Create the Angular 8 app

Now, let's finally start with the Angular app. We will use Node.js and Angular CLI to generate the Angular project and necessary files.

Prerequisites

- Node.js
- Angular CLI
- VS Code

When Node.js is installed, you can open the Node.js command prompt.

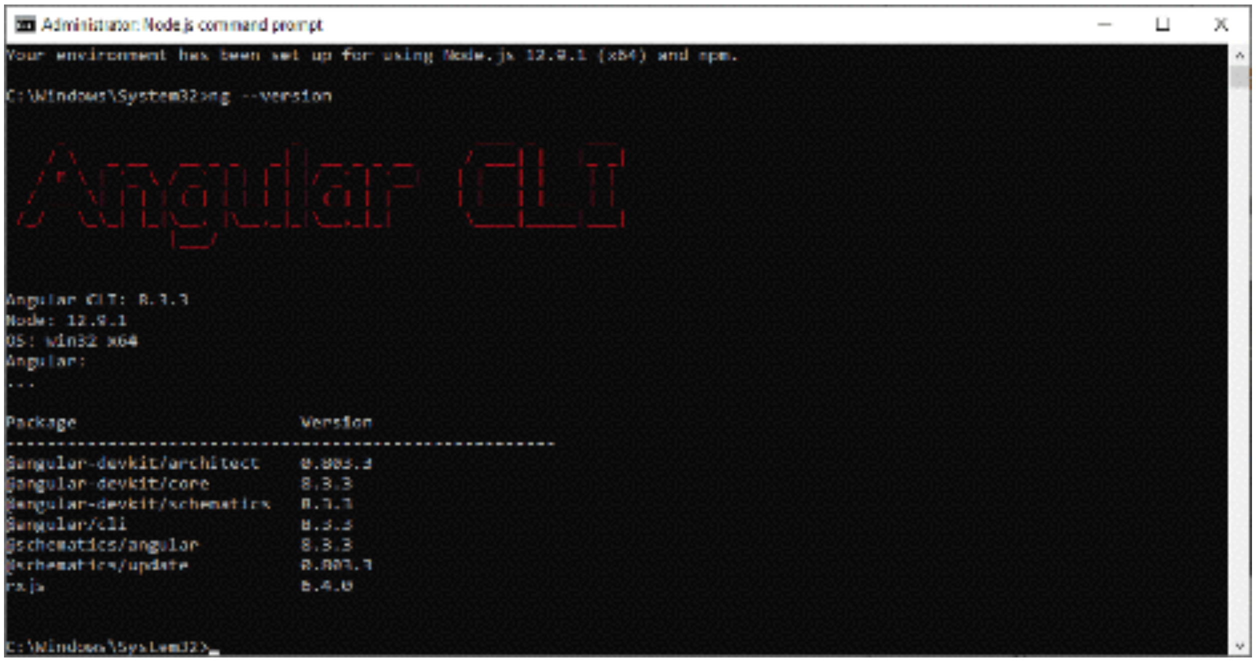
Execute this command in the command prompt to install the Angular 8 CLI:

```
npm install -g @angular/cli
```

This will install the latest Angular CLI globally and it'll take a little while. When you're done, you can check the Angular version with this command:

```
ng --version
```

The Node.js command prompt should look something like this:



```
Your environment has been set up for using Node.js 12.9.1 (x64) and npm.

C:\Windows\System32>ng --version

Angular CLI: 8.3.3
Node: 12.9.1
OS: win32 x64
Angular:
...

Package                      Version
------------------------------------------------------
@angular-devkit/architect    0.803.3
@angular-devkit/core         8.3.3
@angular-devkit/schematics   8.3.3
@angular/cli                 8.3.3
@schematics/angular          8.3.3
@schematics/update           0.803.3
rxjs                         6.4.0

C:\Windows\System32>_
```

Now, let's move to the folder where our Visual Studio back-end is located. Use the cd command to do this:

```
cd c:/projects/blog
```

We will simply call our Angular 8 application **ClientApp**. Let’s execute the command that creates our Angular application:

```
ng new ClientApp
```

We’ll be prompted with some questions. We want to use routing (press Y) and stylesheet format: SCSS. Then let Node do its thing and create the web app. It’ll take a minute or so.

