Module 6: Client-side Development

Student Lab Manual

Instructor Edition (Book Title Hidden Style)

Conditions and Terms of Use

Microsoft Confidential

This training package is proprietary and confidential, and is intended only for uses described in the training materials. Content and software is provided to you under a Non-Disclosure Agreement and cannot be distributed. Copying or disclosing all or any portion of the content and/or software included in such packages is strictly prohibited.

The contents of this package are for informational and training purposes only and are provided "as is" without warranty of any kind, whether express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Training package content, including URLs and other Internet Web site references, is subject to change without notice. Because Microsoft must respond to changing market conditions, the content should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Copyright and Trademarks

Lab 6: Client-side Development

Introduction

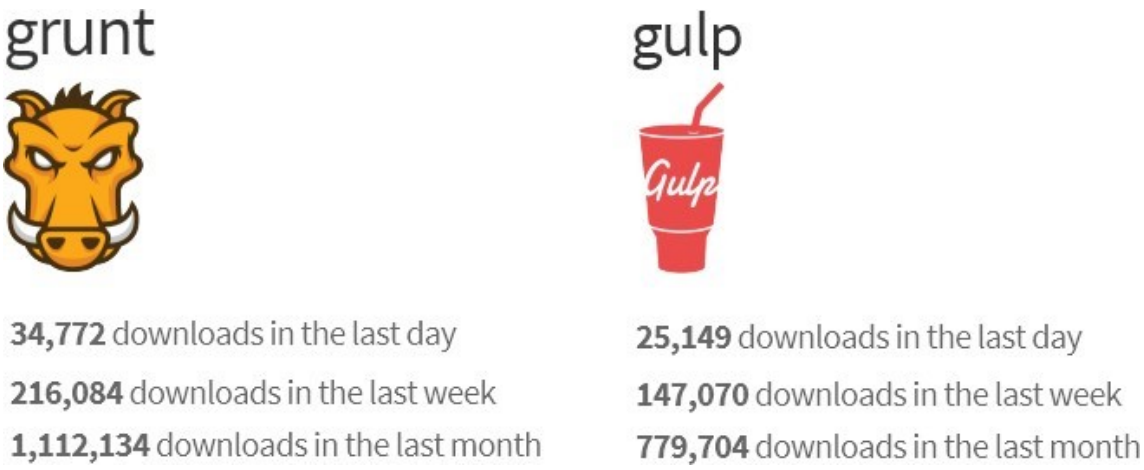
Both Grunt and Gulp are JavaScript task runners that automate script minification, TypeScript compilation, code quality “lint” tools, CSS pre-processors, and just about any repetitive chore that needs doing to support client development.

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 is a screenshot from the npmjs (node package manager) home site downloads.



Objectives

This lab will show you how to:

- Use Grunt and Gulp, JavaScript, and task runners
- Use Visual Studio Task Runner
- Use NPM package manager for client-side
- AngularJS with ASP.NET Model-View-Controller (MVC)

Prerequisites (if applicable)

This lab does not build on previous labs and neither is its end-solution used in the following labs.

System Requirements

To complete this lab, you need:

