JavaScript Web Tools

with Visual Studio 2015 and ASP.NET 5

April 2015.

Microsoft

Readme

Welcome! This document contains a set of articles that introduce the most commonly used JavaScript/frontend web development tools (libraries, task runners, frameworks…). Frontend development ecosystem is growing fast and getting complex, and we hope this brief doc will help you put the pieces together and decide on the ones that best fit your needs.

You'll be able to find updated versions of these articles (and more!) on the asp.net page at http://aka.ms/webdevdocs.

You can get a good understanding just by reading the different articles, but we really recommend you follow the step by step samples to gain exposure to these tools, we created the samples with the free Visual Studio 2015 Preview.

We hope you find it useful!! If you have any question, comment or feedback about the articles, we'll be listening on the **#jswebtoolsdoc** hashtag on twitter.

- Developer Platform team at Microsoft

Contents

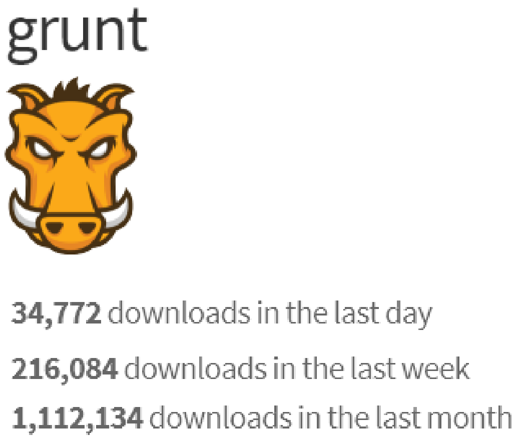
Grunt and Gulp. Task Runners.

Both Grunt and Gulp are JavaScript task runners that automate script minification, TypeScript compilation, code quality “lint” tools, CSS pre-processors, and just about any repetitive chore that needs doing to support client development. Grunt is the Visual Studio 2015 default, but Gulp can be used in its place.

What’s the difference between Grunt and Gulp? Grunt is an early entry in the client-build-tool space. Grunt modules predefine most everyday tasks like linting, minimizing, and unit testing. Grunt is widely adopted and downloaded thousands of times each day.

While Gulp is a later entry to the field, Gulp has gained popularity for crisp performance and elegant syntax. While Grunt tends to read and write files to disk, Gulp uses a stream (Vinyl) object to pipe method results to following methods, allowing calls to be chained together in a fluent syntax.

The statistics below are a snapshot from the npmjs (node package manager) home site downloads.

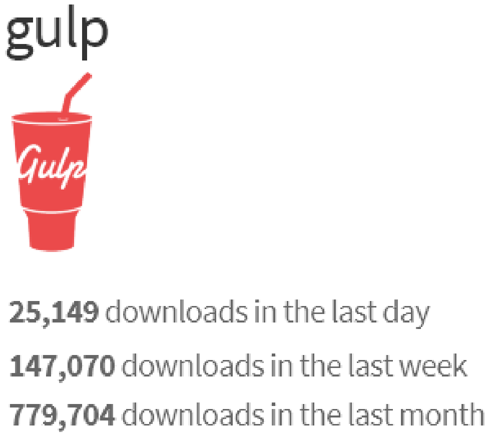


grunt

34,772 downloads in the last day

216,084 downloads in the last week

1,112,134 downloads in the last month



gulp

25,149 downloads in the last day

147,070 downloads in the last week

779,704 downloads in the last month

Using Grunt

Plumbing for NPM (Node Package Manager) and Grunt are ded in the Starter Web project template. This example uses the Empty Web template to show how to automate the client build process from scratch.

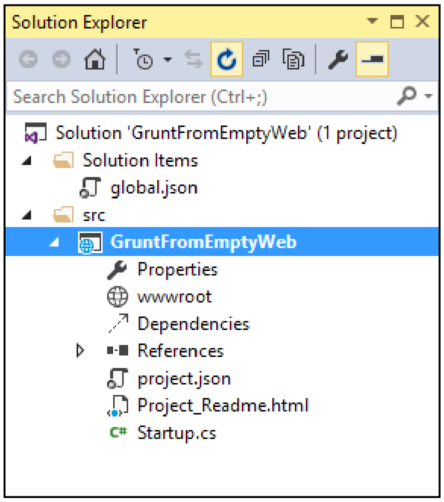
The finished example cleans the target deployment directory, combines JavaScript files, checks code quality, condenses JavaScript file content and deploys to the root of your web application. We will use the following packages:

- **grunt**: The Grunt task runner package.
- **grunt-contrib-clean**: A task that removes files or directories.
- **grunt-contrib-jshint**: A task that reviews JavaScript code quality.
- **grunt-contrib-concat**: A task that joins files into a single file.
- **grunt-contrib-uglify**: A task that minifies JavaScript to reduce size.
- **grunt-contrib-watch**: A task that watches file activity.

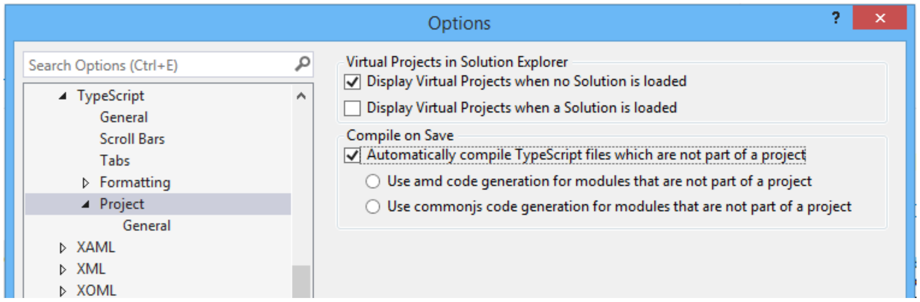
Preparing the Application

To begin, set up a new empty web application and add TypeScript example files. TypeScript files are automatically compiled into JavaScript using default Visual Studio 2015 settings and will be our raw material to process using Grunt.

1. In Visual Studio 2015, create a new **ASP.NET Web Application**.
2. In the **New ASP.NET Project** dialog, select the **ASP.NET 5 Empty** template and click the **OK** button.
3. In the Solution Explorer, review the project structure. The **\src** folder includes empty **wwwroot** and **Dependencies** nodes.



4. Add a new folder named **TypeScript** to your project directory.
5. Before adding any file, let's make sure that Visual Studio 2015 has the option 'compile on save' for TypeScript files checked. *Tools > Options > Text Editor > Typescript > Project*



Virtual Projects in Solution Explorer

☑ Display Virtual Projects when no Solution is loaded

☐ Display Virtual Projects when a Solution is loaded

Compile on Save

☑ Automatically compile TypeScript files which are not part of a project

○ Use amd code generation for modules that are not part of a project

○ Use commonjs code generation for modules that are not part of a project

6. Right-click the **TypeScript** directory and select **Add > New** Item from the context menu. Select the **JavaScript file** item and name the file **Tastes.ts** (note the *.ts extension). Copy the line of TypeScript code below into the file (when you save, a new Tastes.js file will appear with the JavaScript source).

```
enum Tastes { Sweet, Sour, Salty, Bitter }
```

7. Add a second file to the **TypeScript** directory and name it **Food.js**. Copy the code below into the file.

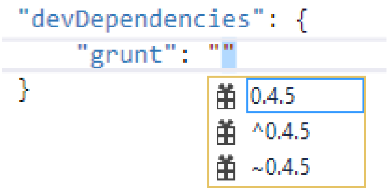
```
class Food {
    constructor(name: string, calories: number) {
        this._name = name;
        this._calories = calories;
    }

    private _name: string;
    get Name() {
        return this._name;
    }

    private _calories: number;
    get Calories() {
        return this._calories;
    }

    private _taste: Tastes;
    get Taste(): Tastes { return this._taste }
    set Taste(value: Tastes) {
        this._taste = value;
    }
}
```

Configuring NPM

Next, configure NPM to download *grunt* and *grunt-tasks*.

1. In the Solution Explorer, right-click the project and select **Add > New Item** from the context menu. Select the **NPM configuration file** item, leave the default name, **package.json**, and click the **Add** button.

2. In the package.json file, inside the **devDepencies** object braces, enter “grunt”. Select **grunt** from the Intellisense list and press the **Enter** key. Visual Studio will quote the grunt package name, and add a colon. To the right of the colon, select the latest stable version of the package from the top of the Intellisense list (press **Ctrl-Space** if Intellisense does not appear).



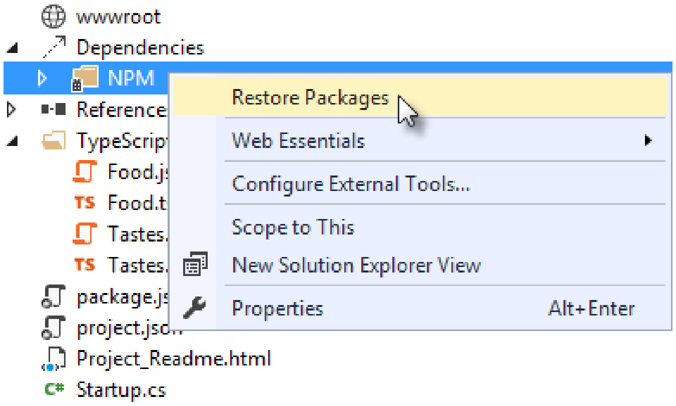
```
"devDependencies": {
    "grunt": ""
}
```

Note: NPM uses semantic versioning to organize dependencies. Semantic versioning, also known as *SemVer*, identifies packages with the numbering scheme **<major>.<minor>.<patch>**. Intellisense simplifies semantic versioning by showing only a few common choices. The top item in the Intellisense list (**0.4.5** in the example above) is considered the latest stable version of the package. The carat ^ symbol matches the most recent major version and the tilde ~ matches the most recent minor version. See the NPM semver version parser reference as a guide to the full expressivity that SemVer provides.

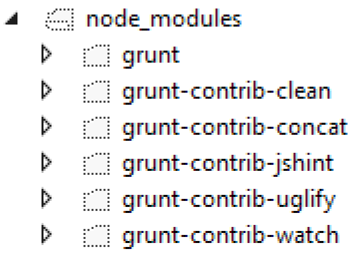
3. Add more dependencies to load grunt-contrib* packages for *clean*, *jshint*, *concat*, *uglify* and *watch* as shown in the example below. The versions do not need to match the example.

```
"devDependencies": {
    "grunt": "0.4.5",
    "grunt-contrib-clean": "0.6.0",
    "grunt-contrib-jshint": "0.11.0",
    "grunt-contrib-concat": "0.5.1",
    "grunt-contrib-uglify": "0.8.0",
    "grunt-contrib-watch": "0.6.1"
}
```

4. Save the **packages.json** file.
5. In Solution Explorer, right-click **Dependencies\NPM** and select the **Restore Packages** context menu option.



The packages for each devDependencies item will download, along with any files that each package requires. You can find the package files in the **node_modules** directory by enabling the **Show All Files** button in the Solution Explorer.



node_modules
- grunt
- grunt-contrib-clean
- grunt-contrib-concat
- grunt-contrib-jshint
- grunt-contrib-uglify
- grunt-contrib-watch

Configuring Grunt

Grunt is configured using a manifest named **gruntfile.js** that defines, loads and registers tasks that can be run manually or configured to run automatically based on events in Visual Studio.

1. Right-click the project and select **Add > New Item**. Select the **Grunt Configuration file** option, leave the default name, **Gruntfile.js**, and click the **Add** button.

   The initial code includes a module definition and the **grunt.initConfig()** method. The initConfig() is used to set options for each package, and the remainder of the module will load and register tasks.

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

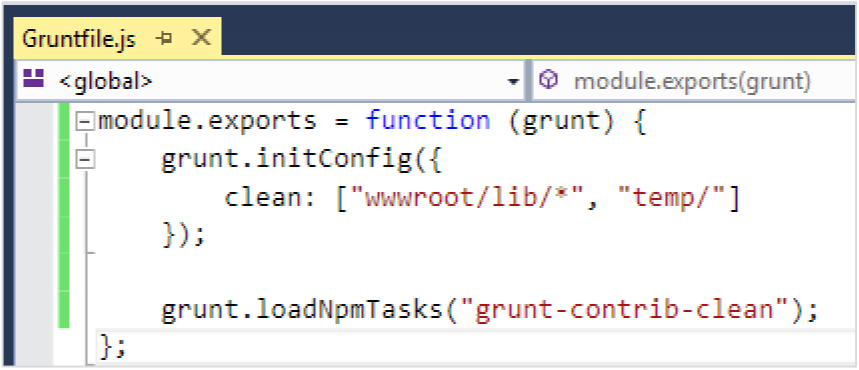
2. Inside **the initConfigt()** method, add options for the **clean** task as shown in the example Gruntfile.js below. The clean task accepts an array of directory strings. This task removes files from wwwroot/lib and removes the entire /temp directory.

```
module.exports = function (grunt) {
    grunt.initConfig({
        clean: ["wwwroot/lib/*", "temp/"],
    });
};
```

3. Below the initConfig() method, add a call to **grunt.loadNpmTasks()**. This will make the task runnable from Visual Studio.

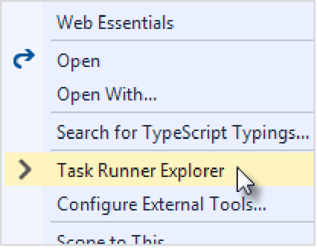
```
grunt.loadNpmTasks("grunt-contrib-clean");
```

4. Save Gruntfile.js. The file should look something like the screenshot below.

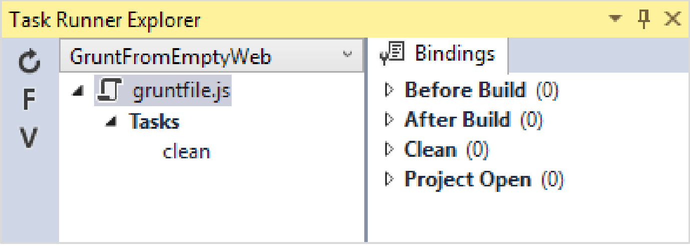


```
module.exports = function (grunt) {
    grunt.initConfig({
        clean: ["wwwroot/lib/*", "temp/"]
    });

    grunt.loadNpmTasks("grunt-contrib-clean");
};
```

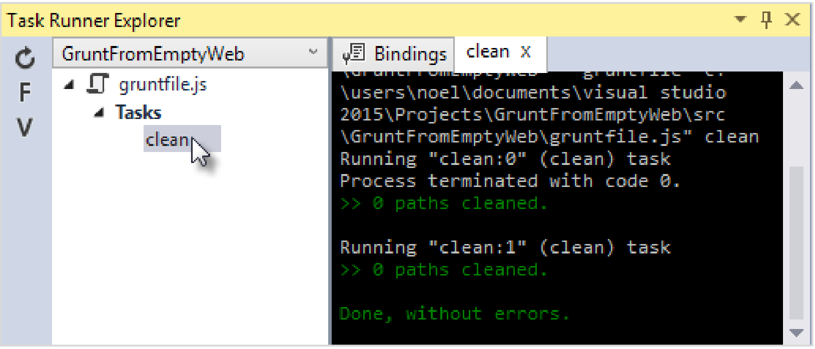
5. Right-click Gruntfile.js and select **Task Runner Explorer** from the context menu. The Task Runner Explorer window will open.



6. Verify that **clean** shows under **Tasks** in the Task Runner Explorer.



7. Right-click the **clean** task and select **Run** from the context menu. A command window displays progress of the task.



```
\users\noel\documents\visual studio
2015\Projects\GruntFromEmptyWeb\src
\GruntFromEmptyWeb\gruntfile.js" clean
Running "clean:0" (clean) task
Process terminated with code 0.
>> 0 paths cleaned.

Running "clean:1" (clean) task
>> 0 paths cleaned.

Done, without errors.
```

Note: There are no files or directories to clean yet. If you like, you can manually create them in the Solution Explorer and then run the clean task as a test.

8. In the initConfig() method, add an entry for **concat** using the code below.

   The src property array lists files to combine, in the order that they should be combined. The dest property assigns the path to the combined file that is produced.

```
concat: {
    all: {
        src: ['TypeScript/Tastes.js', 'TypeScript/Food.js'],
        dest: 'temp/combined.js'
    }
},
```

Note: The **all** property in the code above is the name of a *target*. Targets are used in some Grunt tasks to allow multiple build environments. You can view the built-in targets using Intellisense or assign your own.

9. Add the **jshint** task using the code below.

   The jshint code-quality utility is run against every JavaScript file found in the temp directory.

```
jshint: {
    files: ['temp/*.js'],
    options: {
        '-W069': false,
    }
},
```

Note : The option "-W069" is an error produced by jshint when JavaScript uses bracket syntax to assign a property instead of dot notation, i.e. **Tastes["Sweet"]** instead of **Tastes.Sweet**. The option turns off the warning to allow the rest of the process to continue.

10. Add the **uglify** task using the code below.

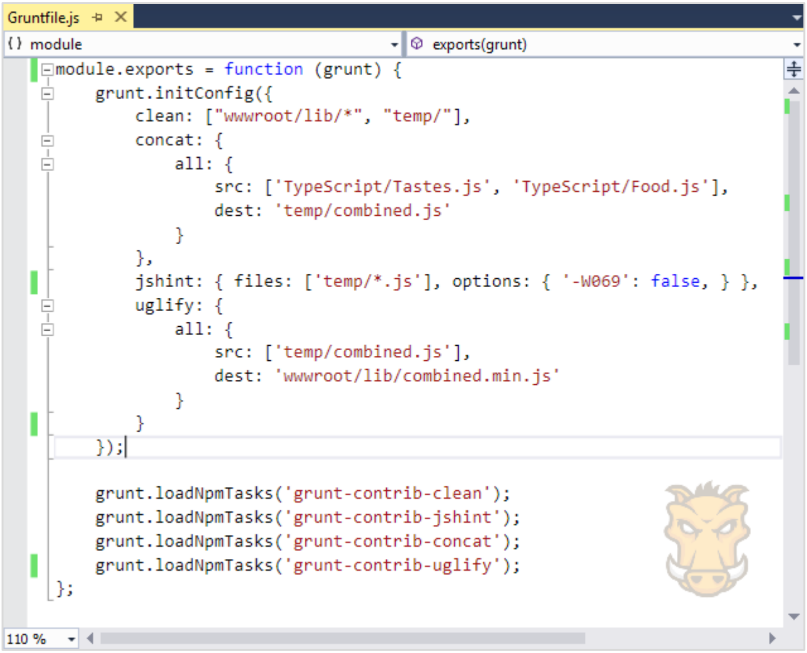
    The task minifies the combined.js file found in the temp directory and creates the result file in wwwroot/lib following the standard naming convention *<file name>.min.js*.

```
uglify: {
    all: {
        src: ['temp/combined.js'],
        dest: 'wwwroot/lib/combined.min.js'
    }
},
```

11. Under the call grunt.loadNpmTasks() that loads grunt-contrib-clean, include the same call for jshint, concat and uglify using the code below.

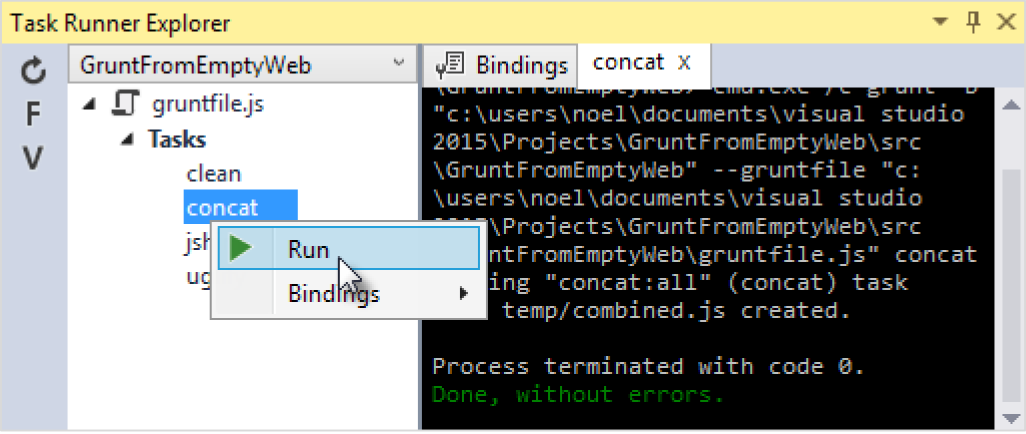
```
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-concat');
grunt.loadNpmTasks('grunt-contrib-uglify');
```

12. Save **Gruntfile.js**. The file should look something like the example below.



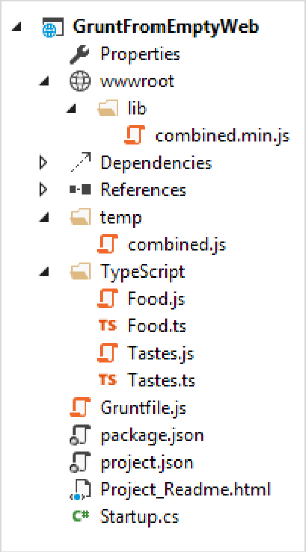
```
module.exports = function (grunt) {
    grunt.initConfig({
        clean: ["wwwroot/lib/*", "temp/"],
        concat: {
            all: {
                src: ['TypeScript/Tastes.js', 'TypeScript/Food.js'],
                dest: 'temp/combined.js'
            }
        },
        jshint: { files: ['temp/*.js'], options: { '-W069': false, } },
        uglify: {
            all: {
                src: ['temp/combined.js'],
                dest: 'wwwroot/lib/combined.min.js'
            }
        }
    });

    grunt.loadNpmTasks('grunt-contrib-clean');
    grunt.loadNpmTasks('grunt-contrib-jshint');
    grunt.loadNpmTasks('grunt-contrib-concat');
    grunt.loadNpmTasks('grunt-contrib-uglify');
};
```

13. Notice that the Task Runner Explorer Tasks list includes **clean**, **concat**, **jshint** and **uglify** tasks. Run each task in order and observe the results in Solution Explorer. Each task should run without errors.



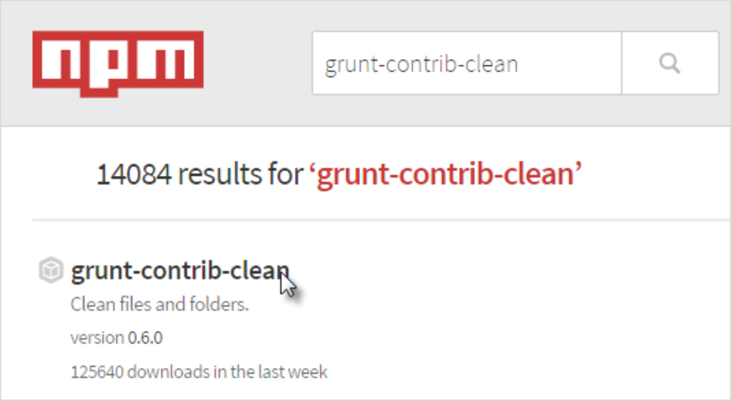
```
\GruntFromEmptyWeb> cmd.exe /c grunt -b
"c:\users\noel\documents\visual studio
2015\Projects\GruntFromEmptyWeb\src
\GruntFromEmptyWeb" --gruntfile "c:
\users\noel\documents\visual studio
    \Projects\GruntFromEmptyWeb\src
   ntFromEmptyWeb\gruntfile.js" concat
   ing "concat:all" (concat) task
    temp/combined.js created.

Process terminated with code 0.
Done, without errors.
```

The concat task creates a new combined.js file and places it into the temp directory. The jshint task simply runs and doesn't produce output. The uglify task creates a new combined.min.js file and places it into wwwroot\lib. On completion, the solution should look something like the screenshot below:



```
◢ GruntFromEmptyWeb
    Properties
  ◢ wwwroot
    ◢ lib
        combined.min.js
  ▷ Dependencies
  ▷ References
  ◢ temp
      combined.js
  ◢ TypeScript
      Food.js
      Food.ts
      Tastes.js
      Tastes.ts
    Gruntfile.js
    package.json
    project.json
    Project_Readme.html
    Startup.cs
```

Note: For more information on the options for each package, visit https://www.npmjs.com/ and lookup the package name in the search box on the main page. For example, you can look up the **grunt-contrib-clean** package to get a documentation link that explains all the parameters.



14084 results for 'grunt-contrib-clean'

grunt-contrib-clean

Clean files and folders.

version 0.6.0

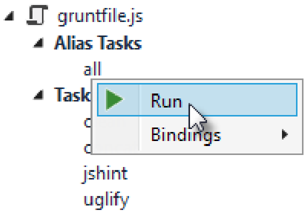
125640 downloads in the last week

All Together Now

Use the Grunt **registerTask()** method to run a series of tasks in a particular sequence. For example, to run the example steps above in the order clean -> concat -> jshint -> uglify, add the code below to the module. The code should be added to the same level as the loadNpmTasks() calls, outside initConfig.

```
grunt.registerTask("all", ['clean', 'concat', 'jshint', 'uglify']);
```

The new task shows up in Task Runner Explorer under Alias Tasks. You can right-click and run it just as you would other tasks. The **all** task will run **clean**, **concat**, **jshint** and **uglify**, in order.



Watching for Changes

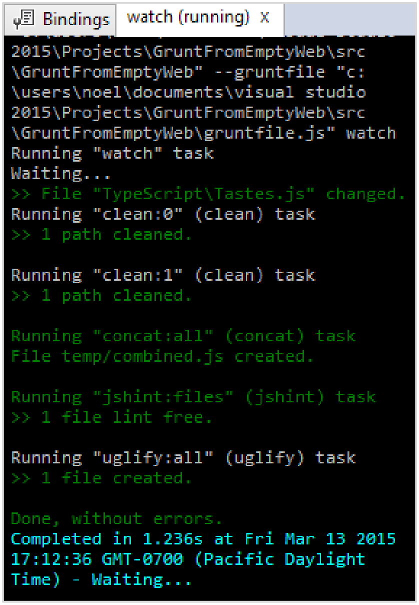
A **watch** task keeps an eye on files and directories. The watch triggers tasks automatically if it detects changes. Add the code below to initConfig to watch for changes to *.js files in the TypeScript directory. If a JavaScript file is changed, **watch** will run the **all** task.

```
watch: {
    files: ["TypeScript/*.js"],
    tasks: ["all"]
}
```

Add a call to **loadNpmTasks()** to show the **watch** task in Task Runner Explorer.

```
grunt.loadNpmTasks('grunt-contrib-watch');
```

Right-click the **watch** task in Task Runner Explorer and select **Run** from the context menu. The command window that shows the watch task running will display a **waiting…** message. Open one of the TypeScript files, add a space, and then save the file. This will trigger the **watch** task and trigger the other tasks to run in order. The screenshot below shows a sample run.

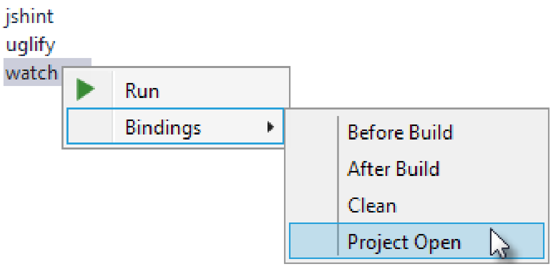


```
2015\Projects\GruntFromEmptyWeb\src
\GruntFromEmptyWeb" --gruntfile "c:
\users\noel\documents\visual studio
2015\Projects\GruntFromEmptyWeb\src
\GruntFromEmptyWeb\gruntfile.js" watch
Running "watch" task
Waiting...
>> File "TypeScript\Tastes.js" changed.
Running "clean:0" (clean) task
>> 1 path cleaned.

Running "clean:1" (clean) task
>> 1 path cleaned.

Running "concat:all" (concat) task
File temp/combined.js created.

Running "jshint:files" (jshint) task
>> 1 file lint free.

Running "uglify:all" (uglify) task
>> 1 file created.

Done, without errors.
Completed in 1.236s at Fri Mar 13 2015
17:12:36 GMT-0700 (Pacific Daylight
Time) - Waiting...
```

Binding to Visual Studio Events

Unless you want to manually start your tasks every time you work in Visual Studio, you can bind tasks to **Before Build**, **After Build**, **Clean**, and **Project Open** events.

Let's bind **watch** so that it runs every time Visual Studio opens. In Task Runner Explorer, right-click the watch task and select **Bindings > Project Open** from the context menu.



Unload and reload the project. When the project loads again, the watch task will start running automatically.

Using Gulp

Gulp configuration is similar to Grunt with some notable differences. The example below parallels the Grunt example using Gulp packages and conventions.

NPM Package Differences

The **devDependencies** defined in package.json are specific to Gulp. To get the same result as the Grunt walk-through, **package.json** should look something like the code below. You will need to change the package versions in the devDependencies list to the latest version. You can get the correct version number using Intellisense (Ctrl-space).

```
{
    "version": "1.0.0",
    "name": "GulpFromEmptyWeb",
    "private": true,
    "devDependencies": {
        "gulp": "3.8.11",
        "gulp-clean": "0.3.1",
        "gulp-jshint": "1.9.2",
        "gulp-concat": "2.5.2",
        "gulp-uglify": "1.1.0",
        "gulp-rename": "1.2.0"
    }
}
```

Gulpfile vs Gruntfile Examples

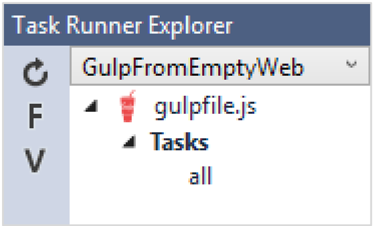
Instead of adding Gruntfile.js to the project, add a JavaScript file to the project and name it **gulpfile.js**. In **gulpfile.js,** assign a series of objects using the node.js **require()** method. Make the assignment for Gulp itself and for every package needed for automation. The code below assigns the same tasks used in the Grunt example:

```
var gulp = require('gulp');
var clean = require('gulp-clean');
var concat = require('gulp-concat');
var jshint = require('gulp-jshint');
var uglify = require('gulp-uglify');
```

Below these assignments in gulpfile.js, call the **gulp** object **task()** method. The first parameter to task() is the name of the task and the second is a function.

```
gulp.task("all", function () {

});
```

Just adding the empty task() method to gulpfile.js displays the **all** task in Task Runner Explorer.



Inside the **task()** function, use the objects defined earlier by **require()** to do the work. The example below cleans any files from the wwwroot/lib directory.

```
gulp.task("all", function () {
     gulp.src('wwwroot/lib/*').pipe(clean());
});
```

The Gulp Stream

Gulp is a streaming object that includes methods **src()**, **pipe()** and **dest()**.

- **src()** defines where the stream is coming from -- wwwroot/lib in our example. The method returns a stream that can be passed to other Gulp plugins.
- **pipe()** pulls data from the stream and writes it to the destination parameter.
- **dest()** outputs streams to files.

The general coding pattern for Gulp looks like this partial example:

```
gulp.src()
    .pipe()
    .pipe()
    .pipe(dest());
```

The src() method gathers the initial raw materials. A series of pipe() calls allow Gulp plugins to operate on the stream. Finally, the dest() method writes out the final results. The advantage to this flow is that only one file read and one file write occur, making the whole process quicker.

All Together

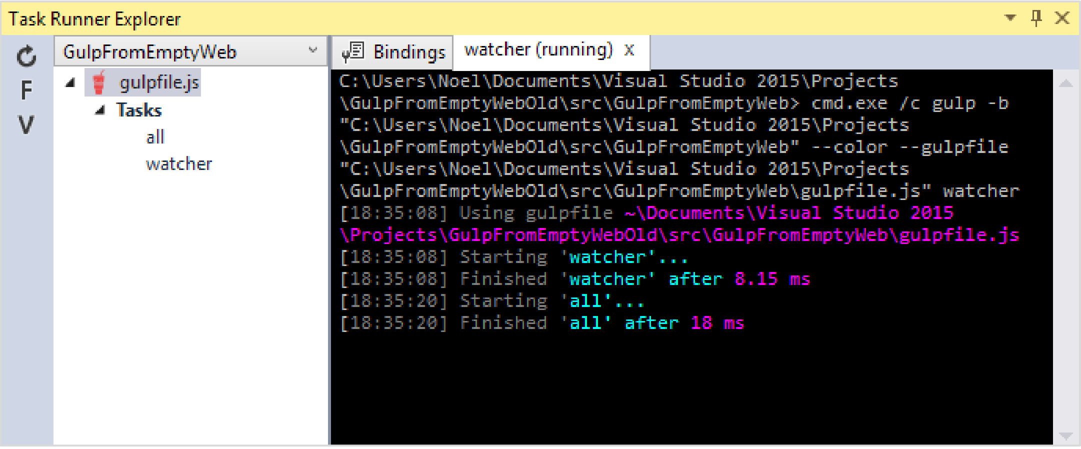
Here's the complete example that concatenates, lints, minifies and writes out the minified file. The processing time is quite fast.

```
gulp.task("all", function () {
     gulp.src('wwwroot/lib/*').pipe(clean());
     gulp.src(['TypeScript/Tastes.js', 'TypeScript/Food.js'])
        .pipe(concat("combined.js"))
        .pipe(jshint())
        .pipe(uglify())
        .pipe(rename({
            extname: '.min.js'
         }))
        .pipe(gulp.dest('wwwroot/lib'))
});
```

Watcher tasks are similar to the Grunt parallel task and are simple to set up. Again, the gulp.task() method names the task that will show in the Task Runner Explorer. The Gulp **watch()** method takes a path or array of paths and second parameter is an array of tasks to run.

```
gulp.task("watcher", function () {
    gulp.watch("TypeScript/*.js", ['all']);
});
```

The Task Runner Explorer running Gulp tasks uses the same interface as Grunt. The screenshot below shows the **watcher** task running.



```
C:\Users\Noel\Documents\Visual Studio 2015\Projects
\GulpFromEmptyWebOld\src\GulpFromEmptyWeb> cmd.exe /c gulp -b
"C:\Users\Noel\Documents\Visual Studio 2015\Projects
\GulpFromEmptyWebOld\src\GulpFromEmptyWeb" --color --gulpfile
"C:\Users\Noel\Documents\Visual Studio 2015\Projects
\GulpFromEmptyWebOld\src\GulpFromEmptyWeb\gulpfile.js" watcher
[18:35:08] Using gulpfile ~\Documents\Visual Studio 2015
\Projects\GulpFromEmptyWebOld\src\GulpFromEmptyWeb\gulpfile.js
[18:35:08] Starting 'watcher'...
[18:35:08] Finished 'watcher' after 8.15 ms
[18:35:20] Starting 'all'...
[18:35:20] Finished 'all' after 18 ms
```

Summary

Both Grunt and Gulp are powerful tasks runners that automate most client-build tasks. Grunt and Gulp both require support from NPM to deliver their packages. While Grunt is configured using Gruntfile.js and Gulp is configured using Gulpfile.js, both build tools play nicely in Visual Studio, automatically sensing changes to the configuration files. Task Runner Explorer detects changes to configuration files and provides a convenient interface to run tasks, view running tasks, and bind tasks to Visual Studio events.

Bower. Package Manager.

Bower is a package manager that delivers groups of files used client-side at run-time. For example, with Bower you can install CSS, fonts, frameworks, and JavaScript libraries from external sources. Bower resolves dependencies and will automatically download and install all the packages you need. For example, if you configure Bower to load the Bootstrap package, the proper jQuery package will automatically come along for the ride.

Visual Studio developers are already familiar with the NuGet package manager, so why not use NuGet instead of adding yet another tool? Mainly because Bower already has a rich eco-system with about 18 thousand packages in play and because ASP.NET is no longer strictly a Windows and Visual Studio space. Bower is accessible within Visual Studio, but also from the command line and IDEs for other environments.

You could also use NPM (Node Package Manager), but NPM typically loads client development, debugging, and testing tools like Less, JSLint, QUnit, and Grunt. While NPM uses a nested dependency tree, Bower has a flatter structure and for that reason tends to be lighter weight. For example, in NPM you can easily have different versions of the same component while Bower doesn't allow two components with the same name.

> *"Bower is optimized for the front-end. Bower uses a flat dependency tree, requiring only one version for each package, reducing page load to a minimum."*
>
> *– from the http://bower.io main page*

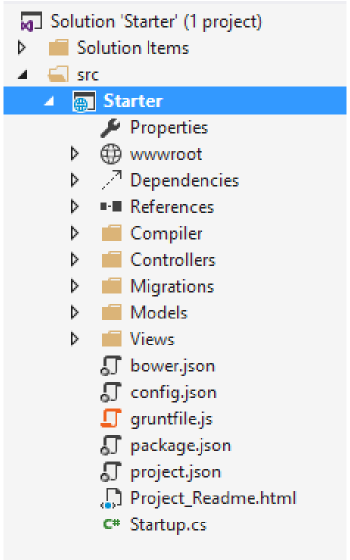
How does Bower Work – the Short Story

Bower works together with the NPM and Grunt (a client-side task runner). NPM first loads Grunt and a Grunt task that supports Bower. Grunt then installs Bower. Finally, Bower resolves dependencies, downloads packages, and deploys the packages to the root of your web site.

Bower packages are essentially Git repositories. Each package has a **bower.json** file that describes the package, such as the name, version, and dependencies. You can add a bower.json file to your own application that defines the packages your application needs.

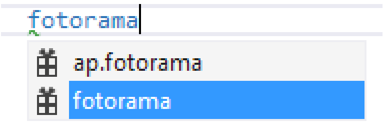
Getting Started

The ASP.NET 5 Starter Web MVC project pre-constructs the client build process for you. The ubiquitous jQuery and Bootstrap packages are installed, and plumbing for NPM, Grunt, and Bower are already in-place. The screenshot below shows the initial project in the Solution Explorer.



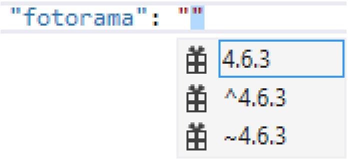
To use Bower, open the **bower.json** file. In the ASP.NET 5 Starter Web project, bower.json is ready to load jQuery, jQuery validation, the Bootstrap framework, and Hammer (supports touch gestures in web pages). Let's add support for photo albums by installing the Fotorama jQuery plugin with Bower.

1. At the end of the **dependencies** properties listing in bower.json, add a comma and type "fotorama". Select "fotorama" from the drop down list.



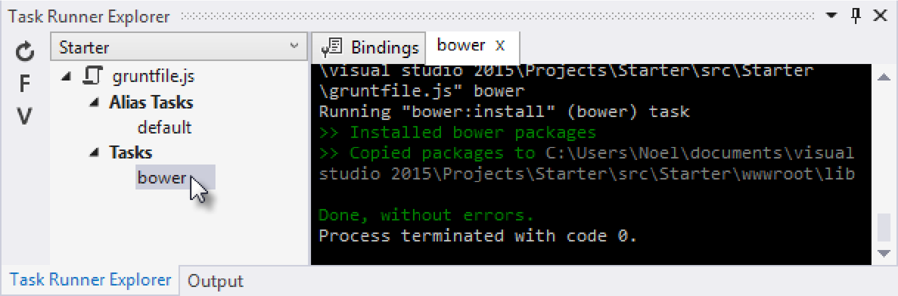
```
fotorama
```

2. Add a colon and then select the latest stable version of the package from the drop down list.



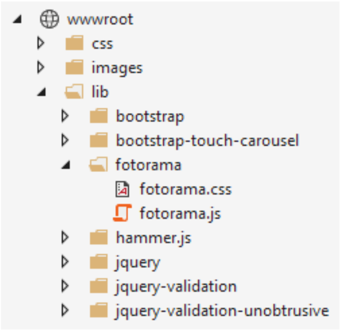
```
"fotorama": ""
```

3. Save the **bower.json** file.
4. In the Solution Explorer, right-click the **Dependencies\Bower** node and select **Restore Packages** from the drop down menu.
5. Navigate to the Task Runner Explorer by either right-clicking **gruntfile.js** and selecting the context menu option or selecting the **View > Other Windows > Task Runner Explorer** menu option.
6. Double-click the **Tasks > bower** option to trigger the Bower deployment.



```
\visual studio 2015\Projects\Starter\src\Starter
\gruntfile.js" bower
Running "bower:install" (bower) task
>> Installed bower packages
>> Copied packages to C:\Users\Noel\documents\visual
studio 2015\Projects\Starter\src\Starter\wwwroot\lib

Done, without errors.
Process terminated with code 0.
```

7. In the Solution Explorer, open the **wwwroot** node. The **\lib** directory should have all the dependencies listed in bower.json including the **\fotorama** directory you just added.