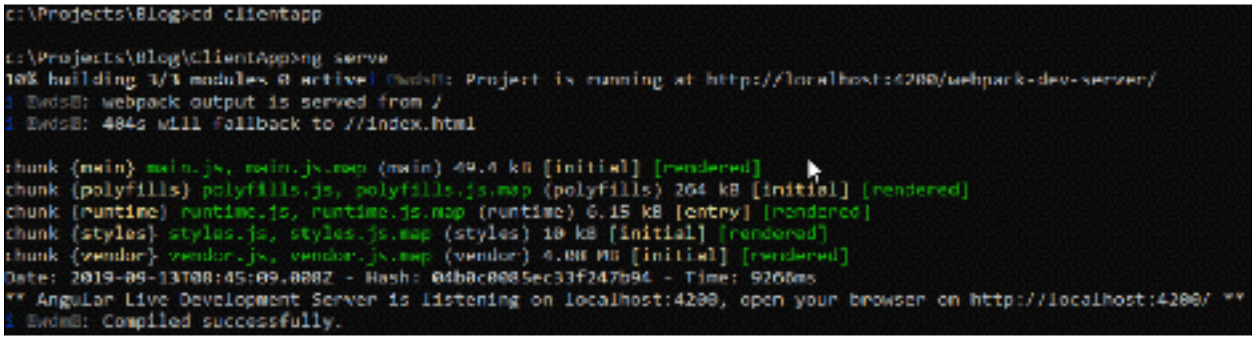
When the app is created, cd command into the app folder:

```
cd ClientApp
```

And then build and run the app with the `ng serve` command:

