser to display the Developer Tools* and *select the Network tab* that appears along the top, and refresh the page. You'll see that initial request will be made with the browser parsing all of the HTML, and then the requests for the images will begin coming through:

Figure 1: Requests loaded through HTTP

By examining the requests that were loaded through HTTP in Figure 1, you can see that the first set of images were loaded as expected, however any subsequent requests required previous requests to complete prior to beginning.

Most browsers can generally handle a small number of concurrent connections to a single domain at a given time, which can be fine for most scenarios. However, for a page that contains a large number of elements (or at least more than the number of maximum connections), the other elements will have to wait for an available connection before being loaded. The example above demonstrates that even while loading just eleven images, it took more than three seconds before the final image even began to download and would be far worse as the number of resources increased. Any delays because of this can often be perceived as slowness by the end-user, when in fact it is simply a limitation of the browser and protocol.

> ***Note:*** *In most browsers, HTTP2 will be supported by default and used if available. If you are running IIS and using Internet Explorer 11, it should be noted that HTTP2 is only supported over secure connections at present. In other browsers, you may need to explicitly enable it.*

Let's now see what type of benefits and improvements that switching the protocol over to HTTP2 yields for the exact same page. In order to take advantage of the protocol, we will need to **change the protocol within the address bar from http to https** and **refresh the page**, which should enable Internet Explorer to meet its requirements to use HTTP2 as follows:

Figure 2: Requests loaded through HTTP2

As you can see in Figure 2, the Developer Tools yield a significantly different shape for requests made using HTTP2. You can see that all of the requests were created and loaded in *parallel*, eliminating any delays from the perspective of the user.

As the HTTP2 request loads, we can see that the Developer Tools now displays a less staggered appearance and may indicate that the HTTP 2 protocol was used for each request. This indicates that all of the requested images were loaded in parallel as opposed to synchronously. Further examination will reveal that the final image began downloading in just 10 milliseconds, which is a significant improvement over HTTP and one that will certainly be noticed by end users.

The introduction of HTTP2 support with .NET 4.6 and IIS should yield significant performance benefits on both existing and future Web Forms applications with little to no changes on the part of the developer.

Leveraging New Language Features in Web Forms using the Roslyn Code DOM Compiler

The Roslyn compiler is arguably one of the cornerstones of a majority of the widespread changes and advances within the .NET ecosystem over the past few years and it's a focal point for the upcoming changes within both .NET 4.6 and .NET 5.0. Roslyn began as an attempt to re-write the previously available compilers (and their related services) for both C# and Visual Basic actually using those respective languages. This would allow Roslyn to actually expose information regarding what was going on behind the scenes during the compilation process and even provide APIs to allow the developer to access this same data.

Exposure to these APIs has provided developers with the ability to generate and further optimize their existing code by receiving feedback from the compiler itself and allows for really cool features such as dynamic compilation, which opens all sorts of new doors for developing ASP.NET applications like the available Code DOM compilers.

In .NET 4.6, Web Forms applications will now use the Roslyn Code DOM compilers natively, which allows developers to write code and take advantage of many of the new available language features such as those found within C# 6.0 that might not otherwise be available using the previous compiler.

Compilation on the Fly within Web Forms in 4.6

Prior to .NET 4.6, Web Forms applications would use the framework-based Code DOM compilers to perform any compilation operations at runtime. This older compiler doesn't currently understand how to use any of the newer language features available like Roslyn does, so you'll be bombarded with errors if you attempt to use any of them without it.

Let's take a look at a very basic ASP.NET Web Forms application that attempts to use a few of the following C# 6.0 features:

```
<%@ Page Language="C#"
AutoEventWireup="true"
CodeBehind="Default.aspx.cs"
Inherits="RoslynCodeDOMExample.
RoslynCodeDOMExample" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/
xhtml">
<head runat="server">
 <title>Roslyn Code DOM Compiler Support
 in .NET 4.6</title>
</head>
<body>
 <form id="form1" runat="server">
  <pre>Example One: String
  Interpolation</pre>
  <%
  // Define two example variables to
  concatenate using C# 6.0 String
  Interpolation
  var firstName = "Rion";
  var lastName = "Williams";

  // Attempt to use the C# 6.0 String
```

```
Interpolation feature for formatting
strings
var output = $"{lastName},
{firstName}";
%>

<!-- Write out the result -->
<%= output %>
<pre>Example Two: Exception Filters
</pre>
<%
 // Define a variable that will be
 "caught" by in the Exception Filter
 var x = 42;

 try
 {
  // Define a null object
  object result = null;
  // Trigger a null reference
  exception
  var stringifiedResult = result.
  ToString();
 }

 catch(NullReferenceException ex) if
 (x == 42){
  // This will explicitly handle the
  exception
  Response.Write("The exception was
  handled in the Null Reference block
  as x was 42.");
 }

 catch(Exception ex)
 {
  // Otherwise fall through to this
  block and handle here
  Response.Write($"x was {x} and not
  42, so it was handled in the generic
  catch block.");
 }
 %>

 <hr />
 <pre>Example Three: nameof
 Expressions</pre>
 <%
 // The nameof keyword can be used to
 help avoid the use of "magic strings"
 within your C# code by
 // resolving the appropriate type of
 a particular object as a string.
 var nameofExample = nameof(Request.
 Cookies);
 %>
 <%= nameofExample %>
</form>
</body>
</html>
```