8. In the Solution Explorer, right-click the **wwwroot** node and select **Add > New Item** from the context menu. Choose the **HTML Page** item, name the page Index.html, and click the **Add** button.
9. Add references to the packages deployed by Bower to the web page.
   - Drag **fotorama.css** from **wwwroot\lib\fotorama** to the head of the page.
   - Drag **jquery.js** to the end of the body tag.
   - Drag **fotorama.js** underneath jquery.js.

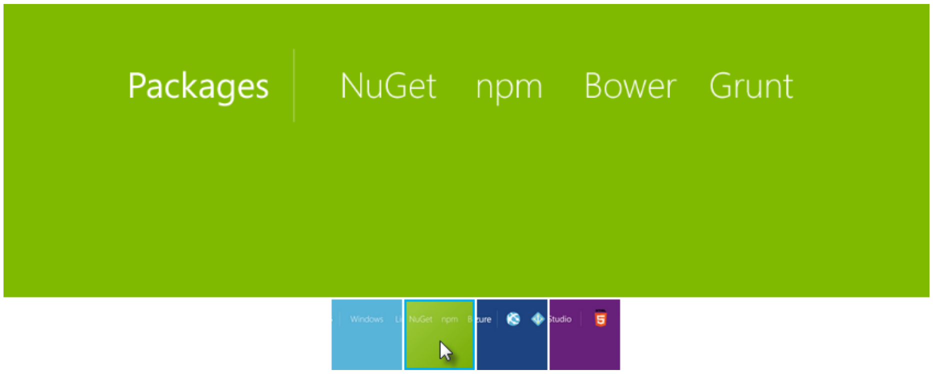
   The page should now look like something like this:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Bower and Fotorama</title>
    <link href="lib/fotorama/fotorama.css" rel="stylesheet" />
</head>
<body>

    <script src="lib/jquery/jquery.js"></script>
    <script src="lib/fotorama/fotorama.js"></script>
</body>
</html>
```

10. Inside the body tag, above the script references, add a **div** element with a class **“fotorama”**. Inside the div element, add a number of **img** tags. The example uses images currently available inside wwwroot\images, but you can add any images on hand.

```
<div class="fotorama" data-nav="thumbs">
    <img src="images/asp-net-banners-01.png" />
    <img src="images/asp-net-banners-02.png" />
    <img src="images/banner-01-azure.png" />
    <img src="images/banner-02-vs.png" />
</div>
```

11. Press **Ctrl-Shift-W** to display the page in the browser. The control displays the images and allows navigation by clicking the thumbnail list below the main image. This quick smoke test demonstrates that Bower deployed the correct packages and dependencies.



Exploring the Client Build Process

The Visual Studio 2015 ASP.NET 5 Starter Web project has everything you need all set up, but it's difficult from a standing start to see how the component processes go together. This next walk-through starts with an ASP.NET Empty project template and adds each piece manually so you can get feel for how Bower is used in a project. See what happens to the project structure and the run-time output as each configuration change is made to the project.

The general steps to use the client-side build process with Bower are:

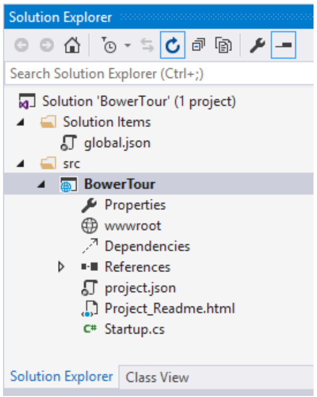
- Define and download packages used in your project.
- Install the packages to the root of your web application.
- Reference packages from your web pages.

Define Packages

The first step is to define the packages your application needs and download them. This example uses Bower to load jQuery and Bootstrap. Start by configuring NPM to install *design-time* packages such as the Grunt task runner. Then use Grunt to run Bower so that Bower installs *run-time* packages jQuery and Bootstrap.

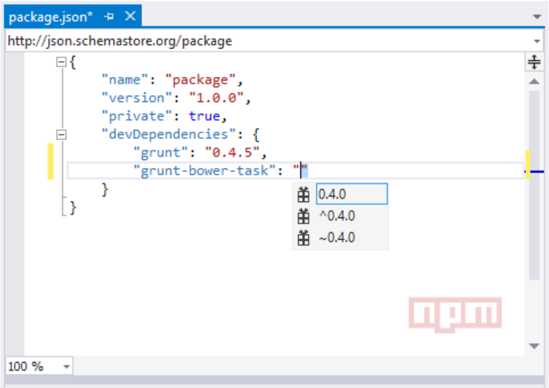
1. In Visual Studio 2015, create a new **ASP.NET Web Application**.
2. In the **New ASP.NET Project** dialog, select the **ASP.NET 5 Empty** template and click the **OK** button.

3. In the Solution Explorer, the empty project **\src** directory now includes a **project.json** file, and a new **wwwroot** and **Dependencies** nodes. The project directory will look like the screenshot below, where the **Properties** and **wwwroot** directories are empty.



4. In the Solution Explorer, enable the **Show All Files** button. At this point, there are no hidden files that are not part of the project.
5. In the Solution Explorer, right-click the project and add
   - NPM configuration file – Package.json
   - Grunt configuration file – Gruntfile.js
   - Bower configuration file – Bower.json

6. The **package.json** file is the NPM package definition that loads all the files, include the grunt and grunt-bower-task dependencies.



```
{
    "name": "package",
    "version": "1.0.0",
    "private": true,
    "devDependencies": {
        "grunt": "0.4.5",
        "grunt-bower-task": ""
    }
}
```

7. In the **gruntfile.js** we should define a task that runs Bower. This is used later to manage run-time packages, like jQuery or Bootstrap, on the client. The **grunt.initConfig** task options dictate that files be copied to the **wwwroot/lib** directory. Grunt loads the **grunt-bower-task** that triggers Bower to install packages to your web application.

```
       module.exports = function (grunt) {
           grunt.initConfig({
               bower: {
                   install: {
                       options: {
                           targetDir: "wwwroot/lib",
                           layout: "byComponent",
                           cleanTargetDir: false
                       }
                   }
               },
           });

           grunt.registerTask("default", ["bower:install"]);

           grunt.loadNpmTasks("grunt-bower-task");
};
```

8. In the Solution Explorer, right-click the **Dependencies\NPM** node and click on restore packages.
9. In the Solution Explorer, view the restored packages:
    - Open the **Dependencies\NPM\ grunt** node to see all packages that Grunt depends on.
    - Open the **node_modules** directory to view the files copied to your local machine during the package restoration.

> **Note**: If you don't see the node_modules directory, make sure that the **Show All Files** button is enabled in Solution Explorer's toolbar.

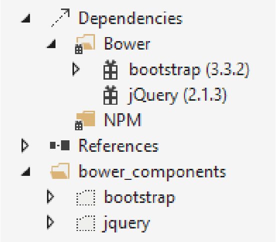
10. Open **bower.json** and remove the **exportsOverride** section for the time being. We will replace it later after you see how Bower deploys files without this section.

11. Add **jquery** and **bootstrap** to the **dependencies** section. The resulting bower.json file should look something like the example here. The versions will change over time, so use the latest stable build version from the drop down list.

```
{
        "name": "bower",
        "license": "Apache-2.0",
        "private": true,
        "dependencies": {
          "jquery": "2.1.3",
          "bootstrap": "3.3.2"
        }
}
```

12. Save the **bower.json** file, right-click on Dependencies > Bower > and click on Restore packages.

The project outline in Solution Explorer should now include **\bootstrap** and **\jQuery** directories in two locations: **\Dependencies\Bower** and **\bower_components**.

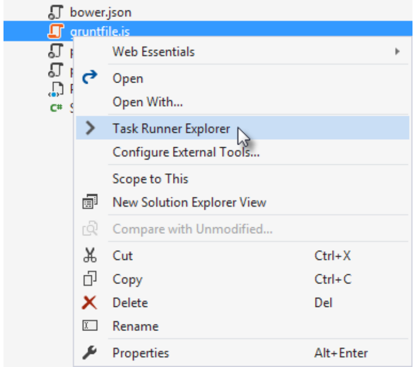


Note: If you don't see the bower_components directory, make sure that the **Show All Files** button is enabled in Solution Explorer's toolbar.

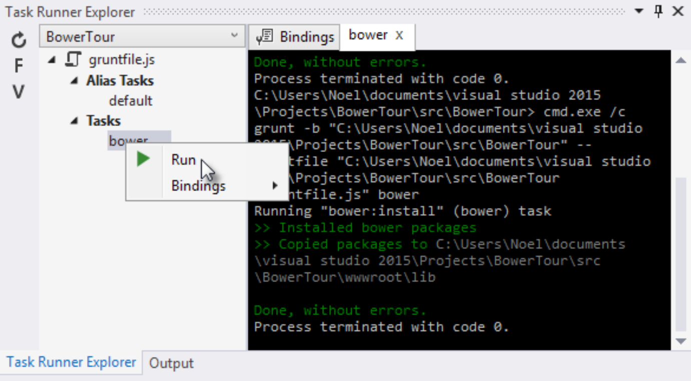
Install Packages to the Web Application

You've installed all the required files to your machine but haven't deployed them yet. Use Bower to copy from **bower_components** to the **\lib** directory under the web application root.

1. Right-click **gruntfile.js** and select **Task Runner Explorer** from the context menu. You can also reach the Task Runner Explorer through the **View > Other Windows** menu.



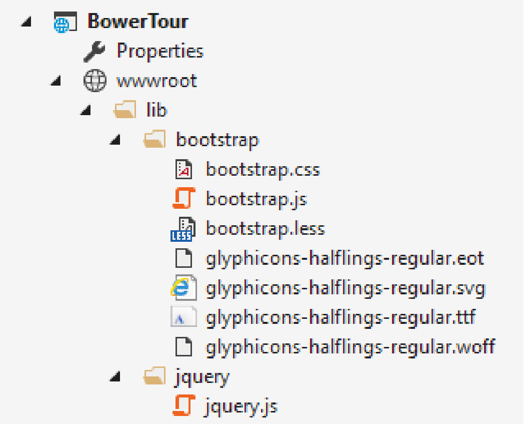
2. In the Task Runner Explorer, right-click the **Tasks > Bower** entry and select **Run** from the context menu. This step copies the Bower packages to the root of the project (the default is wwwroot) under the **\lib** directory.



```
Done, without errors.
Process terminated with code 0.
C:\Users\Noel\documents\visual studio 2015
\Projects\BowerTour\src\BowerTour> cmd.exe /c
grunt -b "C:\Users\Noel\documents\visual studio
     \Projects\BowerTour\src\BowerTour" --
     tfile "C:\Users\Noel\documents\visual studio
     \Projects\BowerTour\src\BowerTour
     ntfile.js" bower
Running "bower:install" (bower) task
>> Installed bower packages
>> Copied packages to C:\Users\Noel\documents
\visual studio 2015\Projects\BowerTour\src
\BowerTour\wwwroot\lib

Done, without errors.
Process terminated with code 0.
```

3. In Visual Studio, open the **wwwroot\lib bootstrap** and **jquery** directories. You should be able to see the deployed files under their respective directories. The files show up in a flat list in their

respective directories.



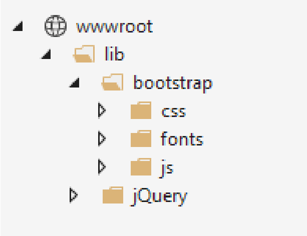
```
BowerTour
    Properties
    wwwroot
        lib
            bootstrap
                bootstrap.css
                bootstrap.js
                bootstrap.less
                glyphicons-halflings-regular.eot
                glyphicons-halflings-regular.svg
                glyphicons-halflings-regular.ttf
                glyphicons-halflings-regular.woff
            jquery
                jquery.js
```

4. In the Solution Explorer, delete the **\wwwroot** node. You will be able to replace it easily in the next step.
5. Open **bower.json** and add the **exportsOverride** element as shown in the listing below.

```
"exportsOverride": {
    "bootstrap": {
        "js": "dist/js/*.*",
        "css": "dist/css/*.*",
        "fonts": "dist/fonts/*.*"
    },
    "jquery": {
        "js": "dist/jquery.{js,min.js,min.map}"
    }
}
```

The exportsOverride element defines source directories and target directories. For example, bootstrap JavaScript files are copied from **\bower_components\bootstrap\dist\js** and copied to **\wwwroot\lib\bootstrap\js**.

6. From the Task Runner Explorer, run the Bower task a second time. The files are now organized under the target \css, \fonts, and \js directories.



```
wwwroot
    lib
        bootstrap
            css
            fonts
            js
        jQuery
```

Reference Packages

Now that Bower has copied the client support packages needed by the application, you can test that an HTML page can use the deployed jQuery and Bootstrap functionality.

1. Right-click the **wwwroot** node and select **Add > New Item > HTML Page** from the context menu.

Note: If you're editing an MVC application, you can add these references to **\Views\Shared_Layout.cshtml.**

2. Add the CSS and JavaScript references.
    - From the Solution Explorer wwwroot node, locate **bootstrap.css** and drag the file into the head tag of the HTML page.
    - Drag **jquery.js** and **bootstrap.js** to the end of the body tag.

    Be sure to keep bootstrap.js below jquery.js so that jQuery is loaded first. The file should look something like the example here:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Bower Example</title>
    <link href="lib/bootstrap/css/bootstrap.css" rel="stylesheet" />
</head>
<body>

    <script src="lib/jquery/jquery.js"></script>
    <script src="lib/bootstrap/js/bootstrap.js"></script>
</body>
</html>
```

Use the Installed Packages

Add jQuery and Bootstrap components to the page to verify that the web application is configured correctly.

1. Inside the body tag, above the script references, add a div element with the Bootstrap jumbotron class and an anchor tag as shown in the snippet below.

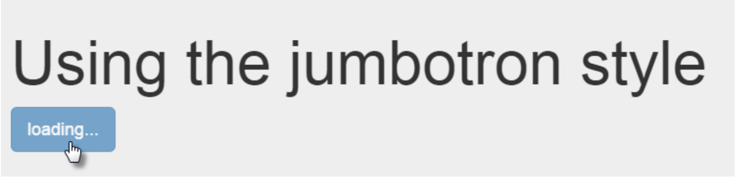
```
<div class="jumbotron">
    <h1>Using the jumbotron style</h1>
    <p><a class="btn btn-primary btn-lg" role="button">
       Stateful button</a></p>
</div>
```

2. Add a script tag below the jQuery and Bootstrap references.

```
<script>
    $(".btn").click(function() {
        $(this).text('loading')
            .delay(1000)
            .queue(function () {
                $(this).text('reset');
                $(this).dequeue()
            });
        }
    );
</script>
```

3. Press **Ctrl-Shift-W** to view the HTML page in the browser. Verify that the jumbotron styling is applied, the jQuery code responds when the button is clicked, and that the Bootstrap button

changes state.



Using the jumbotron style

Summary

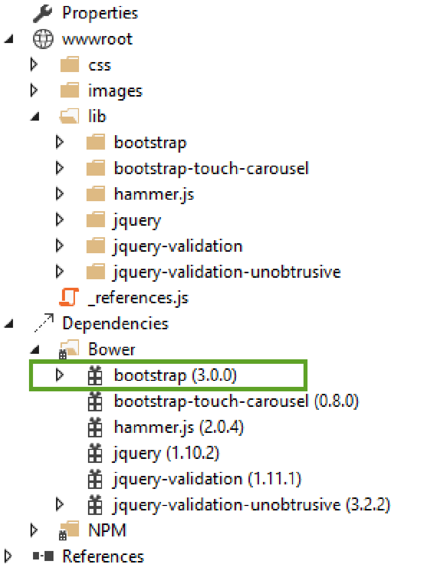
Bower is a package manager optimized to deliver files to the front-end. Bower helps automate searching, downloading and deployment of packages to your web application.

Bootstrap. Responsive Web Framework.

Bootstrap 3 is currently the most popular web framework for developing responsive web applications. It offers a number of features and benefits that can improve your users' experience with your web site, whether you're a novice at front-end design and development or an expert. Bootstrap is deployed as a set of HTML, CSS and JavaScript files, and is designed to help your website or application scale efficiently from phones to tablets to desktops.

Getting Started

There are several ways to get started with Bootstrap. If you're starting a new web application in Visual Studio, you can choose the default starter template for ASP.NET 5, in which case Bootstrap will come pre-installed:



```
  Properties
▲ wwwroot
  ▷ css
  ▷ images
  ▲ lib
    ▷ bootstrap
    ▷ bootstrap-touch-carousel
    ▷ hammer.js
    ▷ jquery
    ▷ jquery-validation
    ▷ jquery-validation-unobtrusive
      _references.js
▲ Dependencies
  ▲ Bower
    ▷ bootstrap (3.0.0)
      bootstrap-touch-carousel (0.8.0)
      hammer.js (2.0.4)
      jquery (1.10.2)
      jquery-validation (1.11.1)
    ▷ jquery-validation-unobtrusive (3.2.2)
  ▷ NPM
▷ References
```

You can also install bootstrap using one of several package managers, such as bower, npm, or NuGet. In each case, the process is essentially the same:

Bower

```
bower install bootstrap
```

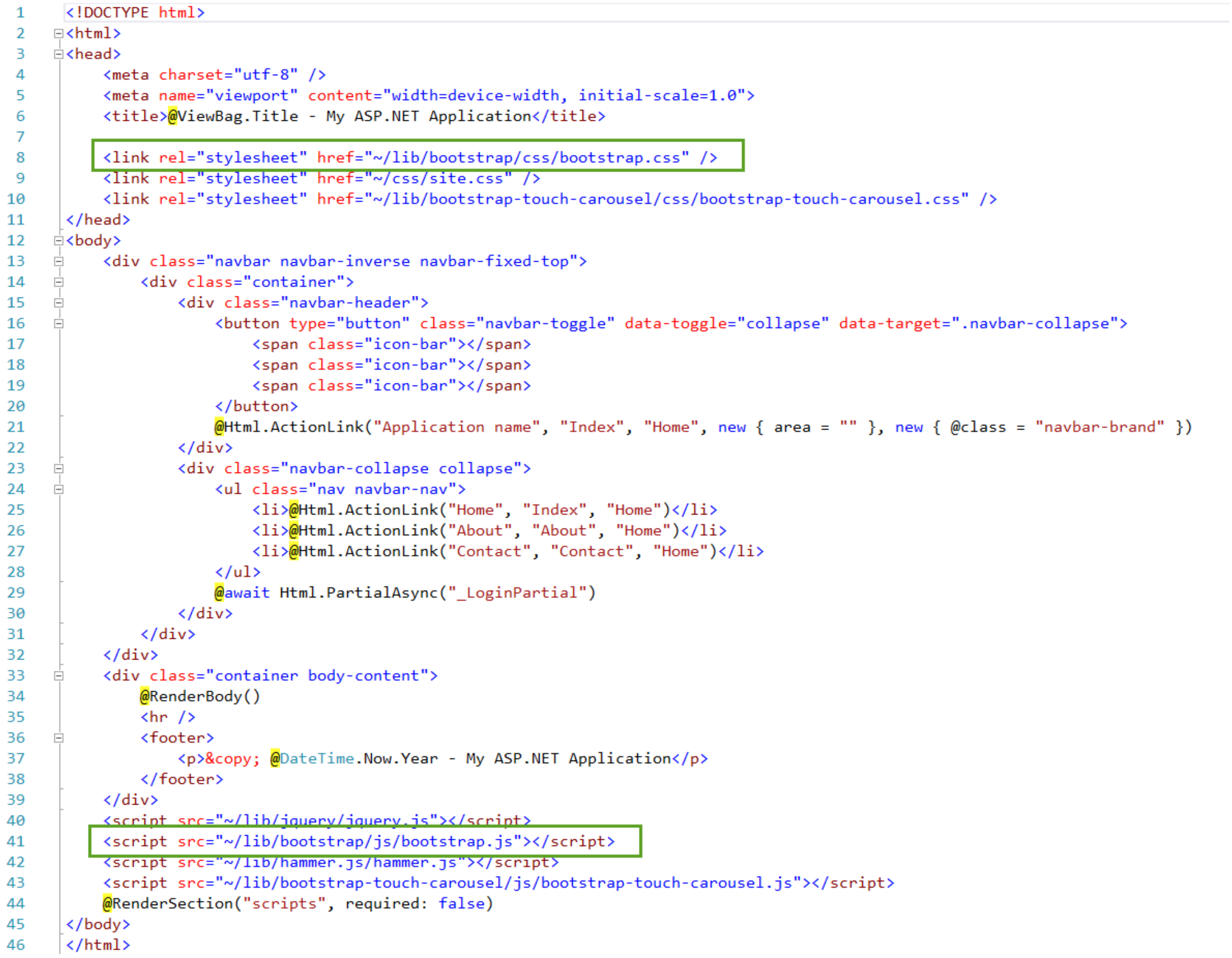
npm

```
npm install bootstrap
```

NuGet

```
Install-Package bootstrap
```

If you're referencing your own local versions of Bootstrap, you'll need to reference them in any pages that will use it. In the default ASP.NET site template, the _Layout.cshtml file does so like this:



```
1   <!DOCTYPE html>
2   <html>
3   <head>
4       <meta charset="utf-8" />
5       <meta name="viewport" content="width=device-width, initial-scale=1.0">
6       <title>@ViewBag.Title - My ASP.NET Application</title>
7   
8       <link rel="stylesheet" href="~/lib/bootstrap/css/bootstrap.css" />
9       <link rel="stylesheet" href="~/css/site.css" />
10      <link rel="stylesheet" href="~/lib/bootstrap-touch-carousel/css/bootstrap-touch-carousel.css" />
11  </head>
12  <body>
13      <div class="navbar navbar-inverse navbar-fixed-top">
14          <div class="container">
15              <div class="navbar-header">
16                  <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
17                      <span class="icon-bar"></span>
18                      <span class="icon-bar"></span>
19                      <span class="icon-bar"></span>
20                  </button>
21                  @Html.ActionLink("Application name", "Index", "Home", new { area = "" }, new { @class = "navbar-brand" })
22              </div>
23              <div class="navbar-collapse collapse">
24                  <ul class="nav navbar-nav">
25                      <li>@Html.ActionLink("Home", "Index", "Home")</li>
26                      <li>@Html.ActionLink("About", "About", "Home")</li>
27                      <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
28                  </ul>
29                  @await Html.PartialAsync("_LoginPartial")
30              </div>
31          </div>
32      </div>
33      <div class="container body-content">
34          @RenderBody()
35          <hr />
36          <footer>
37              <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
38          </footer>
39      </div>
40      <script src="~/lib/jquery/jquery.js"></script>
41      <script src="~/lib/bootstrap/js/bootstrap.js"></script>
42      <script src="~/lib/hammer.js/hammer.js"></script>
43      <script src="~/lib/bootstrap-touch-carousel/js/bootstrap-touch-carousel.js"></script>
44      @RenderSection("scripts", required: false)
45  </body>
46  </html>
```

You can also simply reference Bootstrap using its CDN (Content Delivery Network) links, which is how you may wish to have your application reference bootstrap in production in any case. Simply replace the two references above with these:

```
<!-- Latest compiled and minified CSS -->
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/css/bootstrap.min.css">

<!-- Latest compiled and minified JavaScript -->
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/js/bootstrap.min.js"></script>
```

Note that if you're going to be using any of Bootstrap's jQuery plugins, you will also need to reference jQuery (as shown above on line 40). If you're going the CDN route, you can of course use jQuery's CDN as well:

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>
```

Basic Templates and Features

The most basic Bootstrap template looks very much like the _Layout.cshtml file shown above, and simply includes a basic menu for navigation and a place to render the rest of the page.

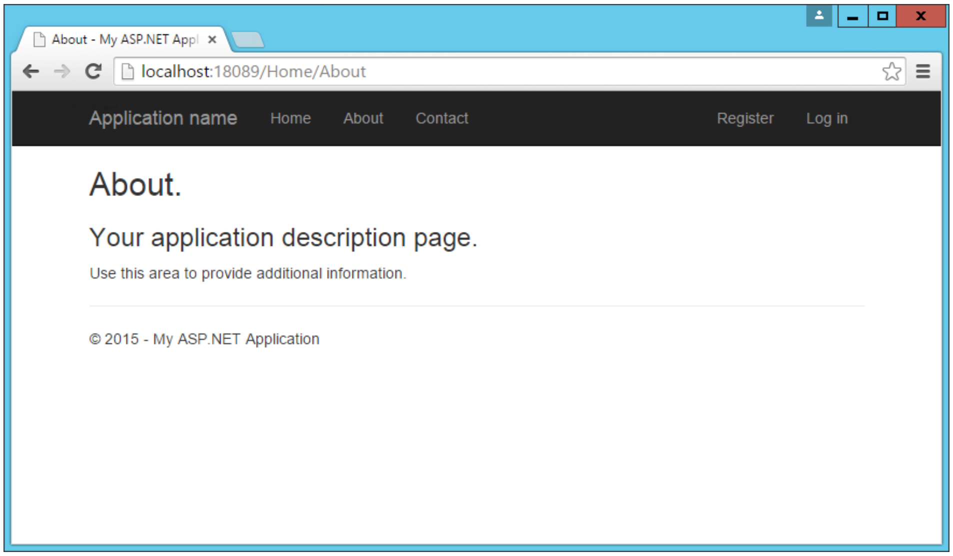
Basic Navigation

The default template uses a set of <div> elements to render a top navbar and the main body of the page. If you're using HTML5, you can replace the first <div> tag with a <nav> tag to get the same effect, but with more precise semantics. Within this first <div> you can see there are several others. First, a <div> with a class of "container", and then within that, two more <div> elements: "navbar-header" and "navbar-collapse". The navbar-header div includes a button that will appear when the screen is below a certain minimum width, showing 3 horizontal lines (a so-called "hamburger icon"). The icon is rendered using pure HTML and CSS; no image is required. This is the code that displays the icon, with each of the <span> tags rendering one of the white bars:

```
<button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
</button>
```

It also includes the application name, which appears in the top left. The main navigation menu is rendered by the <ul> element within the second div, and includes links to Home, About, and Contact. Additional links for Register and Login are added by the _LoginPartial line on line 29.

Below the navigation, the main body of each page is rendered in another <div>, marked with the "container" and "body-content" classes. In the simple default _Layout file shown here, the contents of the page are rendered by the specific View associated with the page, and then a simple <footer> is added to the end of the <div> element. You can see how the built-in About page appears using this template:



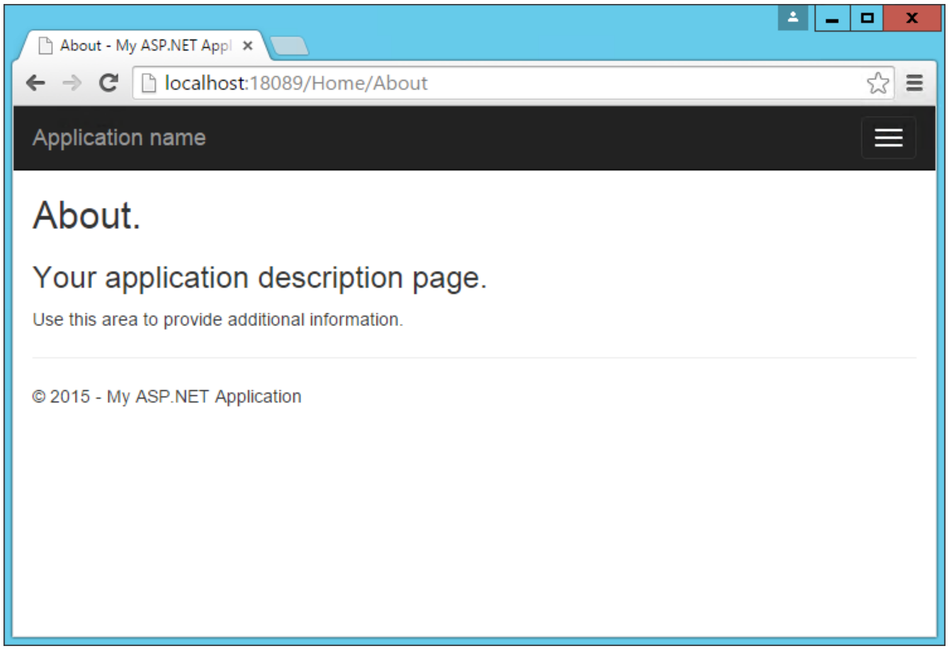
About.

Your application description page.

Use this area to provide additional information.

© 2015 - My ASP.NET Application

The collapsed navbar, with "hamburger" button in the top right, appears when the window drops below a certain width:



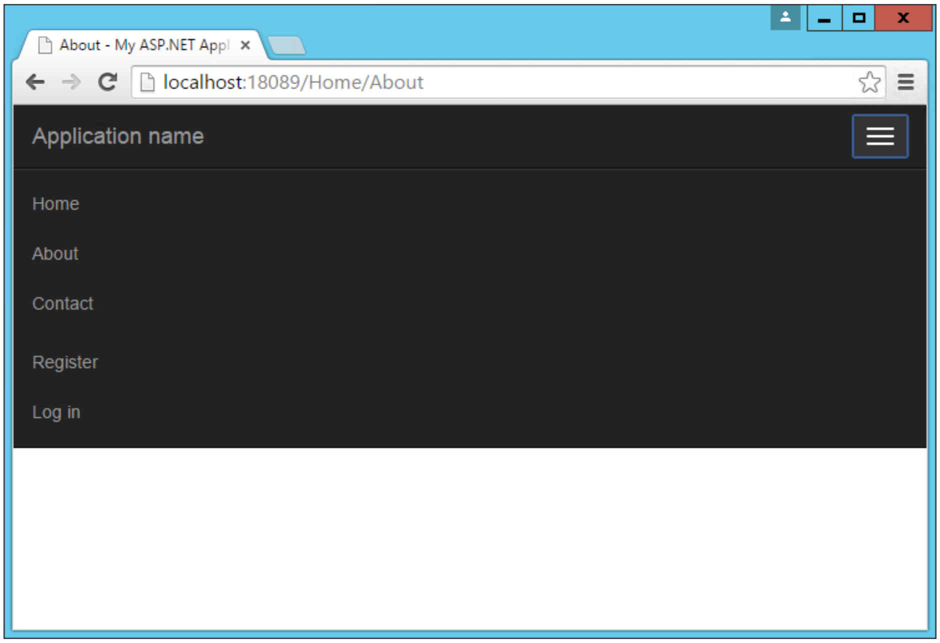
About.

Your application description page.

Use this area to provide additional information.

© 2015 - My ASP.NET Application

Clicking the icon reveals the menu items in a vertical drawer that slides down from the top of the page:



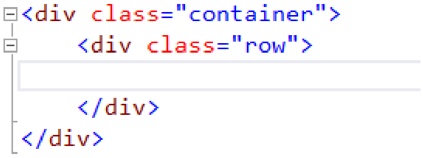
Typography and Links

Bootstrap sets up the site's basic typography, colors, and link formatting in its CSS file. This CSS file includes default styles for tables, buttons, form elements, images, and more. One particularly useful feature is the grid layout system, covered next.

Grids

One of the most popular features of Bootstrap is its grid layout system. Modern web applications should avoid using the <table> tag for layout, instead restricting the use of this element to actual tabular data. Instead, columns and rows can be laid out using a series of <div> elements and the appropriate CSS classes. There are several advantages to this approach, including the ability to adjust the layout of grids to display vertically on narrow screens, such as on phones.

Bootstrap's grid layout system is based on twelve columns. This number was chosen because it can be divided evenly into 1, 2, 3, or 4 columns, and column widths can vary to within 1/12th of the vertical width of the screen. To start using the grid layout system, you should begin with a container <div> and then add a row <div>, as shown here:



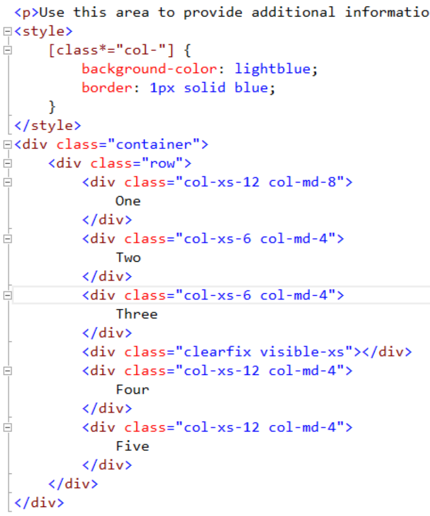
```
<div class="container">
    <div class="row">

    </div>
</div>
```

Next, add additional <div> elements for each column, and specify the number of columns that <div> should occupy (out of 12) as part of a CSS class starting with "col-md-". For instance, if you want to simply have two columns of equal size, you would use a class of "col-md-6" for each one. In this case "md" is short for "medium" and refers to standard-sized desktop computer display sizes. There are four different options you can choose from, and each will be used for higher widths unless overridden (so if you want the layout to be fixed regardless of screen width, you can just specify xs classes).

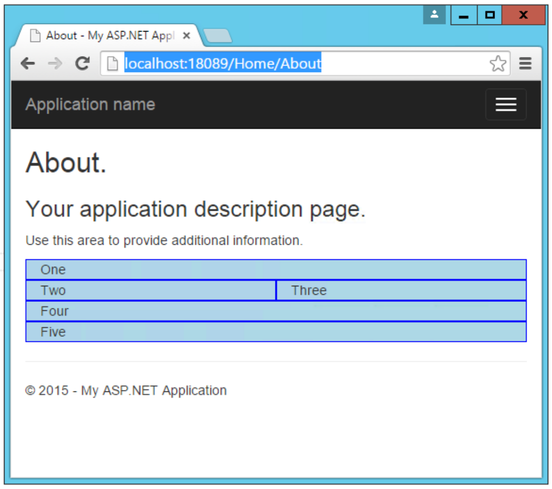
CSS Class Prefix	Device Tier	Width
col-xs-	Phones	<768px
col-sm-	Tablets	>=768px
col-md-	Desktops	>=992px
col-lg-	Larger Desktop Displays	>=1200px

When specifying two columns both with "col-md-6" the resulting layout will be two columns at desktop resolutions, but these two columns will stack vertically when rendered on smaller devices (or a narrower browser window on a desktop), allowing users to easily view content without the need to scroll horizontally.

Bootstrap will always default to a single-column layout, so you only need to specify columns when you want more than one column. The only time you would want to explicitly specify that a <div> take up all 12 columns would be to override the behavior of a larger device tier. When specifying multiple device tier classes, you may need to reset the column rendering at certain points. Adding a clearfix div that is only visible within a certain viewport can achieve this, as shown here:



```
<p>Use this area to provide additional information.</p>
<style>
    [class*="col-"] {
        background-color: lightblue;
        border: 1px solid blue;
    }
</style>
<div class="container">
    <div class="row">
        <div class="col-xs-12 col-md-8">
            One
        </div>
        <div class="col-xs-6 col-md-4">
            Two
        </div>
        <div class="col-xs-6 col-md-4">
            Three
        </div>
        <div class="clearfix visible-xs"></div>
        <div class="col-xs-12 col-md-4">
            Four
        </div>
        <div class="col-xs-12 col-md-4">
            Five
        </div>
    </div>
</div>
```



Application name

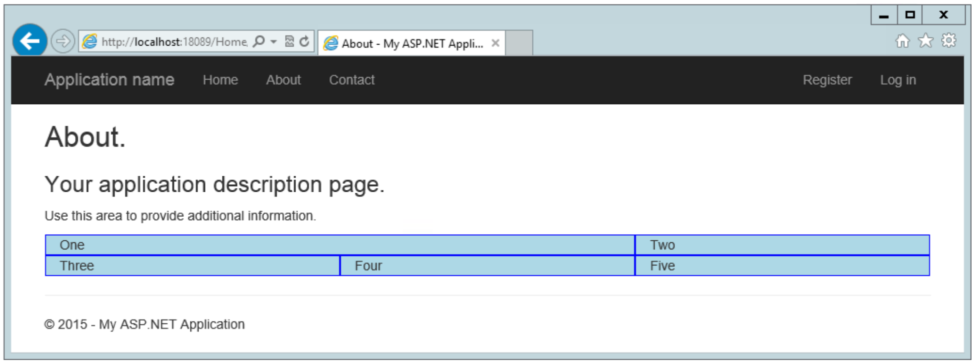
About.

Your application description page.

Use this area to provide additional information.

One	
Two	Three
Four	
Five	

© 2015 - My ASP.NET Application



Application name Home About Contact Register Log in

About.

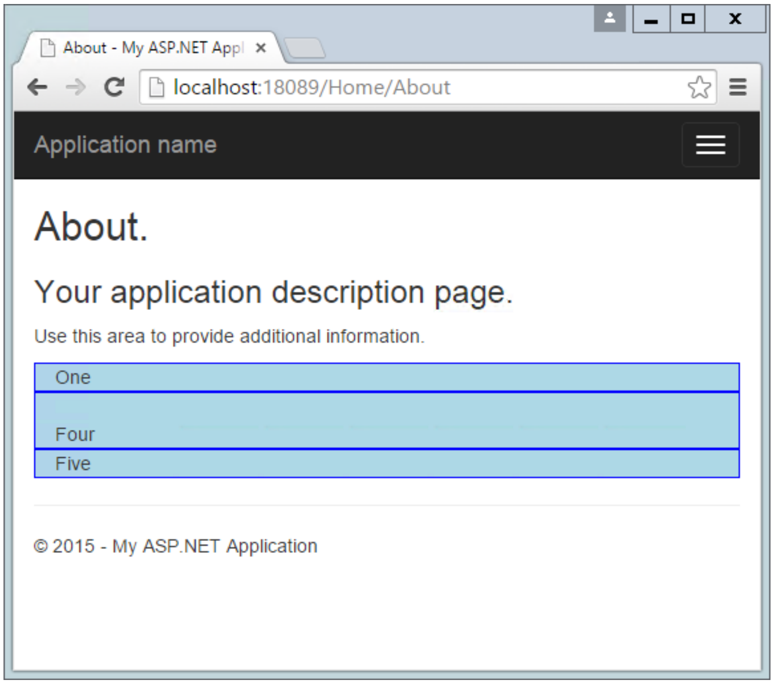
Your application description page.

Use this area to provide additional information.

One		Two
Three	Four	Five

© 2015 - My ASP.NET Application

In the above example, One and Two share a row in the “md” layout, while Two and Three share a row in the “xs” layout. Without the clearfix <div>, Two and Three are not shown correctly in the “xs” view (note that only One, Four, and Five are shown):



Application name

About.

Your application description page.

Use this area to provide additional information.

One

Four

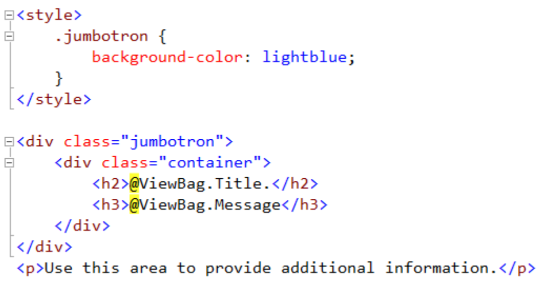
Five

© 2015 - My ASP.NET Application

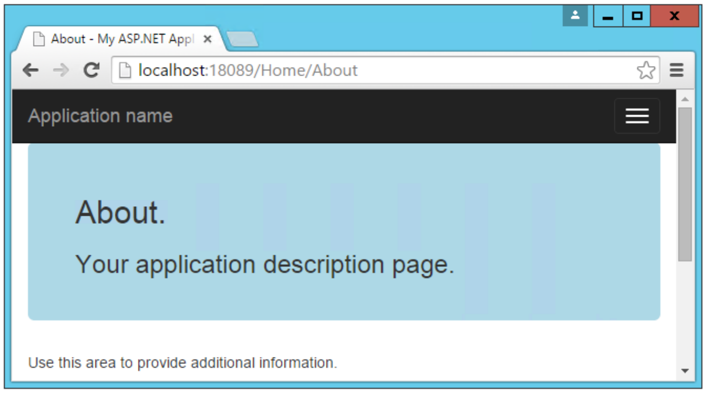
In this example, only a single row <div> was used, and Bootstrap still mostly did the right thing with regard to the layout and stacking of the columns. Typically, you should specify a row <div> for each horizontal row your layout requires, and of course you can nest Bootstrap grids within one another. When you do, each nested grid will occupy 100% of the width of the element in which it is placed, which can then be subdivided using column classes.

Jumbotron

If you've used the default ASP.NET MVC templates in Visual Studio 2012 or 2013, you've probably seen the Jumbotron in action. It refers to a large full-width section of a page that can be used to display a large background image, a call to action, a rotator, or similar elements. To add a jumbotron to a page, simply add a <div> and give it a class of "jumbotron", then place a container <div> inside and add your content. We can easily adjust the standard About page to use a jumbotron for the main headings it displays:



```
<style>
    .jumbotron {
        background-color: lightblue;
    }
</style>

<div class="jumbotron">
    <div class="container">
        <h2>@ViewBag.Title.</h2>
        <h3>@ViewBag.Message</h3>
    </div>
</div>
<p>Use this area to provide additional information.</p>
```



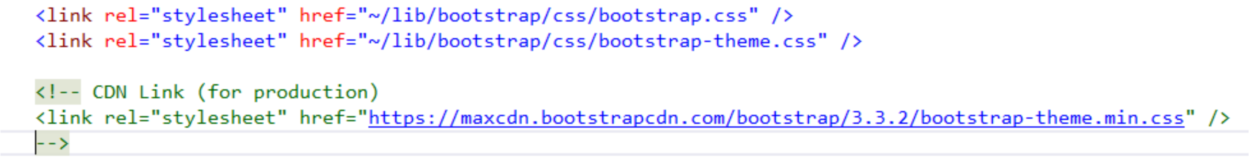
About.

Your application description page.

Use this area to provide additional information.

The Bootstrap Theme

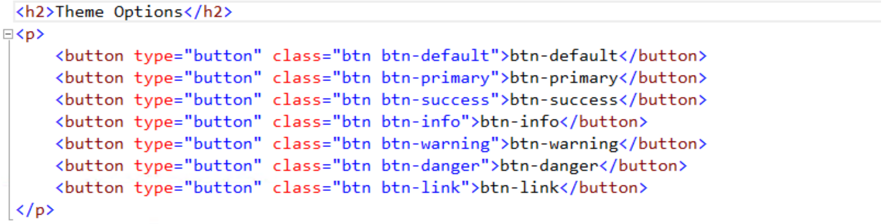
In addition to the included features, you can also apply a standard theme to your site by using the Bootstrap Theme. This theme includes a set of common styles for many standard elements, and is already installed when you installed bootstrap. To start using it, add a reference to its CSS using either its local path or a CDN. Here are both options, using the same default ASP.NET starter application we've been working with thus far:



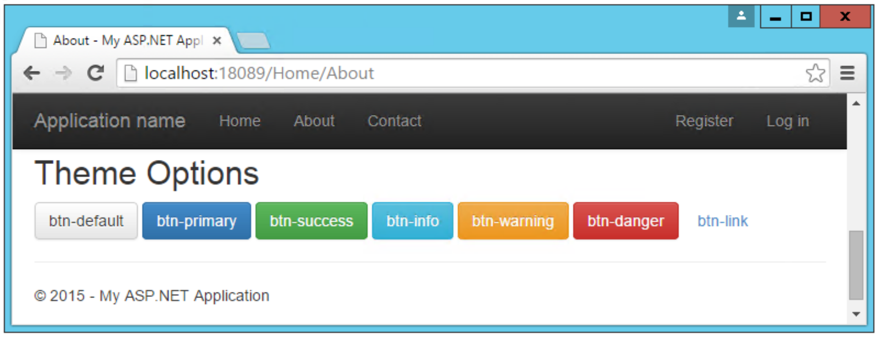
```
<link rel="stylesheet" href="~/lib/bootstrap/css/bootstrap.css" />
<link rel="stylesheet" href="~/lib/bootstrap/css/bootstrap-theme.css" />

<!-- CDN Link (for production)
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/bootstrap-theme.min.css" />
-->
```

With this CSS included, we gain access to a large set of standard CSS classes that we can use to control the appearance of many of our standard UI elements. For instance, any buttons our application requires can use a standard palette of classes to help users visually distinguish between different buttons' behaviors based on their color and appearance.

Buttons

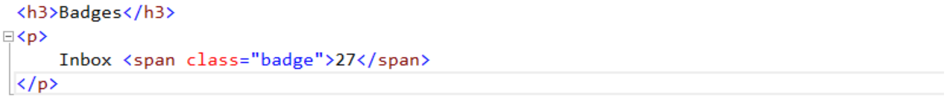


```
<h2>Theme Options</h2>
<p>
    <button type="button" class="btn btn-default">btn-default</button>
    <button type="button" class="btn btn-primary">btn-primary</button>
    <button type="button" class="btn btn-success">btn-success</button>
    <button type="button" class="btn btn-info">btn-info</button>
    <button type="button" class="btn btn-warning">btn-warning</button>
    <button type="button" class="btn btn-danger">btn-danger</button>
    <button type="button" class="btn btn-link">btn-link</button>
</p>
```

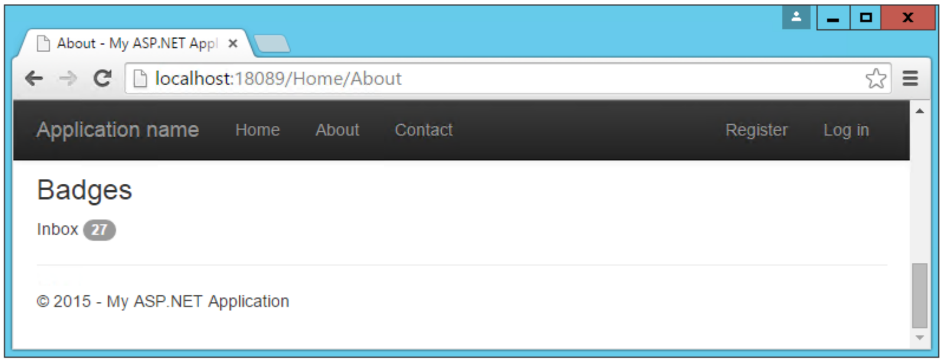


Badges

Badges refer to small, usually numeric callouts next to a navigation item. They can indicate a number of messages or notifications waiting, or the presence of updates. Specifying such badges is as simple as adding a <span> containing the text, with a class of “badge”:

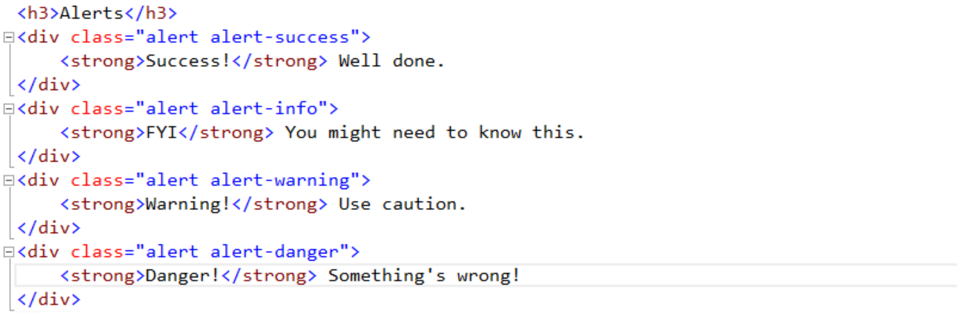


```
<h3>Badges</h3>
<p>
    Inbox <span class="badge">27</span>
</p>
```

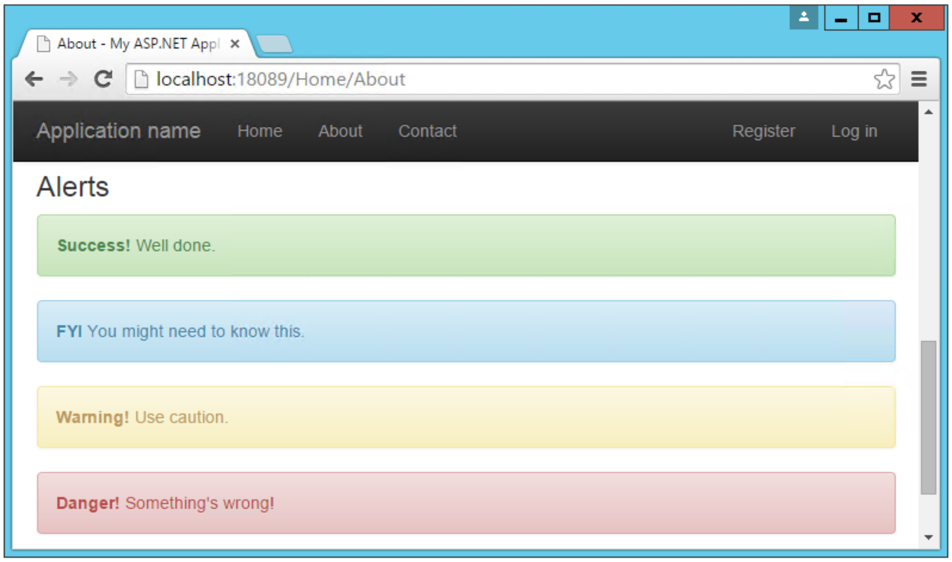


Alerts

You may need to display some kind of notification, alert, or error message to your application’s users. That’s where the standard alert classes come in. There are four different severity levels, with associated color schemes:



```
<h3>Alerts</h3>
<div class="alert alert-success">
    <strong>Success!</strong> Well done.
</div>
<div class="alert alert-info">
    <strong>FYI</strong> You might need to know this.
</div>
<div class="alert alert-warning">
    <strong>Warning!</strong> Use caution.
</div>
<div class="alert alert-danger">
    <strong>Danger!</strong> Something's wrong!
</div>
```



Alerts

Success! Well done.

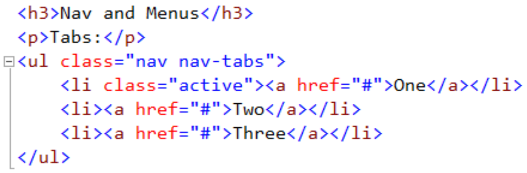
FYI You might need to know this.

Warning! Use caution.

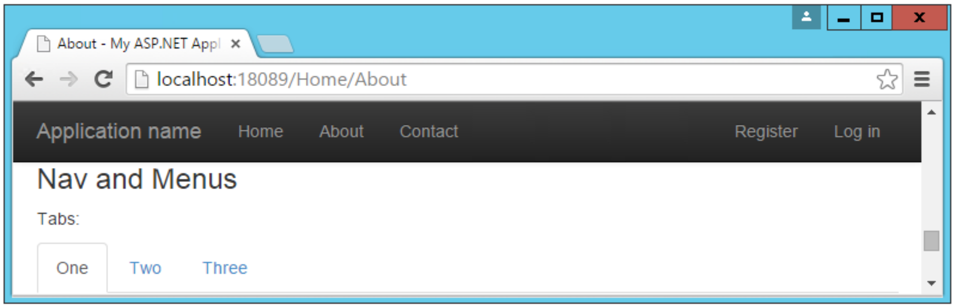
Danger! Something's wrong!

Navbars and Menus

Our layout already includes a standard navbar, but the Bootstrap theme supports additional styling options. We can also easily opt to display the navbar vertically rather than horizontally if that's preferred, as well as adding sub-navigation items in flyout menus. Simple navigation menus, like tab strips, are built on top of <ul> elements. These can be created very simply by just providing them with the CSS classes "nav" and "nav-tabs":



```
<h3>Nav and Menus</h3>
<p>Tabs:</p>
<ul class="nav nav-tabs">
    <li class="active"><a href="#">One</a></li>
    <li><a href="#">Two</a></li>
    <li><a href="#">Three</a></li>
</ul>
```

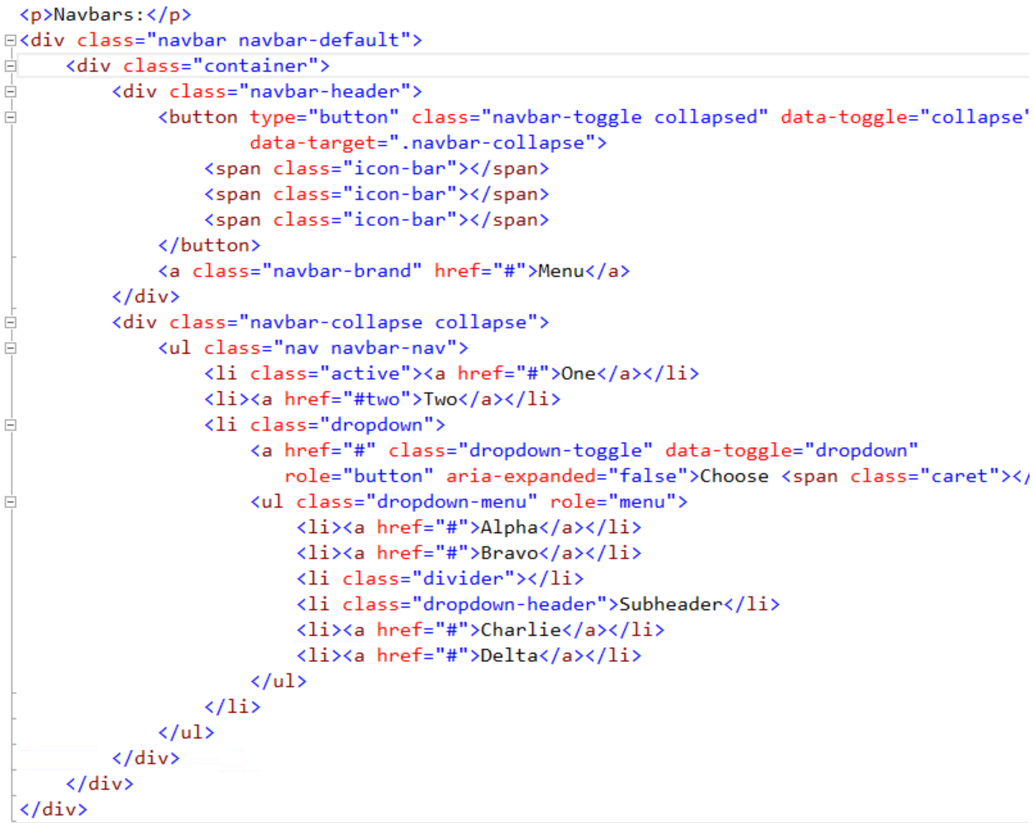


Nav and Menus

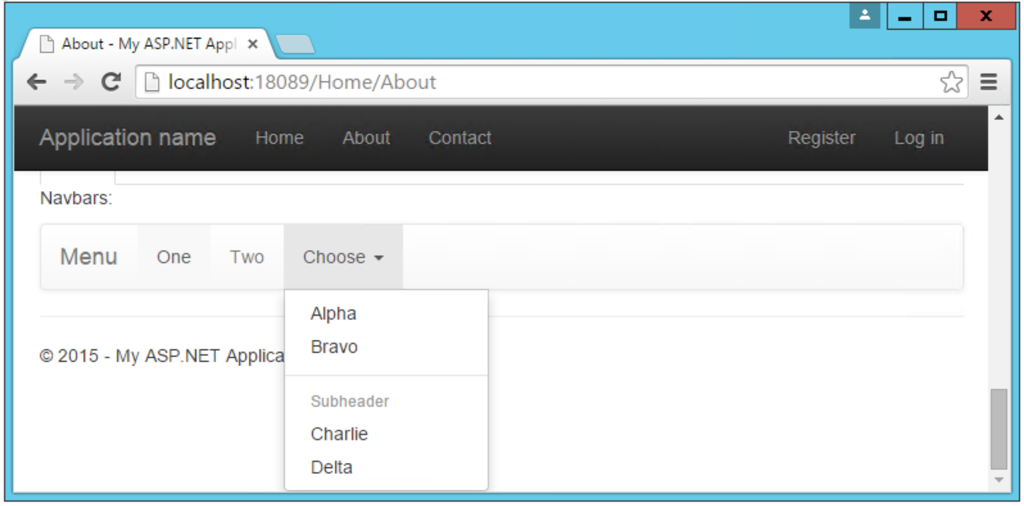
Tabs:

One Two Three

Navbars are built similarly, but are a bit more complex. They start with a <nav> or <div> with a class of “navbar”, within which a container div holds the rest of the elements. Our page includes a navbar in its header already – the one shown below simply expands on this, adding support for a dropdown menu:



```
<p>Navbars:</p>
<div class="navbar navbar-default">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed" data-toggle="collapse'
                    data-target=".navbar-collapse">
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="#">Menu</a>
        </div>
        <div class="navbar-collapse collapse">
            <ul class="nav navbar-nav">
                <li class="active"><a href="#">One</a></li>
                <li><a href="#two">Two</a></li>
                <li class="dropdown">
                    <a href="#" class="dropdown-toggle" data-toggle="dropdown"
                       role="button" aria-expanded="false">Choose <span class="caret"></
                    <ul class="dropdown-menu" role="menu">
                        <li><a href="#">Alpha</a></li>
                        <li><a href="#">Bravo</a></li>
                        <li class="divider"></li>
                        <li class="dropdown-header">Subheader</li>
                        <li><a href="#">Charlie</a></li>
                        <li><a href="#">Delta</a></li>
                    </ul>
                </li>
            </ul>
        </div>
    </div>
</div>
```



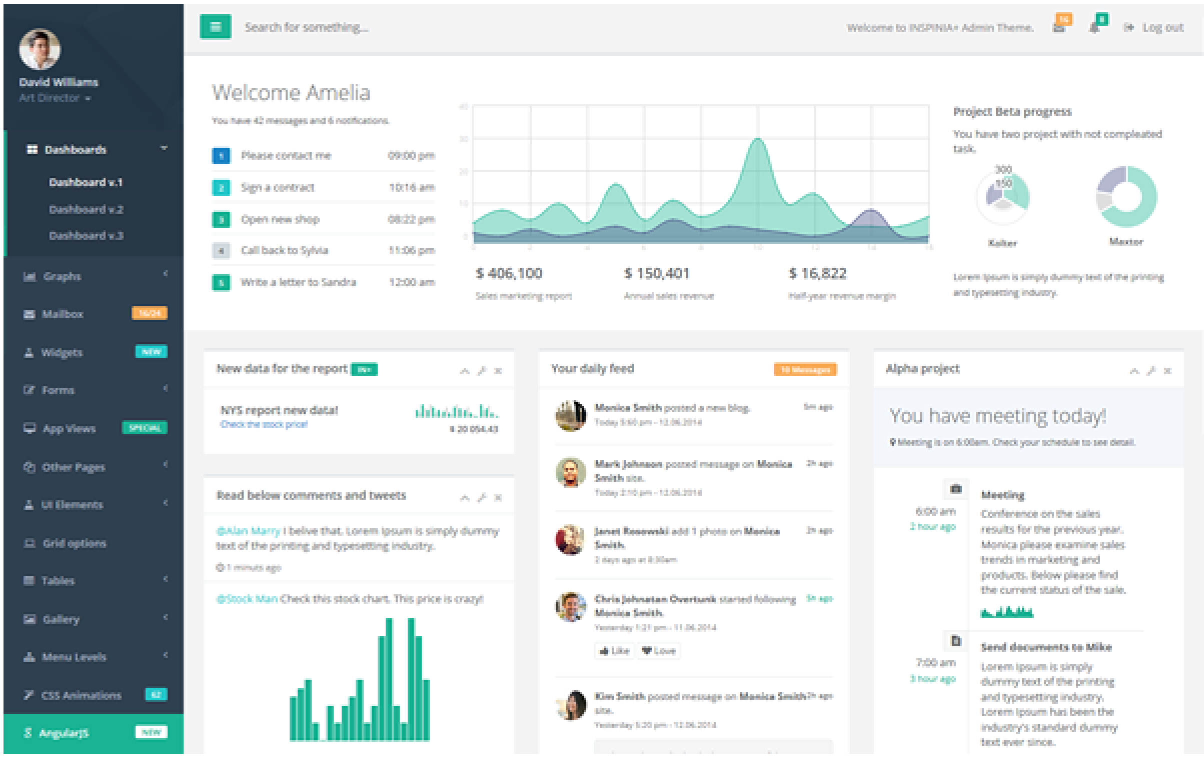
Additional Elements

The default theme can also be used to present HTML tables in a nicely formatted style, including support for striped views. There are labels with styles that are similar to those of the buttons. You can create custom Dropdown menus that support additional styling options beyond the standard HTML <select> element, along with Navbars like the one our default starter site is already using. If you need a progress bar, there are several styles to choose from, as well as List Groups and panels that include a title and content. Explore additional options within the standard Bootstrap Theme here:

http://getbootstrap.com/examples/theme/

More Themes

You can extend the standard Bootstrap Theme by overriding some or all of its CSS, adjusting the colors and styles to suit your own application's needs. If you'd like to start from a ready-made theme, there are several theme galleries available online that specialize in Bootstrap Themes, such as WrapBootstrap.com (which has a variety of commercial themes) and Bootswatch.com (which offers free themes). Some of the paid templates available provide a great deal of functionality on top of the basic Bootstrap theme, such as rich support for administrative menus, and dashboards with rich charts and gauges. An example of a popular paid template is Inspinia, currently for sale for $18, which includes an ASP.NET MVC5 template in addition to AngularJS and static HTML versions. A sample screenshot is shown below.



Welcome Amelia

You have 42 messages and 6 notifications.

1	Please contact me	09:00 pm
2	Sign a contract	10:16 am
3	Open new shop	08:22 pm
4	Call back to Sylvia	11:06 pm
5	Write a letter to Sandra	12:00 am

$ 406,100 Sales marketing report

$ 150,401 Annual sales revenue

$ 16,822 Half-year revenue margin

You have two project with not compleated task.

Lorem Ipsum is simply dummy text of the printing and typesetting industry.

NYS report new data! Check the stock price! $ 20 054.43

Read below comments and tweets

@Alan Marry I belive that. Lorem Ipsum is simply dummy text of the printing and typesetting industry. 1 minuts ago

@Stock Man Check this stock chart. This price is crazy!

Monica Smith posted a new blog. 5m ago Today 5:60 pm - 12.06.2014

Mark Johnson posted message on Monica Smith site. 2h ago Today 2:10 pm - 12.06.2014

Janet Rosowski add 1 photo on Monica Smith. 2h ago 2 days ago at 8:30am

Chris Johnatan Overtunk started following Monica Smith. 5h ago Yesterday 1:21 pm - 11.06.2014

Kim Smith posted message on Monica Smith site. 2h ago Yesterday 5:20 pm - 12.06.2014

You have meeting today!

Meeting is on 6:00am. Check your schedule to see detail.

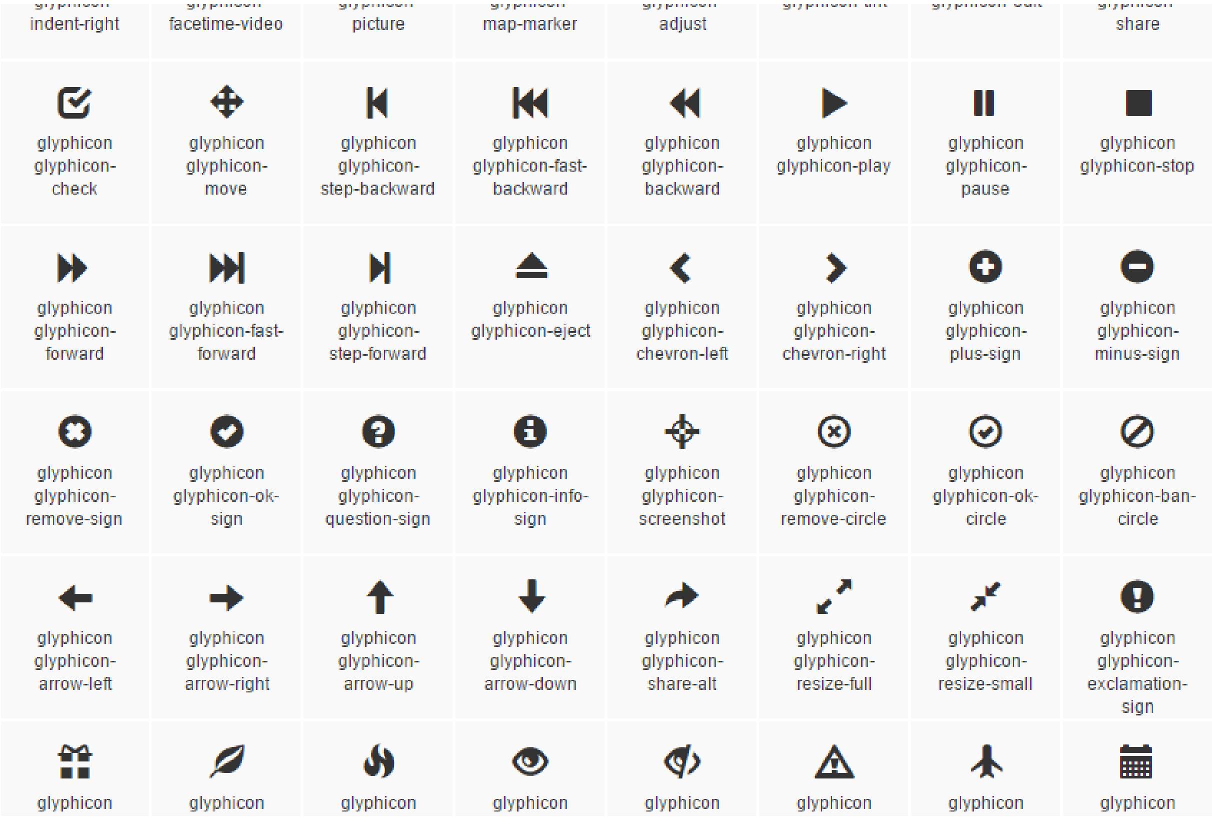
6:00 am 2 hour ago Meeting Conference on the sales results for the previous year. Monica please examine sales trends in marketing and products. Below please find the current status of the sale.

7:00 am 3 hour ago Send documents to Mike Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since.

If you're interested in building your own dashboard, you may wish to start from the free example available here: http://getbootstrap.com/examples/dashboard/.

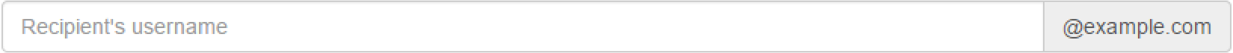
Components

In addition to those elements already discussed, Bootstrap includes support for a variety of built-in UI components. These include icon sets from Glyphicons (http://glyphicons.com), with over 200 icons freely available for use within your Bootstrap-enabled web application. Here's just a small sample:



indent-right	facetime-video	picture	map-marker	adjust			share
glyphicon glyphicon-check	glyphicon glyphicon-move	glyphicon glyphicon-step-backward	glyphicon glyphicon-fast-backward	glyphicon glyphicon-backward	glyphicon glyphicon-play	glyphicon glyphicon-pause	glyphicon glyphicon-stop
glyphicon glyphicon-forward	glyphicon glyphicon-fast-forward	glyphicon glyphicon-step-forward	glyphicon glyphicon-eject	glyphicon glyphicon-chevron-left	glyphicon glyphicon-chevron-right	glyphicon glyphicon-plus-sign	glyphicon glyphicon-minus-sign
glyphicon glyphicon-remove-sign	glyphicon glyphicon-ok-sign	glyphicon glyphicon-question-sign	glyphicon glyphicon-info-sign	glyphicon glyphicon-screenshot	glyphicon glyphicon-remove-circle	glyphicon glyphicon-ok-circle	glyphicon glyphicon-ban-circle
glyphicon glyphicon-arrow-left	glyphicon glyphicon-arrow-right	glyphicon glyphicon-arrow-up	glyphicon glyphicon-arrow-down	glyphicon glyphicon-share-alt	glyphicon glyphicon-resize-full	glyphicon glyphicon-resize-small	glyphicon glyphicon-exclamation-sign
glyphicon	glyphicon	glyphicon	glyphicon	glyphicon	glyphicon	glyphicon	glyphicon

Input groups allow bundling of additional text or buttons with an input element, providing the user with a more intuitive experience:



Recipient's username | @example.com

Breadcrumbs are a common UI component used to show a user their recent history or depth within a site's navigation hierarchy. Add them easily by applying the "breadcrumb" class to any <ol> list element. Include built-in support for pagination by using the "pagination" class on a <ul> element within a <nav>. Add responsive embedded slideshows and video by using <iframe>, <embed>, <video>, or <object> elements, which Bootstrap will style automatically. Specify a particular aspect ratio by using specific classes like "embed-responsive-16by9".

JavaScript Support

Bootstrap's JavaScript library includes API support for the included components, allowing you to control their behavior programmatically within your application. In addition, bootstrap.js includes over a dozen custom jQuery plugins, providing additional features like transitions, modal dialogs, scroll detection (updating styles based on where the user has scrolled in the document), collapse behavior, carousels, and affixing menus to the window so they do not scroll off the screen. There's not sufficient room to cover all of the JavaScript add-ons built into Bootstrap – to learn more please visit http://getbootstrap.com/javascript/.

Summary

Bootstrap provides a web framework that can be used to quickly and productively lay out and style a wide variety of websites and applications. Its basic typography and styles provide a pleasant look and feel that can easily be manipulated through custom theme support, which can be hand-crafted or purchased commercially. It supports a host of web components that in the past would have required expensive third-party controls to accomplish, while supporting modern and open web standards.

Less, Sass & Font Awesome. Styling applications

Users of web applications have increasingly high expectations when it comes to style and overall experience. Modern web applications frequently leverage rich tools and frameworks for defining and managing their look and feel in a consistent manner. Frameworks like Bootstrap can go a long way toward defining a common set of styles and layout options for the web sites. However, most non-trivial sites also benefit from being able to effectively define and maintain styles and cascading style sheet (CSS) files, as well as having easy access to non-image icons that help make the site's interface more intuitive. That's where languages and tools that support Less and Sass, and libraries like Font Awesome, come in.

CSS Preprocessor Languages

Languages that are compiled into other languages, in order to improve the experience of working with the underlying language, are referred to as pre-processors. There are two popular pre-processors for CSS: **Less** and **Sass**. These pre-processors add features to CSS, such as support for variables and nested rules, which improve the maintainability of large, complex stylesheets. CSS as a language is very basic, lacking support even for something as simple as variables, and this tends to make CSS files repetitive and bloated. Adding real programming language features via preprocessors can help reduce duplication and provide better organization of styling rules. Visual Studio provides built-in support for both Less and Sass, as well as extensions that can further improve the development experience when working with these languages.

As a quick example of how preprocessors can improve readability and maintainability of style information, consider this CSS:

```
.header {
   color: black;
   font-weight: bold;
   font-size: 18px;
   font-family: Helvetica, Arial, sans-serif;
}

.small-header {
   color: black;
   font-weight: bold;
   font-size: 14px;
   font-family: Helvetica, Arial, sans-serif;
}
```

Using Less, this can be rewritten to eliminate all of the duplication, using a *mixin* (so named because it allows you to "mix in" properties from one class or rule-set into another):

```
.header {
   color: black;
   font-weight: bold;
   font-size: 18px;
   font-family: Helvetica, Arial, sans-serif;
}

.small-header {
   .header;
```

```
    font-size: 14px;
}
```

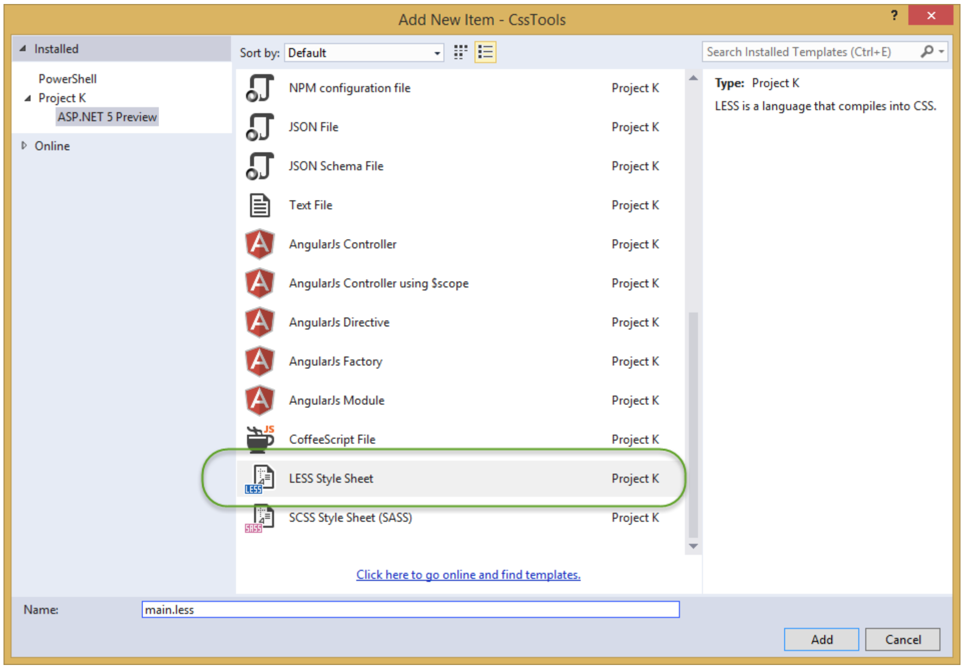
Visual Studio 2015 adds a great deal of built-in support for Less and Sass. You can also add support for earlier versions of Visual Studio by installing the Web Essentials extension.

Less

The Less CSS pre-processor runs using Node.js. You can quickly install it using the Node Package Manager (NPM), with:

```
npm install -g less
```

If you're using Visual Studio 2015, you can get started with Less by adding one or more Less files to your project, and then configuring Gulp (or Grunt) to process them at compile-time. Add a Styles folder to your project, and then add a new Less file called main.less to this folder.



Add New Item - CssTools

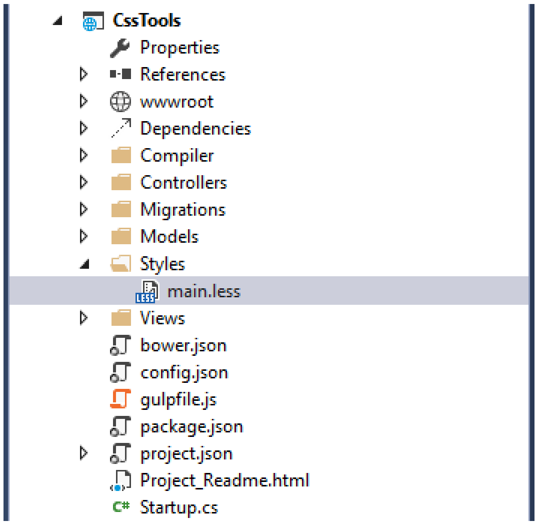
Sort by: Default

Type: Project K

LESS is a language that compiles into CSS.

Name: main.less

Once added, your folder structure should look something like this:



CssTools
Properties
References
wwwroot
Dependencies
Compiler
Controllers
Migrations
Models
Styles
main.less
Views
bower.json
config.json
gulpfile.js
package.json
project.json
Project_Readme.html
Startup.cs

Now we can add some basic styling to the file, which will be compiled into CSS and deployed to the wwwroot folder by Gulp.

Modify main.less to include the following content, which creates a simple color palette from a single base color.

```
@base: #663333;
@background: spin(@base, 180);
@lighter: lighten(spin(@base, 5), 10%);
@lighter2: lighten(spin(@base, 10), 20%);
@darker: darken(spin(@base, -5), 10%);
@darker2: darken(spin(@base, -10), 20%);

body {
    background-color:@background;
}
.baseColor  {color:@base}
.bgLight    {color:@lighter}
.bgLight2   {color:@lighter2}
.bgDark     {color:@darker}
.bgDark2    {color:@darker2}
```

@base and the other @-prefixed items are variables. Each of them represents a color. Except for @base, they are set using color functions: lighten, darken, and spin. Lighten and darken do pretty much what you would expect; spin adjusts the hue of a color by a number of degrees (around the color wheel). The less processor is smart enough to ignore variables that aren't used, so to demonstrate how these variables work, we need to use them somewhere. The classes **.baseColor**, etc. will demonstrate the calculated values of each of the variables in the CSS file that is produced.

Getting Started

If you don't already have one in your project, add a new Gulp/Grunt configuration file, on this sample we will do it with Gulp. Make sure package.json includes gulp in its devDependencies, and add "gulp-less":

```
"devDependencies": {
        "gulp": "3.8.11",
        "gulp-less": "3.0.2",
        "rimraf": "2.3.2"
    }
```

Save your changes to the package.json file, and you should see that the all of the files referenced can be found in the Dependencies folder under NPM. If not, right-click on the NPM folder and select “Restore Packages.”

Now open gulpfile.js. Add a variable at the top to represent less:

```
var gulp = require("gulp"),
            rimraf = require("rimraf"),
            fx = require("fs"),
            less = require("gulp-less");
```

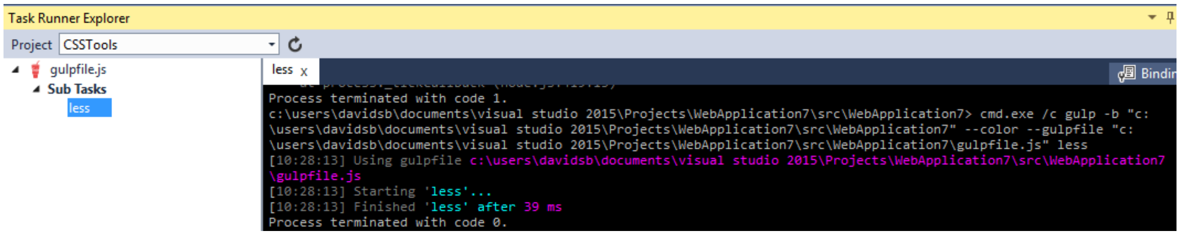
and a variable to be able to access the project properties

```
var project = require('./project.json');
```

Next, add a task to run less, using the syntax shown here:

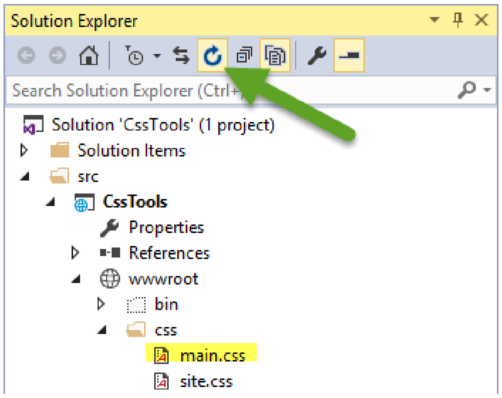
```
gulp.task("less", function () {
    return gulp.src('Styles/main.less')
            .pipe(less())
            .pipe(gulp.dest(project.webroot + '/css'))

});
```

Open the Task Runner Explorer (view>Other Windows > Task Runner Explorer). Among the tasks, you should see a new task named ‘less’. Run it, and you should have output similar to what is shown here:



```
Process terminated with code 1.
c:\users\davidsb\documents\visual studio 2015\Projects\WebApplication7\src\WebApplication7> cmd.exe /c gulp -b "c:
\users\davidsb\documents\visual studio 2015\Projects\WebApplication7\src\WebApplication7" --color --gulpfile "c:
\users\davidsb\documents\visual studio 2015\Projects\WebApplication7\src\WebApplication7\gulpfile.js" less
[10:28:13] Using gulpfile c:\users\davidsb\documents\visual studio 2015\Projects\WebApplication7\src\WebApplication7
\gulpfile.js
[10:28:13] Starting 'less'...
[10:28:13] Finished 'less' after 39 ms
Process terminated with code 0.
```

Now refresh your Solution Explorer and inspect the contents of the wwwroot/css folder. You should find a new file, main.css, there:



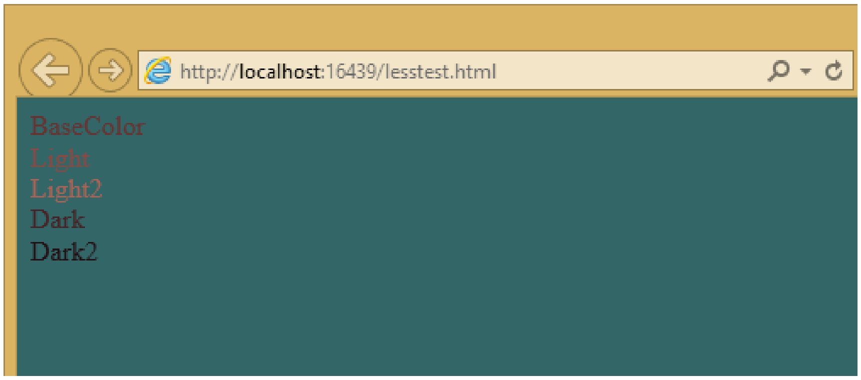
Open main.css and you should see something like the following:

```
body {
  background-color: #336666;
}
.baseColor {
  color: #663333;
}
.bgLight {
  color: #884a44;
}
.bgLight2 {
  color: #aa6355;
}
.bgDark {
  color: #442225;
}
.bgDark2 {
  color: #221114;
}
```

Add a simple HTML page to the wwwroot folder and reference main.css to see the color palette in action.

