# An MVC-Knockout-TypeScript-Webpack Starter project

During a recent project building a relatively complex single page app with KnockoutJS and TypeScript, we ran into a range of problems and decisions which had to be made. The result of what we’ve learned during this project is incorporated in this Knockout-TypeScript-Webpack starter project.

One of the things we ran into in this project was the need for bundling and minification. No problem in a run-of-the mill ASP.NET MVC project, but rather different in single-page application using require syntax in the JavaScript. So instead of doing this at the end of the project, in this starter project, installation of Webpack will be one of the first steps.

I have a backend programming background myself. After doing some work recently with Angular 2 and TypeScript, I was really enthusiastic about TypeScript, because of the similarities with programming in C#. So TypeScript seemed a logical choice for the project.

In the best tradition of JavaScript framework tutorials, the application which we’ll set up through the starter project is single-page application for the booking of superheroes.

## The Visual Studio solution

The Visual Studio solution consists of two projects, Web and Base. Web contains the web application as well as the single-page application. Base is a class library project contains models, controllers and helper classes.

## Prerequisites

So, what do we need to install the starter project?

1. Node.js
2. TypeScript
3. Webpack
4. Knockout

## Install Node.js

Webpack relies on Node, so the first step is to install Node if you haven’t got Node on your machine already.  
<https://nodejs.org/en/download/>

We’ll need a package.json file in the root of the web project, a nice way to set up this file is by running :  
  
npm init

from a command window in admin mode, in the root of the web project. You’ll be asked some questions to set up some default values.

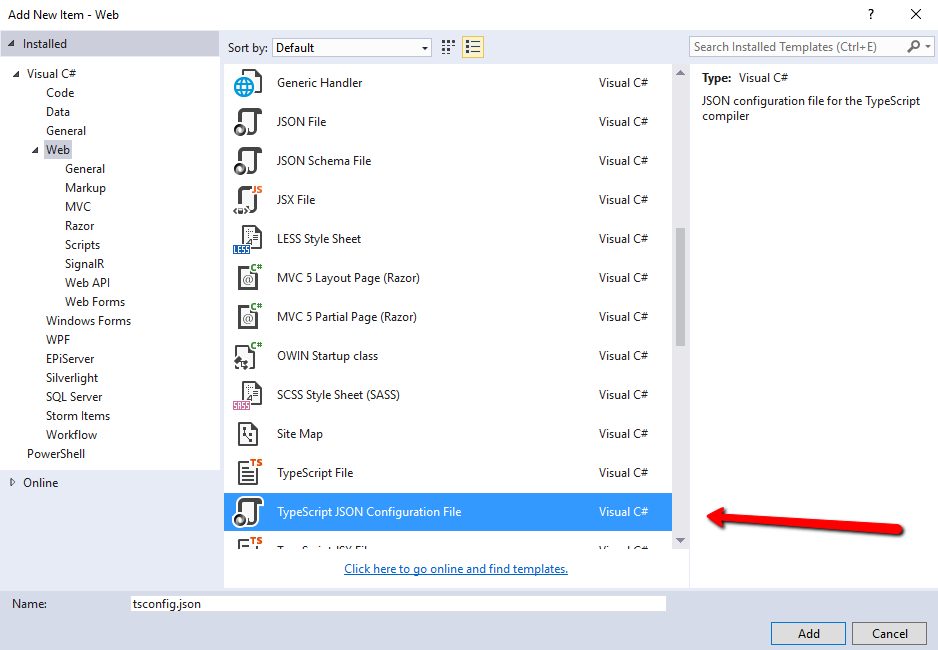
## Install TypeScript

Now we have Node.js installed, we can use the Node.js package manager (NPM) to install TypeScript.

Open a command window in admin mode, in the root of the web project, and type

npm install typescript –-save-dev

Add a TypeScript configuration file to the web project. Right click the Web project, choose “Add”, “New Item” and “TypeScript JSON configuration file”. Don’t change the default name of the project, tsconfig.json



Because we’ll be saving the TypeScript source files in a directory called “source”, we’ll add the key “baseUrl” under the compiler options:   
  
"baseUrl": "./source/"

We’ll leave the target key set at “es5” for maximum compatibility with older browsers, i.e. the TypeScript compiler will transpile the TypeScript code to EcmaScript 5 compatible JavaScript.

## Typings

For TypeScript’s strong typing to work, TypeScript needs some information on the types in various libraries. These are supplied by type definition files, with the extension .d.ts

The files which can be added to your project through Nuget, GitHub (<https://github.com/DefinitelyTyped/DefinitelyTyped> ) or the node package manager. In the starter project a directory with type definition files for some important libraries is included under the root. Type definition files for Knockout, Require and jQuery are included.