- Visual Studio 2019 or higher

Estimated Time to Complete This Lab

60-90 minutes

Exercise 1: Using Grunt, NPM and Visual Studio Task Runner

Objectives

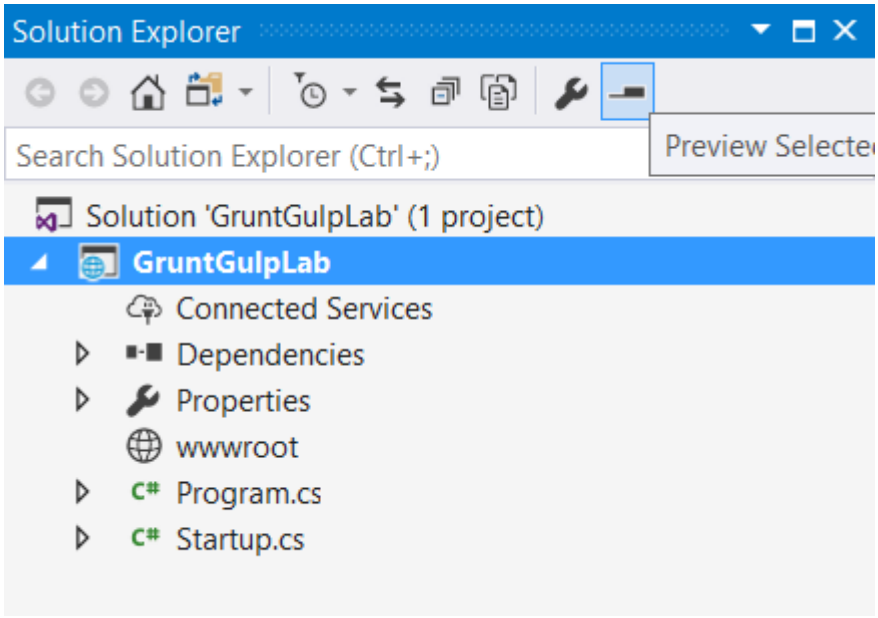
In this exercise, you will:

Use Grunt (JavaScript task runner) using Visual Studio Task Runner to automate script minification, TypeScript compilation, code quality "lint" tools, CSS pre-processors, and just about any repetitive chore that needs to support client development.

Task 1: 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 2017 settings and will be our raw material to process using Grunt.

1. In Visual Studio 2019, create a new **ASP.NET Core Web Application**, name it **GruntGulpLab**.
    - In the **New ASP.NET Project** dialog box, select the **Empty** template and click **OK**.
    - In the Solution Explorer, review the project structure. The solutions folder includes empty **wwwroot** and **Dependencies** nodes.



2. Add a new folder named **TypeScript** to your project directory and add a typescript configuration file:
    - Right-click the **TypeScript** folder > **Add** > **New Item**.
    - Under ASP.NET Core/Web/Scripts templates, choose **TypeScript JSON Configuration File**. Leave the name as **tsconfig.json**. Click Add.

    The presence of a tsconfig.json file in a directory indicates that the directory is the root of a TypeScript project. The tsconfig.json file specifies the root files and the compiler options required to compile the project.

3. Add the "compileOnSave" property to the **tsconfig.json** file and set it to **true**.

```
"compileOnSave": true,
```

This ensures that whenever you change and save your typescript files, compilation will happen and generate the corresponding javascript files.

4. Add a new TypeScript file to the TypeScript folder
    - Right-click the **TypeScript** folder and select **Add > New Item** from the context menu. (You can find it under ASP.NET Core/Web/Scripts)
    - Select the **TypeScript 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 }
      ```

5. Add another TypeScript file to the **TypeScript** folder and name it **Food.ts**.

````
Copy the code below into the **Food.ts** file:


``` typescript
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;
    }

}
```
````

6. Install the **Microsoft.TypeScript.MSBuild** NuGet package. This enables TypeScript compilation in the project

Task 2: Configuring NPM

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

1. Add a new npm Configuration File to the project:

- Right-click the GruntGulpLab project and click **Add > New Item**
- Select the **npm Configuration File** item (under ASP.NET Core/Web/General),
- Leave the default name, **package.json**, and click the **Add** button.

2. In the **package.json** file, specify that we will use the Grunt tooling:

    - Inside the **devDepencies** object braces, enter `"grunt"`.
    - Select **grunt** from the IntelliSense list and press **Enter**.
    - 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 upper section of the IntelliSense list (press **Ctrl-Space** if IntelliSense does not appear).