```
<!DOCTYPE html>

<html>
<head>
    <meta charset="utf-8" />
    <link href="css/main.css" rel="stylesheet" />
    <title></title>
</head>
<body>
    <div>
        <div class="baseColor">BaseColor</div>
        <div class="bgLight">Light</div>
        <div class="bgLight2">Light2</div>
        <div class="bgDark">Dark</div>
        <div class="bgDark2">Dark2</div>
    </div>

</body>
</html>
```

You can see that the 180 degree spin on @base used to produce @background resulted in the color wheel opposing color of @base:



BaseColor
Light
Light2
Dark
Dark2

Less also provides support for nested rules, as well as nested media queries. For example, defining nested hierarchies like menus can result in verbose CSS rules like these:

```
nav {
    height: 40px;
    width: 100%;
}
nav li {
    height: 38px;
    width: 100px;
}
nav li a:link {
    color: #000;
    text-decoration: none;
}
nav li a:visited {
    text-decoration: none;
    color: #CC3333;
}
nav li a:hover {
    text-decoration: underline;
    font-weight: bold;
}
nav li a:active {
    text-decoration: underline;
}
```

Ideally all of the related style rules will be placed together within the CSS file, but in practice there is nothing enforcing this rule except convention and perhaps block comments.

Defining these same rules using Less looks like this:

```
nav {
    height: 40px;
    width: 100%;
    li {
        height: 38px;
        width: 100px;
        a {
            color: #000;
            &:link { text-decoration:none}
            &:visited { color: #CC3333; text-decoration:none}
            &:hover { text-decoration:underline; font-weight:bold}
            &:active {text-decoration:underline}
        }
    }
}
```

Note that in this case, all of the subordinate elements of **nav** are contained within its scope. There is no longer any repetition of parent elements (nav, li, a), and the total line count has dropped as well (though some of that is a result of putting values on the same lines in the second example). It can be very helpful, organizationally, to see all of the rules for a given UI element within an explicitly bounded scope, in this case set off from the rest of the file by curly braces.

The **&** syntax is a Less selector feature, with **&** representing the current selector parent. So, within the a {…} block, **&** represents an **a** tag, and thus **&:link** is equivalent to **a:link**.

Media queries, extremely useful in creating responsive designs, can also contribute heavily to repetition and complexity in CSS. Less allows media queries to be nested within classes, so that the entire class definition doesn't need to be repeated within different top-level @media elements. For example, this CSS for a responsive menu:

```
.navigation {
    margin-top: 30%;
    width: 100%;
}
@media screen and (min-width: 40em) {
    .navigation {
        margin: 0;
    }
}
@media screen and (min-width: 62em) {
    .navigation {
        width: 960px;
        margin: 0;
    }
}
```

Can be better defined in Less as:

```
.navigation {
    margin-top: 30%;
    width: 100%;
    @media screen and (min-width: 40em) {
        margin: 0;
    }
    @media screen and (min-width: 62em) {
        width: 960px;
        margin: 0;
    }
}
```

Another feature of Less that we have already seen is its support for mathematical operations, allowing style attributes to be constructed from pre-defined variables. This makes updating related styles much easier, since the base variable can be modified and all dependent values change automatically.

CSS files, especially for large sites (and especially if media queries are being used), tend to get quite large over time, making working with them unwieldy. Less files can be defined separately, then pulled together using @import directives. Less can also be used to import individual CSS files, as well, if desired.

Mixins can accept parameters, and Less supports conditional logic in the form of *mixin guards*, which provide a declarative way to define when certain *mixins* take effect. A common use for *mixin* guards is to adjust colors based on how light or dark the source color is. Given a *mixin* that accepts a parameter for color, a *mixin* guard can be used to modify the *mixin* based on that color:

```
.box (@color) when (lightness(@color) >= 50%) {
    background-color: #000;
}
.box (@color) when (lightness(@color) < 50%) {
    background-color: #FFF;
}
.box (@color) {
    color: @color;
}

.feature {
    .box (@base);
}
```

Given our current @base value of #663333, this Less script will produce the following CSS:

```
.feature {
  background-color: #FFF;
  color: #663333;
}
```

Less provides a number of additional features, but this should give you some idea of the power of this preprocessing language.

Sass

Sass is similar to Less, providing support for many of the same features, but with slightly different syntax. It is built using Ruby, rather than JavaScript, and so has different setup requirements. The original Sass language did not use curly braces or semicolons, but instead defined scope using white space and indentation. In version 3 of Sass, a new syntax was introduced, SCSS ("Sassy CSS"). SCSS is similar to CSS in that it ignore indentation levels and whitespace, and instead uses semicolons and curly braces.

To install Sass, typically you would first install Ruby (pre-installed on Mac), and then run:

```
gem install sass
```

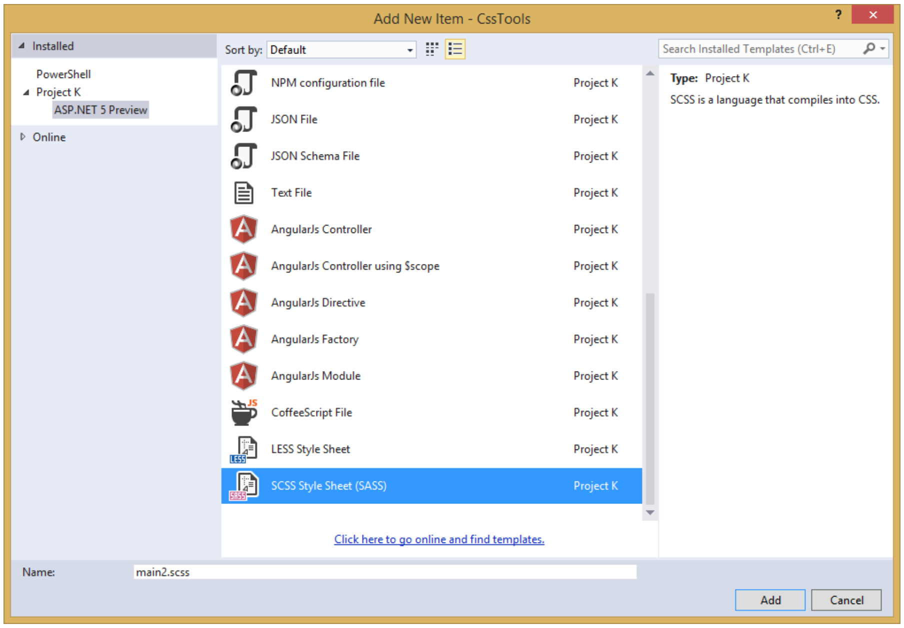
However, assuming you're running Visual Studio, you can get started with Sass in much the same way as you would with Less. Open package.json and add the "gulp-sass" package to "devDependencies" (note that you can include both Less and Sass in your project, but in practice you will typically use one or the other):

```
"devDependencies": {
"gulp": "3.8.11",
"gulp-less": "3.0.2",
"gulp-sass": "1.3.3",
"rimraf": "2.3.2"

}
```

Next, you can modify gulpfile.js to add a sass variable and a task to compile your Sass files and place the results in the wwwroot folder:

```
var gulp = require("gulp"),
       rimraf = require("rimraf"),
       fs = require("fs"),
       less = require("gulp-less"),
       sass = require("gulp-sass");
…
gulp.task("sass", function () {
    return gulp.src('Styles/main2.scss')
        .pipe(sass())
        .pipe(gulp.dest(project.webroot + '/css'))
});
```

Now you can add the Sass file main2.scss to the Styles folder in the root of the project:


Add New Item - CssTools
Installed
PowerShell
Project K
ASP.NET 5 Preview
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
NPM configuration file — Project K
JSON File — Project K
JSON Schema File — Project K
Text File — Project K
AngularJs Controller — Project K
AngularJs Controller using $scope — Project K
AngularJs Directive — Project K
AngularJs Factory — Project K
AngularJs Module — Project K
CoffeeScript File — Project K
LESS Style Sheet — Project K
SCSS Style Sheet (SASS) — Project K
Click here to go online and find templates.
Add
Cancel

Type: Project K

SCSS is a language that compiles into CSS.

Name: main2.scss

Open main2.scss and add the following:

```
$base: #CC0000;
body {
    background-color: $base;
}
```

Save all of your files. Now in Task Runner Explorer, you should see a sass task. Run it, refresh solution explorer, and look in the /wwwroot/css folder. There should be a main2.css file, with these contents:

```
body {
  background-color: #CC0000; }
```

Sass supports nesting in much the same was that Less does, providing similar benefits. Files can be split up by function and included using the @import directive:

```
@import 'anotherfile';
```

Sass supports mixins as well, using the *@mixin* keyword to define them and *@include* to include them, as in this example from sass-lang.com:

```
@mixin border-radius($radius) {
  -webkit-border-radius: $radius;
     -moz-border-radius: $radius;
      -ms-border-radius: $radius;
          border-radius: $radius;
}

.box { @include border-radius(10px); }
```

In addition to mixins, Sass also supports the concept of inheritance, allowing one class to extend another. It's conceptually similar to a mixin, but results in less CSS code. It's accomplished using the *@extend* keyword. First, let's see how we might use mixins, and the resulting CSS code. Add the following to your main2.scss file:

```
@mixin alert {
    border: 1px solid black;
    padding: 5px;
    color: #333333;
}

.success {
    @include alert;
    border-color: green;
}

.error {
    @include alert;
    color: red;
    border-color: red;
    font-weight:bold;
}
```

Examine the output in main2.css after running the sass task in Task Runner Explorer:

```
.success {
  border: 1px solid black;
  padding: 5px;
  color: #333333;
  border-color: green;
 }

.error {
  border: 1px solid black;
  padding: 5px;
  color: #333333;
  color: red;
  border-color: red;
  font-weight: bold;
}
```

Notice that all of the common properties of the alert mixin are repeated in each class. Now replace the alert mixin with a .alert class, and change @include to @extend (remembering to extend .alert, not alert):

```
.alert {
    border: 1px solid black;
    padding: 5px;
    color: #333333;
}

.success {
    @extend .alert;
    border-color: green;
}

.error {
    @extend .alert;
    color: red;
    border-color: red;
    font-weight:bold;
}
```

Run Sass once more, and examine the resulting CSS:

```
.alert, .success, .error {
  border: 1px solid black;
  padding: 5px;
  color: #333333; }

.success {
  border-color: green; }

.error {
  color: red;
  border-color: red;
  font-weight: bold; }
```

Now the properties are defined only as many times as needed, and better CSS is generated.

Sass also includes functions and conditional logic operations, similar to Less. In fact, the two languages' capabilities are very similar.

Less or Sass?

There is still no consensus as to whether it's generally better to use Less or Sass (or even whether to prefer the original Sass or the newer SCSS syntax within Sass). A recent poll conducted on twitter of mostly ASP.NET developers found that the majority preferred to use Less, by about a 2-to-1 margin. Probably the most important decision is to use one of these tools, as opposed to just hand-coding your CSS files. Once you've made that decision, both Less and Sass are good choices.

Font Awesome

In addition to CSS pre-compilers, another great resource for styling modern web applications is Font Awesome. Font Awesome is a toolkit that provides over 500 scalable vector icons that can be freely used in your web applications. It was originally designed to work with Bootstrap, but has no dependency on that framework, or on any JavaScript libraries.

The easiest way to get started with Font Awesome is to add a reference to it, using its public content delivery network (CDN) location:

```
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/font-awesome/4.3.0/css/font-awesome.min.css">
```

Of course, you can also quickly add it to your Visual Studio 2015 project by adding it to the "dependencies" in bower.json:

```
{
"name": "ASP.NET",
"private": true,
"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",
"Font-Awesome": "4.3.0"
}
```

Then, to get the stylesheet added to the wwwroot folder, modify gulpfile.js as follows:

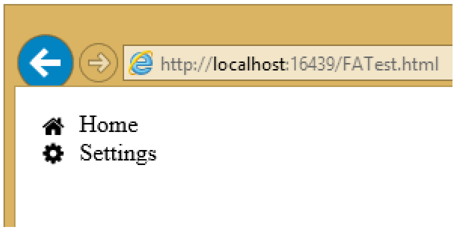
```
gulp.task("copy", ["clean"], function () {
    var bower = {
        "angular": "angular/angular*.{js,map}",
        "boostrap": "bootstrap/dist/**/*.{js,map,css,ttf,svg,woff,eot}",
        "boostrap-touch-carousel": "bootstrap-touch-carousel/dist/**/*.{js,css}",
        "hammer.js": "hammer.js/hammer*.{js,map}",
        "jquery": "jquery/jquery*.{js,map}",
        "jquery-validation": "jquery-validation/jquery.validate.js",
        "jquery-validation-unobtrusive": "jquery-validation-unobtrusive/jquery.validate.unobtrusive.js",
        "font-awesome": "Font-Awesome/**/*.{css,otf,eot,svg,ttf,woff,wof2}"
    };

    for (var destinationDir in bower) {
        gulp.src(paths.bower + bower[destinationDir])
                         .pipe(gulp.dest(paths.lib + destinationDir));
    }
});
```

Once this is in place (and saved), running the 'copy' task in Task Runner Explorer should copy the font awesome fonts and css files to /lib/font-awesome.

Once you have a reference to it on a page, you can add icons to your application by simply applying Font Awesome classes, typically prefixed with "fa-", to your inline HTML elements (such as <span> or <i>). As a very simple example, you can add icons to simple lists and menus using code like this:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title></title>
    <link href="lib/font-awesome/css/font-awesome.css" rel="stylesheet" />
</head>
<body>
    <ul class="fa-ul">
        <li><i class="fa fa-li fa-home"></i> Home</li>
        <li><i class="fa fa-li fa-cog"></i> Settings</li>
    </ul>
</body>
</html>
```

This produces the following in the browser:



Home
Settings

You can view a complete list of the available icons here:

http://fortawesome.github.io/Font-Awesome/icons/

Summary

Modern web applications increasingly demand responsive, fluid designs that are clean, intuitive, and easy to use from a variety of devices. Managing the complexity of the CSS stylesheets required to achieve these goals is best done using a pre-processor like Less or Sass. In addition, toolkits like Font Awesome quickly provide well-known icons to textual navigation menus and buttons, improving the overall user experience of your application.

TypeScript. Enterprise-scale JavaScript.

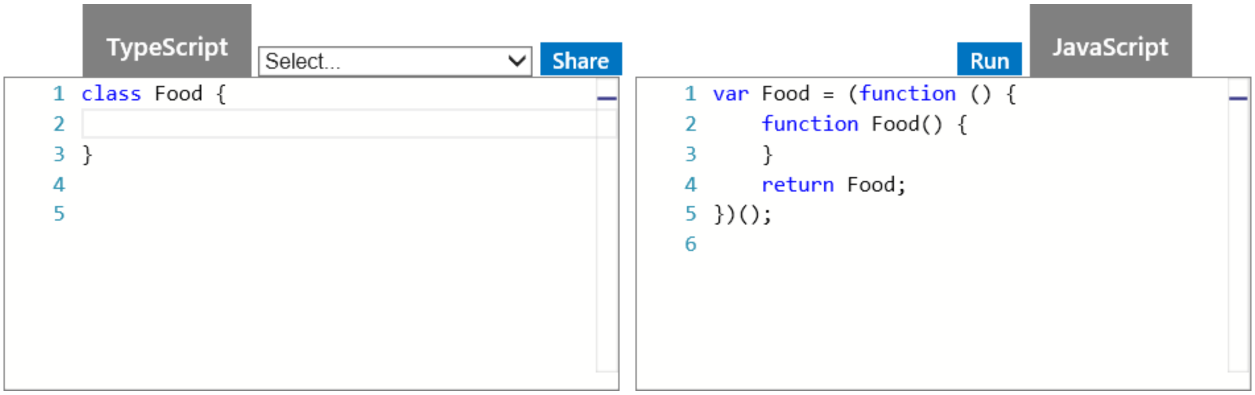
TypeScript helps you build enterprise scale applications on the client with the same smooth development experience, reliability, and scalability that you expect from a .NET language. TypeScript lets you work with familiar abstractions like strong types, interfaces, classes, namespaces, enums, generics, and lambda expressions. TypeScript code editors have intelligent code completion (Intellisense in Visual Studio) and navigation between object definitions and their references. When you're done coding, TypeScript compiles to pure JavaScript, leaving no TypeScript artifacts behind.

TypeScript empowers you to write code that runs on any browser and operating system. You will have access to a world of existing JavaScript libraries like node.js, angular, bootstrap, datejs, and hundreds of others.

Getting Started

TypeScript brings familiar object oriented programming constructs like classes, interfaces, and modules to the world of JavaScript.

You can try TypeScript right now by creating a .ts file in Visual Studio or by running the TypeScript Playground in your browser. As you can see on the image below, the TypeScript Playground translates TypeScript to JavaScript as you type.



```
class Food {

}
```

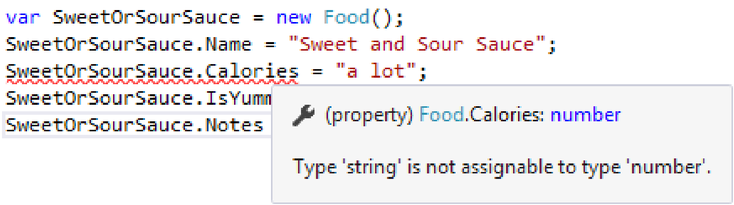
```
var Food = (function () {
    function Food() {
    }
    return Food;
})();
```

Both options, Visual Studio and the playground, run the **tsc.exe** TypeScript compiler on your behalf.

TypeScript is sometimes described as the "syntactic sugar" that makes JavaScript logic easier to express and read. For example, you can define a **Food** class and include properties with **string**, **number**, **Boolean** and **any** type annotations:

```
class Food {
    Name: string;
    Calories: number;
    IsYummy: Boolean;
    Notes: any;
}
```

The Visual Studio screenshot below demonstrates creating a new TypeScript instance of **Food** and assigning properties. If you assign the wrong type value to a property, the TypeScript editor will flag the error right away without waiting to fail at run time.

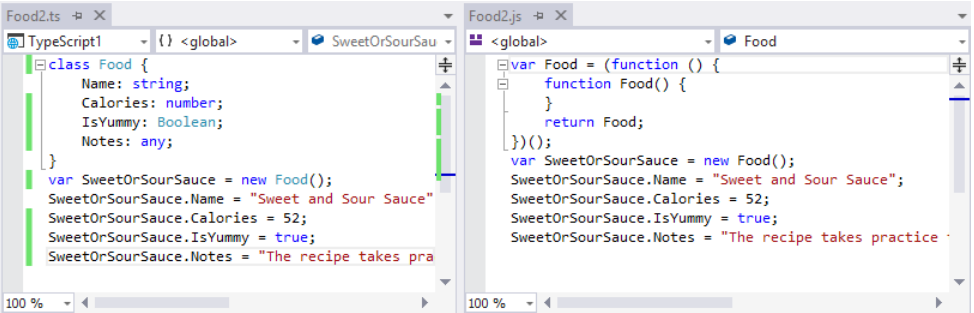


```
var SweetOrSourSauce = new Food();
SweetOrSourSauce.Name = "Sweet and Sour Sauce";
SweetOrSourSauce.Calories = "a lot";
SweetOrSourSauce.IsYumm
SweetOrSourSauce.Notes
```

(property) Food.Calories: number

Type 'string' is not assignable to type 'number'.

1. When the TypeScript compiler generates JavaScript (ensure that Visual Studio 2015 has the option 'compile on save' for TypeScript files checked. *Tools > Options > Text Editor > Typescript > Project*), the type annotations boil away leaving pure JavaScript. Here they are side-by-side in Visual Studio with the TypeScript file **Food.ts** on the left side and the generated JavaScript **Food.js** on the right:



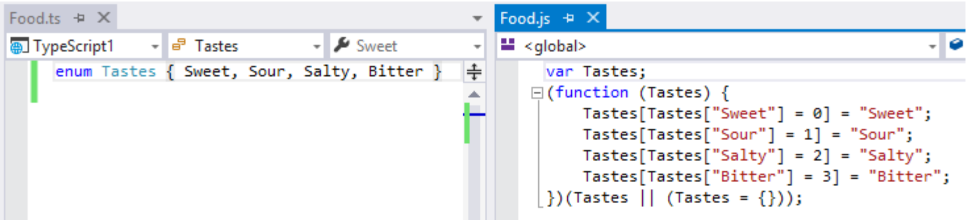
```
class Food {
    Name: string;
    Calories: number;
    IsYummy: Boolean;
    Notes: any;
}
var SweetOrSourSauce = new Food();
SweetOrSourSauce.Name = "Sweet and Sour Sauce"
SweetOrSourSauce.Calories = 52;
SweetOrSourSauce.IsYummy = true;
SweetOrSourSauce.Notes = "The recipe takes pra
```

```
var Food = (function () {
    function Food() {
    }
    return Food;
})();
var SweetOrSourSauce = new Food();
SweetOrSourSauce.Name = "Sweet and Sour Sauce";
SweetOrSourSauce.Calories = 52;
SweetOrSourSauce.IsYummy = true;
SweetOrSourSauce.Notes = "The recipe takes practice
```

The TypeScript **enum** keyword is another a classic example where TypeScript puts a friendly face on JavaScript. The **Tastes** enum below is easy to read at-a-glance:

```
enum Tastes { Sweet, Sour, Salty, Bitter };
```

The equivalent JavaScript is a little crankier and takes longer to read. The brevity and clarity of TypeScript really stands out when compared to the JavaScript output.



```
enum Tastes { Sweet, Sour, Salty, Bitter }
```

```
var Tastes;
(function (Tastes) {
    Tastes[Tastes["Sweet"] = 0] = "Sweet";
    Tastes[Tastes["Sour"] = 1] = "Sour";
    Tastes[Tastes["Salty"] = 2] = "Salty";
    Tastes[Tastes["Bitter"] = 3] = "Bitter";
})(Tastes || (Tastes = {}));
```

Note:

As you can see in the JavaScript output, enum members number starting from zero, but you can assign your own values. You can even treat the enum as a set and test members for inclusion very much like the .NET *flags* annotation:

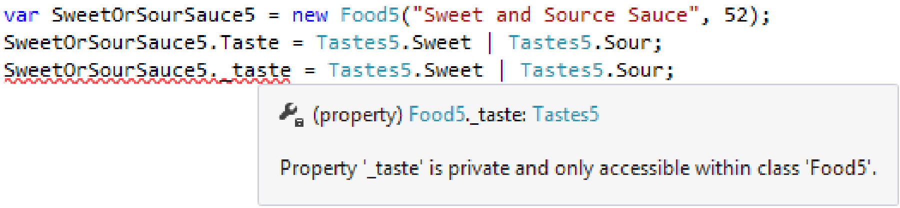
```
// assign each enum value by the power of 2, i.e. 1,2,4,8,16, etc.
enum Tastes4 { Sweet = 1, Sour = 2, Salty = 4, Bitter = 8 }

SweetOrSourSauce4.Taste = Tastes4.Sweet | Tastes4.Sour;

(SweetOrSourSauce4.Taste & Tastes4.Bitter) != 0; // false
(SweetOrSourSauce4.Taste & Tastes4.Salty) != 0; // false
(SweetOrSourSauce4.Taste & Tastes4.Sour) != 0; // true
(SweetOrSourSauce4.Taste & Tastes4.Sweet) != 0; // true
```

Language Features

TypeScript supports standard .NET language features like constructors, property accessors, and access modifiers to control scope. By giving the **Food** class a **constructor** and adding property accessors **get** and **set**, you control the flow of data to properties.

```
enum Tastes5 { Sweet, Sour, Salty, Bitter }

class Food5 {
    constructor(name: string, calories: number) {
    }

    private _name: string;
    get Name() {
        return this._name;
    }

    private _calories: number;
    get Calories() {
        return this._calories;
    }

    private _taste: Tastes5;
    get Taste(): Tastes5 { return this._taste }
    set Taste(value: Tastes5) {
        this._taste = value;
    }
}

var SweetOrSourSauce5 = new Food5("Sweet and Source Sauce", 52);
SweetOrSourSauce5.Taste = Tastes5.Sweet | Tastes5.Sour;
```

Add the **private** access modifier to members that shouldn't be visible outside the class. Access modifiers do not change the JavaScript output. Instead, the TypeScript editor display warnings that properties are not accessible.



```
var SweetOrSourSauce5 = new Food5("Sweet and Source Sauce", 52);
SweetOrSourSauce5.Taste = Tastes5.Sweet | Tastes5.Sour;
SweetOrSourSauce5._taste = Tastes5.Sweet | Tastes5.Sour;
```

(property) Food5._taste: Tastes5

Property '_taste' is private and only accessible within class 'Food5'.

TypeScript also defines a **protected** scope that allows access within the class and derived classes. You can explicitly mark a property with the default **public** access modifier to document that the member is available outside the class.

Inheritance

In TypeScript lingo, inheritance is called *heritage specification*. Use the **extends** keyword to inherit from a base class. The **SweetOrSourSauce** class below extends **Food** and populates its properties in the constructor. The base class constructor is called using the **super** keyword. The constructor signature does not need to match the base class.

```
class SweetOrSourSauce extends Food {
    constructor() {
        super("Sweet and Sour Sauce", 52);
        this.Taste = Tastes.Sweet | Tastes.Sour;
    }
}

var sauce = new SweetOrSourSauce();
```

Interfaces

An interface defines a series of public properties and methods that a class will implement. Interfaces encourage loose coupling that large scale applications require. The **ShelfLifeInterface** below has a Boolean **IsBad** property and a method to age the food a given number of days. The implementation will determine how that aging takes place.

```
interface ShelfLifeInterface {
    IsBad: Boolean;
    Age(days: number);
}
```

The **implements** keyword requires a class to fulfill the properties and methods of an interface. The **Food** class example below **implements** the **ShelfLifeInterface**. Also notice how the protected field **DaysRemaining** as a countdown to when the **IsBad** property becomes true. The **Age()** method decrements from the number of days remaining.

```
class Food implements ShelfLifeInterface {
    // omitted code

    protected DaysRemaining: number;

    get IsBad(): Boolean {
        return this.DaysRemaining < 0;
    }

    Age(days: number) {
        this.DaysRemaining -= days;
    }
}
```

Classes derived from **Food** can assign the protected **DaysRemaining** field and use the implemented **Age()** and **IsBad** members.

```
class Chard extends Food {
    constructor() {
        super("Chard", 7);
        this.Taste = Tastes.Bitter;
        this.DaysRemaining = 2;
    }
}

var salad = new Chard();
salad.Age(3);
if (salad.IsBad) {
    alert("The " + salad.Name + " has gone bad");
}
```

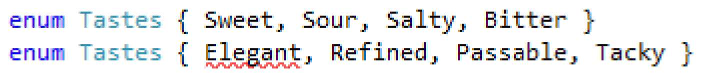
TypeScript allows *structural subtyping* that lets you treat structurally compatible objects as interchangeable. For example, we can create a new **Twinkies** class that has the same structure as the

ShelfLifeInterface, but does not use the **implements** keyword or directly reference the interface. Just the same, you can assign **Twinkies** instance to a **ShelfLifeInterface** variable.

```
// does not implement ShelfLifeInterface
class Twinkies {
    get IsBad(): Boolean {
        // twinkies never go bad
        return false;
    }

    Age(days: number) {
        // does nothing
    }
}

// CupCakes is of type ShelfLifeInterface and
// allows the structural twin Twinkies class to be assigned.
var CupCakes: ShelfLifeInterface = new Twinkies();
```

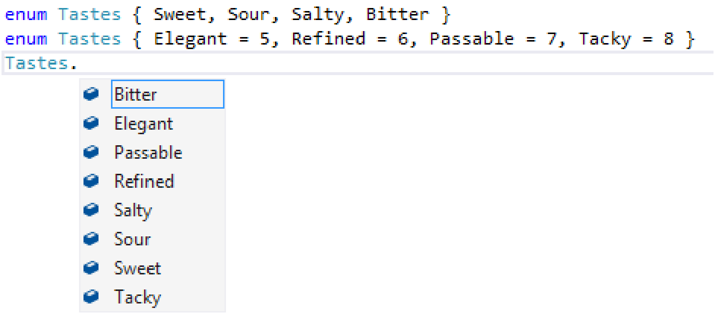
Modules

Modules are analogous to namespaces in .NET and support encapsulation of classes, interfaces, functions, and variables. Like namespaces, modules organize logic and prevent name clashes. Consider the **Tastes** enumeration in previous examples. If you introduce a second **Tastes** with a different fundamental meaning and purpose, you will get an error, but not an error a .NET programmer expects:



```
enum Tastes { Sweet, Sour, Salty, Bitter }
enum Tastes { Elegant, Refined, Passable, Tacky }
```

The error is *In an enum with multiple declarations, only one declaration can omit an initializer for its first enum element*. TypeScript doesn't have an issue with an enum with the same name, it minds that we're adding to the enumeration without telling TypeScript where the new enumeration members should be placed ordinally. TypeScript is perfectly happy if you assign ordinal values for the new members, although the resulting list makes no semantic sense:



```
enum Tastes { Sweet, Sour, Salty, Bitter }
enum Tastes { Elegant = 5, Refined = 6, Passable = 7, Tacky = 8 }
Tastes.
```

To make a member unique, use the **module** keyword to create a distinct context. Use the **export** keyword to mark members that should be visible outside the module. The example below establishes **Flavor** and **Fashion** modules that have distinctive **Tastes** members, allowing you to address **Flavor.Tastes** and **Fashion.Tastes** independently.

```
module Flavor {
    export enum Tastes { Sweet, Sour, Salty, Bitter }
}
```

```
module Fashion {
    export enum Tastes { Elegant, Refined, Passable, Tacky }
}

class Party {
    ChipsTaste: Flavor.Tastes = Flavor.Tastes.Salty;
    DecorTaste: Fashion.Tastes = Fashion.Tastes.Passable;
}
```

As your applications become larger, you can place modules in separate files. Use the **reference** directive to use TypeScript files located at the **path** attribute location.

```
/// < reference path="Flavor.ts" />
/// < reference path="Fashion.ts" />

class Party {
    ChipsTaste: Flavor.Tastes = Flavor.Tastes.Salty;
    DecorTaste: Fashion.Tastes = Fashion.Tastes.Passable;
}
```

Visual Studio configures the TypeScript compiler to satisfy dependencies and creates the minimal JavaScript output. The example generated JavaScript below still includes the reference paths as comments, but no longer requires them.

```
/// < reference path="Flavor.ts" />
/// < reference path="Fashion.ts" />
var Party = (function () {
    function Party() {
        this.ChipsTaste = 2 /* Salty */;
        this.DecorTaste = 2 /* Passable */;
    }
    return Party;
})();
```

Note: The modules we have looked at here are referred to as *internal* modules. Another mechanism called *external* modules supports creating declaration files that encapsulate existing JavaScript.

Generics

The TypeScript implementation of generics will look quite familiar to .NET programmers. Consider the class **List** below of **<T>**. During development, you're assured that items will contain only elements of **<T>**.

```
class List<T> {
    items: Array<T> = new Array<T>();
    add(item: T) {
        this.items.push(item);
    }
}
```

The generics syntax disappears on compilation and leaves the generated JavaScript code without artifacts.

```
var List = (function () {
    function List() {
        this.items = new Array();
    }
    List.prototype.add = function (item) {
        this.items.push(item);
    };
    return List;
})();
```

Once you establish the type of the class, you can operate in perfect type safety. The shoppingList instance is a **List** of string and won't accept other input. Again, this safety only exists during development. If you were to edit the compiled JavaScript, the array will accept whatever you decide to place into it.

```
var shoppingList = new List<string>();
shoppingList.add("Potatoes");
shoppingList.add("Eggs");
shoppingList.add("Artichokes");
shoppingList.add(123); // error: number is not assignable to string
```

You can use the same generic **List<T>** to hold instances of a new **Item** class without changing the **List** logic.

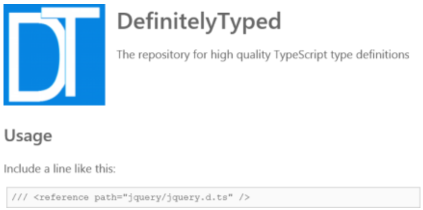
```
class Item {

    constructor(name: string, count: number) {
        this.name = name;
        this.count = count;
    }
    name: string;
    count: number;
}

var shoppingList = new List<Item>();
shoppingList.add(new Item("Bananas", 6));
shoppingList.add(new Item("Eggs", 12));
shoppingList.add(new Item("Milk", 1));
```

The items member is an Array that includes a **forEach()** method. The argument to **forEach()** is a callback function that is neatly handled using a lambda expression.

```
shoppingList.items.forEach(item => {
    console.log(item.count + " of " + item.name);
});
```

TypeScript Ecosystem

The external modules organized in declaration files with the extension .d.ts, functions as an interface to existing JavaScript libraries. The DefinitelyTyped repository has hundreds of files wrapping popular JavaScript libraries like jQuery, angularjs, Boostrap, d3, requirejs, knockout, and node.js to mention a few. You can download the files from Github or Nuget.



DefinitelyTyped

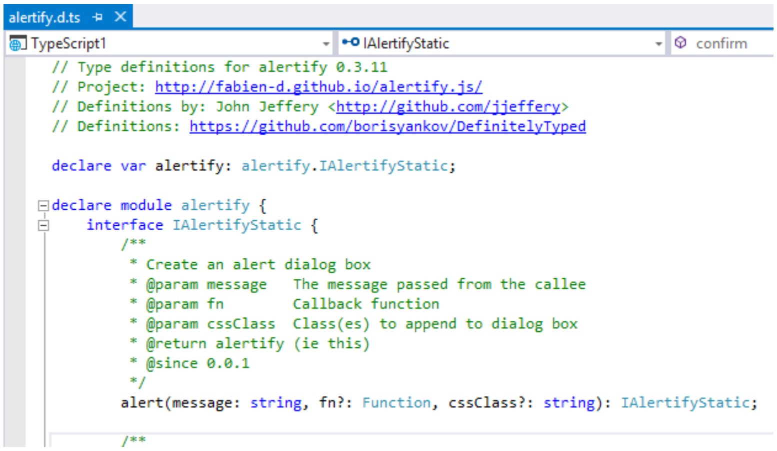
The repository for high quality TypeScript type definitions

Usage

Include a line like this:

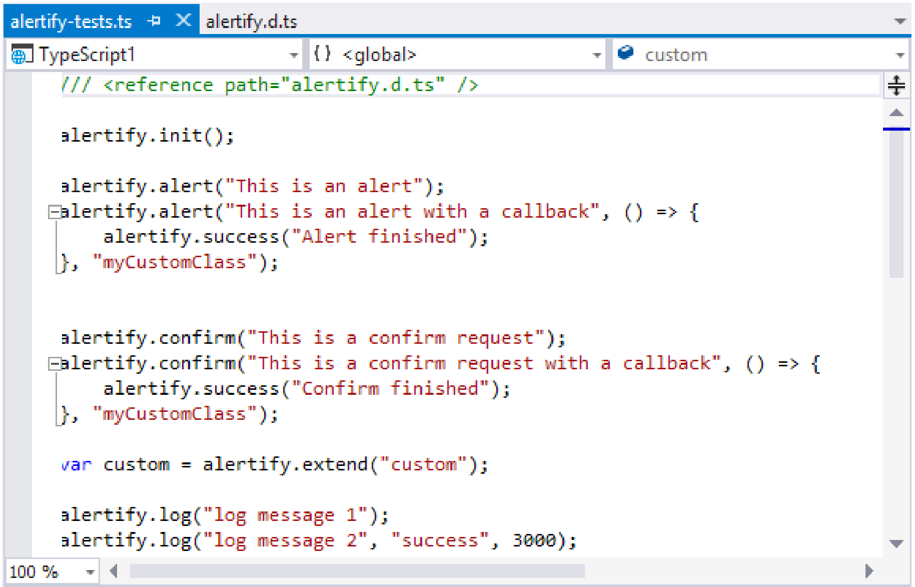
```
/// <reference path="jquery/jquery.d.ts" />
```

For a minimal example, you can use the **alertify.d.ts** from the DefinitelyTyped repository. Alertify is a small, focused library that presents alert and confirmation dialogs.



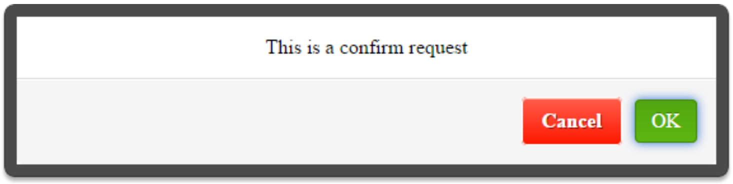
```
// Type definitions for alertify 0.3.11
// Project: http://fabien-d.github.io/alertify.js/
// Definitions by: John Jeffery <http://github.com/jjeffery>
// Definitions: https://github.com/borisyankov/DefinitelyTyped

declare var alertify: alertify.IAlertifyStatic;

declare module alertify {
    interface IAlertifyStatic {
        /**
         * Create an alert dialog box
         * @param message   The message passed from the callee
         * @param fn        Callback function
         * @param cssClass  Class(es) to append to dialog box
         * @return alertify (ie this)
         * @since 0.0.1
         */
        alert(message: string, fn?: Function, cssClass?: string): IAlertifyStatic;

        /**
```

The repository usually includes a tests file companion for each declaration file. In this case, the file is **alertify-tests.ts** and references the declaration file **alertify.d.ts**. These test files are great as informal usage references.



```
/// <reference path="alertify.d.ts" />

alertify.init();

alertify.alert("This is an alert");
alertify.alert("This is an alert with a callback", () => {
    alertify.success("Alert finished");
}, "myCustomClass");


alertify.confirm("This is a confirm request");
alertify.confirm("This is a confirm request with a callback", () => {
    alertify.success("Confirm finished");
}, "myCustomClass");

var custom = alertify.extend("custom");

alertify.log("log message 1");
alertify.log("log message 2", "success", 3000);
```

You will still need to need the underlying JavaScript files that the declaration file is referencing. The **alertify-webpack** package was added to handle styling. Here is the running example:


This is a confirm request

This example is trivial, but using the same mechanism, you can work with a growing bank of powerful frameworks and components. It's like having your own large general store where you can grab the client software components you need, as you need them. For example, you could include the AngularJS framework used to create single-page applications with a client-side model-view-controller architecture. Use the popular BootStrap library to handle styling and you're well on your way.

Note: TypeScript is written in TypeScript and has its own declaration file. This is some very elegant code to learn and get ideas from.

