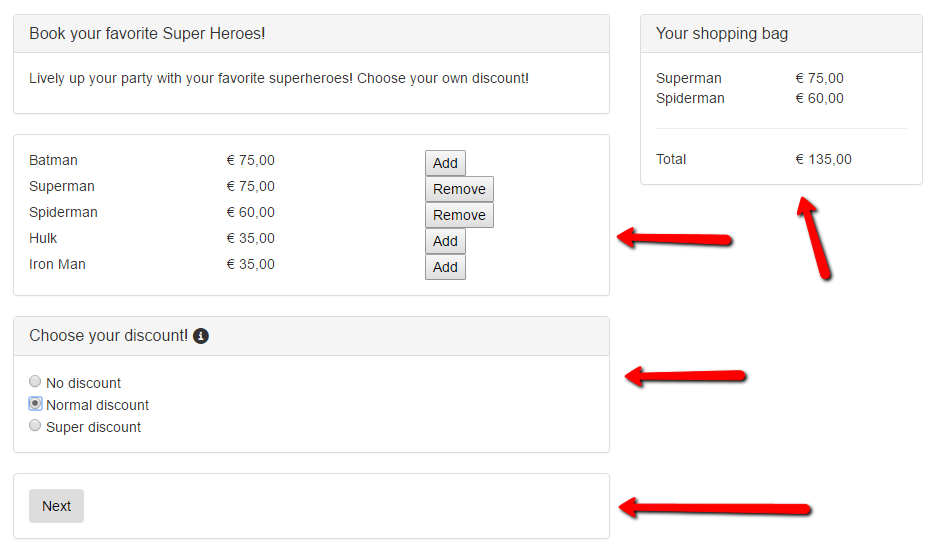
# An MVC-Knockout-TypeScript-Webpack Starter project

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 MVC-Knockout-TypeScript-Webpack starter 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 starter project aims at providing a structure for building up a single page application out of Knockout components. This includes a structure for setting up form steps, which are also Knockout components. The starter project includes a few form steps, as well some other components used in the various form steps, such as a list of superheroes, a shopping bag and a discount selector. The elements in the screenshot indicated by the red arrows are separate Knockout components.



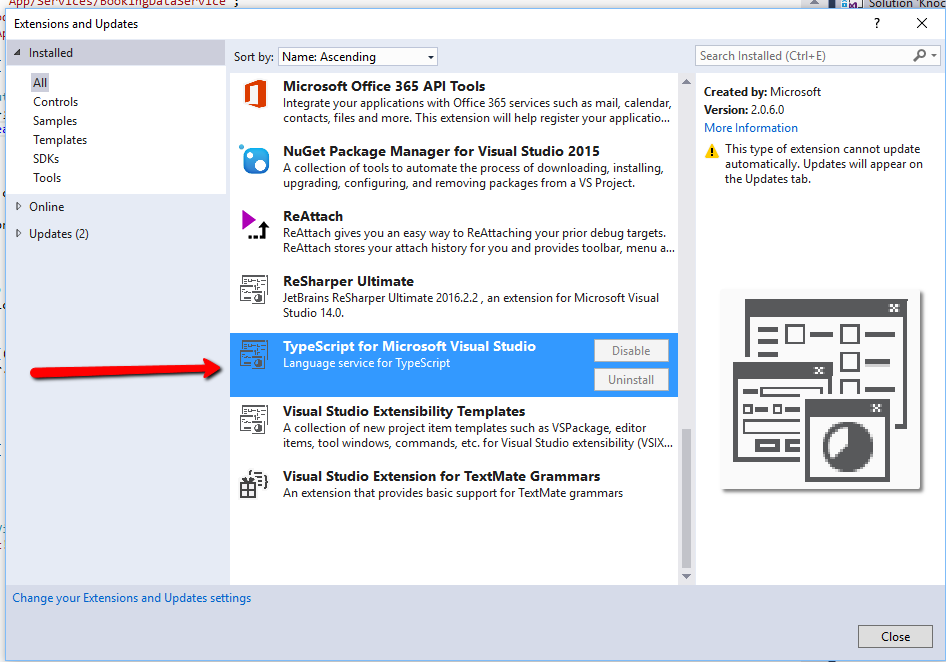
I have a backend programming background myself. After doing some work recently with Angular 2 and TypeScript, I was really enthusiastic about TypeScript, because of the similarities with programming in C#. So TypeScript seemed a logical choice for the project.

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. In this project Webpack is used for bundling and minification.