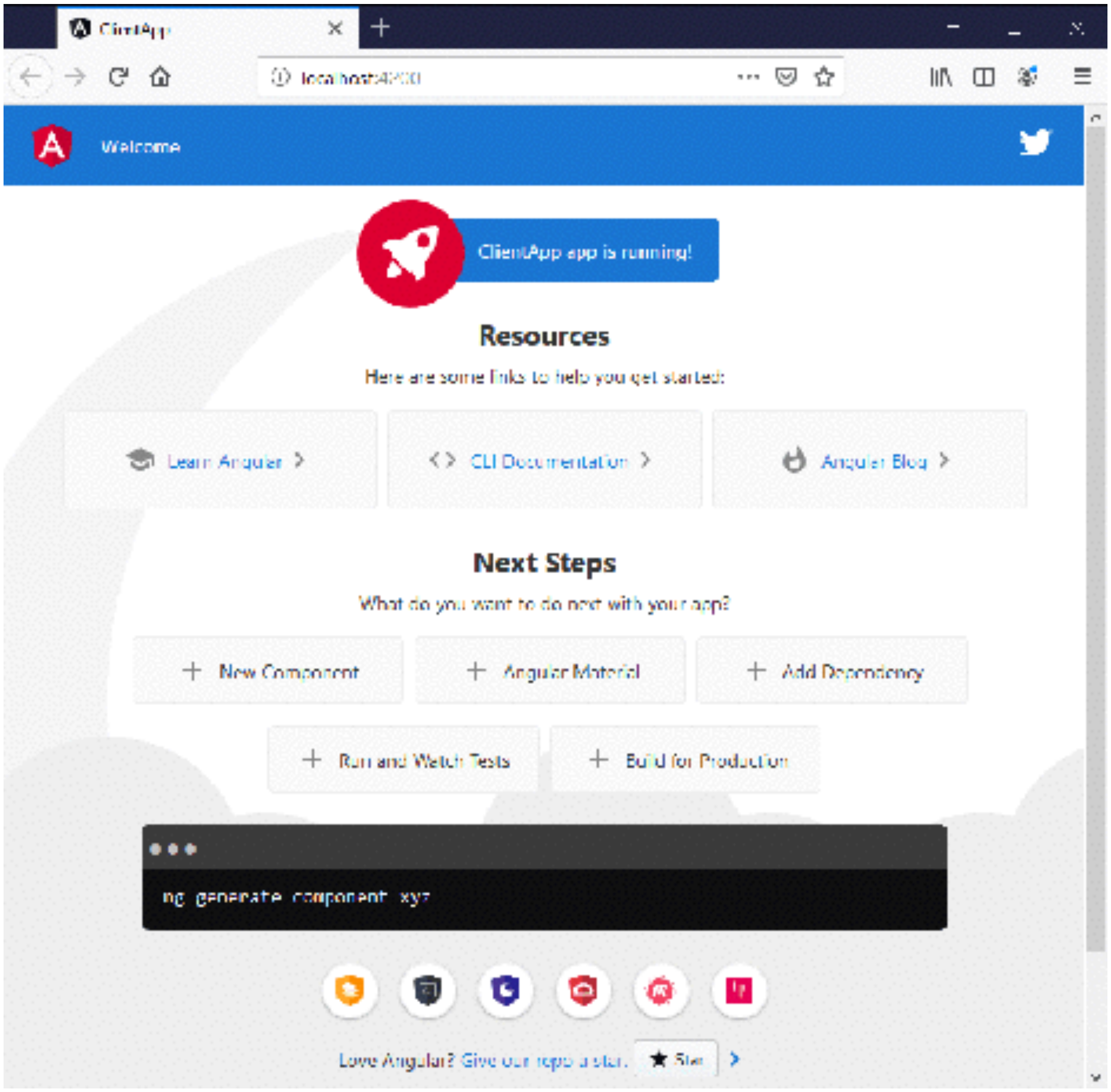
```
ng serve
```

The command prompt will look like this:



```
c:\Projects\Blog>cd clientapp

c:\Projects\Blog\ClientApp>ng serve
10% building 3/3 modules 0 active｢wds｣: Project is running at http://localhost:4200/webpack-dev-server/
｢wds｣: webpack output is served from /
｢wds｣: 404s will fallback to //index.html

chunk {main} main.js, main.js.map (main) 49.4 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 264 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.15 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 10 kB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 4.08 MB [initial] [rendered]
Date: 2019-09-13T08:45:09.080Z - Hash: 04b0c0085ec33f247b94 - Time: 9266ms
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **
｢wdm｣: Compiled successfully.
```

The build succeded and now you can browse to your Angular app with the URL **http://localhost:4200**. The basic Angular 8 app is based on a template and it’ll look something like this:



```
ng generate component xyz
```

If you have a look at the source code, it'll look like this:

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>ClientApp</title>
```

```
<base href="/">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
<app-root></app-root>
<script src="runtime.js" type="module"></script><script src="polyfills.js" type="module"></script><script src="styles.js" type="module"></script><script src="vendor.js" type="module"></script><script src="main.js" type="module"></script></body>
</html>
```

The interesting thing here is `<app-root></app-root>`, which is Angular specific and tells us where our Angular app will execute.

One final command is good to know — it's `Ctrl+C` to close the Angular application, and you should **press it twice** to terminate the batch job and stop ng serve.

One of the nice features of Angular development is **changes you save in front-end files will immediately be reflected in the browser**. For this to happen the app needs to be running.

For some changes though, like adding Bootstrap, you need to restart the application to make it work.

Angular 8 fundamentals

Let’s pause, take a step back and learn some of the Angular 8 fundamentals.

Angular 8 is an open source client-side JavaScript framework, based on TypeScript which is compiled to JavaScript.

The Angular 8 architecture consists of the following:

- Modules
- Components
- Templates, Directives, Data-Binding
- Services and dependency injection
- Routing

You can delve deeper into the Angular architecture here in the official documentation. Here’s a quick rundown though:

Modules

Angular NgModules are fundamental to any Angular application. Every Angular app has a root module called AppModule, which bootstraps and launches the app. Modules can call components and services. The default module is **app.module.ts**.

Components

Components provide us with a class and a view, and are parts of the application. The class is TypeScript based and the view is HTML. All Angular app has at least one component called **app.component.ts**.

Templates, directives, data-binding

A template combines HTML with Angular markup. The directives provide logic and the binding markup connects the application data with the DOM.

Services and dependency injection

Service classes provide application logic that isn't tied to a specific view and shared across the app. They are injectable using the @Injectable() decorator. Component classes are kept nice and tidy using dependency injection.

Routing

The Router NgModule provides a service that defines navigation in the app. It works the same way as a browser's navigation.

Visual Studio 2019 for back-end, VS Code for front-end

While Visual Studio 2019 works very well for back-end as well as front-end, VS Code is actually better for front-end heavy work with frameworks like Angular. I recommend you try VS Code and most instructions for the Angular application in this tutorial, will be for VS Code.

To make front-end and Angular development easier in VS Code, install these extensions. You can do it easiest through the VS Code Extensions module.

- Angular Snippets (Version 8)
- Debugger for Chrome
- TSLint

There's obviously a lot more awesome extensions like Beautify and Path Intellisense that makes your development more productive. It's all up to your preference and style.

In VS Code, make sure you open the folder ClientApp on your disk and work from there.

Add components and services to our Angular app

Let's continue building the Angular app. First, press Ctrl+C twice in the Node.js command prompt, if you haven't closed the connection to your app.

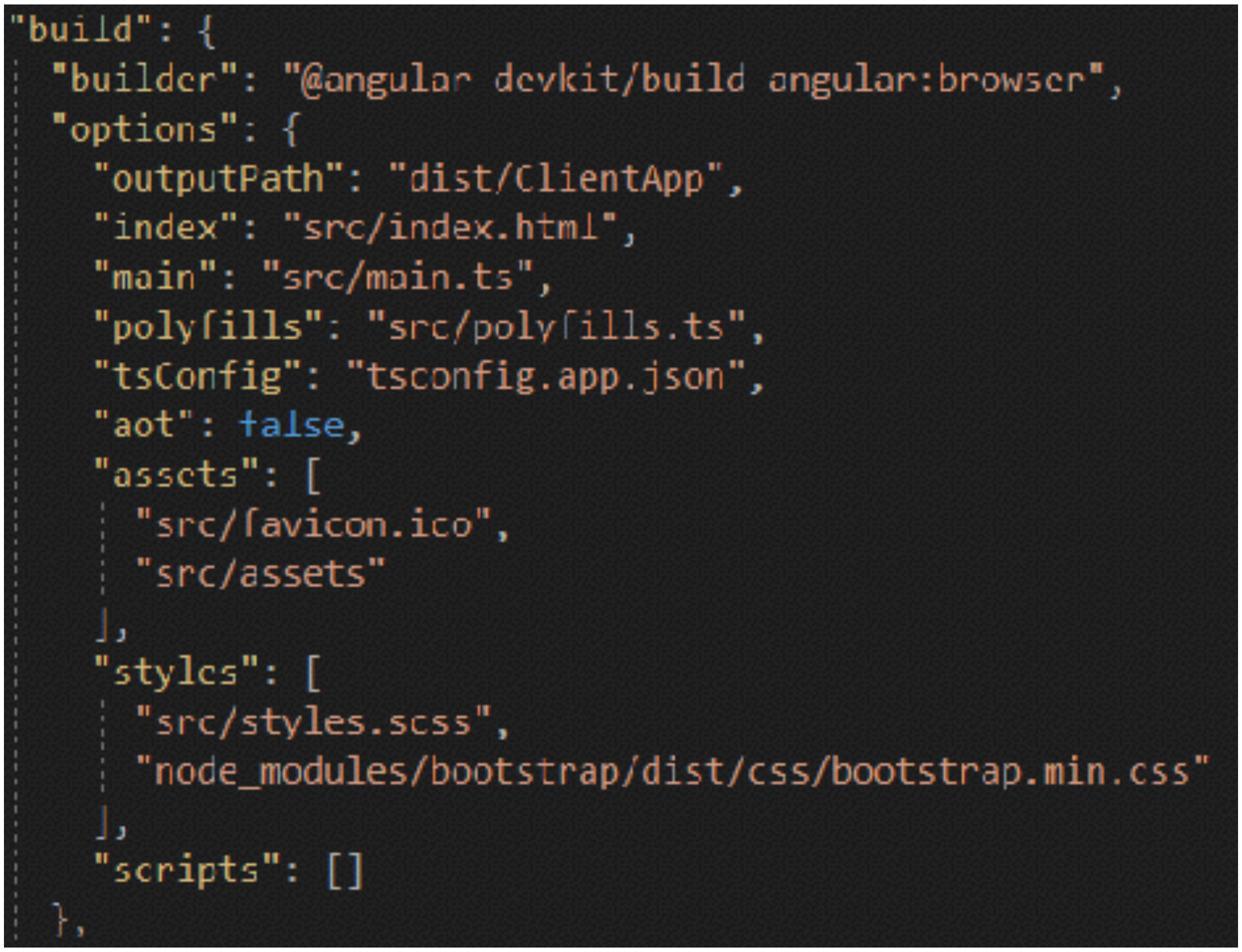
Next, let's add Bootstrap 4 to the application. Execute this command in the Node.js command prompt:

```
npm install bootstrap --save
```

Then find the **angular.json** file and edit the build node to make styles look like this:

```
"styles": [
"src/styles.scss",
"node_modules/bootstrap/dist/css/bootstrap.min.css"
]
```

view raw
angular.json hosted with ❤ by **GitHub**
The angular.json build node should look like this:



```
"build": {
  "builder": "@angular-devkit/build-angular:browser",
  "options": {
    "outputPath": "dist/ClientApp",
    "index": "src/index.html",
    "main": "src/main.ts",
    "polyfills": "src/polyfills.ts",
    "tsConfig": "tsconfig.app.json",
    "aot": false,
    "assets": [
      "src/favicon.ico",
      "src/assets"
    ],
    "styles": [
      "src/styles.scss",
      "node_modules/bootstrap/dist/css/bootstrap.min.css"
    ],
    "scripts": []
  },
```

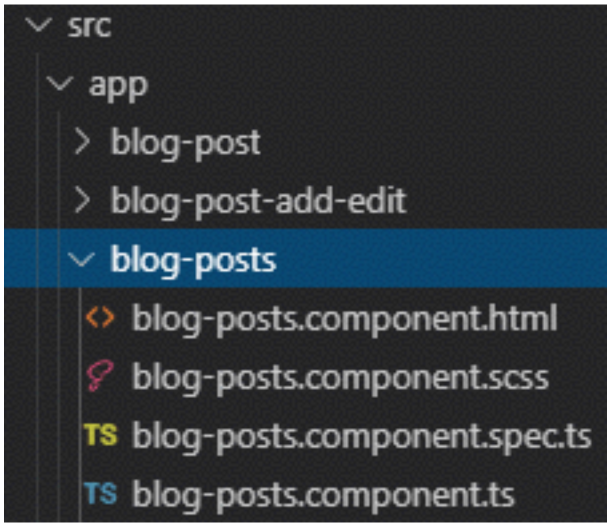
Next up, let’s create our **components**. We will have three components for our blog application:

- BlogPosts — shows all blog posts.
- BlogPost — show a specific blog post.
- BlogPostAddEdit — add new or edit existing blog post.

To create these components, execute the following commands in the Node.js command prompt:

```
ng generate component BlogPosts
ng generate component BlogPost
ng generate component BlogPost-AddEdit
```

Under ClientApp/src/app, the components are now there:



```
src
  app
    blog-post
    blog-post-add-edit
    blog-posts
      blog-posts.component.html
      blog-posts.component.scss
      blog-posts.component.spec.ts
      blog-posts.component.ts
```

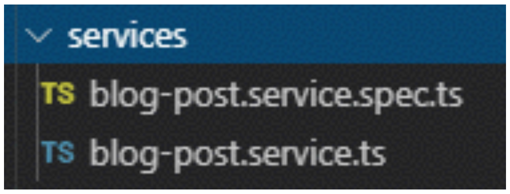
As you can see, we have a .html file, scss file, spec.ts file and component.ts file for each component.

- HTML and SCSS are used for the view.
- spec.ts is for tests.
- component.ts contains our component class and logic.

While we're at it, let's create our service as well, using the command prompt:

```
ng generate service BlogPost
```

Create a new folder under app and call it **services**. Move the two generated service files to the folder:



```
services
  blog-post.service.spec.ts
  blog-post.service.ts
```

Now let's leave our components and services and have a look at the **app.module.ts** file. This is where we import modules and components, declare them and also add providers.

We get a few things for free from the created app. Necessary imports are added and a few modules too. When we add components in the Node.js command prompt, the app.modules.ts file is updated as well. However we don't get help with everything.

For our blog app, we need to manually import some modules on our own and add them. We also need to import and add our service to providers.

Let's update the file to look like this:

```
"styles": [
"src/styles.scss",
"node_modules/bootstrap/dist/css/bootstrap.min.css"
]
```

Necessary modules like HttpClientModule and ReactiveFormsModule are imported. AppRoutingModule and AppComponent were already created for us from the beginning.

Just make sure to declare components, add modules to imports and also add our service to providers.

Just one thing on import and export.

TypeScript uses the module concept from EcmaScript 2015. Modules are executed in their own scope and not in the global scope. To make one module's classes, variables, functions etc visible to other modules, export is used. Also to use some of these from another module, an import is needed.

Setup routing

Open **app-routing.module.ts**. Here you have the routes setup with no routes configured:

```
const routes: Routes = [];
```

Update the file to look like this instead:

```
<ng-container *ngIf="(blogPost$ | async) as blogPost; else loading">

<h1>{{ blogPost.title }}</h1>
<div>{{ blogPost.body }}</div>
<ul>
<li>{{ blogPost.creator }}</li>
<li>{{ blogPost.dt }}</li>
</ul>

</ng-container>
<ng-template #loading>Loading…</ng-template>
```

We import necessary component and update Routes with paths and tell what components will be loaded in those paths.

```
{ path: '', component: BlogPostsComponent, pathMatch: 'full' }
```

This tells us we will load the BlogPostsComponent on the app start page.

```
{ path: '**', redirectTo: '/' }
```

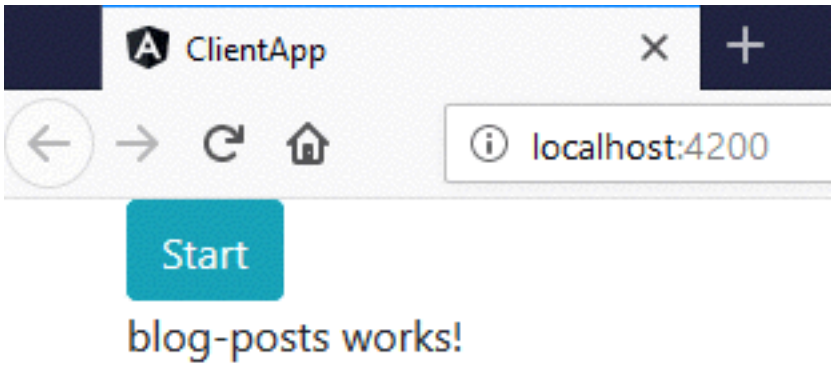
This tells us all invalid paths for the application will be redirected to the start page.

Open **app.component.html** and update the file to look like this:

```
<div class="container">

<a [routerLink]="['/']" class="btn btn-info">Start</a>
<router-outlet></router-outlet>