If your application isn't currently running on .NET 4.6 or it isn't leveraging the Roslyn compiler through another route such as a NuGet package, your application should build, but then it will throw a compilation error when the newer language features are loaded:

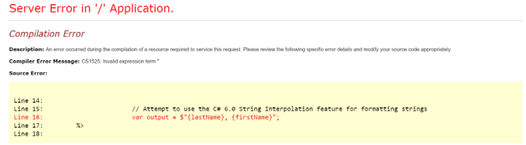


Figure 3: Compilation errors will be thrown when new features aren't recognized

On the other hand, if we use that same application and target .NET 4.6, we can see that it compiles just as it did previously, however when it hits the newer C# 6.0 features, it uses Roslyn to compile and handle this new code on the fly and process it as expected:

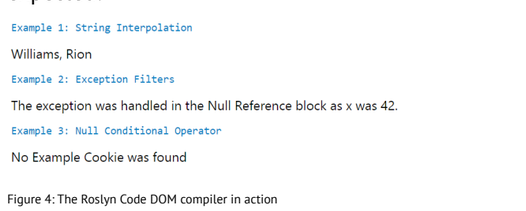


Figure 4: The Roslyn Code DOM compiler in action

Just when you thought your application would see the new language features that you are using and throw a compilation error, the Roslyn Code DOM compiler steps in and compiles these new features on the fly without issue.

While a feature like String Interpolation probably isn't a game-changer, it is just one of the many new features introduced in C# 6.0. Since you now have access to all of these via the Code DOM compilers, try experimenting with some of the more practical ones like the null-conditional operator and exception filters.

It should be noted that you should be able to use the Microsoft.CodeDom.Providers.DotNetCompilerPlatform NuGet package available from within your application if you aren't currently targeting .NET 4.6 or beyond to add this additional functionality in without committing to a new framework version.

Added Support for Asynchronous Model Binding

.NET 4.5 introduced the popular task-based async / await methods that have allowed developers to easily create asynchronous functionality within their applications. This same version of the framework also gave rise to model binding as well, which allows a variety of data-specific controls (e.g. GridViews, Repeaters, ListViews, etc.) to be populated by actual methods as opposed to data sources. Both of these changes provided developers with an improved flexibility about how they go about designing their applications and use them.

With the release of .NET 4.6, both of these concepts are expanded upon with the introduction of asynchronous model binding support. This added support allows you to decorate the various methods that govern your data sources with async and await calls, thereby creating a more asynchronous and efficient application.

Binding Data to Entity Framework Easily and Asynchronously

Although ADO.NET connections can still be seen in a wide range of applications, the rise of ORMs in recent years has made interacting with data much easier without being a SQL wizard. Entity Framework and Code First provide several built-in methods that allow developers to interact with their data asynchronously. For this reason, asynchronous model binding can be a helpful feature for developers working with Entity Framework as well as those that simply want to make their data transactions a bit more efficient.

Getting started with asynchronous model binding is quite easy and doesn't demand too much work on your part for either new or existing applications. You'll just need to ensure that the following things happen:

- **Ensure that your application is targeting .NET 4.6 or higher** so that you can take advantage of this feature.
- **Set the Async directive set to "true" at the top of any of your pages** that are targeting asynchronous methods.
- **Decorate any asynchronous methods with the appropriate async / await keywords** (i.e. add *async* to the signature of the method and preface any asynchronous calls within it using *await*).

We can demonstrate this same functionality by constructing a single ASPX page that hooks into an imaginary data context and handles all of the necessary CRUD operations that you commonly find within an application in an asynchronous manner:

```
<%@ Page Language="C#"
AutoEventWireup="true" Async="true"
CodeBehind="Default.aspx.cs" Inherits=
"AsynchronousModelBindingExample.Default"
%>
<!DOCTYPE html>
<html xmlns=
"http://www.w3.org/1999/xhtml">
<head runat="server">
 <title>Asynchronous Model Binding
 Support in .NET 4.6</title>
</head>
<body>
 <form id="form1" runat="server">
  <!-- Define a GridView to wire up all
  of the available methods for the Grid
  (e.g. Select, Update, Delete) -->
  <asp:GridView ID="WidgetsGrid"
  runat="server"
  DataKeyNames="WidgetID" ItemType=
  "AsynchronousModelBindingExample
  .Models.Widget"
  SelectMethod="GetWidgets"
  UpdateMethod="UpdateWidget"
  DeleteMethod="DeleteWidget">
  <Columns>
   <asp:DynamicField DataField=
   "WidgetID" />
   <asp:DynamicField DataField=
   "WidgetName" />
   <asp:TemplateField HeaderText=
   "Working Parts">
    <ItemTemplate>
     <asp:Label Text="<%#
     Item.Parts.Count(p =>
     p.IsFunctioning) %>"
runat="server">
     </asp:Label>

</ItemTemplate>
  </asp:TemplateField>
```

```
  </Columns>
 </asp:GridView>
 </form>
</body>
</html>
```

There are a few things worth noting within this otherwise traditional GridView, especially if you are unfamiliar with the Model Binding that was introduced in .NET 4.5:

- **ItemType** – This is a strongly-typed class that corresponds to the objects that will be used to populate the rows within the Grid.