Summary

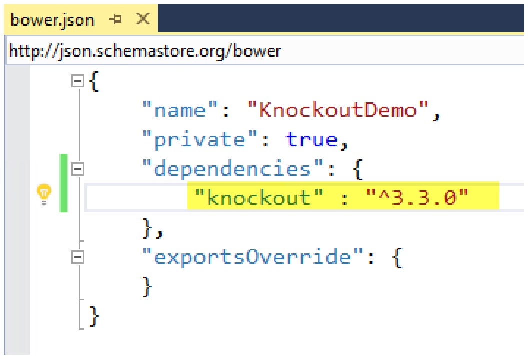
TypeScript brings a rich development environment to help you build robust client-side applications, using familiar, tried-and-true mechanisms. The tooling provides support for Intellisense, Peek and Goto Definition, Find All References and pop-up hint windows. You can use modules to organize your code into distinct partitions that you can combine without complication to produce large-scale applications. Language features like type safety and generics help you build reusable components that you can rely on. The large and diverse TypeScript ecosystem of existing code and new tooling makes TypeScript a great fit for web developers.

Knockout.js . MVVM Library.

Knockout is a popular JavaScript library that simplifies the creation of complex data-based user interfaces. It can be used alone or with other libraries, such as jQuery. Its primary purpose is to bind UI elements to an underlying data model defined as a JavaScript object, such that when changes are made to the UI, the model is updated, and vice versa. Knockout facilitates the use of a Model-View-ViewModel (MVVM) pattern in a web application's client-side behavior. The two main concepts one must learn when working with Knockout's MVVM implementation are Observables and Bindings.

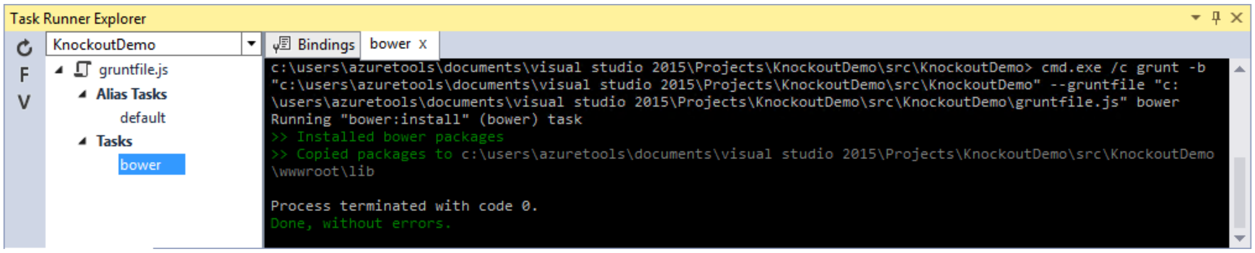
Getting Started

Knockout is deployed as a single JavaScript file, so installing and using it is very straightforward. In Visual Studio 2015, you can simply add knockout as a dependency and Visual Studio will use bower to retrieve it. Assuming you already have bower and grunt configured (the ASP.NET 5 Starter Template comes with them already set up), open *bower.json* in your ASP.NET 5 project, and add the knockout dependency as shown here:



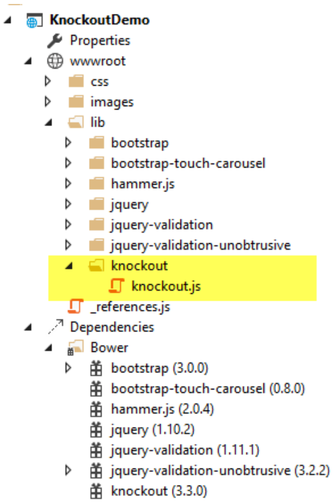
```
{
    "name": "KnockoutDemo",
    "private": true,
    "dependencies": {
        "knockout" : "^3.3.0"
    },
    "exportsOverride": {
    }
}
```

With this in place, you can then manually run bower by opening the Task Runner Explorer (under View > Other Windows > Task Runner Explorer) and then under Tasks, right-click on bower and select Run. The result should appear similar to this:



```
c:\users\azuretools\documents\visual studio 2015\Projects\KnockoutDemo\src\KnockoutDemo> cmd.exe /c grunt -b
"c:\users\azuretools\documents\visual studio 2015\Projects\KnockoutDemo\src\KnockoutDemo" --gruntfile "c:
\users\azuretools\documents\visual studio 2015\Projects\KnockoutDemo\src\KnockoutDemo\gruntfile.js" bower
Running "bower:install" (bower) task
>> Installed bower packages
>> Copied packages to c:\users\azuretools\documents\visual studio 2015\Projects\KnockoutDemo\src\KnockoutDemo
\wwwroot\lib

Process terminated with code 0.
Done, without errors.
```

Now if you look in your project's *wwwroot* folder, you should see knockout installed under the lib folder.



```
KnockoutDemo
  Properties
  wwwroot
    css
    images
    lib
      bootstrap
      bootstrap-touch-carousel
      hammer.js
      jquery
      jquery-validation
      jquery-validation-unobtrusive
      knockout
        knockout.js
    _references.js
  Dependencies
    Bower
      bootstrap (3.0.0)
      bootstrap-touch-carousel (0.8.0)
      hammer.js (2.0.4)
      jquery (1.10.2)
      jquery-validation (1.11.1)
      jquery-validation-unobtrusive (3.2.2)
      knockout (3.3.0)
```

It's recommended that in your production environment you reference knockout via a Content Delivery Network, or CDN, as this increases the likelihood that your users will already have a cached copy of the file and thus will not need to download it at all. Knockout is available on several CDNs, including the Microsoft Ajax CDN, here:

http://ajax.aspnetcdn.com/ajax/knockout/knockout-3.3.0.js

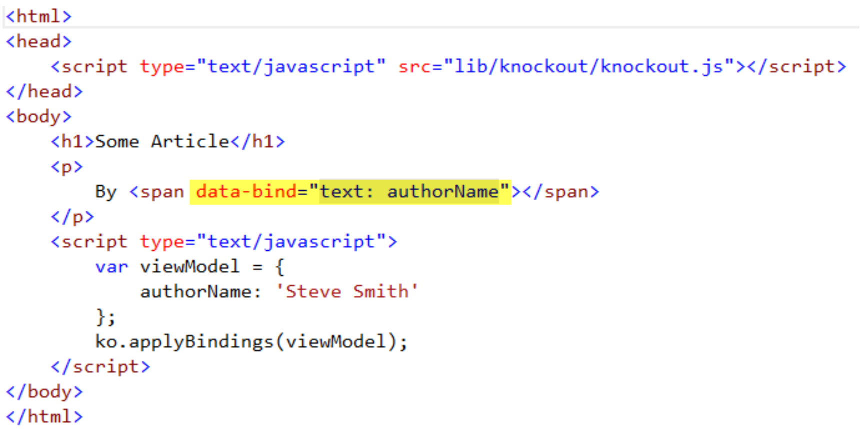
To include Knockout on a page that will use it, simply add a <script> element referencing the file from wherever you will be hosting it (with your application, or via a CDN):

```
<script type="text/javascript" src="knockout-3.3.0.js"></script>
```

Observables, ViewModels, and Simple Binding

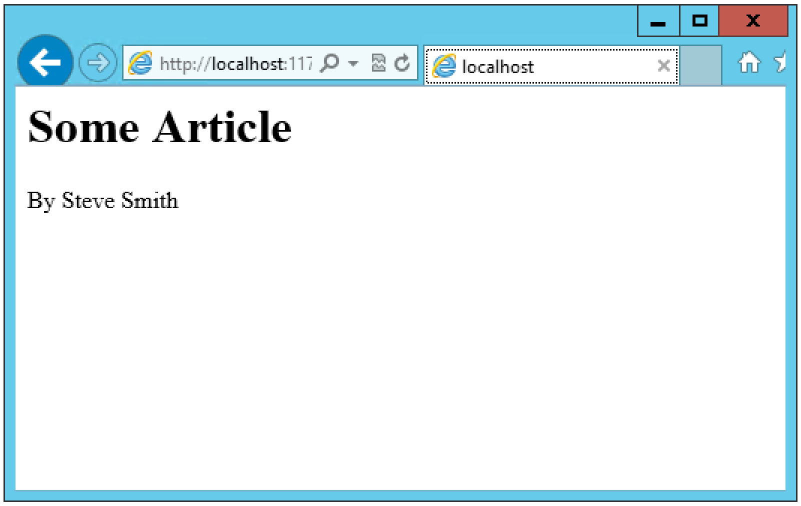
You may already be familiar with using JavaScript to manipulate elements on a web page, either via direct access to the DOM or using a library like jQuery. Typically this kind of behavior is achieved by writing code to directly set element values in response to certain user actions. With Knockout, a declarative approach is taken instead, through which elements on the page are bound to properties on an object. Instead of writing code to manipulate DOM elements, user actions simply interact with the ViewModel object, and Knockout takes care of ensuring the page elements are synchronized.

As a simple example, consider the following page. It includes a <span> element with a data-bind attribute indicating that the text content should be bound to authorName. Next, in a JavaScript block a variable viewModel is defined with a single property, authorName, set to some value. Finally, a call to ko.applyBindings is made, passing in this viewModel variable.



```
<html>
<head>
    <script type="text/javascript" src="lib/knockout/knockout.js"></script>
</head>
<body>
    <h1>Some Article</h1>
    <p>
        By <span data-bind="text: authorName"></span>
    </p>
    <script type="text/javascript">
        var viewModel = {
            authorName: 'Steve Smith'
        };
        ko.applyBindings(viewModel);
    </script>
</body>
</html>
```

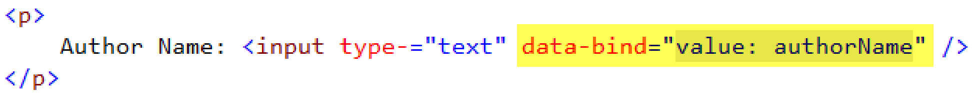
When viewed in the browser, the content of the <span> element is replaced with the value in the viewModel variable:



Some Article

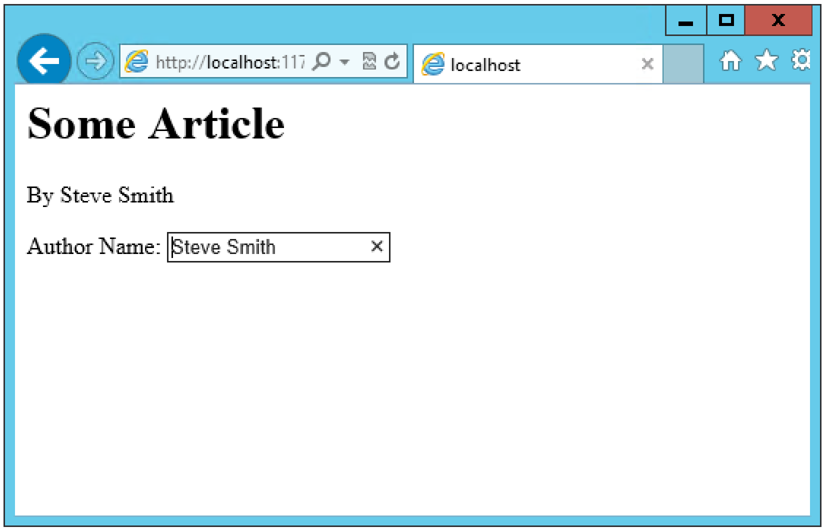
By Steve Smith

So, this is great, and we have simple one-way binding working. Notice that nowhere in the code did we write JavaScript to assign a value to the span's contents. If we want to manipulate the ViewModel, we can take this a step further and add an HTML input textbox, and bind to its value, like so:



```
<p>
    Author Name: <input type-="text" data-bind="value: authorName" />
</p>
```

Reloading the page, we see that this value is indeed bound to the input box:



Some Article

By Steve Smith

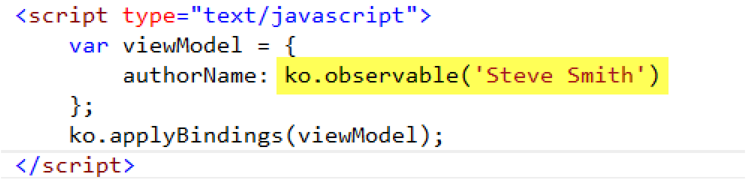
Author Name: Steve Smith

However, if we change the value in the textbox, the corresponding value in the <span> element doesn't change. Why not?

The issue is that nothing notified the <span> that it needed to be updated. Simply updating the ViewModel isn't by itself sufficient, unless the ViewModel's properties are wrapped in a special type. We need to use **observables** in the ViewModel for any properties that need to have changes automatically updated as they occur. By changing the ViewModel to use ko.observable("value") instead of just "value", the ViewModel will update any HTML elements that are bound to its value whenever a change occurs. Note that input boxes don't update their value until they lose focus, so you won't see changes to bound elements as you type.

Note: Adding support for live updating after each keypress is simply a matter of adding *valueUpdate: "afterkeydown"* to the data-bind attribute's contents.

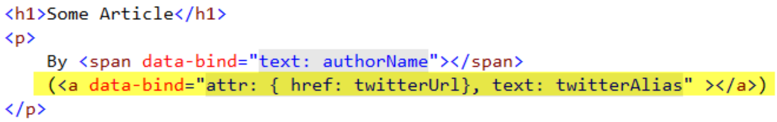
Our viewModel, after updating it to use ko.observable:



```
<script type="text/javascript">
    var viewModel = {
        authorName: ko.observable('Steve Smith')
    };
    ko.applyBindings(viewModel);
</script>
```

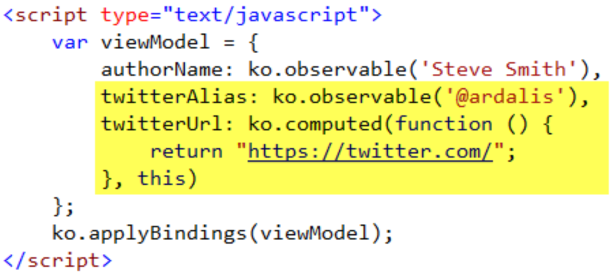
Knockout supports a number of different kinds of bindings. So far we've seen how to bind to text and to value. You can also bind to any given attribute. For instance, to create a hyperlink with an anchor tag, the 'src' attribute can be bound to the viewModel. Knockout also supports binding to functions. To demonstrate this, let's update the viewModel to include the author's twitter handle, and display the twitter handle as a link to the author's twitter page. We'll do this in three stages.

First, add the HTML to display the hyperlink, which we'll show in parentheses after the author's name:



```
<h1>Some Article</h1>
<p>
    By <span data-bind="text: authorName"></span>
    (<a data-bind="attr: { href: twitterUrl}, text: twitterAlias" ></a>)
</p>
```

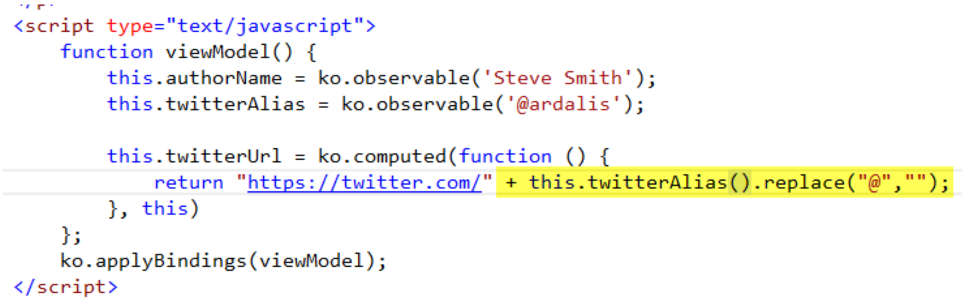
Next, update the viewModel to include the twitterUrl and twitterAlias properties:



```
<script type="text/javascript">
    var viewModel = {
        authorName: ko.observable('Steve Smith'),
        twitterAlias: ko.observable('@ardalis'),
        twitterUrl: ko.computed(function () {
            return "https://twitter.com/";
        }, this)
    };
    ko.applyBindings(viewModel);
</script>
```

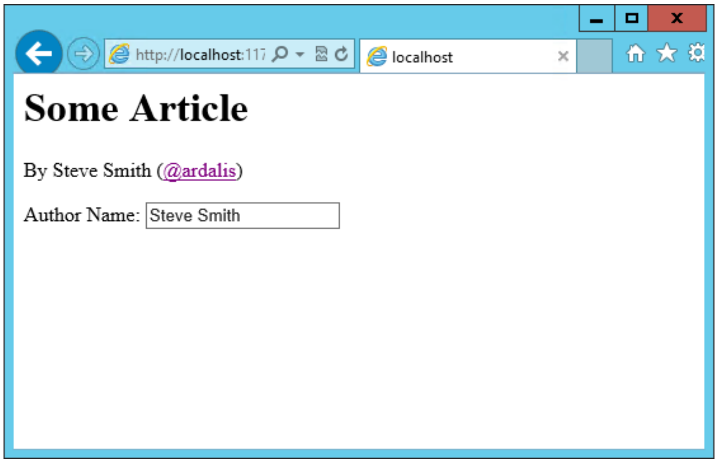
Notice that at this point we haven't yet updated the twitterUrl to go to the correct URL for this twitter alias – it's just pointing at twitter.com. Also notice that we're using a new Knockout function, *computed*, for twitterUrl. This is an observable function that will notify any UI elements if it changes. However, for it to have access to other properties in the viewModel, we need to change how we are creating the viewModel, so that each property is its own statement.

The revised viewModel declaration is shown here. It is now declared as a function. Notice that each property is its own statement now, ending with a semicolon. Also notice that to access the twitterAlias property value, we need to execute it, so its reference includes ().



```
<script type="text/javascript">
    function viewModel() {
        this.authorName = ko.observable('Steve Smith');
        this.twitterAlias = ko.observable('@ardalis');

        this.twitterUrl = ko.computed(function () {
            return "https://twitter.com/" + this.twitterAlias().replace("@","");
        }, this)
    };
    ko.applyBindings(viewModel);
</script>
```

The result works as expected in the browser:



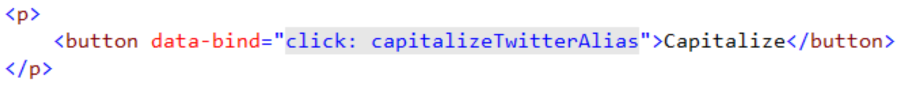
Some Article

By Steve Smith (@ardalis)

Author Name: Steve Smith

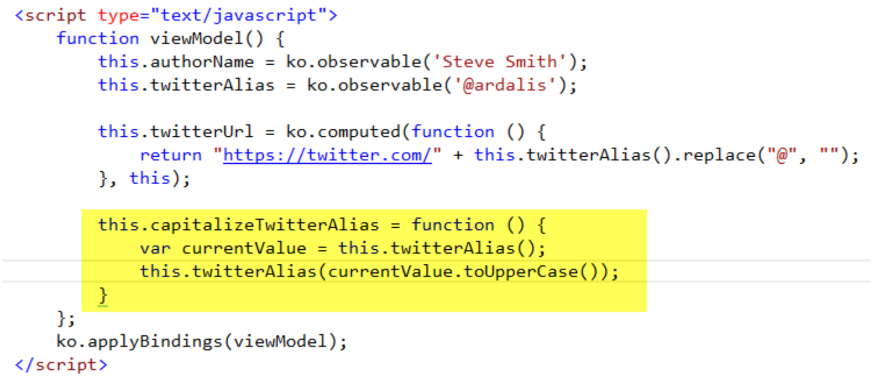
Knockout also supports binding to certain UI element events, such as the click event. This allows you to easily and declaratively bind UI elements to functions within the application's viewModel. As a simple example, we can add a button that, when clicked, modifies the author's twitterAlias to be all caps.

First, we add the button, binding to the button's click event, and referencing the function name we're going to add to the viewModel:



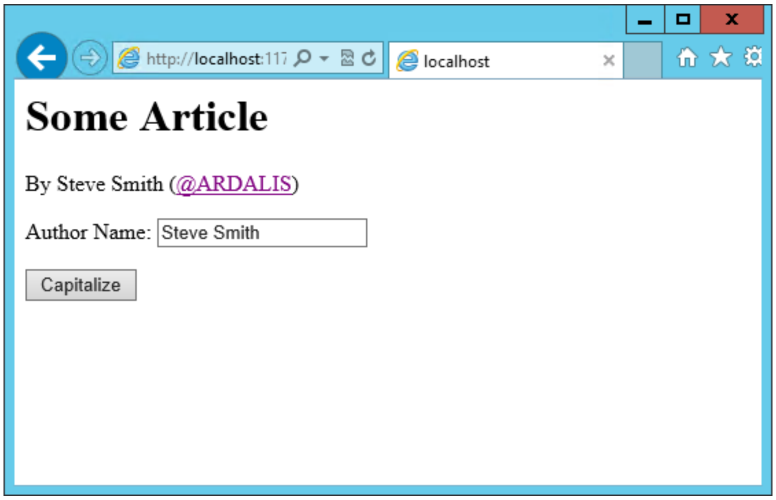
```
<p>
    <button data-bind="click: capitalizeTwitterAlias">Capitalize</button>
</p>
```

Then, add the function to the viewModel, and wire it up to modify the viewModel's state. Notice that to set a new value to the twitterAlias property, we call it as a method and pass in the new value.



```
<script type="text/javascript">
    function viewModel() {
        this.authorName = ko.observable('Steve Smith');
        this.twitterAlias = ko.observable('@ardalis');

        this.twitterUrl = ko.computed(function () {
            return "https://twitter.com/" + this.twitterAlias().replace("@", "");
        }, this);

        this.capitalizeTwitterAlias = function () {
            var currentValue = this.twitterAlias();
            this.twitterAlias(currentValue.toUpperCase());
        }
    };
    ko.applyBindings(viewModel);
</script>
```

Running the code and clicking the button modifies the displayed link as expected:



Some Article

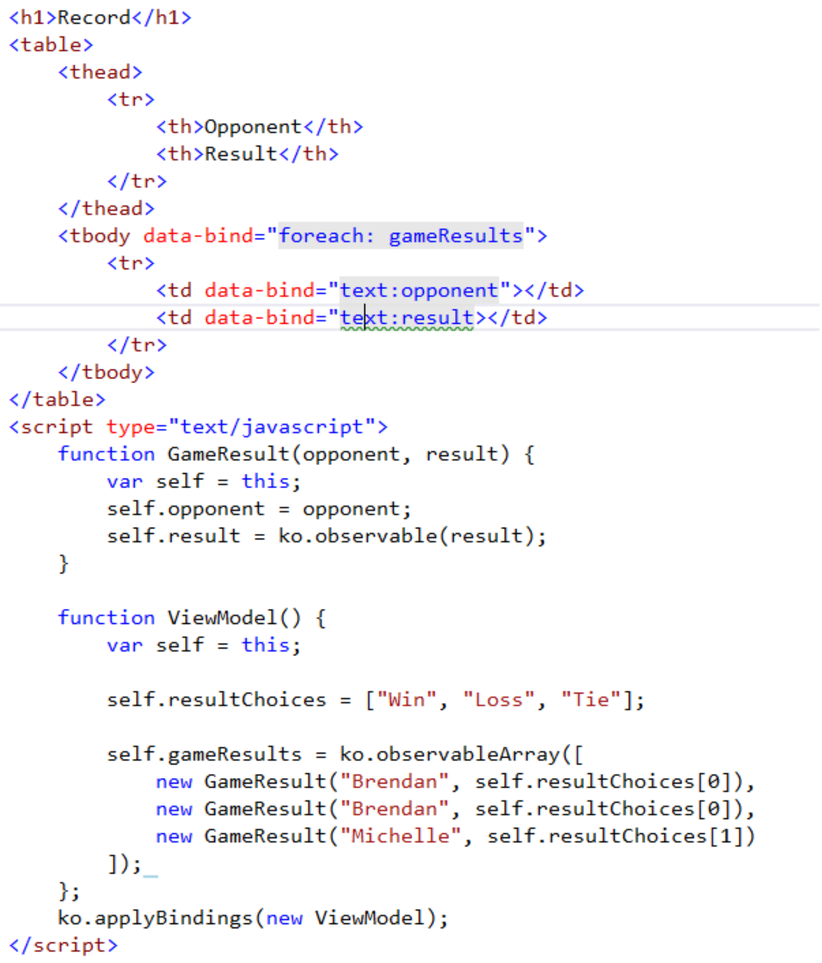
By Steve Smith (@ARDALIS)

Author Name: Steve Smith

Capitalize

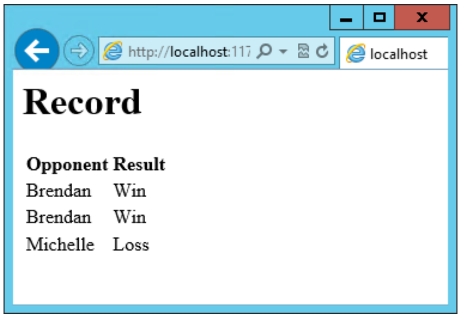
Control Flow

Knockout includes bindings that can perform conditional and looping operations. Looping operations are especially useful for binding lists of data to UI lists, menus, and grids or tables. The *foreach* binding will iterate over an array. When used with an *observable array*, it will automatically update the UI elements when items are added or removed from the array, without re-creating every element in the UI tree. The following example uses a new viewModel which includes an observable array of game results. It is bound to a simple table with two columns using a foreach binding on the <tbody> element. Each <tr> element within <tbody> will be bound to an element of the gameResults collection.



```
<h1>Record</h1>
<table>
    <thead>
        <tr>
            <th>Opponent</th>
            <th>Result</th>
        </tr>
    </thead>
    <tbody data-bind="foreach: gameResults">
        <tr>
            <td data-bind="text:opponent"></td>
            <td data-bind="text:result></td>
        </tr>
    </tbody>
</table>
<script type="text/javascript">
    function GameResult(opponent, result) {
        var self = this;
        self.opponent = opponent;
        self.result = ko.observable(result);
    }

    function ViewModel() {
        var self = this;

        self.resultChoices = ["Win", "Loss", "Tie"];

        self.gameResults = ko.observableArray([
            new GameResult("Brendan", self.resultChoices[0]),
            new GameResult("Brendan", self.resultChoices[0]),
            new GameResult("Michelle", self.resultChoices[1])
        ]);
    };
    ko.applyBindings(new ViewModel);
</script>
```

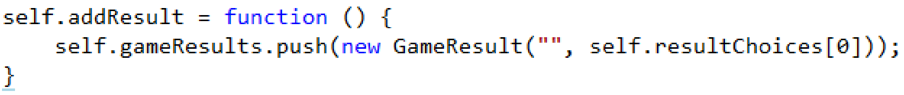
Notice that this time we're using ViewModel with a capital "V" because we expect to construct it using "new" (in the applyBindings call). When executed, the page results in the following output:



Record

```
Opponent Result
Brendan   Win
Brendan   Win
Michelle  Loss
```

To demonstrate that the observable collection is working, let's add a bit more functionality. We can include the ability to record the results of another game to the ViewModel, and then add a button and some UI to work with this new function. First, let's create the addResult method:

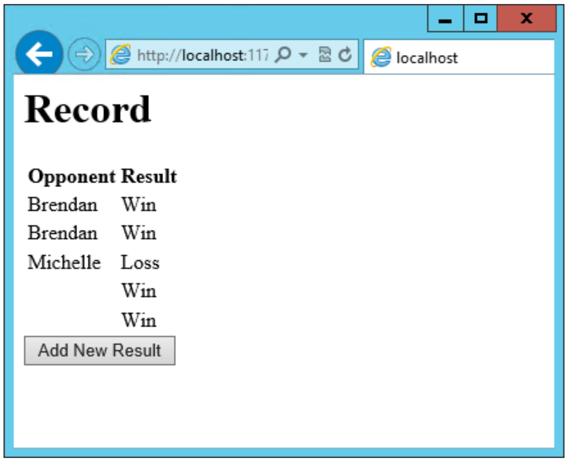


```
self.addResult = function () {
    self.gameResults.push(new GameResult("", self.resultChoices[0]));
}
```

Bind this method to a button using the *click* binding:

```
    </tbody>
</table>
<button data-bind="click: addResult">Add New Result</button>
```

Open the page in the browser and click the button a couple of times, resulting in a new table row with each click:



Record

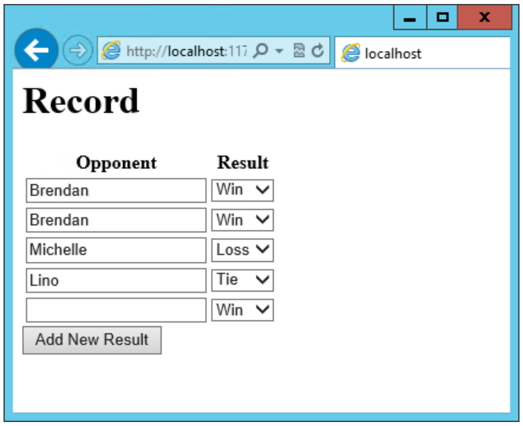
Opponent	Result
Brendan	Win
Brendan	Win
Michelle	Loss
	Win
	Win

There are a few ways to support adding new records in the UI, typically either inline or in a separate form. We can easily modify the table to use textboxes and dropdownlists so that the whole thing is editable. Just change the <tr> element as shown:

```
<tbody data-bind="foreach: gameResults">
    <tr>
        <td><input data-bind="value:opponent" /></td>
        <td><select data-bind="options: $root.resultChoices, value:result, optionsText: $data"></select></td>
    </tr>
</tbody>
```

Note that $root refers to the root ViewModel, which is where the possible choices are exposed. $data refers to whatever the current model is within a given context – in this case it refers to an individual element of the resultChoices array, each of which is a simple string.

With this change, the entire grid becomes editable:

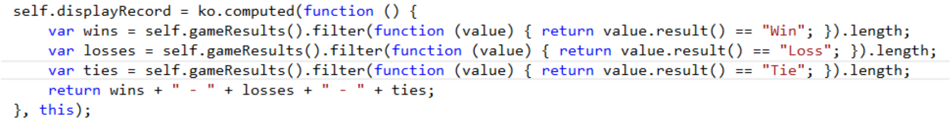


Record

Opponent	Result
Brendan	Win
Brendan	Win
Michelle	Loss
Lino	Tie
	Win

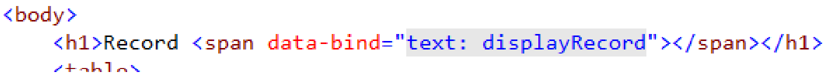
Now, if we weren't using Knockout, we could achieve all of this using jQuery, but most likely it would not be nearly as efficient. Knockout tracks which bound data items in the ViewModel correspond to which UI elements, and only updates those elements that need to be added, removed, or updated. It would take significant effort to achieve this ourselves using jQuery or direct DOM manipulation, and even then if we then wanted to display aggregate results (such as a win-loss record) based on the table's data, we would need to once more loop through it and parse the HTML elements. With Knockout, displaying the win-loss record is trivial. We can perform the calculations within the ViewModel itself, and then display it with a simple text binding and a <span>.

To build the win-loss record string, we can use a computed observable. Note that references to observable properties within the ViewModel must be function calls, otherwise they will not retrieve the value of the observable (i.e. gameResults() not gameResults in the code shown):



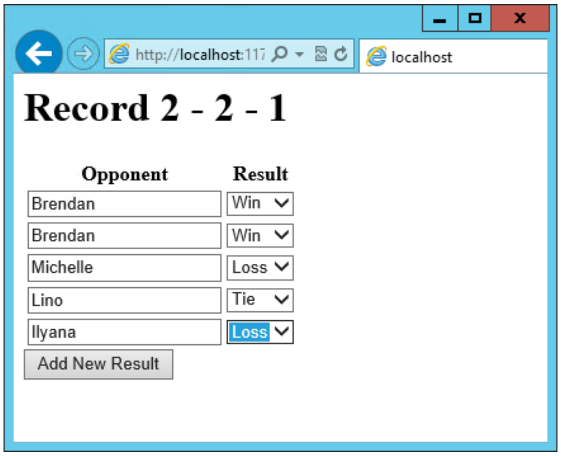
```
self.displayRecord = ko.computed(function () {
    var wins = self.gameResults().filter(function (value) { return value.result() == "Win"; }).length;
    var losses = self.gameResults().filter(function (value) { return value.result() == "Loss"; }).length;
    var ties = self.gameResults().filter(function (value) { return value.result() == "Tie"; }).length;
    return wins + " - " + losses + " - " + ties;
}, this);
```

Bind this function to a span within the <h1> element at the top of the page:



```
<body>
    <h1>Record <span data-bind="text: displayRecord"></span></h1>
    <table>
```

The result:



Record 2 - 2 - 1

Opponent	Result
Brendan	Win
Brendan	Win
Michelle	Loss
Lino	Tie
Ilyana	Loss

Adding rows or modifying the selected element in any row's Result column will update the record shown at the top of the window.

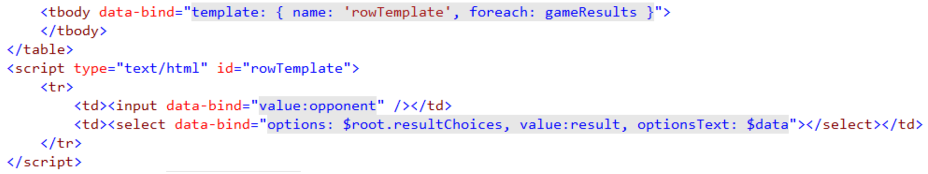
In addition to binding to values, you can also use almost any legal JavaScript expression within a binding. For example, if a UI element should only appear under certain conditions, such as when a value exceeds a certain threshold, you can specify this logically within the binding expression:

<div data-bind="visible: customerValue > 100"></div>

This <div> would only be visible when the customerValue is over 100.

Templates

Knockout has support for templates, so that you can easily separate your UI from your behavior, or incrementally load UI elements into a large application on demand. We can update our previous example to make each row its own template by simply pulling the HTML out into a template and specifying the template by name in the data-bind call on <tbody>.



```
    <tbody data-bind="template: { name: 'rowTemplate', foreach: gameResults }">
    </tbody>
</table>
<script type="text/html" id="rowTemplate">
    <tr>
        <td><input data-bind="value:opponent" /></td>
        <td><select data-bind="options: $root.resultChoices, value:result, optionsText: $data"></select></td>
    </tr>
</script>
```

Knockout also supports other templating engines, such as the jQuery.tmpl library and Underscore.js's templating engine.

Components

Components allow you to organize and reuse UI code, usually along with the ViewModel data on which the UI code depends. To create a component, you simply need to specify its template and its viewModel, and give it a name. This is done by calling ko.components.register(). In addition to defining the templates and viewmodel inline, they can be loaded from external files using a library like **require.js**, resulting in very clean and efficient code.

Communicating with APIs

Knockout can work with any data in JSON format. A common way to retrieve and save data using Knockout is with jQuery, which supports the $.getJSON() function to retrieve data, and the $.post() method to send data from the browser to an API endpoint. Of course, if you prefer a different way to send and receive JSON data, Knockout will work with it as well.

Summary

Knockout provides a simple, elegant way to bind UI elements to the current state of the client application, defined in a ViewModel. Knockout's binding syntax uses the data-bind attribute, applied to HTML elements that are to be processed. Knockout is able to efficiently render and update large data sets by tracking UI elements and only processing changes to affected elements. Large applications can break up UI logic using templates and components, which can be loaded on demand from external files. Currently version 3, Knockout is a stable JavaScript library that can improve web applications that require rich client interactivity.

Backbone.js. MVC Library

Backbone.js is a light-weight JavaScript library that helps structure web applications.

Backbone is not a framework that requires a particular approach. Instead, you can pick Backbone tools and combine them with other libraries. For example, you could use a Backbone View object to render data on a web page or you could swap in React.js to paint the page instead. You could use jQuery to bind Backbone views to DOM objects on the page or use the light-weight, mobile-friendly, Zepto library in its place.

Backbone has only one hard dependency, Underscore.js. The underscore.js library "…provides a whole mess of useful functional programming helpers without extending any built-in objects". Underscore is handy on its own, particularly for manipulating Backbone collections.

Getting Started

While underscore.js is the only hard dependency, you will want to display Backbone views. This example will use jQuery, but you could use Ender, Zepto, or some other DOM library. Download the jQuery, Underscore and Backbone JavaScript files and copy them to your web application directory. You can also use NPM, Gulp, Grunt, and Bower client build tools in Visual Studio 2015 to automate the process and handle dependencies.

```
<!DOCTYPE html>

<html>
<head>
    <meta charset="utf-8" />
    <title>Backbone Example</title>
</head>
<body>

    <script src="lib/jquery/js/jquery.js"></script>
    <script src="lib/underscore/underscore.js"></script>
    <script src="lib/backbone/backbone.js"></script>

</body>
</html>
```

Backbone Objects

Backbone includes classes for models, collections, views and routes.

- **Model**: Stores data attributes, supports validation, change events and communication with a server.
- **Collection**: Contains series of Model classes that can be manipulated and rendered as a group.
- **View**: Renders Model and Collection instances.
- **Route**: Reacts to URL changes and calls functions depending on the URL pattern.

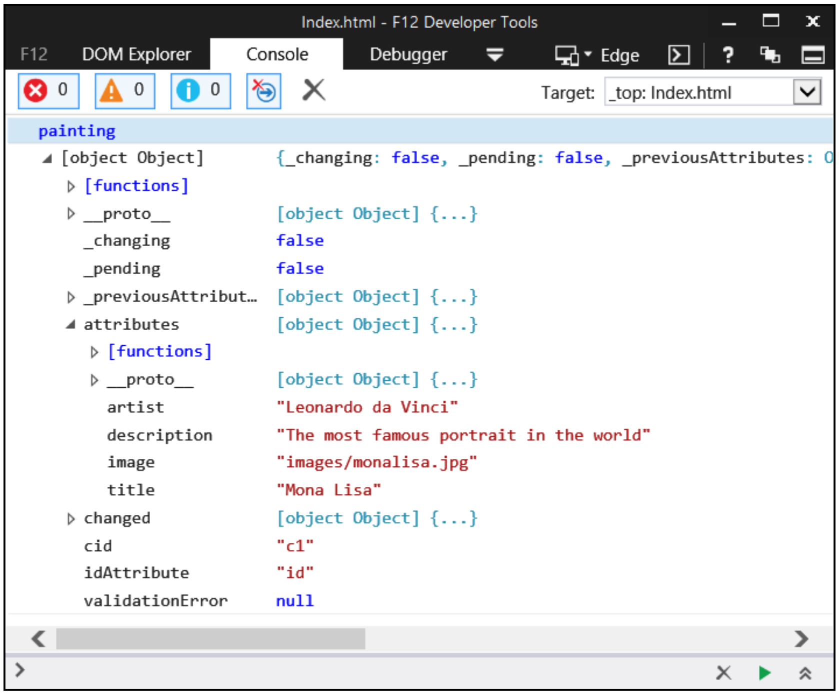
Models

Models are the foundation of a web application. While you could use JSON literals as models, you would miss out on the ability to inherit from other models, respond to events, validate attributes, and communicate with a server.

