[Skip to main content](https://keepinguptodate.com/pages/2019/12/using-typescript-with-knockout/)

[](https://keepinguptodate.com/)

**Latest discoveries and ponderings of a web focused software engineer**



**Using TypeScript With Knockout**

DECEMBER 14, 2019  |  TYPESCRIPT, KNOCKOUT

Knockout is a minimalist, mature and proven library for creating web applications. It isn’t as feature rich as some of the more modern libraries & frameworks but it does what it does well, primarily being binding HTML elements against a data model. Even in 2019, I believe Knockout has its place and is still used on some very large projects, including Microsoft’s Azure Portal.

As of v3.5.0 (released Feb 2019), Knockout has built-in TypeScript definitions. <https://github.com/knockout/knockout/blob/master/build/types/knockout.d.ts>

TypeScript provides a huge amount of benefits over JavaScript including type safety, improved IDE tooling and additional constructs like interfaces and enums. If you have an existing project that uses Knockout, you can move over to TypeScript now and start enjoying these benefits.

This article will start with covering using TypeScript and Knockout’s type definitions to create view models, components and bindings that are strongly typed. We’ll then cover creating a new project from scratch that uses webpack and Babel to transpile TypeScript down to JavaScript (ES5).

The code for this article is available on the GitHub repo: <https://github.com/JonUK/knockout-typescript>

**Using strong types with Knockout**

We’re going to create a Person class that will be used by a read-only and an editable component to display & allow editing of a person’s details.

Create the file src/models/person.ts.

import \* as ko from 'knockout';   
import {  
 Observable, ObservableArray, PureComputed  
} from 'knockout';   
   
class Person {   
 firstName: Observable<string>;   
 lastName: Observable<string>;   
 favouriteFilms: ObservableArray<string>;   
   
 fullName: PureComputed<string> = ko.pureComputed(  
 () => this.firstName() + ' ' + this.lastName());   
   
 constructor(firstName: string, lastName: string, favouriteFilms: string[] | null) {   
 this.firstName = ko.observable(firstName);   
 this.lastName = ko.observable(lastName);   
 this.favouriteFilms = ko.observableArray(favouriteFilms || []);   
 }   
}   
   
export default Person;

The types Observable, ObservableArray and Purecomputed have all been pulled in from knockout. First name is an Observable of type string so if you try to set the firstName observable to a null, a boolean, a number etc, you’ll receive a TypeScript error. Similarly, the PureComputed must always return a string and favouriteFilms can only contain an array of string.

Let’s now create the read-only component to show the person’s details. Create the file src/components/PersonReadOnly.ts.

import Person from '../models/person';   
import template from './PersonReadOnly.html';   
   
class PersonReadOnly {   
 person: Person;   
   
 constructor(params: { person: Person }) {   
 this.person = params.person;   
 }   
}   
   
// The default export returns the component details object to register with KO   
export default { viewModel: PersonReadOnly, template: template };

The template (which we’ll create next) is imported which will result in a HTML string being included in the webpack bundle.

Create the template src/components/PersonReadOnly.html for the read-only component.

<div>   
   
 <h2 class="h4 mt-4 mb-3">PersonReadOnly Component</h2>   
   
 <table class="table">   
 <thead>   
 <tr>   
 <th scope="col">Item</th>   
 <th scope="col">Value</th>   
 </tr>   
 </thead>   
 <tbody>   
 <tr>   
 <th scope="row">First name</th>   
 <td data-bind="text: person.firstName"></td>   
 </tr>   
 <tr>   
 <th scope="row">Last name</th>   
 <td data-bind="text: person.lastName"></td>   
 </tr>   
 <tr>   
 <th scope="row">Full name</th>   
 <td data-bind="text: person.fullName"></td>   
 </tr>   
 <tr>   
 <th scope="row">Favourite films</th>   
 <td data-bind="films: person.favouriteFilms"></td>   
 </tr>   
 </tbody>   
 </table>   
   
</div>

The first name observable, last name observable and full name computed are all bound using the built-in text binding. The favourite films observable array is however bound with a custom films binding which we’ll create next.