## Download the project from Github

You can download the project from Github at   
git@github.com:johnligt/Knockout-TypeScript-Webpack-Starter.git

## The Visual Studio solution

The project is set up as a Visual Studio solution, consisting of two projects, Web and Data. Web contains the web application as well as the single-page application. Base is a class library project containing some additional models and helper classes for data provision. Because we well be working with TypeScript make sure you have TypeScript installed, through Tools > Extensions and updates  
  


## Installation

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

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

But because we use a package.json file in which all dependencies are configured, the installation of Node.js is probably the only thing which you need to do. So, the first step is to install Node if you haven’t got Node on your machine already.  
<https://nodejs.org/en/download/>

To install the various dependencies which are configured in package.json, open a command window in admin mode, in the root of the web project, and run:  
  
npm install

## TypeScript

The project includes a TypeScript configuration file. This file overrides the TypeScript build setting in your project properties.

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

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.

## Webpack

Webpack is also listed as a dependency in package.json, so it should be installed by running   
  
npm install

For more information on Webpack see:

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

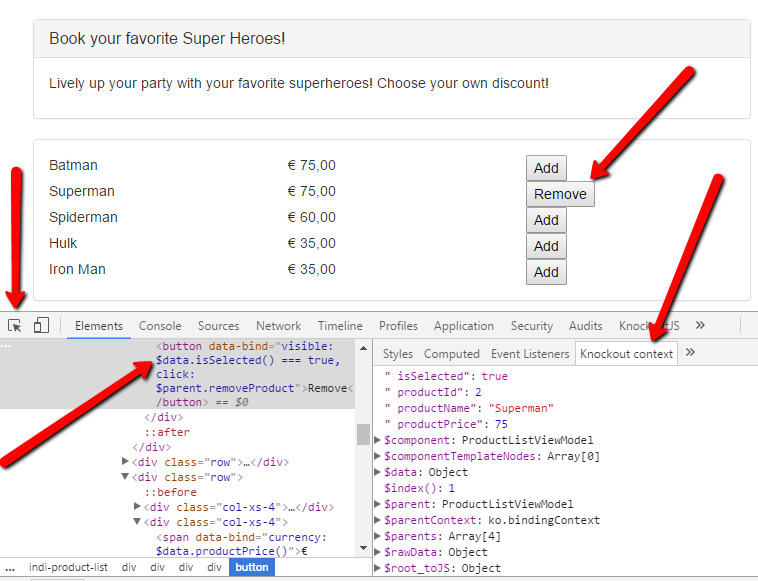
A file called webpack.config.js is included in the root directory of the web project.

## Debugging Knockout in Chrome

To enable debugging of Knockout in Chrome, install the Knockoutjs Context Debugger, available through:

<https://chrome.google.com/webstore/detail/knockoutjs-context-debugg/oddcpmchholgcjgjdnfjmildmlielhof>

This tool enables you to select an element in the page through the Chrome developer tools, and view the properties and values of the related Knockout elements, if any.



## Making Knockout globally available to enable debugging

To enable debugging with the Knockoutjs Context Debugger 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

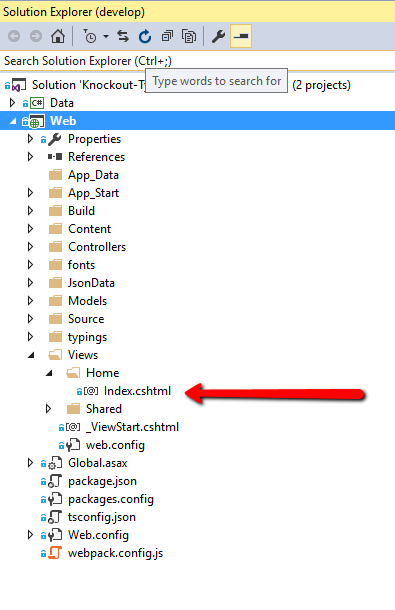
import ko = require("knockout");

at the top of the TypeScript files.