To create your own model, add a **<script>** tag in your HTML page and use the **Backbone.Model.extend()** function to create a model class. Then use the **new** keyword to create an instance of the class. If you pass a JSON object to the constructor, the model *attributes* will be set automatically.

```
<script>
    var Painting = Backbone.Model.extend();

    var painting = new Painting({
        title: "Mona Lisa",
        image: "images/monalisa.jpg",
        artist: "Leonardo da Vinci",
        description: "The most famous portrait in the world"
    });
</script>
```

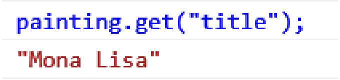
Use a browser console to dump the properties of the model. The Internet Explorer Developer Tools console in the screenshot below shows the attributes of the painting model object.



```
painting
◢ [object Object]          {_changing: false, _pending: false, _previousAttributes: O
   ▷ [functions]
   ▷ __proto__             [object Object] {...}
     _changing             false
     _pending              false
   ▷ _previousAttribut…    [object Object] {...}
   ◢ attributes            [object Object] {...}
      ▷ [functions]
      ▷ __proto__          [object Object] {...}
        artist             "Leonardo da Vinci"
        description        "The most famous portrait in the world"
        image              "images/monalisa.jpg"
        title              "Mona Lisa"
   ▷ changed               [object Object] {...}
     cid                   "c1"
     idAttribute           "id"
     validationError       null
```

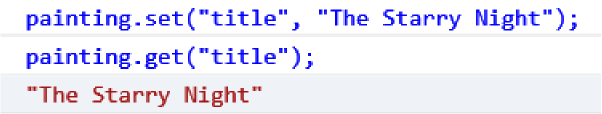
Reading and Writing Model Attributes

Use the **get()** method and pass the name of an attribute to retrieve the attribute value. The browser console output below shows the painting model title property value.



```
painting.get("title");
"Mona Lisa"
```

To modify a model attribute, call **set()** and pass both the attribute and a value:



```
painting.set("title", "The Starry Night");
painting.get("title");
"The Starry Night"
```

Avoid assigning Model attributes directly. Using **set()** and **get()** methods ensures that the Model class fires the **change** event and that model attributes are valid.

Validation

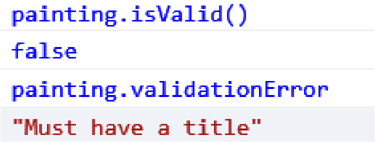
To verify the model's integrity, implement the **validate()** method. The method is undefined by default so you're free to use any criteria to confirm model state. Return a message only if the validation fails.

```
var Painting = Backbone.Model.extend({
    validate: function (attrs, options) {
        if (attrs.title === " ") {
            return "Must have a title";
        }
    }
    // ...
});
```

The Model's validate() method *does not* fire automatically when an attribute is set. Instead, you must pass the **validate: true** option to the set() method:

```
var painting = new Painting();
painting.set("title", " ", { validate: true });
```

You can also use the Model **isValid()** method to trigger validation and return false if errors are returned from validate(). Model **validationError** is handy for getting text for the last validation failure. The screenshot of the browser console below shows the output where the painting object title attribute is blank.



```
painting.isValid()
false
painting.validationError
"Must have a title"
```

Events

Backbone objects support a number of built-in events for models, collections and routes. For example, the **invalid** event fires when the **validate()** method returns an error message. Use the **on()** method to listen for an event name and a function that will handle the event. The code snippet below establishes an **invalid** event handler and logs the error message when model validation fails.

```
painting.on("invalid", function (model, error) {
    console.log(error);
});

painting.set("title", "", { validate: true });
```

You can also hook up events in during the Model's initialize() method.

```
var Painting = Backbone.Model.extend({
    initialize: function (options) {
        this.on("invalid", function (model, error) {
```

```
                    console.log(error);
                });
            },
            //...
        });
```

Constructors, initialize() and calling ancestor methods

When an instance of a class is created, the **constructor()** method is called before the model is accessible. At the end of the constructor, when the model is available, the **initialize()** method is called. If you decide to override the **constructor()**, be sure to call the ancestor's constructor. JavaScript doesn't have a constructor() or super() built-in. You must call the inherited constructor through the JavaScript prototype of the Backbone object.

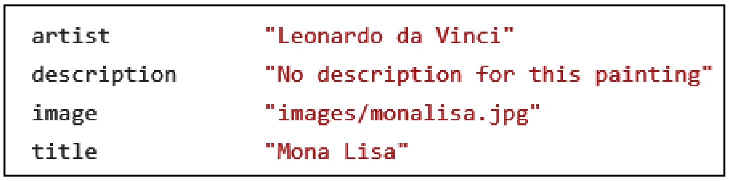
```
var Painting = Backbone.Model.extend({
    constructor: function (options) {
        console.log("Called before the model is built ");
        // call the ancestor constructor:
        Backbone.Model.prototype.constructor.apply(this, options);
    },
    initialize: function (options) {
        console.log("Called from constructor after model is available ");
    },
    //...
});
```

Default Attributes

You can avoid some issues with invalid data just by setting attribute values up front. Set initial values and help document attributes by overriding Model **defaults**. Any values not set during initialization are filled in by their defaults.

```
        var Painting = Backbone.Model.extend({
            defaults: {
                title: "",
                image: "images/default.jpg",
                artist: "Unknown",
                description: "No description for this painting"
            },
        });

        var painting = new Painting({
            title: "Mona Lisa",
            image: "images/monalisa.jpg",
            artist: "Leonardo da Vinci"
        });
```

Here is the output for the example in the browser console:



```
artist            "Leonardo da Vinci"
description       "No description for this painting"
image             "images/monalisa.jpg"
title             "Mona Lisa"
```

Here is the code listing for the painting model example we will use going forward.

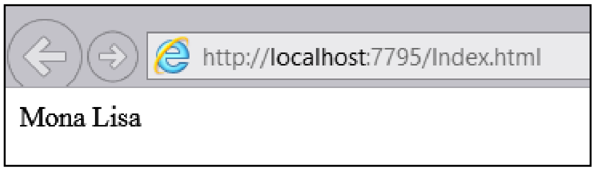
```
var Painting = Backbone.Model.extend({
    initialize: function (options) {
        this.on("invalid", function (model, error) {
            console.log(error);
        });
    },
    defaults: {
        title: "",
        image: "images/default.jpg",
        artist: "Unknown",
        description: "No description for this painting"
    },
    validate: function (attrs, options) {
        if (this.get("title") === "") {
            return "Must have a title";
        }
    }
});
```

Views

Backbone views paint model content to the browser page. Override the **View render()** function to assign the **View.el** or **View.$el** attributes with new HTML. The el attribute represents a DOM element for the view while $el is a jQuery selector for the view element. The example below extends the **Backbone.View** object and overrides the render() function to add the custom title attribute of the model. You should return **this** to allow chaining.

```
// Extend view to customize the render function
var PaintingView = Backbone.View.extend({
    render: function () {
        this.$el.append(this.model.get("title"));
        return this;
    }
});

// create an instance of the view
var paintingView = new PaintingView({ model: painting });

// inject the view's HTML into the DOM
$('body').html(paintingsView.render().el);
```

The render output shows as **<div>Mona Lisa</div>** in the browser.



Mona Lisa

By default, the View el represents a <div> tag, but you can override **tagName**, **id** and **className** to customize the element type, id attribute and CSS class, respectively.

```
var PaintingView = Backbone.View.extend({
    tagName: "h1",
    id: "painting",
    className: "painting-frame",
    render: function () {
        this.$el.append(this.model.get("title"));
        return this;
    }
});
```

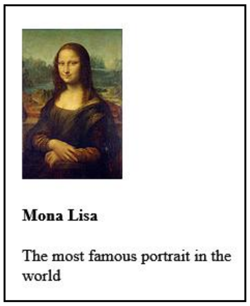
The markup is rendered as this HTML fragment:

<h1 class="painting-frame" id="painting">Mona Lisa</h1>

Using JavaScript and jQuery (or a jQuery replacement), you can render arbitrarily complex elements. The example below represents elements for the model's title, description and image, and uses jQuery to style elements inline, but you can also assign CSS classes for your custom styles or add Bootstrap to help set the page's look-and-feel.

```
var PaintingView = Backbone.View.extend({
    render: function () {
        var title = $('<h4 />').append(this.model.get("title"));
        var description = $('<p />').append(this.model.get("description"));
        var image = $('<img />',
                {
                    src: this.model.attributes.image
                });
        this.$el.css("width", "220px");
        this.$el.append(image).append(title).append(description);
        return this;
    }
});
```

Here is the view rendered in the browser:



Mona Lisa

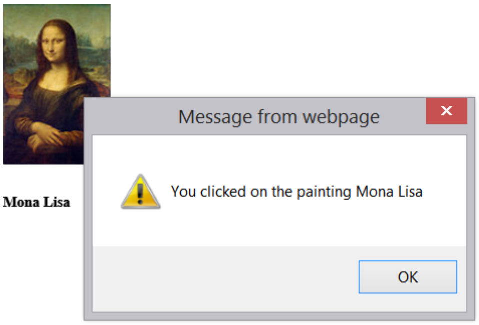
The most famous portrait in the world

View Events

The View class's **events** attribute takes a list of *delegated* events. Each attribute of the object assigned to **events** has an event name and the name of a callback function. What is the advantage of using **events** over the **on** function? The delegated events are bound to the view so that **this** still refers to the View object when the callback is invoked. The View example below responds to the DOM **click** event and fires the **clickHandler** callback.

```
var PaintingView = Backbone.View.extend({
    events: {
        'click': 'clickHandler'
    },
    clickHandler: function () {
        alert('You clicked on the painting ' +
            this.model.get("title"))
    }
    // ...
});
```

The example running in the browser displays the View's model data.



You clicked on the painting Mona Lisa

Collections

A Backbone **Collection** holds a list of Model objects that can be manipulated and rendered as a whole. The example below specifies the type of model it expects, although the **model** attribute is not required.

```
var Paintings = Backbone.Collection.extend({
    model: Painting,
});
```

To populate a collection, you can pass an array of model instances to the constructor or use the **add()** method. The example below creates an instance of the Paintings collection, then uses the add() method to fill the list.

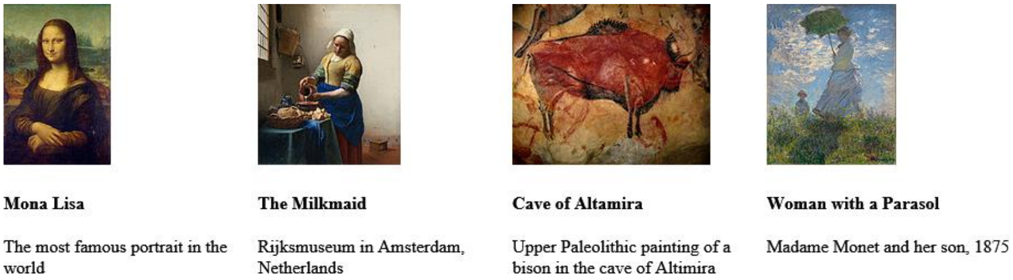
```
var paintings = new Paintings();

paintings.add([
    new Painting({
        title: "Mona Lisa",
        image: "images/monalisa.jpg",
        artist: "Leonardo da Vinci",
        description: "The most famous portrait in the world"
    }),

    new Painting({
        title: "The Milkmaid",
        image: "images/themilkmaid.jpg",
        artist: "Johannes Vermeer",
        description: "Rijksmuseum in Amsterdam, Netherlands"
    }),

    new Painting({
        title: "Cave of Altamira",
        image: "images/cave.jpg",
        artist: "Unknown",
        description: "Upper Paleolithic painting of a bison in the cave of Altimira"
    }),

    new Painting({
        title: "Woman with a Parasol",
        image: "images/womanwithparasol.jpg",
        artist: "Claude Monet",
        description: "Madame Monet and her son, 1875"
    })
```

```
        ]);
```

The Underscore library is required for Backbone and useful on its own to manipulate collections. For example, the **each()** method operates on collection model instances. The example below creates a new view for each model in the collection.

```
        paintings.each(function (painting) {
            var view = new PaintingView({ model: painting });
            var element = view.render().$el
                .css("float", "left")
                .css("margin-right", "10px");

            $('body').append(element);
        });
```

The example outputs a View for each Painting model instance:



Mona Lisa

The most famous portrait in the world

The Milkmaid

Rijksmuseum in Amsterdam, Netherlands

Cave of Altamira

Upper Paleolithic painting of a bison in the cave of Altimira

Woman with a Parasol

Madame Monet and her son, 1875

Views can represent Collections as well as Models, so you can wrap the logic inside the render() function. By default, **this** will refer to the global object (usually the browser window). Pass **this** as a second parameter to the **each()** function so that **this** will represent the View.

```
        paintings.each(function (painting) {
            var view = new PaintingView({ model: painting });
            var element = view.render().$el
                .css("float", "left")
                .css("margin-right", "10px");

            $('body').append(element);
        });
```

To delete a painting from the list, call the **remove()** function and pass an instance of a model. The example below uses the **findWhere()** function to locate the painting titled “Cave of Altamira” and deletes it from the collection.

```
        var cavePainting = paintings.findWhere({ "title": "Cave of Altamira" });
        paintings.remove(cavePainting);
```

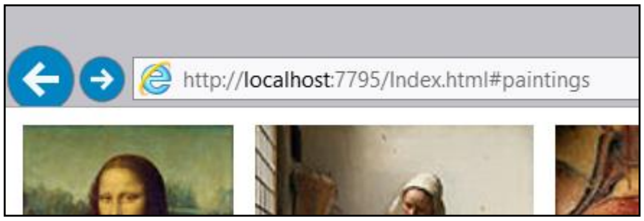
Routers

Backbone provides a simple client-side Router class that trigger actions when the browser URL changes. The **routes** attribute contains route names and the functions that should run for each route. After creating an instance of a router, call the **Backone.history start()** function.

The example below extends Backbone.Router and responds to a “paintings” route. When a fragment of the browser URL contains “paintings”, a function renders the PaintingsView .

```
var GalleryRouter = Backbone.Router.extend({
    routes: {
        'paintings': function () {
            var view = new PaintingsView({ collection: paintings });
            $('#app').html(view.render().el);
        }
    }
});

var router = new GalleryRouter();
Backbone.history.start();
```

To navigate using the router, add a hash symbol and the route identifier to the URL. The example below navigates to the ‘paintings’ route, instantiates the PaintingsView and renders the view to the page.



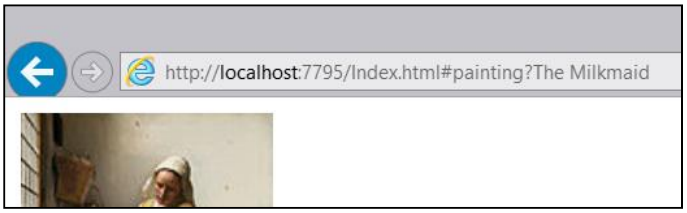
http://localhost:7795/Index.html#paintings

Note: Backbone Router class can also accommodate the HTML5 History API. For supported browsers, make the call to start history setting the pushState option to true. Backbone.history.start({pushState: true}); See the Backbone History documentation for more details.

You can further extend the router to include a detail page that will take a query string. The handler function example below uses the query string to filter the collection and present a view for a single model.

```
var GalleryRouter = Backbone.Router.extend({
    routes: {
        'paintings': function () {
            var view = new PaintingsView({ collection: paintings });
            $('#app').html(view.render().el);
        },

        'painting': function (query) {
            var title = decodeURI(query);
            var painting = paintings.findWhere({ "title": title })
            var view = new PaintingView({ model: painting });
            $('#app').html(view.render().el);
        }
    }
});
```

The route for a single painting is invoked using something like the path shown in the screen shot below.



Use the Backbone **history** class **navigate()** function to update the browser URL. The first parameter to **navigate()** is the target for the URL. The **trigger** option calls the **route()** function. The example below adds a click handler to each painting view. When the collection view of the paintings is showing and the user clicks a single painting, the click event handler fires and navigates to the view for that painting.

```
var PaintingView = Backbone.View.extend({
    events: {
        'click': 'clickHandler'
    },
    clickHandler: function () {
        var target = "painting?" + this.model.attributes.title;
        Backbone.history.navigate(target, { trigger: true })
    }
    // ...
});
```

To define a default route, add a route “*path” at the end of your list of routes.

```
var GalleryRouter = Backbone.Router.extend({
    routes: {
        // ...

        '*path': function () {
            var view = new PaintingsView({ collection: paintings });
            $('#app').html(view.render().el);
        }
    }
});
```

Summary

Backbone is a library that helps structure applications with models, views, collections and routers. Backbone will not dictate a particular approach but instead allows you to pick-and-choose functionality from the library. Backbone requires only underscore.js and can use any library to bind views to the DOM.

Angular.js. SPA Framework.

AngularJS is a modern JavaScript framework from Google to work with Single Page Applications (SPA). AngularJS is open sourced under MIT license and the development progress of AngularJS can be followed on its GitHub repository.

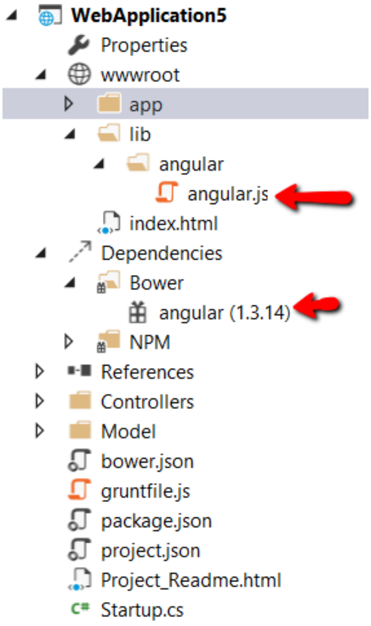
AngularJS is not a DOM manipulation library like jQuery but it uses a subset of jQuery called jQLite. AngularJS is primarily based on declarative HTML attributes that you can add to your HTML tags. You can try AngularJS in your browser using the Code School website.

V1.3.x is the current stable version and Angular team is working towards a big rewrite of AngularJS in its V2.0 which is currently still in development. This article focuses on Angular 1.X with a note of where Angular is header with 2.0.

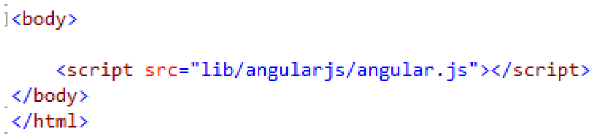
Getting Started

Installation

There are several ways to add AngularJS to your application. If you're starting a new web application in Visual Studio 2015 and ASP.NET 5, you can add AngularJS using built-in NPM and Bower support.



Add the <script> tag to your main html file



```
<body>

    <script src="lib/angularjs/angular.js"></script>
</body>
</html>
```

Key components

Below are the few major components of AngularJS, we will cover these topics in more detail in this article.

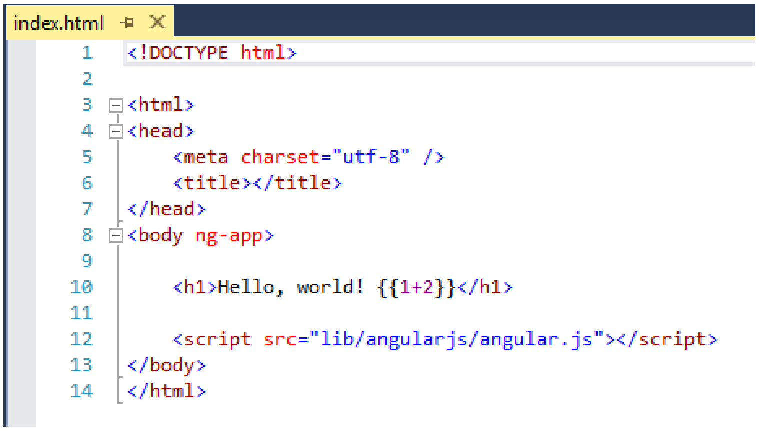
- Directives
- Data Binding
- Templates and Expressions
- Repeaters
- $scope
- Modules
- Controllers
- Services / Factories / Providers
- Routing
- Event Handlers

Directives

AngularJS uses directives to extend HTML with custom attributes and elements. AngularJS directives are defined via **data-ng-*** or **ng-*** prefixes. There are two types of AngularJS directives:

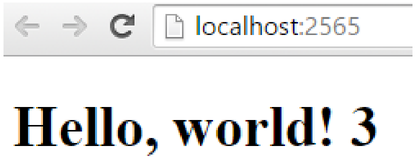
- Primitive Directives: These are pre-defined by the Angular team and are part of the AngularJS framework.
- Custom Directives: These are custom directives that you can define.

One of the primitive directives used in all AngularJS applications is the **ng-app** directive used to bootstrap the AngularJS application. This directive can be added to the whole document body or to specific pieces of the body. Let's see an example in action:



```html
<!DOCTYPE html>

<html>
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body ng-app>

    <h1>Hello, world! {{1+2}}</h1>

    <script src="lib/angularjs/angular.js"></script>
</body>
</html>
```

We decorated the body tag with **ng-app** directive to indicate this page is an AngularJS application. The **{{1+3}}** is an Angular data binding expression that will learn more about in this article. Here is the result if you run this application:



Hello, world! 3

Other primitive directives in AngularJS include:

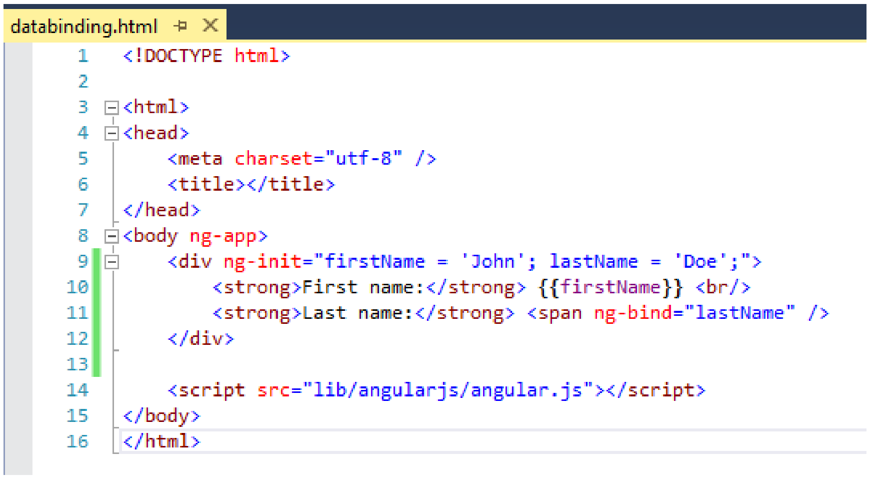
- **ng-controller**: Determines which JavaScript controller is bound to which view.
- **ng-model**: Determines what model the value of an HTML field properties is bound to.
- **ng-init**: Used to initialize application data in the form of an expression for the current scope.
- **ng-if**: If clause in your AngularJS applications, generally used with an expression.
- **ng-repeat**: Repeats given HTML over a set of data.
- **ng-show**: Shows or hides the given HTML element based on the expression provided.

For a full list of all primitive directives supported in AngularJS please read directive documentation section on the AngularJS documentation website.

Data Binding

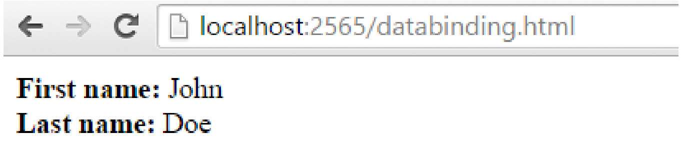
AngularJS provides data binding support out-of-the-box using either the **ng-bind** directive or a data binding expression syntax such as {{expression}}. AngularJS supports two-way data binding where data from a model is kept in synchronization with a view template at all time. Any changes to the view are automatically reflected in the model and any changes in the model are reflected in the view.

Here is an example with AngularJS data binding in action:



```
<!DOCTYPE html>

<html>
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body ng-app>
    <div ng-init="firstName = 'John'; lastName = 'Doe';">
        <strong>First name:</strong> {{firstName}} <br/>
        <strong>Last name:</strong> <span ng-bind="lastName" />
    </div>

    <script src="lib/angularjs/angular.js"></script>
</body>
</html>
```

When you run this file, you will see First Name and Last Name data bound to the view:

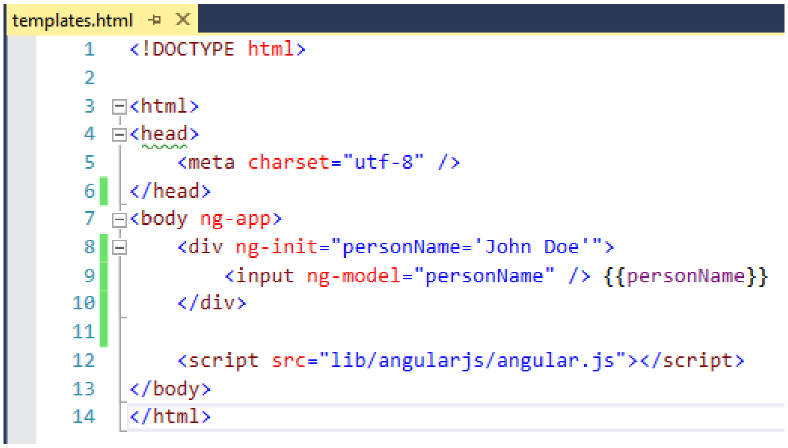


First name: John
Last name: Doe

Templates

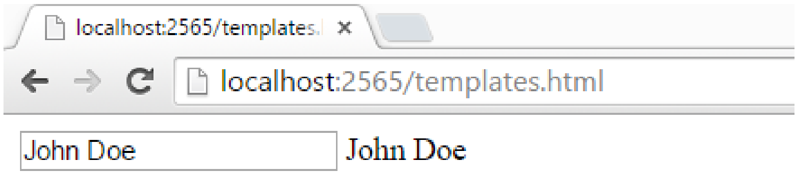
Templates in AngularJS are just plain HTML pages decorated with AngularJS directives and artifacts. A template in AngularJS is a mixture of directives, expressions, filters, and controls that combine with HTML to form the view.

Below is an example of a simple AngularJS template:



```
<!DOCTYPE html>

<html>
<head>
    <meta charset="utf-8" />
</head>
<body ng-app>
    <div ng-init="personName='John Doe'">
        <input ng-model="personName" /> {{personName}}
    </div>

    <script src="lib/angularjs/angular.js"></script>
</body>
</html>
```

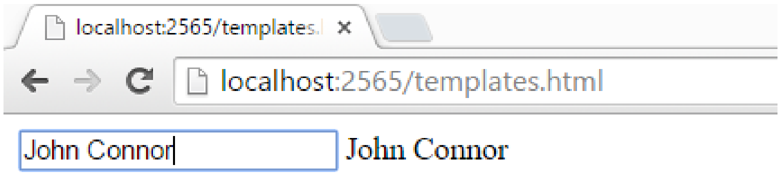
The template has AngularJS directives like **ng-app**, **ng-init**, **ng-model** and data binding expression syntax to bind the **personName** property. Running in the browser, the view looks like the screenshot below:



localhost:2565/templates.html

John Doe John Doe

If you change the name by typing in the input field, you will see the text next to the input field dynamically update, showing Angular two-way data binding in action.



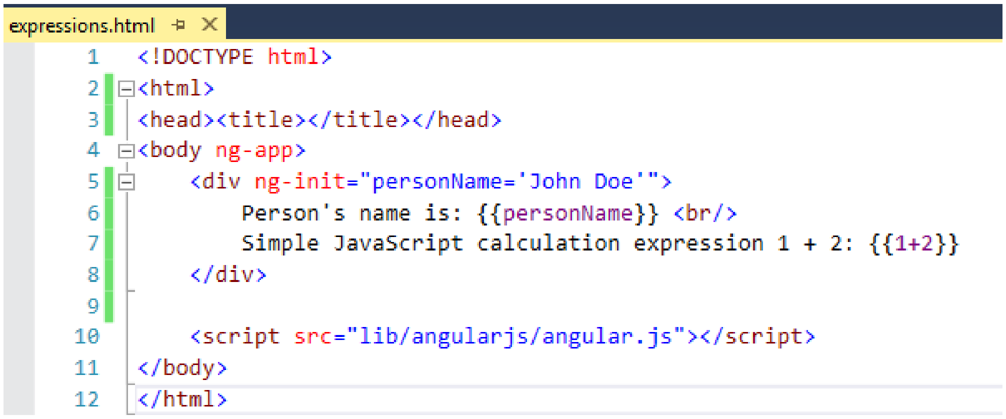
localhost:2565/templates.html

John Connor John Connor

Expressions

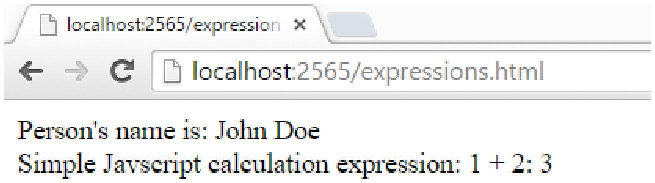
Expressions in AngularJS are JavaScript -like code snippets that are written inside the **{{ expression }}** syntax. The data from these expressions are bound to HTML the same way as ng-bind directives. The main difference between AngularJS expressions and regular JavaScript expressions is that AngularJS expressions are evaluated against the **$scope** object in AngularJS.

The AngularJS expressions in the sample below bind personName and a simple JavaScript calculated expression:



```
<!DOCTYPE html>
<html>
<head><title></title></head>
<body ng-app>
    <div ng-init="personName='John Doe'">
        Person's name is: {{personName}} <br/>
        Simple JavaScript calculation expression 1 + 2: {{1+2}}
    </div>

    <script src="lib/angularjs/angular.js"></script>
</body>
</html>
```

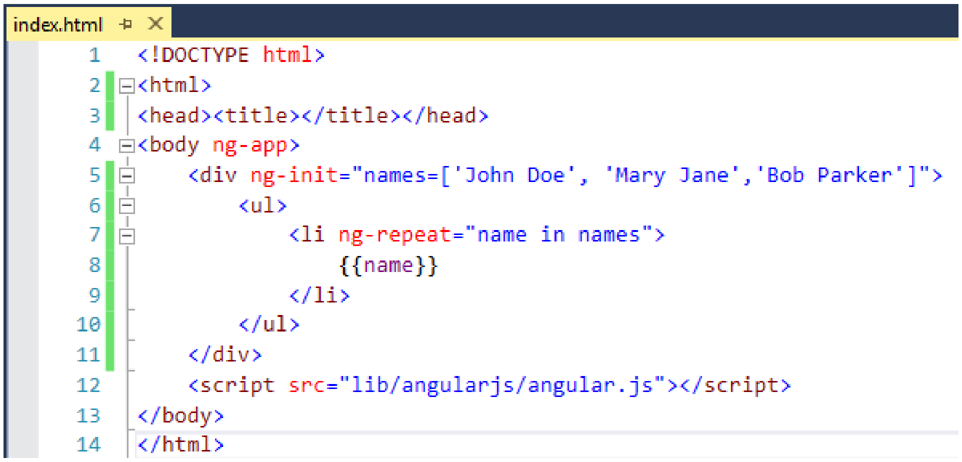
The example running in the browser displays the personName data and the results of the calculation:



```
Person's name is: John Doe
Simple Javscript calculation expression: 1 + 2: 3
```

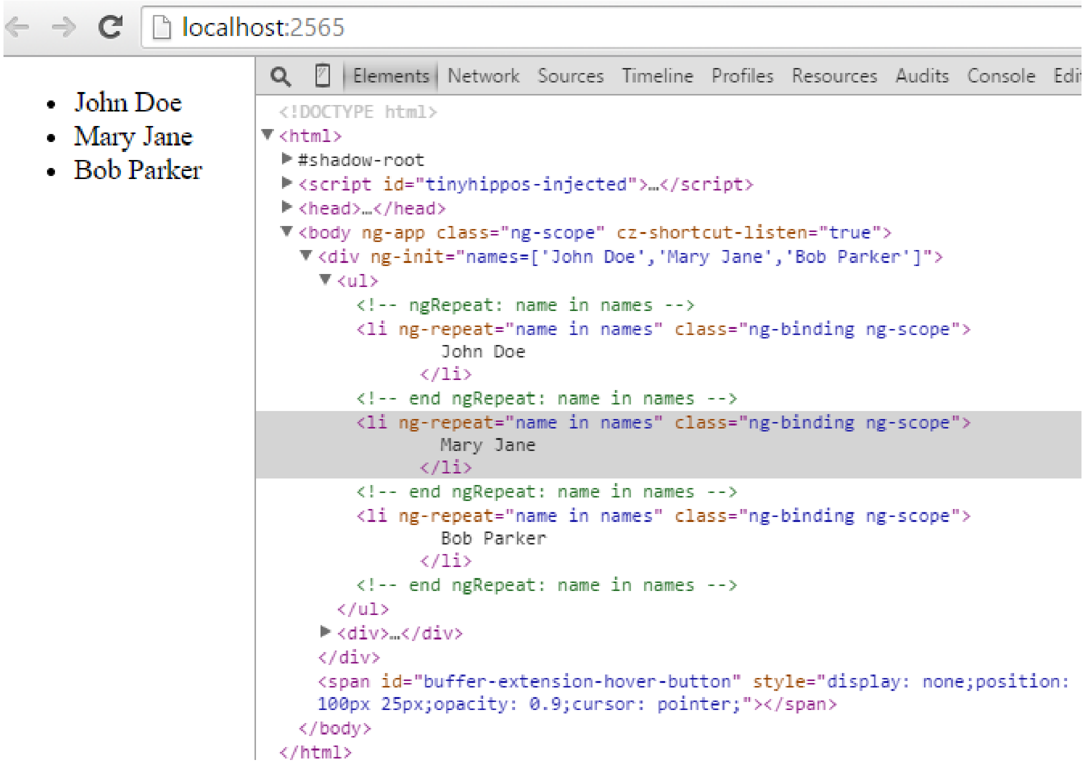
Repeaters

Repeating in AngularJS is done via a primitive directive called **ng-repeat**. The ng-repeat directive repeats a given HTML element in a view over the length of a repeating data array. Repeaters in AngularJS can repeat over an array of strings or objects. Here is a sample usage of repeating over an array of strings:



```
<!DOCTYPE html>
<html>
<head><title></title></head>
<body ng-app>
    <div ng-init="names=['John Doe', 'Mary Jane','Bob Parker']">
        <ul>
            <li ng-repeat="name in names">
                {{name}}
            </li>
        </ul>
    </div>
    <script src="lib/angularjs/angular.js"></script>
</body>
</html>
```

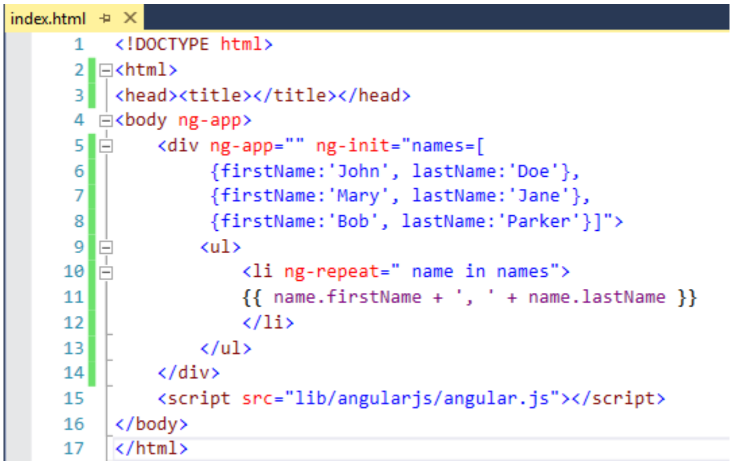
The repeat directive outputs a series of list items in an unordered list:



- John Doe
- Mary Jane
- Bob Parker

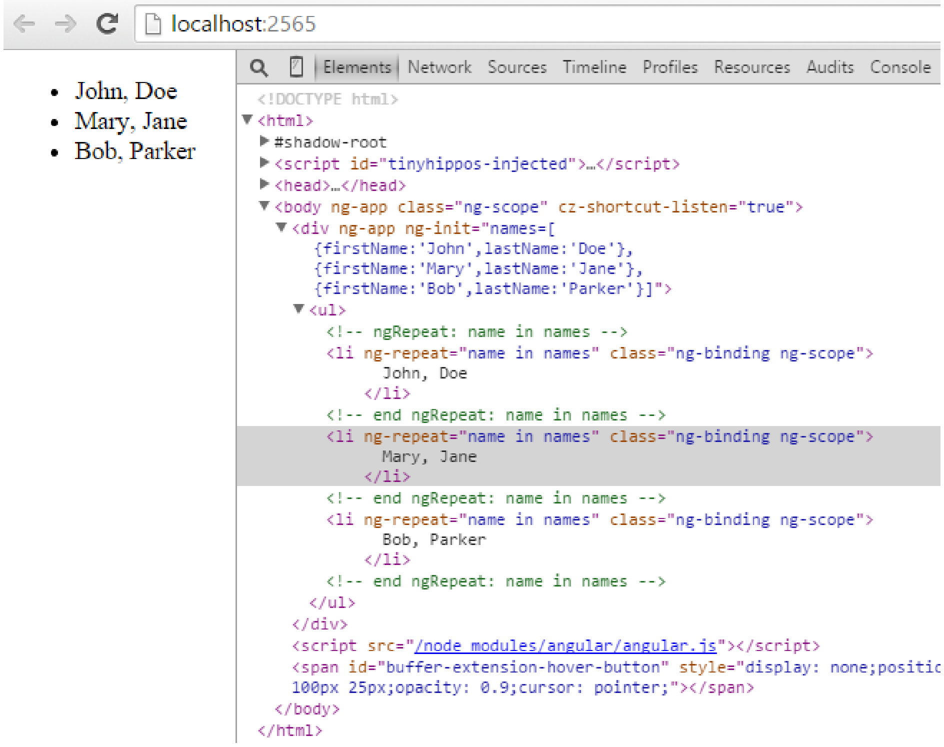
```
<!DOCTYPE html>
▼<html>
  ▶#shadow-root
  ▶<script id="tinyhippos-injected">…</script>
  ▶<head>…</head>
  ▼<body ng-app class="ng-scope" cz-shortcut-listen="true">
    ▼<div ng-init="names=['John Doe','Mary Jane','Bob Parker']">
      ▼<ul>
          <!-- ngRepeat: name in names -->
          <li ng-repeat="name in names" class="ng-binding ng-scope">
                  John Doe
                </li>
          <!-- end ngRepeat: name in names -->
          <li ng-repeat="name in names" class="ng-binding ng-scope">
                  Mary Jane
                </li>
          <!-- end ngRepeat: name in names -->
          <li ng-repeat="name in names" class="ng-binding ng-scope">
                  Bob Parker
                </li>
          <!-- end ngRepeat: name in names -->
        </ul>
      ▶<div>…</div>
      </div>
      <span id="buffer-extension-hover-button" style="display: none;position:
      100px 25px;opacity: 0.9;cursor: pointer;"></span>
    </body>
  </html>
```

Here is an example that repeats over an array of objects. The ng-init directive establishes a **names** array, where each element is an object contains first and last names. The ng-repeat assignment, **name in names**, outputs a list item for every array element.



```
<!DOCTYPE html>
<html>
<head><title></title></head>
<body ng-app>
    <div ng-app="" ng-init="names=[
            {firstName:'John', lastName:'Doe'},
            {firstName:'Mary', lastName:'Jane'},
            {firstName:'Bob', lastName:'Parker'}]">
        <ul>
            <li ng-repeat=" name in names">
            {{ name.firstName + ', ' + name.lastName }}
            </li>
        </ul>
    </div>
    <script src="lib/angularjs/angular.js"></script>
</body>
</html>
```

The HTML output is, as you would expect, a series of list items in an unordered list:



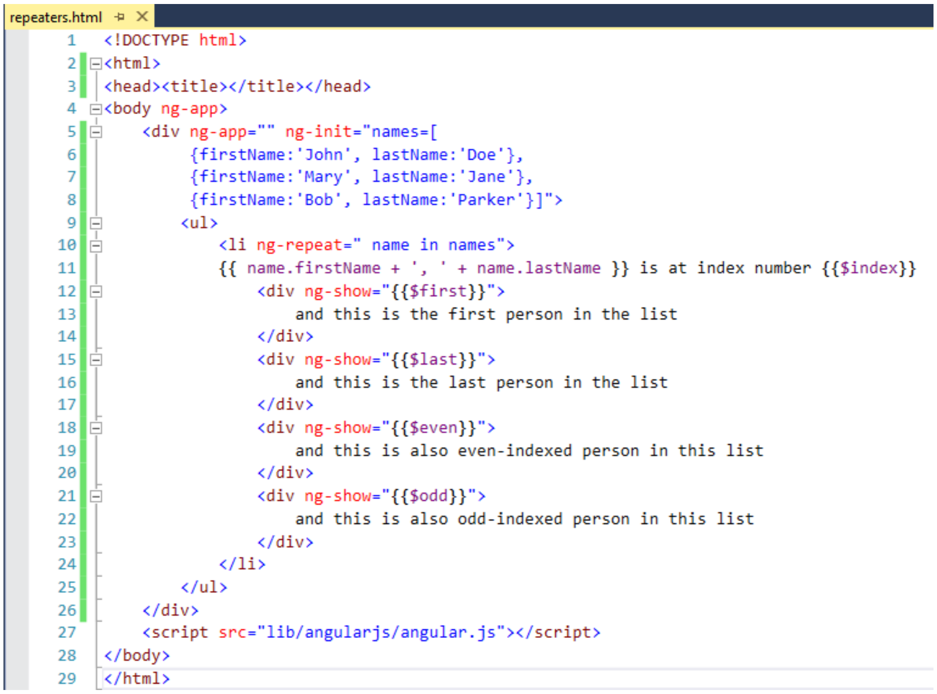
- John, Doe
- Mary, Jane
- Bob, Parker

```
<!DOCTYPE html>
▼<html>
  ▶#shadow-root
  ▶<script id="tinyhippos-injected">…</script>
  ▶<head>…</head>
  ▼<body ng-app class="ng-scope" cz-shortcut-listen="true">
    ▼<div ng-app ng-init="names=[
        {firstName:'John',lastName:'Doe'},
        {firstName:'Mary',lastName:'Jane'},
        {firstName:'Bob',lastName:'Parker'}]">
      ▼<ul>
          <!-- ngRepeat: name in names -->
          <li ng-repeat="name in names" class="ng-binding ng-scope">
              John, Doe
            </li>
          <!-- end ngRepeat: name in names -->
          <li ng-repeat="name in names" class="ng-binding ng-scope">
              Mary, Jane
            </li>
          <!-- end ngRepeat: name in names -->
          <li ng-repeat="name in names" class="ng-binding ng-scope">
              Bob, Parker
            </li>
          <!-- end ngRepeat: name in names -->
        </ul>
      </div>
      <script src="/node_modules/angular/angular.js"></script>
      <span id="buffer-extension-hover-button" style="display: none;positic
      100px 25px;opacity: 0.9;cursor: pointer;"></span>
    </body>
</html>
```

Use the **$index**, **$even**, **$odd**, **$first** and **$last** directives to determine where you are in the loop.