Create the file src/bindings/filmsBinding.ts.

import \* as ko from 'knockout';  
import { BindingHandler, ObservableArray } from 'knockout';  
  
const filmsBinding = {  
 init: (element: HTMLElement, valueAccessor: () => ObservableArray): void => {  
 const valueUnwrapped = ko.unwrap(valueAccessor());  
 const isPopulatedArray = Array.isArray(valueUnwrapped) && valueUnwrapped.length > 0;  
 const text = isPopulatedArray ? valueUnwrapped.join(', ') : 'Unknown';  
   
 element.textContent = text;  
 }  
} as BindingHandler;  
  
export default filmsBinding;

Now we’ll create the editable component. Create the file src/components/PersonEditable.ts.

import Person from '../models/person';   
import template from './PersonEditable.html';   
   
class PersonReadOnly {   
 person: Person;   
   
 constructor(params: { person: Person }) {   
 this.person = params.person;   
 }   
}   
   
// The default export returns the component details object to register with KO   
export default { viewModel: PersonReadOnly, template: template };

And create the editable component template src/components/PersonEditable.html.

<div>   
   
 <h2 class="h4 mt-4 mb-3">PersonEditable Component</h2>   
   
 <form>   
 <div class="form-group">   
 <label for="firstName">First name</label>   
 <input data-bind="textInput: person.firstName"  
 type="text" id="firstName" class="form-control">   
 </div>   
   
 <div class="form-group">   
 <label for="lastName">Last name</label>   
 <input data-bind="textInput: person.lastName"  
 type="text" id="lastName" class="form-control">   
 </div>   
   
 </form>   
   
</div>

We can now reference and use these components in index.ts.

import \* as ko from 'knockout';   
   
import Person from "./models/person";   
import PersonReadOnly from './components/PersonReadOnly';   
import PersonEditable from './components/PersonEditable';   
   
import filmsBinding from './bindings/filmsBinding';   
   
ko.components.register('person-read-only', PersonReadOnly);   
ko.components.register('person-editable', PersonEditable);   
   
ko.bindingHandlers.films = filmsBinding;   
   
class AppViewModel {   
 person: Person;   
   
 constructor() {   
   
 // These values are hard-coded but could come from a server API request with JSON response   
 this.person = new Person(   
 'Jon',   
 'Keeping',   
 ['The Matrix', 'The Shawshank Redemption', 'Upgrade']   
 )   
 }   
}   
   
   
ko.applyBindings(  
 new AppViewModel(),  
 document.getElementById('app'));

Here we’re importing and then registering the components and the custom binding. We’re also creating a instance of the Person class on the person property of AppViewModel. All that’s left to do now is update the webpack template to use these 2 components.

Update the <div id="app"> element in webpack-template/index.html to contain:

<div id="app">   
   
 <div class="row">   
 <div class="col-sm-12 col-lg-7">   
 <person-read-only params="person: person"></person-read-only>   
 </div>   
 <div class="col-sm-12 col-lg-5 pl-lg-5">   
 <person-editable params="person: person"></person-editable>   
 </div>   
 </div>   
   
</div>

Now when you run npm run serve and reload the browser you’ll see these strongly typed Knockout components in action.

The code for this article is available on the GitHub repo: <https://github.com/JonUK/knockout-typescript>

**Using Knockout Validation with types**

The package [Knockout Validation](https://github.com/Knockout-Contrib/Knockout-Validation) is very popular with Knockout but unfortunately hasn’t been updated with the latest type definitions.

[A GitHub pull request has been created to provide the latest types](https://github.com/Knockout-Contrib/Knockout-Validation/pull/670) but at this time (April 2021), this pull request has yet to be merged.