- **Select, Update and Delete Methods** – Each of these methods is going to point to a server-side method defined within the code-behind that is used to handle performing the respective operation (e.g. retrieving data, updating or deleting records).

- **Item within the ItemTemplate** – The Item keyword can be used within Item templates to function as a strongly-typed object, similar to the iterator within a foreach loop through a collection.

We can take a look behind the scenes at the code-behind to see the asynchronous events that each of these various methods (e.g. Select, Update and Delete) correspond to and how they are built within an asynchronous environment:

```
using AsynchronousModelBindingExample.
Models;
using System;
using System.Data.Entity;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Web.UI.WebControls;

namespace
AsynchronousModelBindingExample
{
 public partial class Default : System.
Web.UI.Page
 {
  protected void Page_Load(object
  sender, EventArgs e)
  {

  }
```

```
public async Task<SelectResult>
GetWidgets()
{
 // Create your data context to
 access your data (replace with your
 existing context
 // in your real-world application)
 using (var context = new
 ImaginaryContext())
 {
  // Build a query to pull your
  widgets (including the Parts related
  to each Widget)
  var query = context.Widgets.
  Include("Parts");
  // Return a SelectResult to bind
  the data to the GridView (using the
  asyncronous
  // ToListAsync() method exposed by
  Entity Framework)
  return new SelectResult(query.
  Count(), await query.ToListAsync());
 }
}
public async Task UpdateWidget(Guid
widgetId, CancellationToken token)
{
 using (var context = new
 ImaginaryContext())
 {
  // Attempt to retrieve your current
  Widget through the context
  var widget = await context.Widgets.
  FirstAsync(w => w.WidgetID ==
  widgetId);

  // Check if the requested Widget was
  found
  if (widget != null)
  {
   // Update your Widget here, adding
   any custom logic as necessary

   TryUpdateModel(widget);
   // Ensure the model is valid, if so
   update and save the Widget
   if (ModelState.IsValid)
   {
    await context.SaveChangesAsync();
   }
  }
  // Otherwise, the widget was not
  found, display an error to the user
  ModelState.AddModelError("", $"The
  Widget with ID '{widgetId}' was not
  found");
 }
}
```

```
public async Task DeleteWidget(Guid
widgetId, CancellationToken token)
{
 using (var context = new
 ImaginaryContext())
 {
  // Attempt to retrieve your current
  Widget through the context
  var widget = await context.Widgets.
  FirstAsync(w => w.WidgetID ==
  widgetId);

  // Check if the requested Widget was
  found
  if (widget != null)
  {
   // Delete the requested Widget
   context.Widgets.Remove(widget);
   await context.SaveChangesAsync();
  }
  // Otherwise, the widget was not
  found, display an error to the user
  ModelState.AddModelError("", $"The
  Widget with ID '{widgetId}' was not
  found");
 }
}
```

That's really it. After updating the return types for each of the methods to leverage Task objects and indicating which asynchronous methods should be awaited, everything should just work. The addition of these asynchronous features will allow developers to quickly and easily take advantage of the benefits that the async / await keywords provide with relatively little change to their applications.

Summary

The improvements that were introduced within this article should hopefully serve as a reminder that Web Forms **is still** an important component of the ASP.NET ecosystem and that Microsoft is dedicated to continuing to support and improve upon it. We can take one last glance to see exactly how each of them affected the Web Forms developer:

- **HTTP2 Protocol Support** should aid in improving performance and reducing server latency for all Web Forms applications moving forward; a benefit that should be felt by both servers and end-users alike.

- The **Roslyn Code DOM Compiler** will allow developers to take advantage of brand-new language features found in both C# and Visual Basic and easily integrate them into their applications on the fly.

- **Asynchronous Model Binding** now provides Web Forms developers with the ability to easily take advantage of the benefits of Model Binding coupled with async / await support to build more efficient applications.

These recent enhancements, coupled with the entire 4.6 branch being dedicated to preserving it, should allow Web Forms developers to breathe a sigh of relief. Web Forms isn't dying. While it may no longer have the spotlight shown upon it as it once did, it's continuing to improve and it looks like it will still play an active part in .NET development for the foreseeable future ■

Download the entire source code from GitHub at
bit.ly/dncm18-aspwebforms

About the Author

Rion Williams is a Senior Software Developer and Microsoft MVP with a passion for building cool applications, helping fellow developers within the community and staying on the edge emerging development trends. If you want to see what he is up to, you can follow him Twitter as @rionmonster, visit his blog at rion.io or if its important enough, send him an e-mail (e-mail address left off as an exercise for the reader).