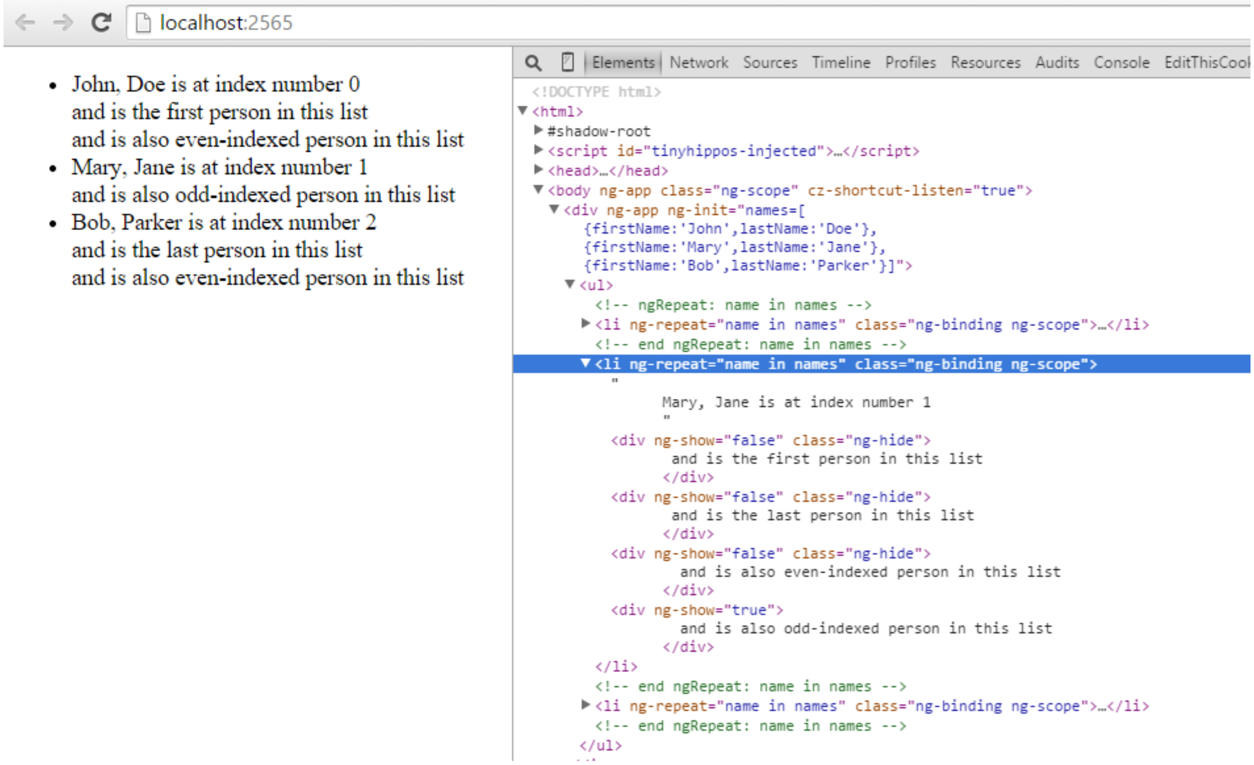
- Use $index in the ng-repeat loop to determine which index position your loop currently is on.
- Use $even in the ng-repeat loop to determine whether the current index in your loop is an even indexed row. Similarly, use $odd to determine if the current index is an odd indexed row.
- Use $first in the ng-repeat loop to determine whether the current index in your loop is the first row. Similarly, use $last to determine if the current index is the last row.

Below is a sample that shows $index, $even, $odd, $first, $odd in action:



```
<!DOCTYPE html>
<html>
<head><title></title></head>
<body ng-app>
    <div ng-app="" ng-init="names=[
         {firstName:'John', lastName:'Doe'},
         {firstName:'Mary', lastName:'Jane'},
         {firstName:'Bob', lastName:'Parker'}]">
        <ul>
            <li ng-repeat=" name in names">
            {{ name.firstName + ', ' + name.lastName }} is at index number {{$index}}
                <div ng-show="{{$first}}">
                    and this is the first person in the list
                </div>
                <div ng-show="{{$last}}">
                    and this is the last person in the list
                </div>
                <div ng-show="{{$even}}">
                    and this is also even-indexed person in this list
                </div>
                <div ng-show="{{$odd}}">
                    and this is also odd-indexed person in this list
                </div>
            </li>
        </ul>
    </div>
    <script src="lib/angularjs/angular.js"></script>
</body>
</html>
```

Here is the result running in the browser:



- John, Doe is at index number 0
  and is the first person in this list
  and is also even-indexed person in this list
- Mary, Jane is at index number 1
  and is also odd-indexed person in this list
- Bob, Parker is at index number 2
  and is the last person in this list
  and is also even-indexed person in this list

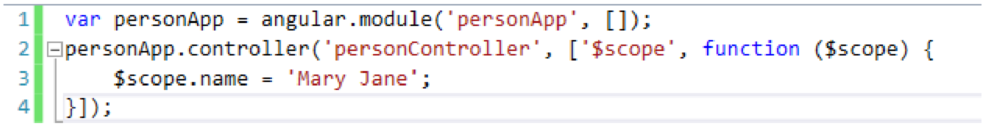
```
<!DOCTYPE html>
▼<html>
  ▶#shadow-root
  ▶<script id="tinyhippos-injected">…</script>
  ▶<head>…</head>
  ▼<body ng-app class="ng-scope" cz-shortcut-listen="true">
    ▼<div ng-app ng-init="names=[
        {firstName:'John',lastName:'Doe'},
        {firstName:'Mary',lastName:'Jane'},
        {firstName:'Bob',lastName:'Parker'}]">
      ▼<ul>
          <!-- ngRepeat: name in names -->
        ▶<li ng-repeat="name in names" class="ng-binding ng-scope">…</li>
          <!-- end ngRepeat: name in names -->
        ▼<li ng-repeat="name in names" class="ng-binding ng-scope">
            "
                  Mary, Jane is at index number 1
                  "
            <div ng-show="false" class="ng-hide">
                   and is the first person in this list
                  </div>
            <div ng-show="false" class="ng-hide">
                   and is the last person in this list
                  </div>
            <div ng-show="false" class="ng-hide">
                    and is also even-indexed person in this list
                  </div>
            <div ng-show="true">
                    and is also odd-indexed person in this list
                  </div>
          </li>
          <!-- end ngRepeat: name in names -->
        ▶<li ng-repeat="name in names" class="ng-binding ng-scope">…</li>
          <!-- end ngRepeat: name in names -->
        </ul>
```

$scope

$scope is a JavaScript object that acts as glue between the view (template) and the controller (explained below). A view template in AngularJS only knows about the values that are set on the $scope object from the controller.

Note: In the MVVM world, the $scope object in AngularJS is often defined as the ViewModel. The AngularJS team refers to the $scope object as the Data-Model.

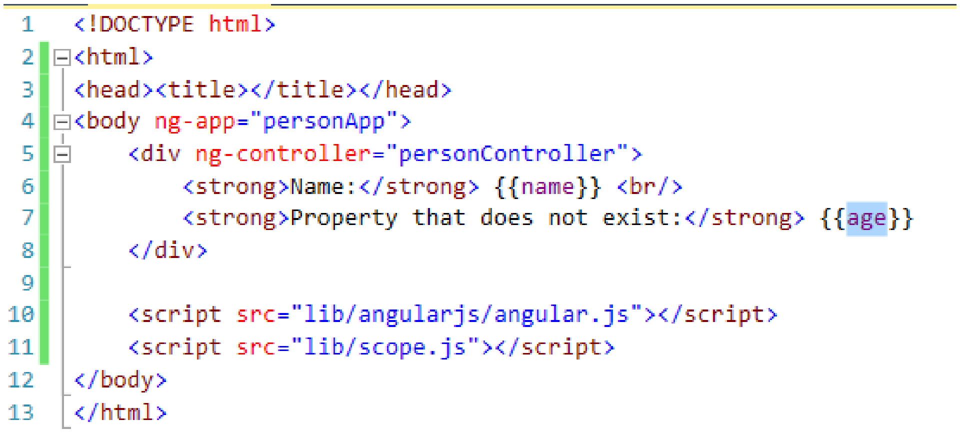
Below is a simple example showing how to set properties on $scope.



```
1  var personApp = angular.module('personApp', []);
2 personApp.controller('personController', ['$scope', function ($scope) {
3      $scope.name = 'Mary Jane';
4  }]);
```

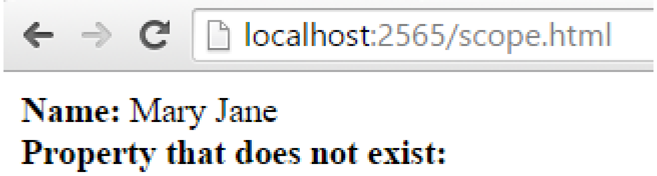
Observe the $scope parameter passed to the controller on line #3. This object is what the view knows about. In line #4, we are setting a property called "name" to "Mary Jane".

What happens when a particular property is not found by the view? The view defined below refers to **name** and **age** properties:



```
1   <!DOCTYPE html>
2   <html>
3   <head><title></title></head>
4   <body ng-app="personApp">
5       <div ng-controller="personController">
6           <strong>Name:</strong> {{name}} <br/>
7           <strong>Property that does not exist:</strong> {{age}}
8       </div>
9
10      <script src="lib/angularjs/angular.js"></script>
11      <script src="lib/scope.js"></script>
12  </body>
13  </html>
```

Notice in line #9 that we are asking Angular to show the "name" property using expression syntax. Line #10 refers to "age", a property that does not exist. The running example shows the name set to "Mary Jane" and nothing for age.



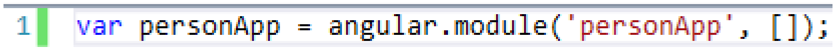
localhost:2565/scope.html

Name: Mary Jane
Property that does not exist:

Modules

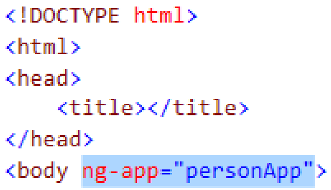
A **module** in AngularJS is a collection of controllers, services, directives etc. The **angular.module()** function call is used to create, register and retrieve modules in AngularJS. All modules, including those shipped by the AngularJS team and third party libraries should be registered using the angular.module() function.

Below is a snippet of code that shows how to create a new module in AngularJS. The first parameter is the name of the module. The second parameter defines dependencies on other modules. Later in this articles we will be showing how to pass these dependencies in angular.module() method call.



```
var personApp = angular.module('personApp', []);
```

Use the **ng-app** directive to represent an AngularJS module on the page. To use a module, assign the name of the module, “personApp” in this example, to the ng-app directive in our template.



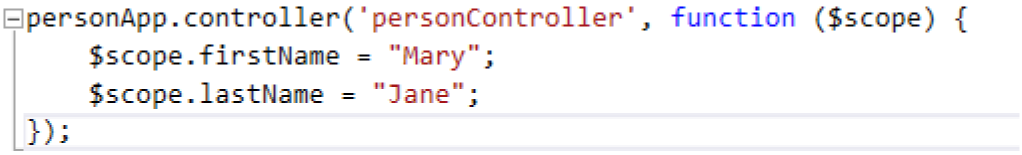
```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body ng-app="personApp">
```

Note: Before Angular V1.3 having a module was an optional (but still highly recommended) feature. Starting from Angular V1.3, The Angular team mandated the use of modules in order to work with controllers.

Controllers

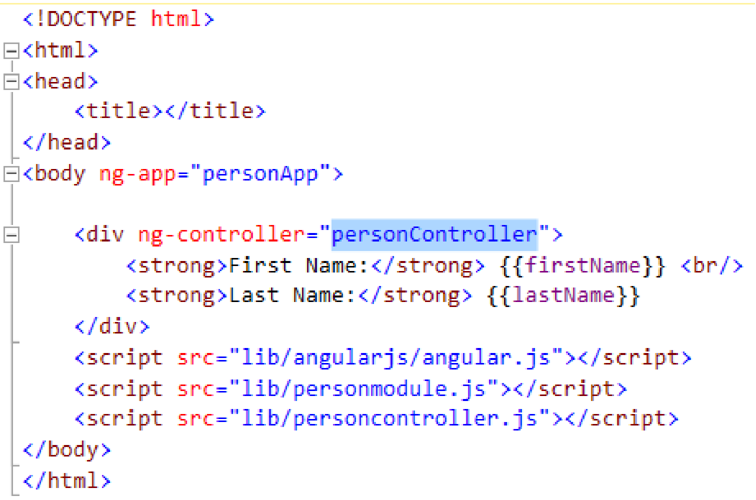
Controllers in AngularJS are the first point of entry for your code. The **<module name>.controller()** function call is used to create and register controllers in AngularJS. The **ng-controller** directive is used to represent an AngularJS controller on the HTML page. The role of the controller in Angular is to set state and behavior of the data model ($scope). Controllers should not be used to manipulate the DOM directly.

Below is a snippet of code that registers a new controller. The **personApp** variable in the snippet is an Angular module.



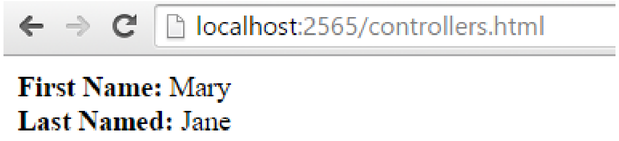
```
personApp.controller('personController', function ($scope) {
    $scope.firstName = "Mary";
    $scope.lastName = "Jane";
});
```

The view using the ng-controller directive assigns the controller name:



```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body ng-app="personApp">

    <div ng-controller="personController">
        <strong>First Name:</strong> {{firstName}} <br/>
        <strong>Last Name:</strong> {{lastName}}
    </div>
    <script src="lib/angularjs/angular.js"></script>
    <script src="lib/personmodule.js"></script>
    <script src="lib/personcontroller.js"></script>
</body>
</html>
```

The page shows “Mary” and “Jane” that correspond to the firstName and lastName properties assigned to the $scope object.

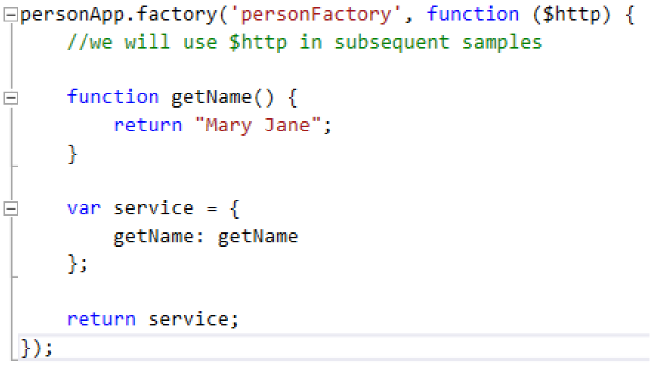


First Name: Mary
Last Named: Jane

Services

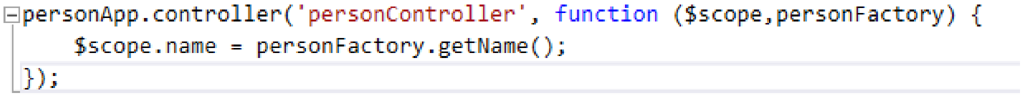
Services in AngularJS are commonly used shared code that are abstracted away into a file that can be used throughout the lifetime of an angular application. Services are lazily instantiated, meaning that there will not be an instance of a service unless a component that depends on the service gets used. Factories in AngularJS are created using the **myApp.factory()** function call, where **myApp** is the module.

Below is an example that shows how to use factories in AngularJS:



```
personApp.factory('personFactory', function ($http) {
    //we will use $http in subsequent samples

    function getName() {
        return "Mary Jane";
    }

    var service = {
        getName: getName
    };

    return service;
});
```

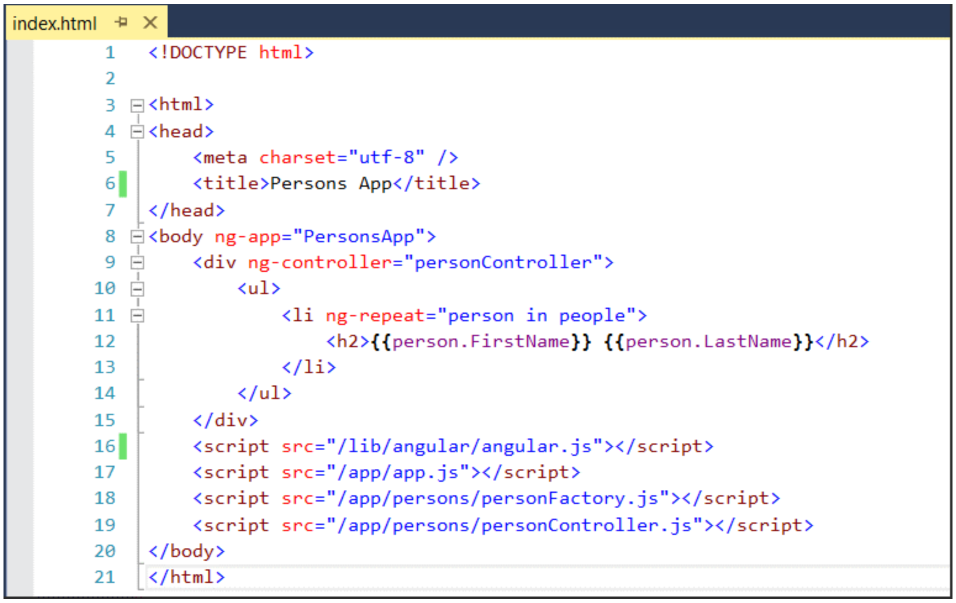
To call this factory from the controller, pass **personFactory** as a parameter to the controller() function:



```
personApp.controller('personController', function ($scope,personFactory) {
    $scope.name = personFactory.getName();
});
```

Using Services to talk to a REST endpoint

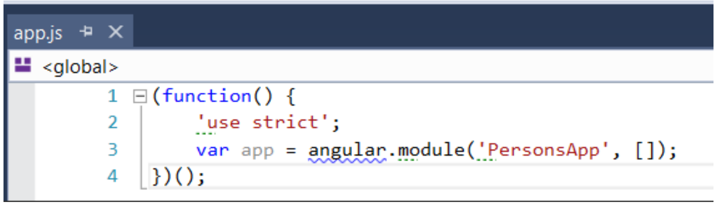
Below is an end-to-end example using services in AngularJS to interact with an ASP.NET MVC's REST endpoint. The example gets data from the service built using ASP.NET 5 and displays the data in a view template. Let's start with the view first:



```
1  <!DOCTYPE html>
2  
3  <html>
4  <head>
5      <meta charset="utf-8" />
6      <title>Persons App</title>
7  </head>
8  <body ng-app="PersonsApp">
9      <div ng-controller="personController">
10         <ul>
11             <li ng-repeat="person in people">
12                 <h2>{{person.FirstName}} {{person.LastName}}</h2>
13             </li>
14         </ul>
15     </div>
16     <script src="/lib/angular/angular.js"></script>
17     <script src="/app/app.js"></script>
18     <script src="/app/persons/personFactory.js"></script>
19     <script src="/app/persons/personController.js"></script>
20 </body>
21 </html>
```

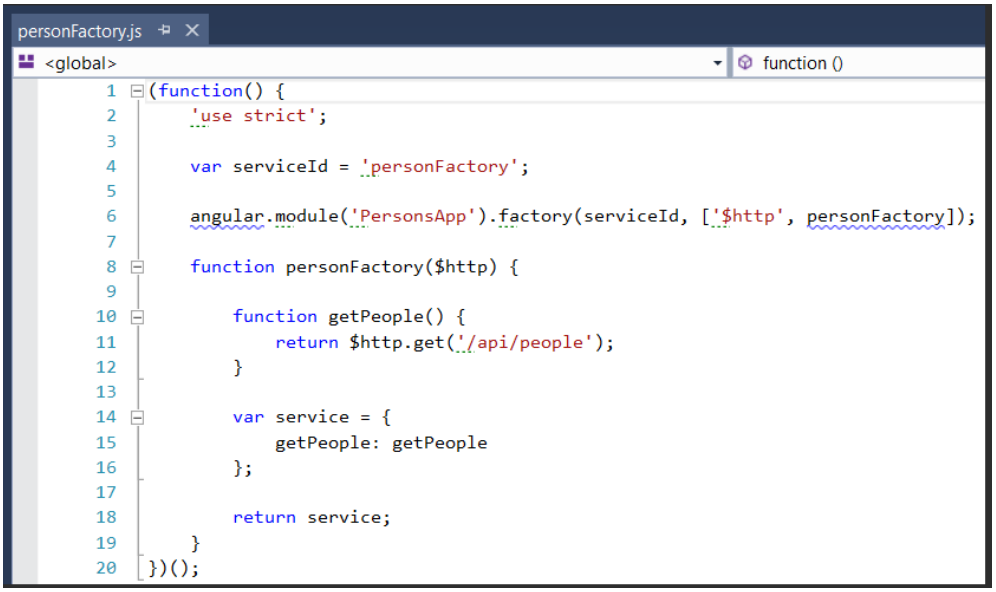
In this view, we have an Angular module called "PersonsApp" and a controller called "personController". We are using **ng-repeat** to iterate over the list of persons. Notice that lines #17 to #19 reference three new files: **app.js**, **personFactory.js** and **personController.js**.

The app.js file is used to register the PersonsApp module. The syntax is similar to previous examples. We are using the angular.module() function to create a new instance of the module that we will be working with.



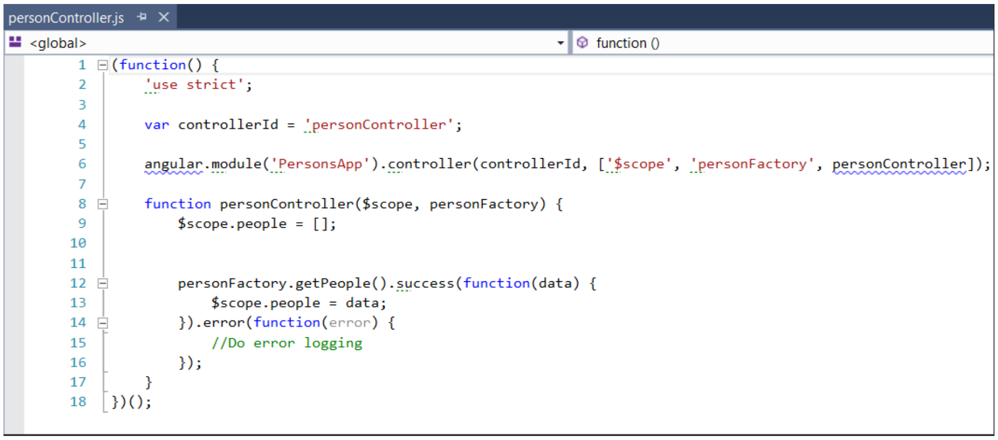
```
(function() {
    'use strict';
    var app = angular.module('PersonsApp', []);
})();
```

Let's take a look at personFactory.js. We are calling the module's factory() method to create a factory. Line #11 shows the built-in Angular **$http** service retrieving people information from a web service.



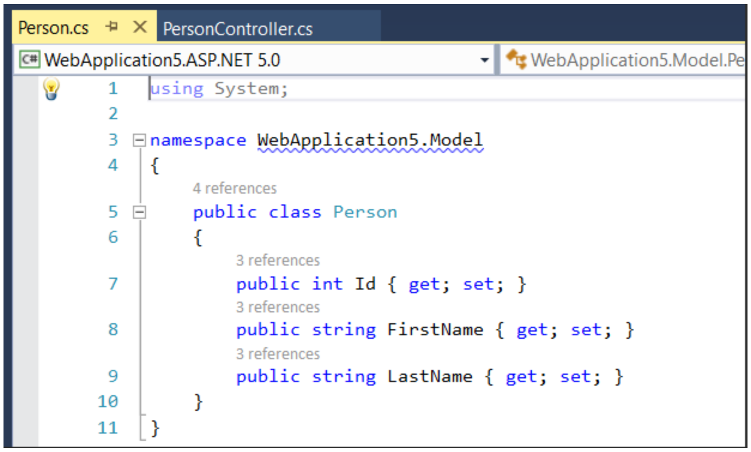
```javascript
(function() {
    'use strict';

    var serviceId = 'personFactory';

    angular.module('PersonsApp').factory(serviceId, ['$http', personFactory]);

    function personFactory($http) {

        function getPeople() {
            return $http.get('/api/people');
        }

        var service = {
            getPeople: getPeople
        };

        return service;
    }
})();
```

In personController.js, we are calling the module's controller() method to create the controller. The $scope **people** property is assigned data returned from the personFactory (line #13).



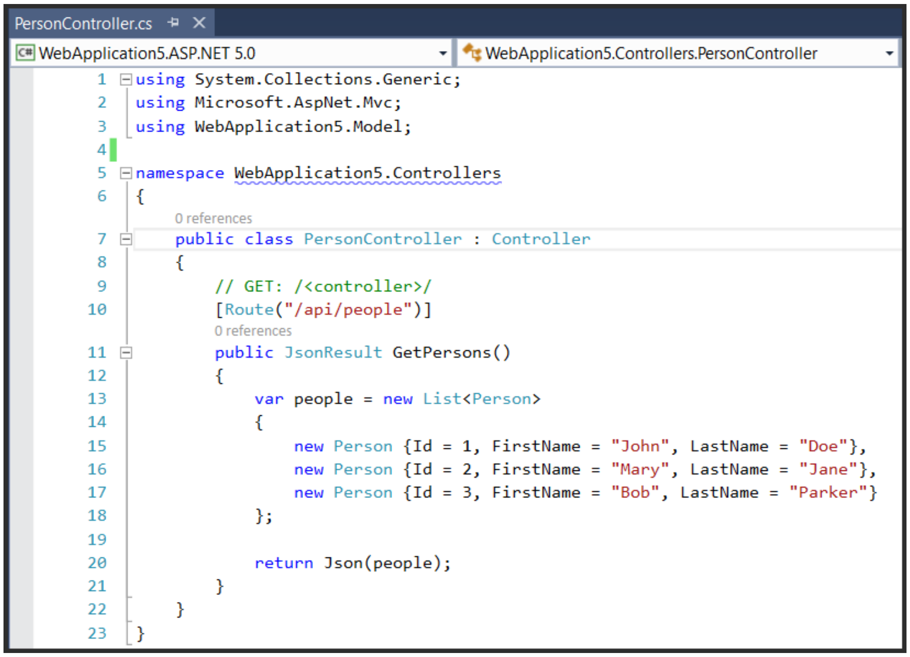
```javascript
 1  (function() {
 2      'use strict';
 3  
 4      var controllerId = 'personController';
 5  
 6      angular.module('PersonsApp').controller(controllerId, ['$scope', 'personFactory', personController]);
 7  
 8      function personController($scope, personFactory) {
 9          $scope.people = [];
10  
11  
12          personFactory.getPeople().success(function(data) {
13              $scope.people = data;
14          }).error(function(error) {
15              //Do error logging
16          });
17      }
18  })();
```

Let's take a quick look at the ASP.NET 5 web service and the model behind it. The Person model is a plain POCO (Plain Old CLR Object) with Id, FirstName and LastName properties.



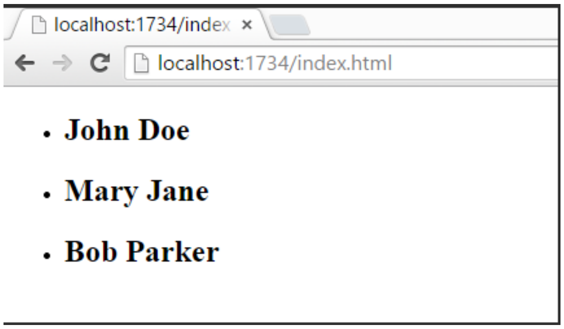
```csharp
using System;

namespace WebApplication5.Model
{
    4 references
    public class Person
    {
        3 references
        public int Id { get; set; }
        3 references
        public string FirstName { get; set; }
        3 references
        public string LastName { get; set; }
    }
}
```

The Person controller returns a JSON of list of Persons.


PersonController.cs
WebApplication5.ASP.NET 5.0
WebApplication5.Controllers.PersonController
0 references
0 references

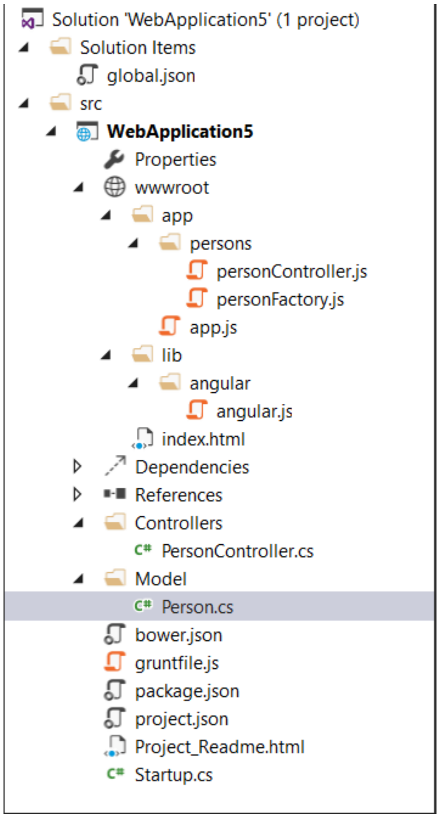
```
using System.Collections.Generic;
using Microsoft.AspNet.Mvc;
using WebApplication5.Model;

namespace WebApplication5.Controllers
{
    public class PersonController : Controller
    {
        // GET: /<controller>/
        [Route("/api/people")]
        public JsonResult GetPersons()
        {
            var people = new List<Person>
            {
                new Person {Id = 1, FirstName = "John", LastName = "Doe"},
                new Person {Id = 2, FirstName = "Mary", LastName = "Jane"},
                new Person {Id = 3, FirstName = "Bob", LastName = "Parker"}
            };

            return Json(people);
        }
    }
}
```

Let’s see the application in action:



- John Doe
- Mary Jane
- Bob Parker

Here is a snapshot of the application structure in Visual Studio:



```
Solution 'WebApplication5' (1 project)
  Solution Items
    global.json
  src
    WebApplication5
      Properties
      wwwroot
        app
          persons
            personController.js
            personFactory.js
          app.js
        lib
          angular
            angular.js
        index.html
      Dependencies
      References
      Controllers
        PersonController.cs
      Model
        Person.cs
      bower.json
      gruntfile.js
      package.json
      project.json
      Project_Readme.html
      Startup.cs
```

Note: For more on structuring AngularJS applications, see John Papa's Angular Style Guide.

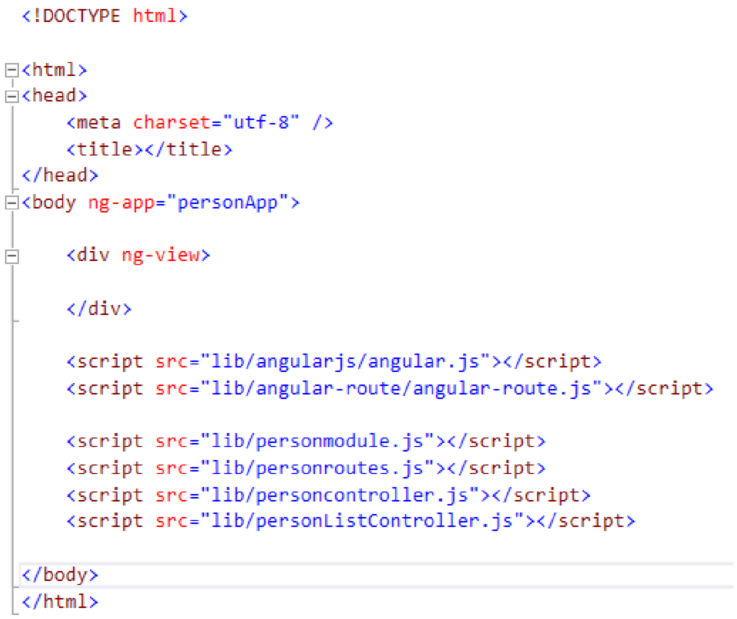
Note: To create AngularJS module, controller, factory, directive and view files easily, be sure to check out Sayed Hashimi's SideWaffle template pack for Visual Studio. Sayed Hashimi is a Senior Program Manager on the Visual Studio Web Team at Microsoft and SideWaffle templates are considered a gold standard. At the moment of this writing, SideWaffle is only available for Visual Studio 2012 and 2013.

Routing and Multiple Views

AngularJS has a built-in route provider to handle SPA (Single Page Application) based navigation. To work with routing in AngularJS you have to add the **angular-route** using NPM or Bower.

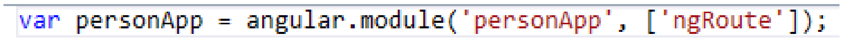
After you install the package, add the script reference your view.

Now let’s take the Person App we have been building and add navigation to it. Starting with the main view, add a reference to **angular-route** (see line #13). Also add a div marked with the **ng-view** directive (see line #8) as a placeholder to place views in.



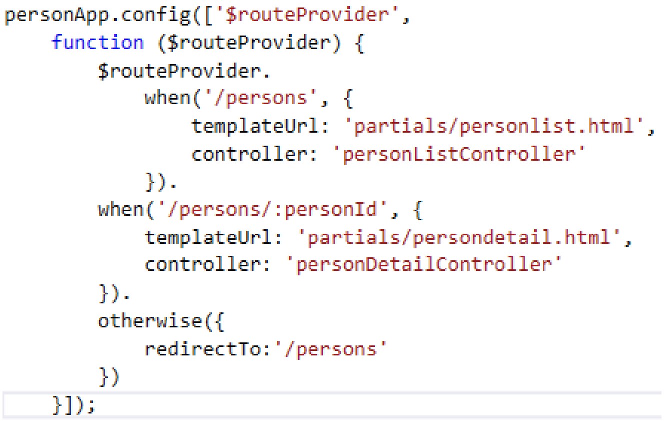
```
<!DOCTYPE html>

<html>
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body ng-app="personApp">

    <div ng-view>

    </div>

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

    <script src="lib/personmodule.js"></script>
    <script src="lib/personroutes.js"></script>
    <script src="lib/personcontroller.js"></script>
    <script src="lib/personListController.js"></script>

</body>
</html>
```

Let’s take a look at personModule.js file to see how we are instantiating the module with routing. We are passing **ngRoute** as a library into the module. This module handles routing in our application.



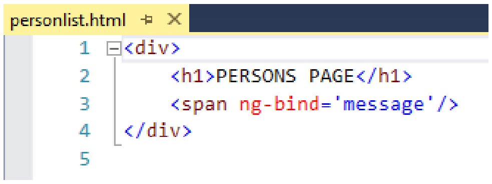
```
var personApp = angular.module('personApp', ['ngRoute']);
```

The personRoutes.js file (see below) defines routes based on the route provider. Lines 4-7 define navigation by effectively saying, when a URL with “/persons” is requested, use a template called “partials/personList” by working through “personListController”. Lines 8-11 indicate a detail page with a route parameter of **personId**. If the URL doesn’t match one of the patterns, Angular defaults to the “/persons” view.



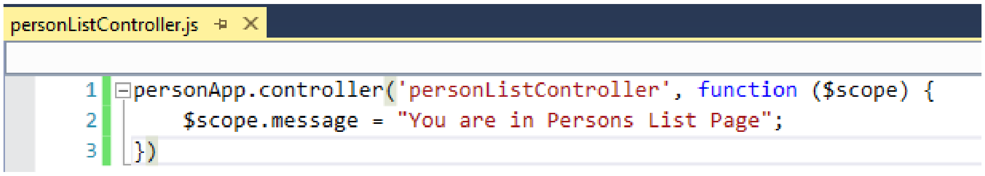
```
personApp.config(['$routeProvider',
    function ($routeProvider) {
        $routeProvider.
            when('/persons', {
                templateUrl: 'partials/personlist.html',
                controller: 'personListController'
            }).
        when('/persons/:personId', {
            templateUrl: 'partials/persondetail.html',
            controller: 'personDetailController'
        }).
        otherwise({
            redirectTo:'/persons'
        })
    }]);
```

The personlist.html is a partial view, which only has HTML that is needed to show person list.



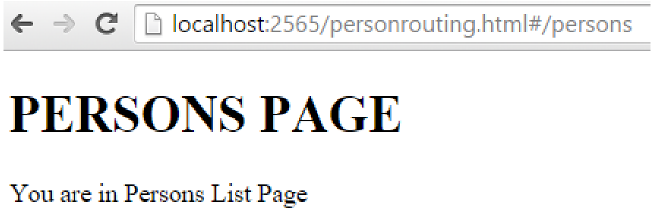
```
<div>
    <h1>PERSONS PAGE</h1>
    <span ng-bind='message'/>
</div>
```

The controller is defined by using the module's controller() function.



```
personApp.controller('personListController', function ($scope) {
    $scope.message = "You are in Persons List Page";
})
```

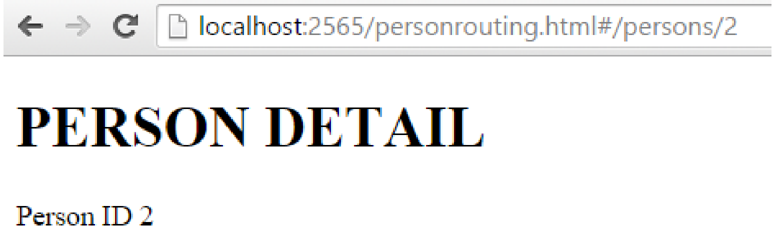
If we run this application and go to /persons URL we will see


localhost:2565/personrouting.html#/persons

PERSONS PAGE

You are in Persons List Page

If we navigate to a detail page, for example, **/persons/2**, we will see the detail partial view:


localhost:2565/personrouting.html#/persons/2

PERSON DETAIL

Person ID 2

Event Handlers

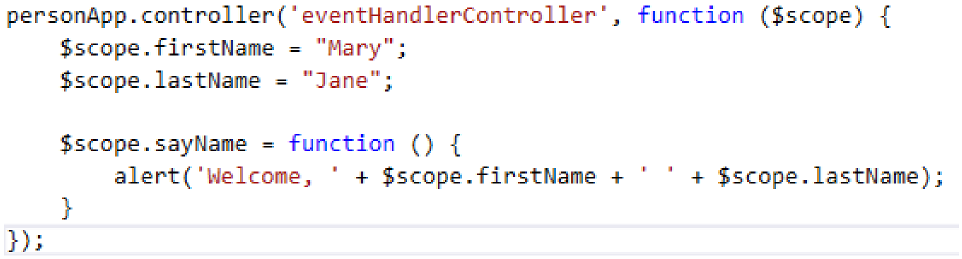
There are a number of directives in AngularJS that add event-handling capabilities to the input elements in your HTML DOM. Below is a list of these events that are built into AngularJS.