For us, this means we’ll need to add a TypeScript type definition file to our project. Create the file src/knockout.validation.d.ts and copy the content from the [GitHub pull request file knockout.validation.d.ts](https://github.com/Knockout-Contrib/Knockout-Validation/blob/ac04d921a2e76d237e3e244a799f4248d9477808/dist/knockout.validation.d.ts)

With the TypeScript type definitions in place, we can install the actual package.

npm install --save knockout.validation

From our main entry point src/main.ts, include this package and configure it as required.

import \* as ko from 'knockout';  
require('knockout.validation');  
  
...  
  
ko.validation.init({  
 errorElementClass: 'is-invalid',  
 errorMessageClass: 'invalid-feedback',  
 decorateInputElement: true  
});  
  
...

Within our components, we can now apply validation rules.

name: Observable<string> = ko.observable('Bananas')  
 .extend({ required: true });  
  
 productCode: Observable<string> = ko.observable()  
 .extend({  
 required: true,  
 minLength: 5,  
 pattern: {  
 message: 'Please enter letters and digits only',  
 params: /^[A-Za-z0-9]\*$/  
 }  
 })  
  
 stockCount: Observable<number> = ko.observable()  
 .extend({  
 min: 1,  
 max: 100  
 })

**Creating a new Knockout project with TypeScript support**

Fire up your terminal / command prompt and let’s get stuck in!

Create a new folder for our project.

mkdir knockout-typescript  
cd knockout-typescript

From within the new folder, initialize a new package.json.

npm init -y

Now we have our newly created package.json we can install Knockout and TypeScript.

npm install --save knockout  
npm install --save-dev typescript

TypeScript is installed with the --save-dev argument to save it as a development dependency as it’s not needed in production.

Our project is going to use webpack to:

* Handle transpiling TypeScript to Javascript and bundling into a single file
* Generate a HTML page that includes a reference to the output bundle
* During development, provide a web server that supports live reloading

Let’s install webpack and the other related packages we’re going to use.

npm install --save-dev webpack webpack-cli webpack-dev-server html-webpack-plugin

We’ll use Babel (with webpack) to actually transpile our TypeScript to JavaScript which can then be run by web browsers. Let’s install Babel and other related packages.

npm install --save-dev @babel/core @babel/preset-env @babel/preset-typescript @babel/plugin-proposal-class-properties babel-loader

We’re going to use Knockout components with HTML templates and want these templates to be bundled up by webpack. The effect this will have is when the page loads, the component templates will be ready immediately without any additional calls to the server. We’ll use the webpack loader html-loader to include our HTML in the webpack bundle. npm install --save-dev html-loader

We now need to add the webpack configuration file webpack.config.js.

const path = require('path');   
const HtmlWebpackPlugin = require('html-webpack-plugin');   
   
module.exports = {   
 mode: 'development',   
 entry: {   
 app: './src/index.ts'   
 },   
 devtool: 'source-map', // Generate separate source map files   
 devServer: {   
 contentBase: './dist',   
 overlay: true // Show errors in overlay on the website   
 },   
 module: {   
 rules: [   
 {   
 test: /\.(js|ts)$/,   
 exclude: /node\_modules/,   
 loader: 'babel-loader'   
 },   
 {   
 test: /\.html$/, // All Knockout.js component HTML templates   
 use: 'html-loader' // Adds the component templates to the bundle   
 }   
 ]   
 },   
 plugins: [   
 new HtmlWebpackPlugin({   
 template: './webpack-template/index.html'   
 }),   
 ],   
 resolve: {   
 extensions: ['.js', '.ts']   
 }   
};

This configuration file instructs webpack to use babel-loader to transpile TypeScript (and ES6+) files down to a JavaScript (ES5) bundle which is output to the dist folder. HTML files are also included in the bundle by html-loader. The app entry file src/index.ts and the webpack template HTML file webpack-template/index.html used to generate dist/index.html don’t exist yet so lets create these now.

Create the file src/index.ts.

alert('An alert from index.ts');

This is just some test code which will allow us to check everything is working correctly. Later we’ll add some real code here.

Create the file webpack-template/index.html.