    > ***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 - pick the latest version.

```
"devDependencies": {

    "grunt": "1.0.1",

    "grunt-contrib-clean": "^2.0.0",
    "grunt-contrib-jshint": "^2.1.0",
    "grunt-contrib-concat": "^1.0.1",
    "grunt-contrib-uglify": "^4.0.1",
    "grunt-contrib-watch": "^1.1.0"
}
```

> *Note the comma at the end of each line listing the package*

4. Save the **packages.json** file. Visual Studio will automatically restore packages. The final list should show as below.

Task 3: 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. Add a Grunt Configuration file to your project:

    - Right-click the project and select **Add > New Item**.
    - Select the **JavaScript File** item template

- Change the name to *Gruntfile.js,* and click the **Add** button.

  Add the following code to Gruntfile.js. The `initConfig()` function sets options for each package, and the remainder of the module loads and register tasks.

  This is what you should see in the **Gruntfile.js** file:

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

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

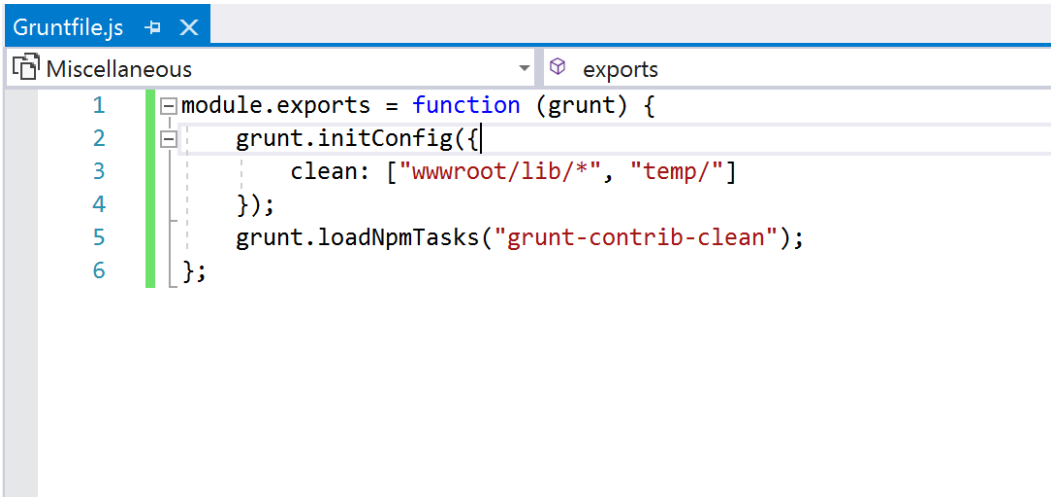
```
module.exports = function (grunt) {
grunt.initConfig({
    clean: ["wwwroot/lib/*", "temp/"]
    });
};
```