## The host MVC view for the single page application

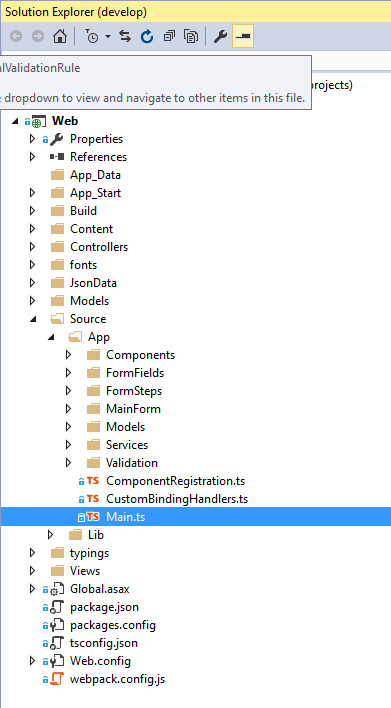
The original project from which this starter project is derived is an ASP.NET MVC application with a commercial CMS system. In this project, we’ve kept things simple, we just have one MVC controller and view to host the single page application. But in your real-world application this could be any kind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

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



## Structure of 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”.



To generate the JavaScript files which the project uses, we need to run Webpack. We do this by opening a command window in the root of the web project, and typing:

webpack --watch

This will generate the file app.bundle.js in the Build folder. (If you want to stop the watch process, type Ctrl-C in the command window.)

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

## Webpack configuration

Configuration of Webpack is done through the webpack.config.js file in the root of the Web project.

One of the things configured is the entry point of the application. In our case 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

entry: {

app: "./App/Main.ts"

}

This path is relative to the context, which is set through the context key:

context: path.resolve("./Source/")

## KnockoutJS

The starter application is built up out of Knockout components, supported by some additional service files. 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. This is the only component which is directly referred to in the Razor view “~/Views/Home/Index.cshtml”. This component hosts all other components, directly or indirectly, as components may also host other components. 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. Because the form steps are also components, they are also registered in this file.

For instance, the MainForm component is registered like this:

ko.components.register("indi-main-form", new MainFormComponent());

In the Razor view “~/Views/Home/Index.cshtml” we refer to this component like so:  
  
<indi-main-form></indi-main-form>

In the TypeScript object “MainFormComponent” the component is instantiated with the model and view of the component:  
  
export class MainFormComponent {

constructor() {

return {

viewModel: MainFormViewModel,

template: require("text!./MainFormView.html")

}

}

}

This structure is used for all Knockout components in the project.

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

The following things will have to happen in the Main.ts file:

1. Load the file in which the Knockout components are registered.
2. Load the data needed for the application (heroes, prices).
3. Bind the BookingData Knockout viewmodel.
4. Initialization of Knockout validation.

## Form steps

The starter application provides basic plumbing for multiple form steps and navigation through those form steps.

The app consists of four form steps: product selection, entering personal details, a “check and submit” step, and a “thank you” step. The form steps are also implemented as Knockout components, inheriting from a FormStepBase component. Remove them and add your own form steps as needed.

Adding a form step requires the following steps:

1. Add a subfolder to the /Source/App/FormSteps/ folder with the name of your form step.
2. Add the viewmodel and template files.
3. Register the form step component in ComponentRegistration.ts
4. Add your form step to FormStepEnum
5. Add your form step to MainFormView.html, as a nested component, and assign an order value to the form step.

MainFormView.html in the starter project looks like this:

<div class="row">

<div class="col-md-8">

<indi-formstep-selectproducts params="order: 0"></indi-formstep-selectproducts>

<indi-formstep-personaldetails params="order: 1"></indi-formstep-personaldetails>

<indi-formstep-check params="order: 2"></indi-formstep-check>

<indi-formstep-thanks params="order: 3"></indi-formstep-thanks>

<indi-navigation></indi-navigation>

</div>

<div class="col-md-4">

<indi-shopping-bag></indi-shopping-bag>

</div>

</div>

Every form step viewmodel has two properties with which the visibility of the form step can be manipulated, active and visible. These properties are inherited from FormStepBase.

this.visible = ko.observable(false);

this.active = ko.observable(true);

To disable a form step due to some business rule, you can set active to false, and the form step will never be shown.

To toggle the visibility as the user steps through the form, the “visible” property is set to true or false. This is taken care of in Navigation.ts and FormStepsManager.ts

## Validation

For validation, the Knockout-Validation library is used. More information on this library can be found at <https://github.com/Knockout-Contrib/Knockout-Validation> or <https://www.pluralsight.com/courses/knockout-validation-library>

Because validation usually happens when moving to the next form step, a validation model needs to be set up for each form step. This model is instantiated in FormStepBase. Every nested component which has to be validated needs a reference to this validation model, so form field components such as the text field and textarea components (in the folder \Source\App\FormFields\FieldBase.ts) inherit from a base class FieldBase, which sets up the reference to the validation model of the current form step.