<!DOCTYPE html>   
<html lang="en">   
<head>   
 <meta charset="utf-8">   
 <meta name="viewport" content="width=device-width, initial-scale=1" />   
 <title>Knockout TypeScript</title>   
   
 <link   
 rel="stylesheet"   
 href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css"   
 integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm"   
 crossorigin="anonymous">   
   
</head>   
<body>   
   
<main class="container">   
 <h1 class="mt-4">Using TypeScript With Knockout</h1>   
   
 <!-- The element Knockout will bind to -->  
 <div id="app">  
 </div>  
</main>   
   
</body>   
</html>

As well as the configuration file for webpack, we also need to add configuration files for Babel TypeScript.

Create the file Babel configuration file .babelrc.

{   
 "presets": [   
 "@babel/preset-env",   
 "@babel/typescript"   
 ],   
 "plugins": [   
 "@babel/proposal-class-properties"   
 ]   
}

Create the TypeScript configuration file tsconfig.json

{   
 "compilerOptions": {   
 /\* Basic Options \*/   
 "target": "esnext",   
 "module": "commonjs",  
 "allowJs": true,   
 "noEmit": true,  
 "isolatedModules": true,  
   
 /\* Strict Type-Checking Options \*/  
 "strict": true,  
   
 /\* Module Resolution Options \*/  
 "moduleResolution": "node",  
 "baseUrl": "./src",  
   
 /\* Advanced Options \*/   
 "forceConsistentCasingInFileNames": true  
 },   
 "include": ["src"]   
}

We’re almost ready to test our page out by running the webpack dev server. Just before we do that though, let’s update the scripts section in package.json to include a new entries for starting the webpack dev server and for performing a production build.

{  
 ...  
 "scripts": {   
 "serve": "webpack-dev-server --open",   
 "build": "webpack --mode production",   
 },  
 ...  
}

Good work! Now you can run npm run serve to start dev server and load the site. You should see the JavaScript alert that we added in index.ts.

The code for this article is available on the GitHub repo: <https://github.com/JonUK/knockout-typescript>

**Contents**

1. [Using strong types with Knockout](https://keepinguptodate.com/pages/2019/12/using-typescript-with-knockout/#using-strong-types-with-knockout)
2. [Using Knockout Validation with types](https://keepinguptodate.com/pages/2019/12/using-typescript-with-knockout/#using-knockout-validation-with-types)
3. [Creating a new Knockout project with TypeScript support](https://keepinguptodate.com/pages/2019/12/using-typescript-with-knockout/#creating-a-new-knockout-project-with-typescript-support)

**Recent Articles**

[Creating a Google Lighthouse custom audit that checks DOM elements](https://keepinguptodate.com/pages/2021/08/custom-lighthouse-audit/)

AUGUST 30, 2021  |  LIGHTHOUSE, CUSTOM AUDIT

[Automating Google Lighthouse audits and uploading results to Azure](https://keepinguptodate.com/pages/2021/07/automating-google-lighthouse-upload-to-azure/)

JULY 26, 2021  |  LIGHTHOUSE, DOCKER, AZURE

[Get Code Metrics On Your JavaScript Project Using SonarQube](https://keepinguptodate.com/pages/2021/01/code-metrics-on-your-javascript-project/)

JANUARY 30, 2021  |  JAVASCRIPT, CODE METRICS, SONARQUBE

[Creating A .NET 5 ASP.NET Core App Running In Docker On A Mac](https://keepinguptodate.com/pages/2020/12/net5-aspnet-docker-mac/)

DECEMBER 19, 2020  |  .NET 5, ASP.NET CORE, DOCKER, MAC

[BEM Class Names Checker](https://keepinguptodate.com/pages/2020/05/bem-class-names-checker/)

MAY 31, 2020  |  BEM, CSS

**About**



**Jon Keeping**

Software Engineer. Google Certified Mobile Web Specialist.

© Keeping Up To Date



**This site uses cookies**

Cookies are used to provide comments functionality, videos and to measure site performance.

OK