## Install Webpack

Install Webpack using the Node Package Manager, from the command line, in admin mode, from the root of the web project:

npm install webpack –-save-dev

For more on installing Webpack see:

<https://webpack.github.io/docs/tutorials/getting-started/>

Add a file called webpack.config.js to the root directory of the web project. Copy it from the sample project. More on all the options later.

For Webpack and TypeScript to work, additional packages are necessary. To get started use the package.json file from the sample project, and run

npm install

This will install a range of required packages.

## Install KnockoutJS

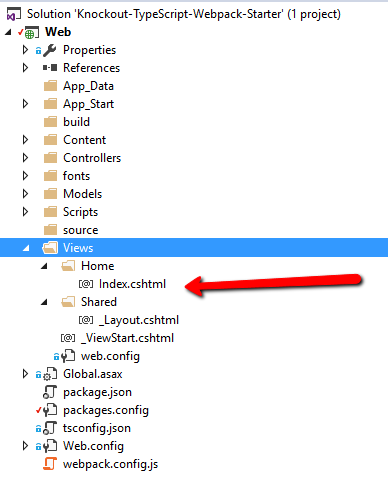
We’ll install KnockoutJS through NPM:

npm install knockout –-save-dev

## Set up a host MVC view for the single page application

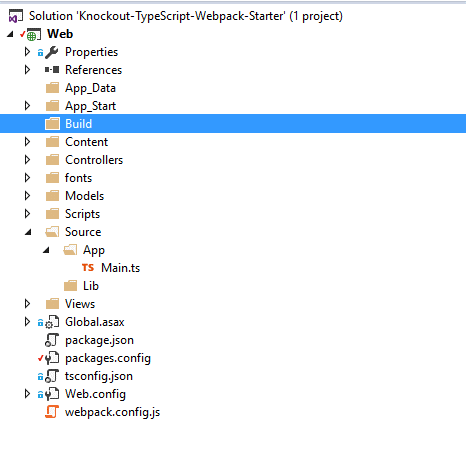
The original project from which this starter project is derived was an ASP.NET MVC application with a commercial CMS system. In this project, we’ve kept things simple, we just have one MVC controller and view to host the single page application. But in your real-world application this could be any kind of web page.

So we have some basic ASP.NET MVC ingredients: a HomeController, a corresponding MVC view, and a \_Layout.cshtml. So Views > Home > Index.cshtml will contain the starting point of our single page application.



## Setting up the single page application

The basic folder structure for the single page application is a folder called “source” for the TypeScript files and JavaScript libraries, and an output directory called “build”, both under the root. The application specific files will be places under a directory called “app”, the libraries will placed under a directory called “lib”.



So what is the first step?

The first thing to do is load the file which will initialize the application. This is Main.ts, in the root of the App folder. We do this by referring to this file in webpack.config.js, by setting the entry key to Main.ts

If we run the webpack command in the console, the file app.bundle.js will be generated in the Build folder.

We refer to this file in our \_Layout.cshtml file:   
  
<script src="~/Build/app.bundle.js"></script>

To check if all this works, you can put this code in Main.ts:

export class Main {

constructor() {

alert("Hello World");

}

}

let main = new Main();

## Setting up KnockoutJS

Now we’ve set up the entry point of our application it is time to add Knockout to the mix.

In this application, every element will consist of a Knockout component. For more information on Knockout components, see:

<http://knockoutjs.com/documentation/component-overview.html>

Every component consists of a model and a view. In this project, the model and view of every component are kept together in a single directory.

The starting point is a component called MainForm. Eventually, this component will host other components, which may also host other components, etc. MainForm is mainly a container for the other components, in which the actual work will be done.

To be able to use the Knockout components, each component will have to be registered first. Because the number of components in a large single page application may be substantial, a separate file is use for component registration. This file is called ComponentRegistration.ts.

We also want to set up a main view model for the application. In this case, we will use this model to hold the booking details which result from the various choices the customer will make during the booking process.

This model, called BookingData, will live in a separate TypeScript file, called BookingData.ts

So initially, the following things will have to happen in the Main.ts file:

1. Bind the BookingData Knockout viewmodel.
2. Load the file in which the Knockout components are registered.

### Debugging Knockout in Chrome

TODO

### Making Knockout globally available, enable debugging

To enable debugging of Knockout in Chrome when using Webpack, the “ko” variable needs to be made available as a global object. We do this using the “expose-loader” (<https://github.com/webpack/expose-loader>) . This loader is referenced in package.json. It is used in Main.ts, the starting point in the project:  
require("expose?ko!knockout");

After globally exposing the “ko” variable in this way, we don’t need to reference it anymore in the various TypeScript files which use Knockout, i.e. it is not necessary anymore to place

import ko = require("knockout");

at the top of the TypeScript files.

## The FormSteps

Some basis plumbing for supporting multiple form steps and navigating through those form steps is provided.

The app consists of three form steps: product selection, entering personal details, and a check and submit step. The form steps are also implemented as Knockout components, inheriting from a FormStepBase component.