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

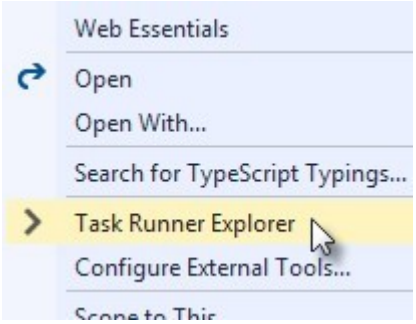
```
module.exports = function (grunt) {
    grunt.initConfig({
        clean: ["wwwroot/lib/*", "temp/"]
    });

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

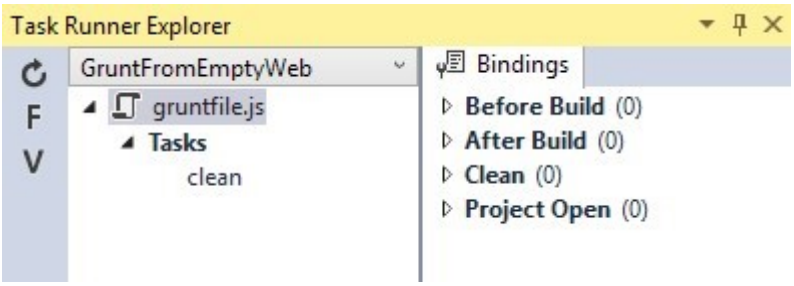
4. Save **Gruntfile.js**. The file should look something like the following screenshot:.



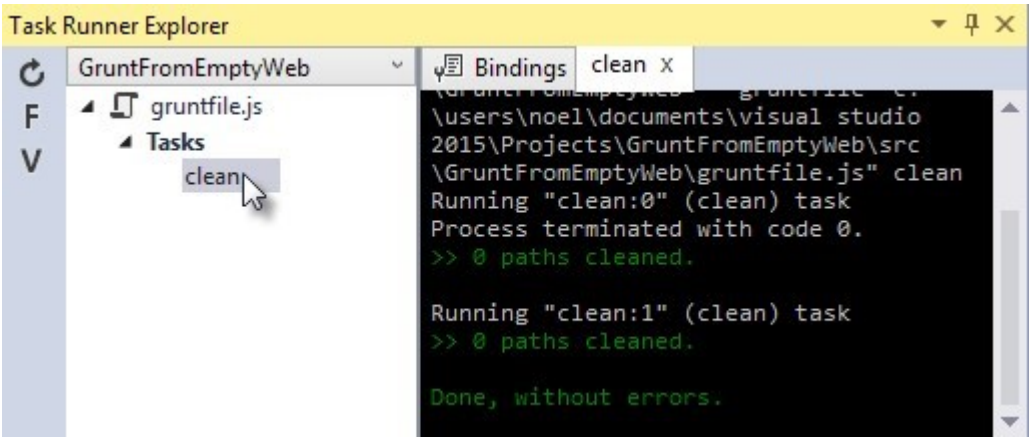
5. Right-click Gruntfile.js and select **Task Runner Explorer** from the context menu. The Task Runner Explorer window will open.



6. Verify that **clean** shows under **Tasks** in the Task Runner Explorer. Note that you may need to click the **Refresh** button (upper-left).



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



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.

Also, you might see a “clean:0” and “clean:1”. This corresponds to the files we listed in our “clean” array earlier: clean: ["wwwroot/lib/*", "temp/"]

8. Inside 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. Inside the **initConfig()** method, add the **jshint** task using the code below.

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

```
``` js
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, that is **Tastes["Sweet"]** instead of **Tastes.Sweet**. The option turns off the warning to allow the rest of the process to continue.
```

10. Inside the **initConfig()** method, 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*.

```
``` 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 similar to the example in the following screenshot:

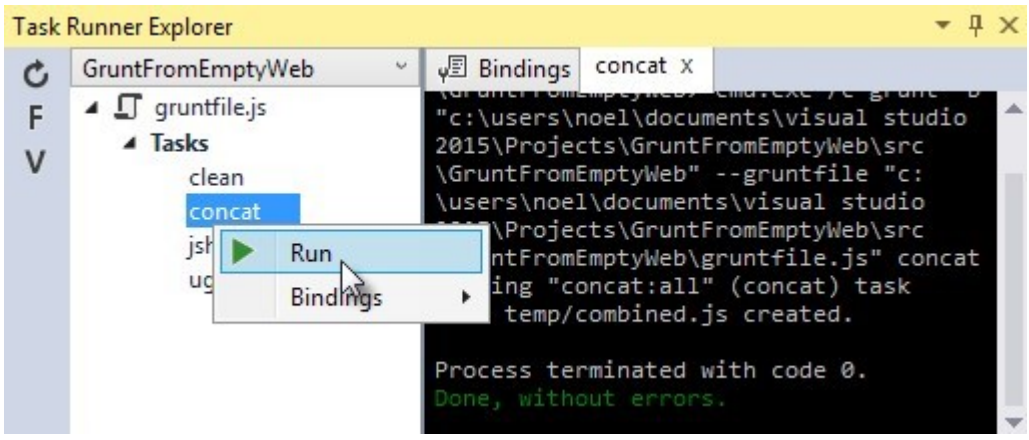
Miscellaneous | options

```
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'
                }
            }

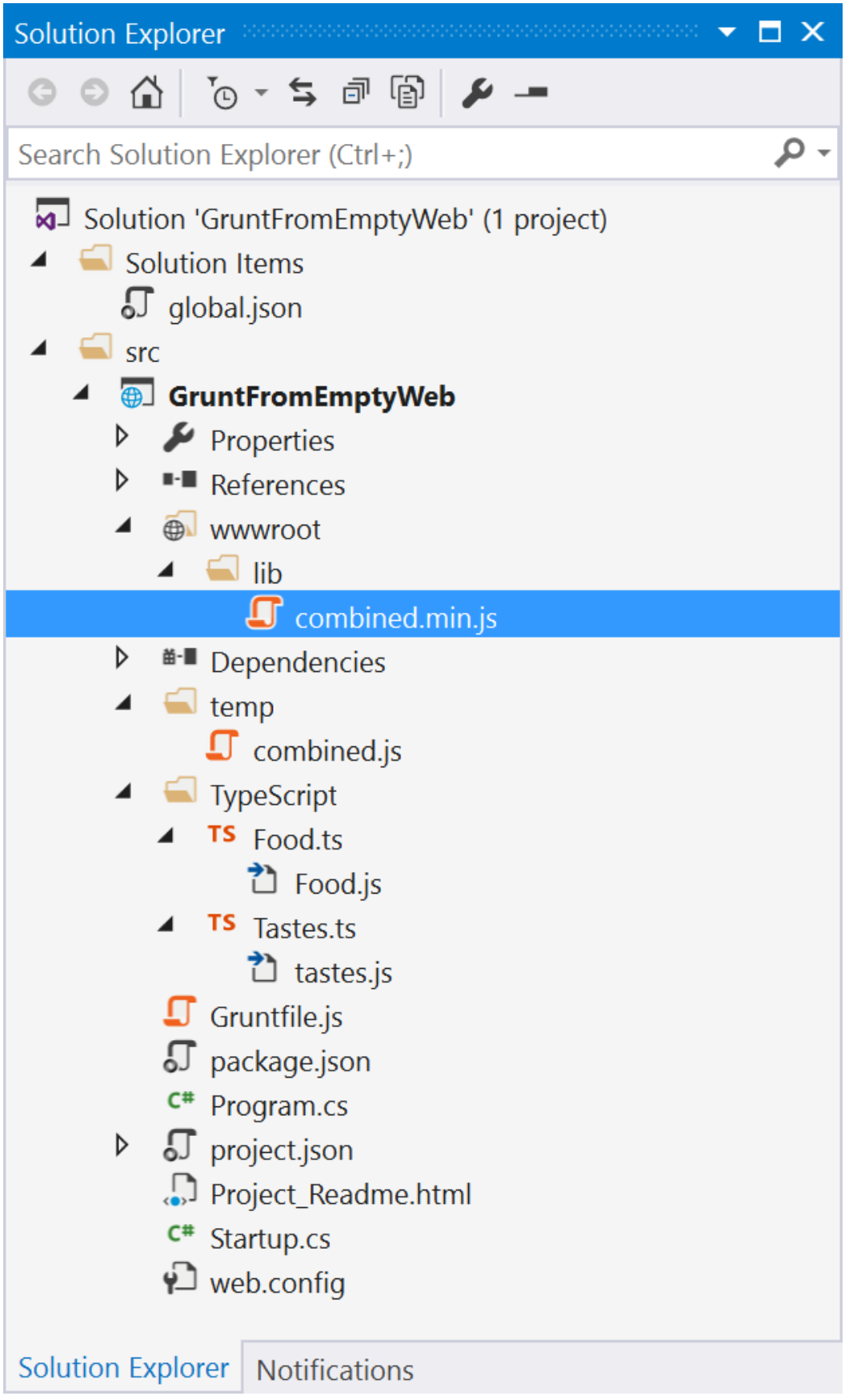
    });

    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.



The concat task creates a new combined.js file and places it into the temp directory. The jshint task simply runs and does not produce the output. The uglify task creates a new combined.min.js file and places it into wwwroot\lib. On completion, the solution should look similar to the following screenshot:


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'GruntFromEmptyWeb' (1 project)
Solution Items
global.json
src
GruntFromEmptyWeb
Properties
References
wwwroot
lib
combined.min.js
Dependencies
temp
combined.js
TypeScript
Food.ts
Food.js
Tastes.ts
tastes.js
Gruntfile.js
package.json
Program.cs
project.json
Project_Readme.html
Startup.cs
web.config
Solution Explorer
Notifications

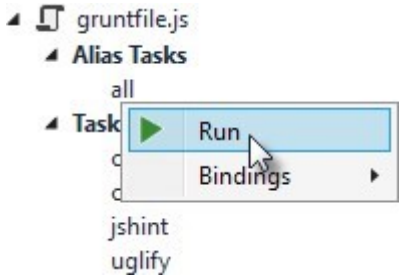