</div>
```

The <router-outlet></router-outlet> element will be replaced by the correct component and this file will be used for all components in the app.

Now let's build and run the app again using the ng serve command in the Node.js command prompt. When Node is done compiling, go to http://localhost:4200. The start page will now look like this:



This is our BlogPostsComponent in action. Try browsing to http://localhost:4200/add as well and you'll get the view for our BlogPostAddEditComponent.

If you try to browse to a path that doesn't exist, you're redirected to the start page again.

Different ways to build and run the application

We have two different ways that we can use to build and run our Angular application:

- Node.js command prompt and ng serve.
- Visual Studio F5 command and IIS Express.

This is good to know. The simplest thing to do is to just use Visual Studio to build and run our Angular app as well as the back-end. To make the Angular app work, we need to edit **Startup.cs** to allow SPA static files.

In Startup.cs, we already have commented out configuration for SPA. In the ConfigureServices method, uncomment the services.AddSpaStaticFiles section:

```
services.AddSpaStaticFiles(configuration =>
{
configuration.RootPath = "ClientApp/dist";
});
```

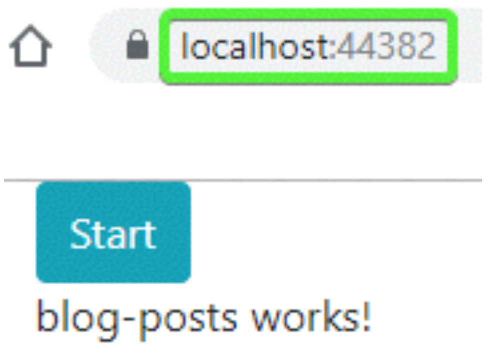
In the Configure method, uncomment the app.UseSpaStaticFiles() line and app.UseSpa() section. Since before, we already have app.UseMvc():

```
app.UseSpaStaticFiles();

app.UseMvc(routes =>
{
routes.MapRoute(
name: "default",
template: "{controller}/{action=Index}/{id?}");
});

app.UseSpa(spa =>
{
// To learn more about options for serving an Angular SPA from ASP.NET Core,
// see https://go.microsoft.com/fwlink/?linkid=864501

spa.Options.SourcePath = "ClientApp";

if (env.IsDevelopment())
{
spa.UseAngularCliServer(npmScript: "start");
}
});
```

Also, let's update **environment.ts**. Add appUrl to the environment constant, it should look like this:

```
export const environment = {
production: false,
appUrl: 'https://localhost:44382/'
};
```

Now in Visual Studio 2019, press F5 and your Angular app AND back-end will be up and running on the same address, on IIS Express:



blog-posts works!

Whether you want to use the Node.js command prompt to build and run the Angular application is up to you. Just remember the back-end needs to be up and running too.

Visual Studio building and running both the front-end and back-end means one less thing to think about.

Create blog post model and service methods

We need a blog post model to work with in TypeScript. Let's create a new folder called models and then a TypeScript file(right click the folder -> New file in VS Code) and name it **blogpost.ts**.

Copy and paste this BlogPost model class into blogposts.ts:

```
export class BlogPost {
postId?: number;
creator: string;
title: string;
body: string;
dt: Date;
}
```

Our BlogPost model will now be available across the application.

Angular 8 service CRUD tasks

Our Angular service will call our back-end and carry out these tasks:

- Create blog post.
- Show all blog posts / show a single blog post.
- Update an existing blog post.
- Delete a blog post.

Now let's go back to our previously created service, located in the services folder. Open **blog-post.service.ts** and edit the file to look like this:

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/
http';
import { Observable, throwError } from 'rxjs';
import { retry, catchError } from 'rxjs/operators';
import { environment } from 'src/environments/environment';
import { BlogPost } from '../models/blogpost';

@Injectable({
providedIn: 'root'
})
export class BlogPostService {
```

```
myAppUrl: string;
myApiUrl: string;
httpOptions = {
headers: new HttpHeaders({
'Content-Type': 'application/json; charset=utf-8'
})
};
constructor(private http: HttpClient) {
this.myAppUrl = environment.appUrl;
this.myApiUrl = 'api/BlogPosts/';
}

getBlogPosts(): Observable<BlogPost[]> {
return this.http.get<BlogPost[]>(this.myAppUrl +
this.myApiUrl)
.pipe(
retry(1),
catchError(this.errorHandler)
);
}

getBlogPost(postId: number): Observable<BlogPost> {
return this.http.get<BlogPost>(this.myAppUrl + this.myApiUrl
+ postId)
.pipe(
retry(1),
catchError(this.errorHandler)
);
}

saveBlogPost(blogPost): Observable<BlogPost> {
return this.http.post<BlogPost>(this.myAppUrl +
this.myApiUrl, JSON.stringify(blogPost), this.httpOptions)
.pipe(
retry(1),
catchError(this.errorHandler)
);
}

updateBlogPost(postId: number, blogPost):
Observable<BlogPost> {
return this.http.put<BlogPost>(this.myAppUrl + this.myApiUrl
+ postId, JSON.stringify(blogPost), this.httpOptions)
.pipe(
retry(1),
catchError(this.errorHandler)
);
}

deleteBlogPost(postId: number): Observable<BlogPost> {
```

```
return this.http.delete<BlogPost>(this.myAppUrl +
this.myApiUrl + postId)
.pipe(
retry(1),
catchError(this.errorHandler)
);
}

errorHandler(error) {
let errorMessage = '';
if (error.error instanceof ErrorEvent) {
// Get client-side error
errorMessage = error.error.message;
} else {
// Get server-side error
errorMessage = `Error Code: ${error.status}\nMessage: $
{error.message}`;
}
console.log(errorMessage);
return throwError(errorMessage);
}
}
```