- ng-click
- ng-dbl-click
- ng-mousedown
- ng-mouseup
- ng-mouseenter
- ng-mouseleave
- ng-mousemove
- ng-keydown
- ng-keyup
- ng-keypress

- ng-change

Note: You can add your own event handlers using the custom directives feature in AngularJS.

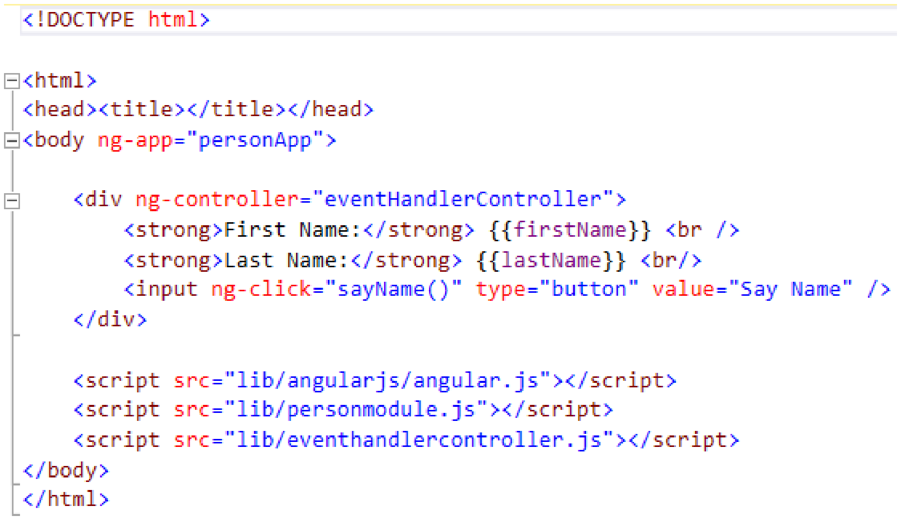
Let's look at **ng-click** event in more detail:



```
personApp.controller('eventHandlerController', function ($scope) {
    $scope.firstName = "Mary";
    $scope.lastName = "Jane";

    $scope.sayName = function () {
        alert('Welcome, ' + $scope.firstName + ' ' + $scope.lastName);
    }
});
```

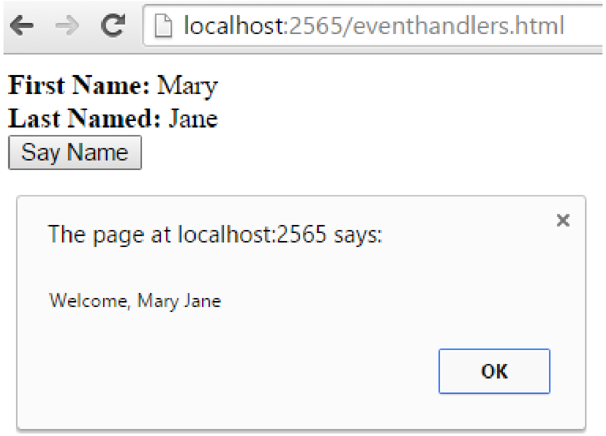
Notice in the eventHandlerController we now have a new **sayName()** function (line #6). All the method is doing for now is showing a JavaScript alert to the user with a welcome message.

The view binds a controller function to an AngularJS event. Line #11 has a new <input> element of type button marked with the **ng-click** angular directive that calls our **sayName()** function which is part of $scope object passed to this view.



```
<!DOCTYPE html>

<html>
<head><title></title></head>
<body ng-app="personApp">

    <div ng-controller="eventHandlerController">
        <strong>First Name:</strong> {{firstName}} <br />
        <strong>Last Name:</strong> {{lastName}} <br/>
        <input ng-click="sayName()" type="button" value="Say Name" />
    </div>

    <script src="lib/angularjs/angular.js"></script>
    <script src="lib/personmodule.js"></script>
    <script src="lib/eventhandlercontroller.js"></script>
</body>
</html>
```

The running example demonstrates that the controller sayName() function is called automatically when the button is clicked.



First Name: Mary
Last Named: Jane

The page at localhost:2565 says:

Welcome, Mary Jane

For more detail on AngularJS built-in event handler directives, be sure to head to the documentation website of AngularJS

Future, Angular 2.0

Angular 2.0 is the next version of AngularJS, which is totally reimagined with ES6 and mobile in mind. It's built using Microsoft's TypeScript language. Angular 2.0 is supposed to hit RTM towards the end of calendar year 2015. There are lot of things that changed in Angular 2.0 when compared to 1.X but Angular team is working hard to provide guidance to developers and things will become more obvious closer to the release. If you wish to play with Angular 2.0 now, Angular team released a new website Angular.io to show the progress and provide early documentation to gather feedback.

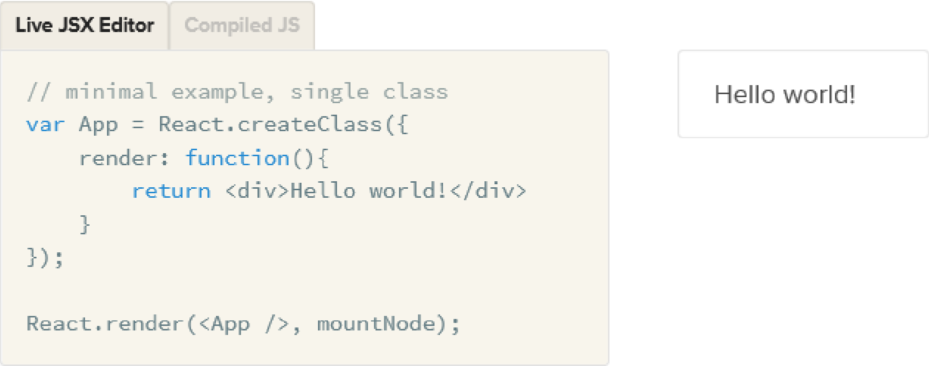
ReactJS. Reusable UI Components.

The ReactJS JavaScript library helps build reusable components for the user interface. You can use React with MVC and AngularJS frameworks, or simply replace a small slice of an existing user interface.

There is certain discussion around React's design. For example, Instagram's Pete Hunt argues in React: Rethinking best practices, that splitting markup and code is not separation of concerns, but merely separation of technologies. Whatever your take on React, the library is popular enough to get 19k stars on GitHub at the time of this writing. The React ecosystem has attracted development of components, add-ins, utilities, templates, training, and connectivity to other popular frameworks.

Getting Started with React

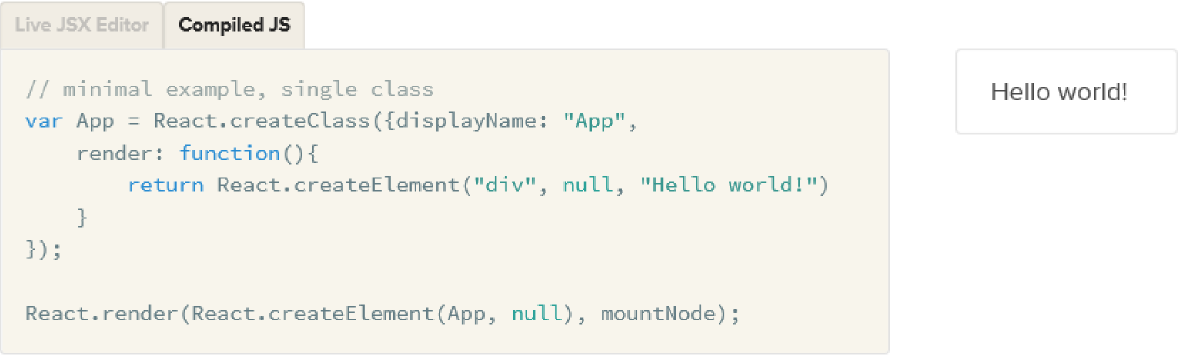
React is expressed with JavaScript or using an XML like syntax called JSX that compiles to JavaScript. You can see both side-by-side at Facebook's react home page. The JSX code appears as a mix of JavaScript and HTML syntax. As you enter JSX, the results render to the right of the editor.



```
// minimal example, single class
var App = React.createClass({
    render: function(){
        return <div>Hello world!</div>
    }
});

React.render(<App />, mountNode);
```

Hello world!

JSX syntax is optional. You can write JavaScript that works with React directly, without having to use JSX. The screenshot below shows the equivalent JavaScript. Each JSX element, such as <div>, has been converted to a React.createElement() JavaScript function.



```
// minimal example, single class
var App = React.createClass({displayName: "App",
    render: function(){
        return React.createElement("div", null, "Hello world!")
    }
});

React.render(React.createElement(App, null), mountNode);
```

Hello world!

Minimal React Class

You can use JSFiddle to try your hand at React without having to set up an environment. Facebook maintains a JSFiddle integration library that lets you play with the syntax in real time.

To render a minimal React component using JSX, first create a class using the React **createClass()** function. The class must have a **render()** function that returns markup.

```
// minimal example, single class
var App = React.createClass({
    render: function(){
        return <div>Hello world!</div>
    }
});
```

Next, call the React **render()** method. The first argument is a React class and the second is a DOM element. When React renders the component, it creates the markup for the class and inserts the markup into the DOM element.

```
React.render(<App />, document.getElementById('container'));
```

Project Setup

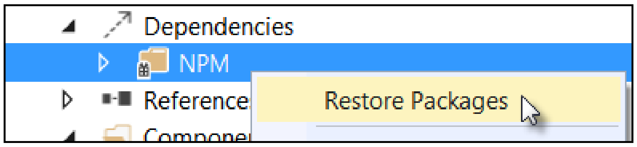
These examples use the Visual Studio client-side build process to deploy the React library and compile JSX code on the fly.

1. In Visual Studio, select **File > New Project** from the main menu.
2. Select the **ASP.NET Web Application** project and click **OK**.
3. Select the **ASP.NET 5 Empty** template and click **OK**.
4. In the Solution Explorer, right-click the project node and select **Add > New Item** from the context menu.
5. Select **NPM configuration file**, leave the defaults, and click the **Add** button.
6. Open the package.json file and edit the devDependencies node to include **grunt**, **grunt-contrib-copy**, **grunt-exec**, **react**, **react-tools,** and **zxcvbn2**. The devDependencies node should look something like the example below.

```
"devDependencies": {
  "grunt": "0.4.5",
  "grunt-contrib-copy": "0.8.0",
  "grunt-exec": "0.4.6",
  "react": "0.13.1",
  "react-tools": "0.13.1",
  "zxcvbn2": "1.0.0"
}
```

Note: Grunt packages manipulate files and push the results out to your web site. The **react-tools** package contains a **jsx** command that compiles your jsx code into pure JavaScript. The **react** package contains **react.js**, a file used at runtime by the React JavaScript on your page.

7. In the Solution Explorer, right-click the **Dependencies > NPM** node and select **Restore Packages** from the context menu.

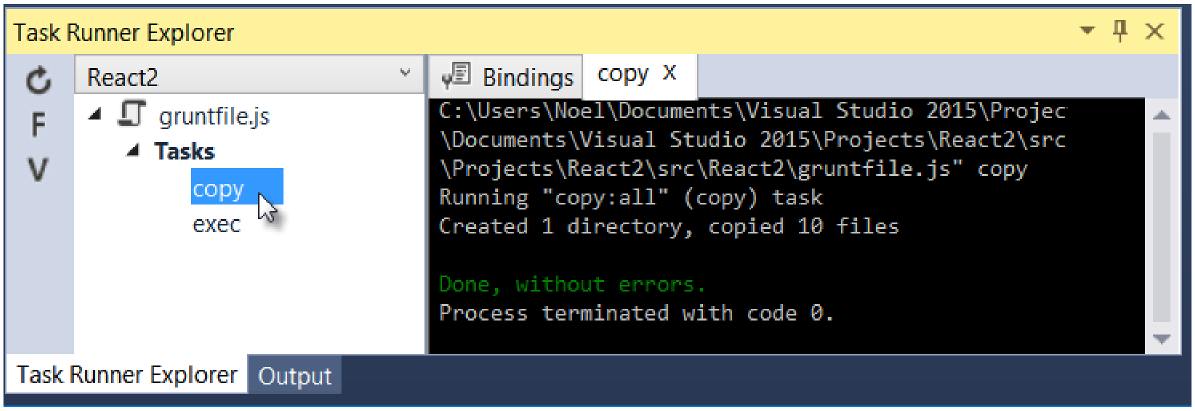


8. In the Solution Explorer, right-click the project node and select **Add > New Item** from the context menu.
9. Select the **Grunt configuration file**, leave the defaults, and click the **Add** button.
10. Open Gruntfile.js for editing and replace the entire contents of the file with the code below.

```
module.exports = function (grunt) {
    grunt.initConfig({
        copy: {
            all: {
                files: [
                {   // required to use React classes and rendering on the client
                    expand: true,
                    cwd: 'node_modules/react/dist/',
                    src: "react.js",
                    dest: 'wwwroot/lib/react/'
                },
                {   // zxcvbn2 is used to measure password strength,
                    // used in a demo and not a direct part of react.
                    expand: true, cwd: 'node_modules/', src: "zxcvbn2/**",
                    dest: 'wwwroot/lib/'
                }, ]
            }
        },
        exec: {    // compiles jsx to JavaScript, results are placed in wwwroot/js
               compile: 'node node_modules/react-tools/bin/jsx -x jsx -w --no-cache-dir
               components/ wwwroot/js'
        }
    });

    grunt.loadNpmTasks("grunt-exec");
    grunt.loadNpmTasks("grunt-contrib-copy");
};
```

The Grunt **copy** task pushes JavaScript libraries to your web site. The **exec** task runs the **jsx** compiler. The compiler converts jsx files in the **/components** folder to JavaScript and places the results in **wwwroot/js**. The command also includes a file watch option so that anytime you change a jsx file in /components, the compiler automatically kicks in and sends the new results to the wwwroot/js folder.

11. In the Solution Explorer, right-click **Gruntfile.js** and select **Task Runner Explorer** from the context menu. Double-click the **copy** task and let it run.

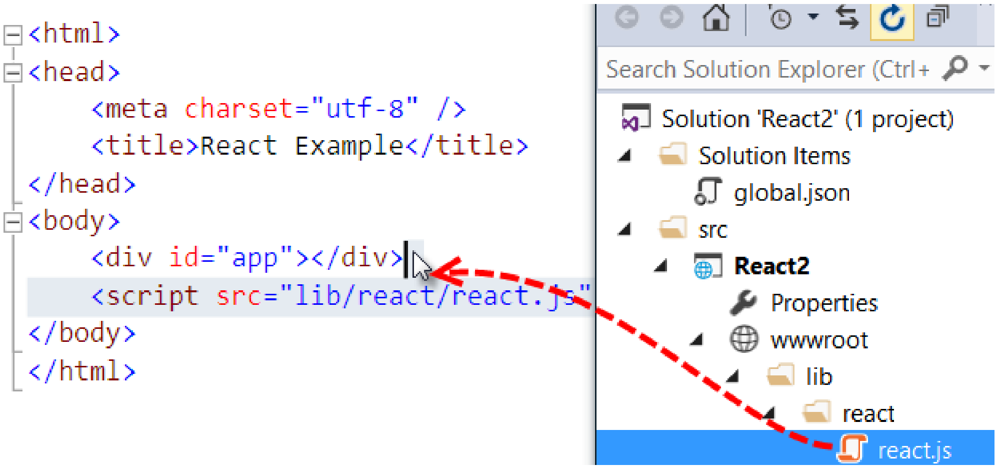


```
C:\Users\Noel\Documents\Visual Studio 2015\Projec
\Documents\Visual Studio 2015\Projects\React2\src
\Projects\React2\src\React2\gruntfile.js" copy
Running "copy:all" (copy) task
Created 1 directory, copied 10 files

Done, without errors.
Process terminated with code 0.
```

12. In the Solution Explorer, right-click the **wwwroot** node and select **Add > New Item** from the context menu. Select the **HTML Page** item, name the page “Index”, and click the **Add** button.
13. Open Index.html and replace the contents with the markup below. The page has a single **<div>** element that has an id attribute **app**.

```
<!DOCTYPE html>

<html>
<head>
    <meta charset="utf-8" />
    <title>React Example</title>
</head>
<body>
    <div id="app"></div>
</body>
</html>
```

14. In the Solution Explorer, open **wwwroot\lib\react** and drag **react.js** to the body of your HTML page, just below the <div>.



```
<html>
<head>
    <meta charset="utf-8" />
    <title>React Example</title>
</head>
<body>
    <div id="app"></div>
    <script src="lib/react/react.js"
</body>
</html>
```

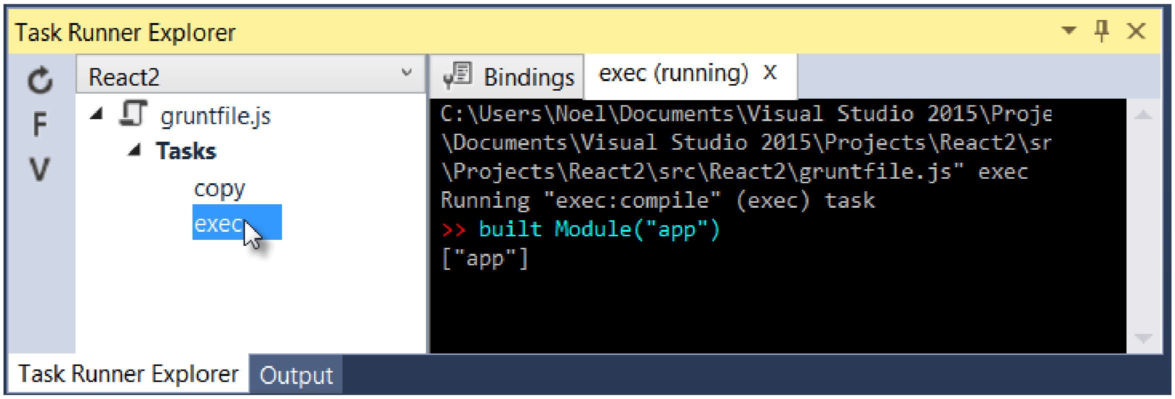
15. In the Solution Explorer, right-click the project node and select **Add > New Folder**. Rename the folder **Components**.
16. Right-click the Components folder and select **Add > New Item** from the context menu.
17. Select the **JavaScript File** type and name it **app.jsx** (note the jsx file extension).

18. Open app.jsx for editing and add the JSX code below.

    The code creates a class called App. The App **render()** function returns a <div> element. The **React.render()** binds the React **App** class to the **app** DOM element.

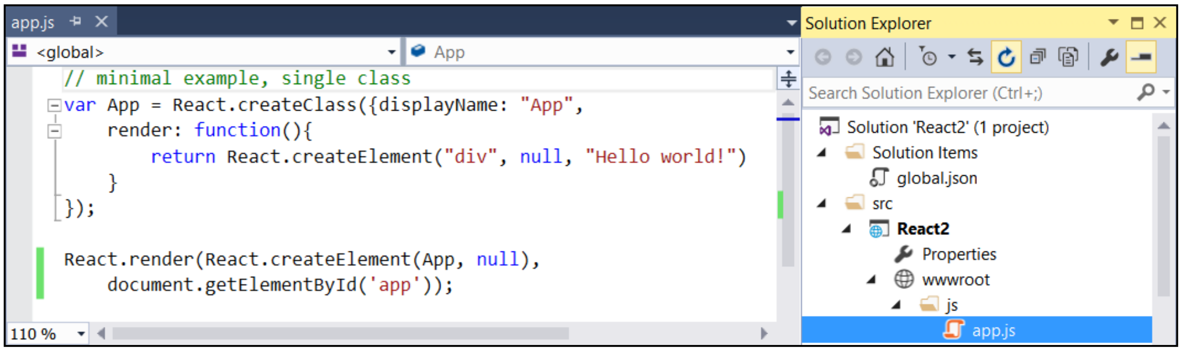
```
// minimal example, single class
var App = React.createClass({
    render: function(){
        return <div>Hello world!</div>
    }
});

React.render(<App />, document.getElementById('app'));
```

19. In the Task Runner Explorer, double-click the **exec** task to run the JSX compiler. The compiler consumes **Components/app.jsx** and produces **wwwroot/js/app.js**.



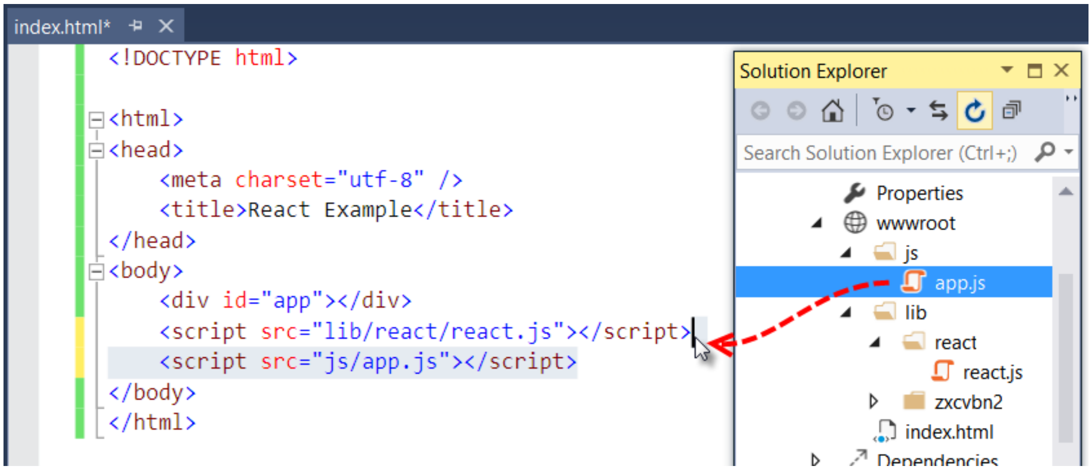
```
C:\Users\Noel\Documents\Visual Studio 2015\Proje
\Documents\Visual Studio 2015\Projects\React2\sr
\Projects\React2\src\React2\gruntfile.js" exec
Running "exec:compile" (exec) task
>> built Module("app")
["app"]
```

20. Verify that the compiled code in **wwwroot/js/app.js** looks like the JavaScript below.



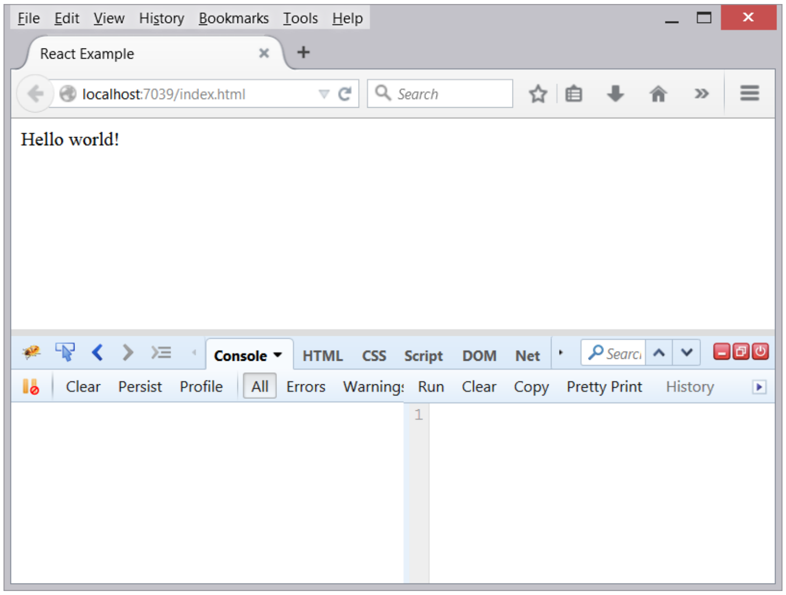
```
// minimal example, single class
var App = React.createClass({displayName: "App",
    render: function(){
        return React.createElement("div", null, "Hello world!")
    }
});

React.render(React.createElement(App, null),
    document.getElementById('app'));
```

21. Open Index.html and drag **app.js** from wwwroot/js to the page, just below the reference to react.js. The page should look like the screenshot below.



```
<!DOCTYPE html>

<html>
<head>
    <meta charset="utf-8" />
    <title>React Example</title>
</head>
<body>
    <div id="app"></div>
    <script src="lib/react/react.js"></script>
    <script src="js/app.js"></script>
</body>
</html>
```

22. Save your files.
23. In the Solution Explorer, right-click **Index.html**, then select **View In Browser** from the context menu. The page displays the Hello world message. The screenshot below shows the example running in Firefox with no errors.


File Edit View History Bookmarks Tools Help
React Example
localhost:7039/index.html
Search
Console HTML CSS Script DOM Net
Search
Clear Persist Profile All Errors Warnings Run Clear Copy Pretty Print History
1

Hello world!

Now that the build mechanism is set, you can change the JSX code at will. Your changes will automatically compile and be pushed out to your website for viewing.

Nesting Classes

React components are composable and can be deeply nested. For example, you can create a Login component and nest it inside the App component. Let’s rewrite app.jsx to include a Login component.

1. Add the code below to define a **Login** class to the top of the file. The snippet below defines the required call to React.createClass(), and implements the render() function.

```
var Login = React.createClass({
    render: function(){
        return <div>
        </div>
    }
});
```

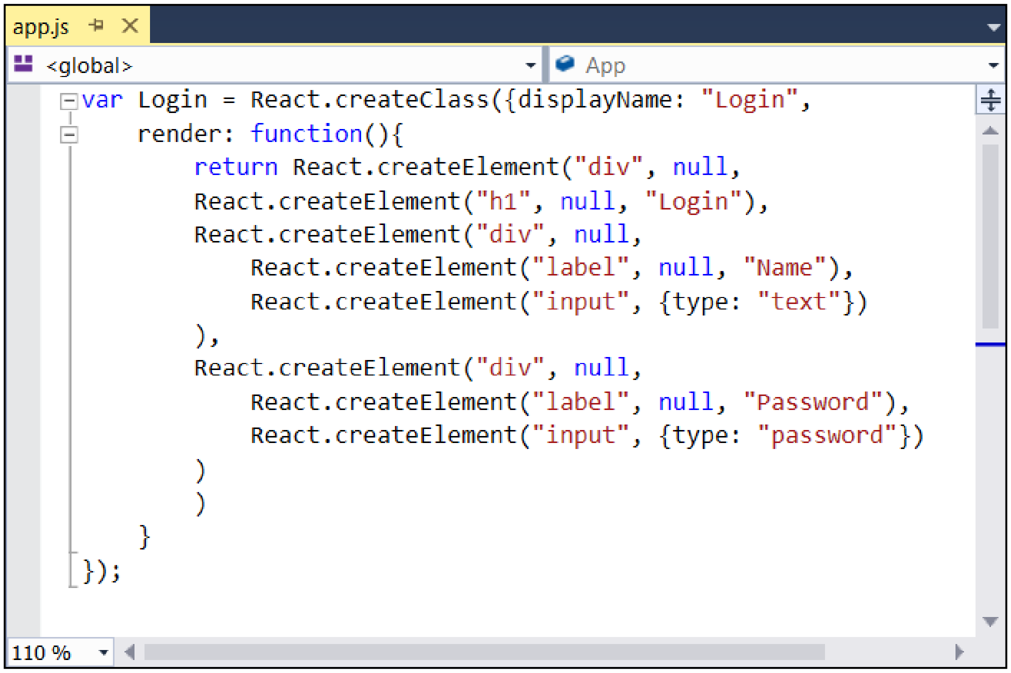
Note: To avoid compiler errors, the markup returned by render() should start on the same line as the **return** key word.

2. Using the example code below, add an **<h1></h1>** element to hold the login title. Below the title, define labels and inputs for the login name and password.

```
var Login = React.createClass({
    render: function(){
        return <div>
              <h1>Login</h1>
              <div>
                     <label>Name</label>
                     <input type='text' />
              </div>
              <div>
                     <label>Password</label>
                     <input type='password' />
              </div>
        </div>
    }
});
```

Note: Notice that the collection of elements in the render() function are enclosed in a single outer div. If you remove the outer div, the compiler will generate a parsing error “Adjacent JSX elements must be wrapped in an enclosing tag”.

The compiled app.js includes the generated Login class, but it has not been bound to the DOM and will not render.



```
var Login = React.createClass({displayName: "Login",
    render: function(){
        return React.createElement("div", null,
        React.createElement("h1", null, "Login"),
        React.createElement("div", null,
            React.createElement("label", null, "Name"),
            React.createElement("input", {type: "text"})
        ),
        React.createElement("div", null,
            React.createElement("label", null, "Password"),
            React.createElement("input", {type: "password"})
        )
        )
    }
});
```

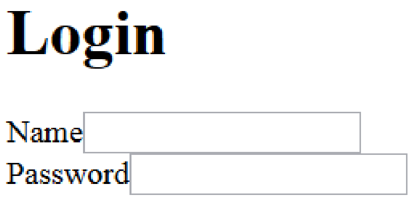
3. Change the App class to return the **Login** component.

```
var App = React.createClass({
    render: function(){
        return <Login />
    }
});
```

4. Review **app.jsx**. The code should look like the complete example below. The Login class renders the title, name and password elements, App renders the Login component, and the call to React.render() at the bottom of the page binds the App component to an element in the DOM.

```
var Login = React.createClass({
    render: function(){
        return <div>
               <h1>Login</h1>
               <div>
                      <label>Name</label>
                      <input type='text' />
               </div>
               <div>
                      <label>Password</label>
                      <input type='password' />
               </div>
        </div>
    }
});

var App = React.createClass({
    render: function(){
        return <Login />
    }
});

React.render(<App />, document.getElementById('app'));
```

5. View **Index.html** to see the Login component rendered in the browser.



Login

Name

Password

Class Properties

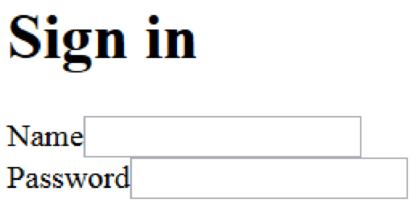
React *props* allow an owner component to set properties on components it owns. In the last example, App was the owner component and Login the owned component. Soft-code the Login header by including an attribute on the Login element during the App render() as shown in the JSX snippet below.

```
var App = React.createClass({
    render: function(){
        return <Login title="Sign in"  />
    }
});
```

When App assigns the **title** attribute, React creates **props.title** in the Login component. The **title** attribute is expressed inside the Login class as **{this.props.title}** where the curly braces contain JavaScript. You can re-write the Login render function to look like the example below:

```
<h1>{this.props.title}</h1>
```

The running example displays the assigned title value in the heading.



Sign in

Name

Password

Property Validation

Properties can be validated to verify they exist, are of a certain type, or even your own custom validation, using React **propTypes**. You can check primitive types like array, boolean, function, number object, and string. React also supports arrays of type, instances of a type, enumerations, and unions. Search the react.js source file for **var ReactPropTypes = {** to see all possible property types that you can validate.

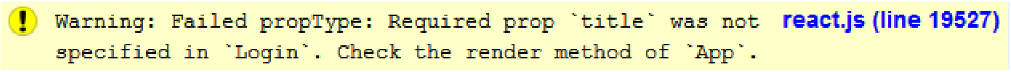
The propTypes member takes an object with the properties you are expecting. The propTypes example below defines an object that includes a **title** member and assigns a React.PropTypes member. This example requires the title property, and that the value must be a string value.

```
var Login = React.createClass({
       propTypes: {
              title: React.PropTypes.string.isRequired
       },...
```

Test this by removing the title attribute from the Login element:

```
var App = React.createClass({
    render: function(){
        return <Login />
    }
});
```

The jsx file will compile without error but will generate a warning at runtime. Use the console of the developer tools for your browser to display the message.



```
Warning: Failed propType: Required prop `title` was not   react.js (line 19527)
specified in `Login`. Check the render method of `App`.
```

Default Properties

Another way to guarantee valid property values is to assign default properties. Use the **getDefaultProps()** function to return an object that includes the properties to assign.

```
getDefaultProps: function() {
       return {
              title: 'Login'
       };
},...
```

If the owner component does not assign the title attribute, getDefaultProps() will initialize properties before they are checked by propTypes validation.

Class State

React props are immutable and cannot be changed other than by initial owner assignment. If part of the component changes independently from its owner, use the React *state* object to store and communicate values. When component state changes, the component and its children are re-rendered, keeping the component state and the DOM in sync. Whenever the user types into the input, the DOM onChange event fires, and the input value displays in the console.

This next example shows how to manage state. When the user types into the password text input, the password strength is calculated and stored in a state variable. When state changes, the component renders to show the password strength in a label.

Encapsulate Password in its own Component

First, refactor Login to extract password functionality into its own component.

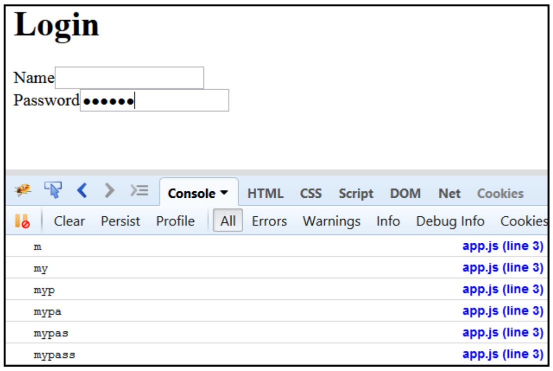
1. Create a new Login password class using the code below.

```
var Password = React.createClass({
       onChange: function(e){
              console.log(e.target.value);
       },
       render: function(){
              return <div>
                     <label>Password</label>
                     <input type='password' onChange={this.onChange} />
           </div>
       }
});
```

2. Change the Login component to render the Password component.

```
var Login = React.createClass({
    render: function(){
        return <div>
              <h1>{this.props.title}</h1>
              <div>
                     <label>Name</label>
                     <input type='text' />
              </div>
              <Password />
        </div>
    }
});

var App = React.createClass({
    render: function(){
        return <Login title="Login" />
    }
});

React.render(<App />, document.getElementById('app'));
```

3. Run the application in the browser. The console in the screenshot below shows password input contents as the user types.



Login

Name

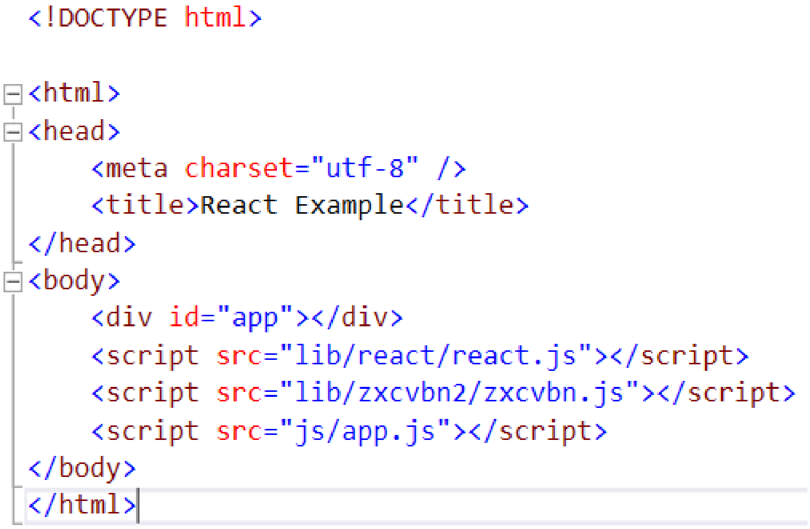
Password ••••••

```
m                                   app.js (line 3)
my                                  app.js (line 3)
myp                                 app.js (line 3)
mypa                                app.js (line 3)
mypas                               app.js (line 3)
mypass                              app.js (line 3)
```

Add Password Strength Checking

Next, add the Dropbox **zxcvbn** library to judge password strength. The key function in the library is **zxcvbn(password)**. It takes a password string and returns a score from zero to four. The example retrieves a description for strength ("weak", "good", etc.) and assigns that string to a state variable. The Password component reflects the strength state in a label.

1. Drag the **zxcvbn.js** file from **wwwroot/lib/zxcvbn2** and drop it into **index.html**, just before the app.js script reference. The markup should look like the screenshot below.



```
<!DOCTYPE html>

<html>
<head>
    <meta charset="utf-8" />
    <title>React Example</title>
</head>
<body>
    <div id="app"></div>
    <script src="lib/react/react.js"></script>
    <script src="lib/zxcvbn2/zxcvbn.js"></script>
    <script src="js/app.js"></script>
</body>
</html>
```

2. On app.jsx, add **strengthNames** array declaration to the top of the file.

```
var strengthNames = ['Very weak', 'Weak', 'Good', 'Strong', 'Very Strong'];
```

3. Remove the code from Password **onChange** so that it is an empty function.
4. Add the code below to the onChange function.
   - First call the **zxcvbn()** function, passing the text from the input and assigning the local **score** variable.
   - Use the score variable to index into the **strengthNames** array and assigns result to the local **strength** variable.

- Finally call the React component's built-in **setState()** method. Pass an object that creates a new strength member and assigns it.

```
onChange: function(e){
    // get a numeric score from 0 to 4
    var score = zxcvbn(e.target.value).score;
    // get a name for the score, i.e. "weak", "good", etc.
    var strength = strengthNames[score];
    // create a state member called strength
    // and assign it the name.
    this.setState({strength: strength});
},
```

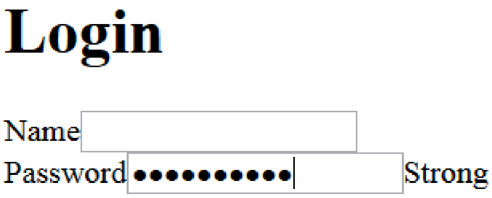
5. Add a call to **getInitialState()** to populate state.strength with a default value.

```
getInitialState: function(){
       return {strength: "Very weak"};
},
```

6. Use the new state member during the Password render. Add a new <label> element to the render output. Assign {this.state.strength} to the <label> text. When state.strength is updated during the onChange event, the Password component renders and the label displays the new value.

```
render: function(){
       return <div>
              <label>Password</label>
              <input type='password' onChange={this.onChange} />
              <label>{this.state.strength}</label>
       </div>
}
```

7. View index.html in the browser. The label changes as the length and complexity of the password changes.



Login

Name

Password •••••••••• Strong

Style

React *inline styles* are objects passed to the **style** attribute of an element. The example below defines margin and text color for the <input> element. Hyphenated styles are camel-cased, so that **margin-left** becomes **marginLeft**.

```
var inputStyle = {
       color: 'purple',
       marginLeft: '10',
       marginRight: '20'
};

...

render: function(){
       return <div>
               <input type='password' style={inputStyle} />
       </div>
}
```

See the React inline styles documentation for more details.

If you don’t care for React inline styles, you can add CSS classes to your markup as you would with any web page. Just don’t call them classes. React cannot have JavaScript reserved words in markup, including *class*. Instead, use **className**. The example below assigns the Bootstrap row style to a <div> element.

<div className="row"></div>

Summary

The React JavaScript library helps build reusable components to construct user interfaces. React code can be written directly as JavaScript or can be expressed in JSX syntax as a combination of markup and code. React classes describe the layout and behavior of a component. Each React class must minimally include a render() function. The React.render() function is required to bind a React class to an element of the DOM.