Note: For more information on the options for each package, see: https://www.npmjs.com/ and lookup the package name in the search box on the main page. For example, you can look up the **grunt-contrib-clean** package to get a documentation link that explains all the parameters.

Task 4: All Together Now

1. 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, below **initConfig().**

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

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



Task 5: Watching for Changes

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

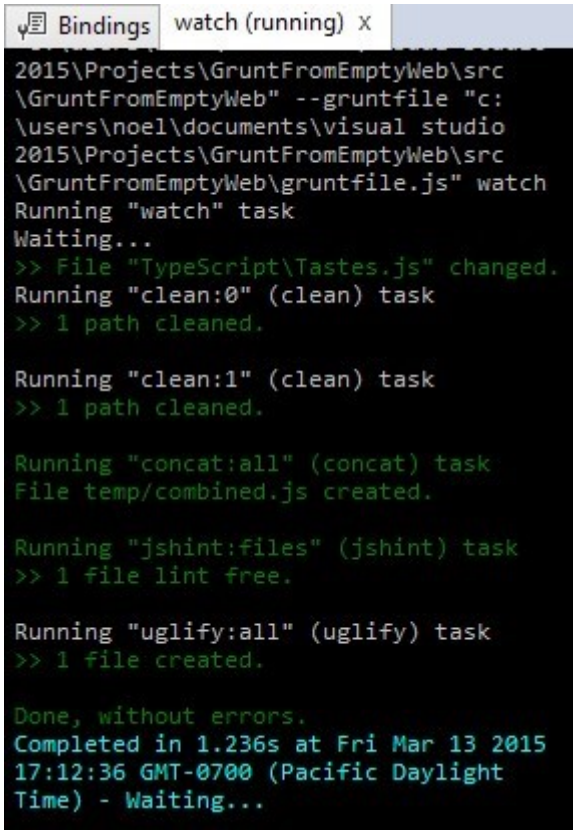
```
watch: {
    files: ["TypeScript/\*.js"], tasks: ["all"]
}
```

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

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

3. 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, make a change that is not just whitespace, and then save the file. This will trigger the **watch** task and trigger the other tasks to run in order. The following screenshot shows a sample run.

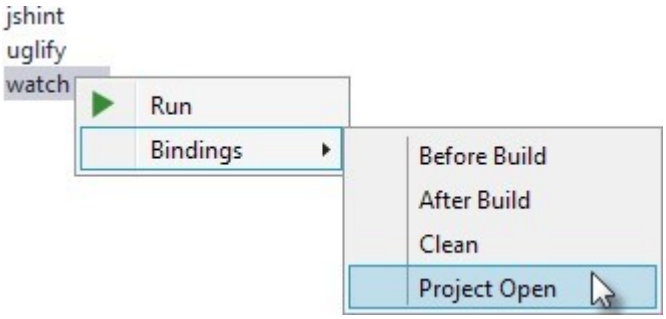



```
> Note: If a change in TypeScript file does not reflect in updated js file, update the
following setting in Tools > Options
>
> Then go to JavaScript/TypeScript > Project > General > Automatically compile
TypeScript files which are not part of a project
>
>    ![](Media/Mod_02_11/2019-11-20-21-43-37.png)
```

Task 6: Binding to Visual Studio Events

1. 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.

2. 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.



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

Exercise 2: Using Gulp

Objectives

In this exercise, you will:

Use Gulp (JavaScript task runner) using Visual Studio Task Runner to automate tasks that were automated by Grunt in the previous exercise.

Task 1: Configuring NPM

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

1. The **devDependencies** defined in package.json are specific to Gulp. To get the same result as the Grunt walk-through, **package.json** should add the following gulp packages. You can get the correct version number using IntelliSense (Ctrl + space).

```
"gulp": "4.0.2",
"gulp-clean": "^0.4.0",
"gulp-jshint": "^2.1.0",
"gulp-concat": "^2.6.1",
"gulp-uglify": "^3.0.2",
"gulp-rename": "^1.4.0"
```

Task 2: Configuring Gulp

1. Instead of adding Gruntfile.js to the project, add a **Gulp Configuration File** to the project and name it **gulpfile.js**. In **gulpfile.js,** assign a series of objects using the **node.js require()** method.

   At the top of **gulpfile.js**, 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');
var rename = require('gulp-rename');
```

2. 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.

```
``` js
gulp.task('all', function () {

});
```
```

3. Add the empty task() method to gulpfile.js and it will display the **all** task in Task Runner Explorer. (Refresh the Task Runner Explorer if needed)

4. Inside the **task()** function, use the objects defined earlier by **require()** method 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());

});
```

Task 3: All Together

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.

1. Here is the complete code that concatenates, lints, minifies and writes the minified file. The processing time is quite fast. Add this code to your file.

```
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'));
});
```

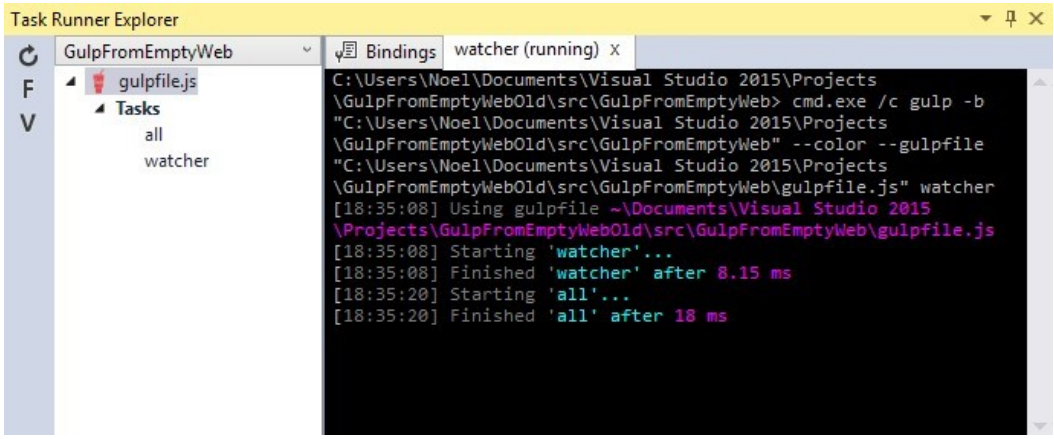
2. Watcher tasks are similar to the Grunt parallel task and are simple to set up. Again, the gulp.task() method names the task that will show in the Task Runner Explorer. The Gulp **watch()** method takes a path or array of paths and second parameter is an array of tasks to run.

   Add the "watcher" task code to the end of our **gulpfile.js**.

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

   Your file should look like the code below:

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



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. ===