We already injected the service into the providers array in app.module.ts, meaning the service can be used right away across the application.

Observables in Angular

The Angular HttpClient methods use **RxJS observables**. Observables provide support for passing messages between publishers and subscribers in your application. They are powerful and have several advantages and are therefore used extensively in Angular.

When we've created (published) an observable, we need to use the subscribe() method to receive notifications. We then get a Subscription object we can work with. Also, we can use unsubscribe() to stop receiving notifications.

We make our BlogPostService injectable via the @Injectable() decorator. We will inject the service into our components later.

For our service's post and put methods, we will send application/json.

Then we use the pipe() method for each service call. Here we can pass in operator functions for data transformation in our observable collection. We add retry and catchError to our pipe method.

It's very common to subscribe to observables in Angular. This is fine, but you have to remember to unsubscribe too. pipe does that automatically for you, freeing up memory resources and preventing leaks.

Update components to show service data

Over to our three blog components. Let’s start with BlogPostsComponent which will list all our blog posts. Update the file **blog-posts.component.ts** to look like this:

```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs';
import { BlogPostService } from '../services/blog-
post.service';
import { BlogPost } from '../models/blogpost';

@Component({
selector: 'app-blog-posts',
templateUrl: './blog-posts.component.html',
styleUrls: ['./blog-posts.component.scss']
})
export class BlogPostsComponent implements OnInit {
blogPosts$: Observable<BlogPost[]>;

constructor(private blogPostService: BlogPostService) {
}

ngOnInit() {
this.loadBlogPosts();
}

loadBlogPosts() {
this.blogPosts$ = this.blogPostService.getBlogPosts();
}

delete(postId) {
const ans = confirm('Do you want to delete blog post with id: ' + postId);
if (ans) {
this.blogPostService.deleteBlogPost(postId).subscribe((data) => {
this.loadBlogPosts();
});
}
}
}
```

We dependency inject BlogPostService in the constructor and in loadBlogPosts() we simply call our Angular service.

Since the service getBlogPosts() method gives us an Observable<BlogPost[]> back, we assign it to this component's blogPost$ object. It's common practice to name observable objects with a $ sign at the end.

In the delete() method we need to subscribe to our observable instead to execute the action and then reload the blog post list.

Now open **blog-posts.component.html** and update it to look like this:

```
<h1>Blog posts</h1>
```

```
<p *ngIf="!(blogPosts$ | async)"><em>Loading...</em></p>
<p>
<a [routerLink]="['/add']" class="btn btn-primary float-
right mb-3">New post</a>
</p>

<table class="table table-sm table-hover" *ngIf="(blogPosts$
| async)?.length>0">
<thead>
<tr>
<th>#</th>
<th>Title</th>
<th>Creator</th>
<th>Date</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr *ngFor="let blogPost of (blogPosts$ | async)">
<td>{{ blogPost.postId }}</td>
<td><a [routerLink]="['/blogpost/',
blogPost.postId]">{{ blogPost.title }}</a></td>
<td>{{ blogPost.creator }}</td>
<td>{{ blogPost.dt | date: "dd.MM.y" }}</td>
```

```
<td><a [routerLink]="['/blogpost/edit/', blogPost.postId]"
class="btn btn-primary btn-sm float-right">Edit</a></td>
<td><a [routerLink]="" (click)="delete(blogPost.postId)"
class="btn btn-danger btn-sm float-right">Delete</a></td>
</tr>
</tbody>
</table>
```

We use the AsyncPipe to subscribe to our observables. When we want to display our observable value in our HTML template file we use this syntax:

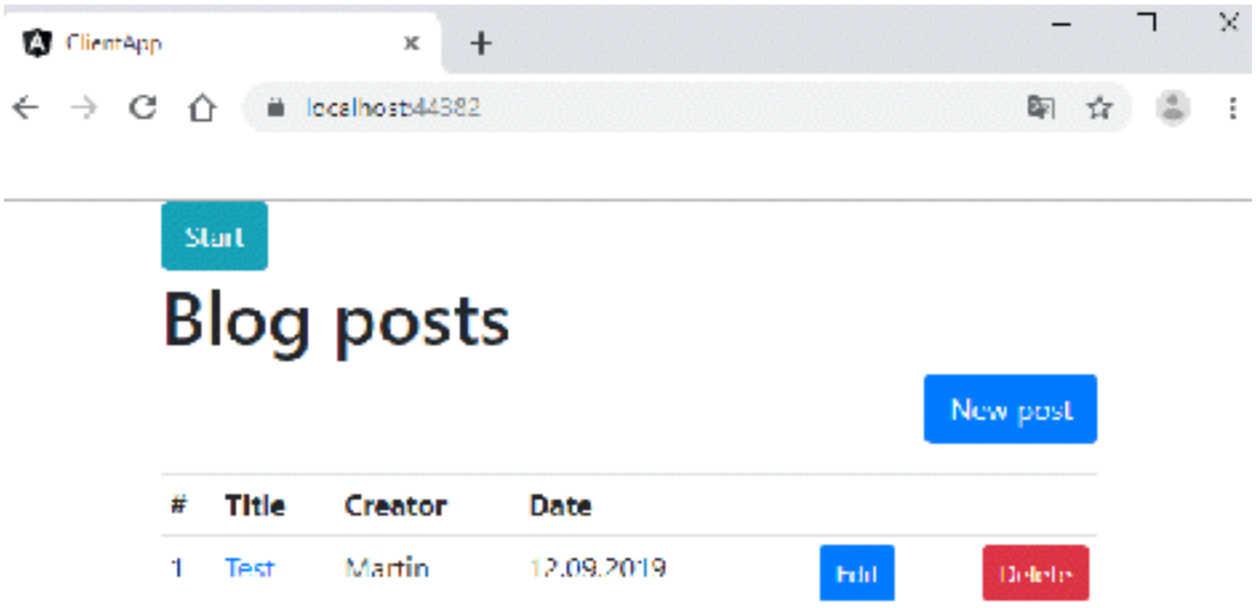
(blogPosts$ | async)

ngIf and ngFor are **structural directives** which change the DOM structure by adding or removing elements.

The routerLink directive lets us link to specific routes in our app.

You can press F5 in Visual Studio 2019 or use the Node.js command prompt and ng serve to launch the app. If you use Node.js to launch the app, make sure the back-end is launched as well in the background (using Visual Studio F5 command).

Since we've manually added a blog post in Postman before, we should now see this:



Blog posts

#	Title	Creator	Date		
1	Test	Martin	12.09.2019	Edit	Delete

Excellent!

Next up is **blog-post.component.ts** to view a single blog post. Edit the file to look like this:

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Observable } from 'rxjs';
import { BlogPostService } from '../services/blog-post.service';
import { BlogPost } from '../models/blogpost';

@Component({
selector: 'app-blog-post',
templateUrl: './blog-post.component.html',
styleUrls: ['./blog-post.component.scss']
})
```

```
export class BlogPostComponent implements OnInit {
blogPost$: Observable<BlogPost>;
postId: number;

constructor(private blogPostService: BlogPostService,
private avRoute: ActivatedRoute) {
const idParam = 'id';
if (this.avRoute.snapshot.params[idParam]) {
this.postId = this.avRoute.snapshot.params[idParam];
}
}

ngOnInit() {
this.loadBlogPost();
}

loadBlogPost() {
this.blogPost$ =
this.blogPostService.getBlogPost(this.postId);
}
}
```

As it’s a single blog post we want to show, we fetch the id key from the url querystring with the built in ActivatedRoute component, and pass it to the service getBlogPost() method.

Now open **blog-post.component.html** and edit it to look like this:

```html
<ng-container *ngIf="(blogPost$ | async) as blogPost;
else loading">

<h1>{{ blogPost.title }}</h1>
<div>{{ blogPost.body }}</div>
<ul>
<li>{{ blogPost.creator }}</li>
<li>{{ blogPost.dt }}</li>
</ul>

</ng-container>
<ng-template #loading>Loading…</ng-template>
```

We use the AsyncPipe again and also use the alias blogPost so we don't have to write blogPost | async everywhere we want to access a blogPost property. We also provide a loading screen.

We're getting closer. Now we just need a way to create new blog posts and edit existing ones. Open **blog-post-add-edit.component.ts** and edit it to look like this:

```typescript
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from
'@angular/forms';
import { Router, ActivatedRoute } from '@angular/router';
import { BlogPostService } from '../services/blog-
post.service';
import { BlogPost } from '../models/blogpost';
```

```
@Component({
selector: 'app-blog-post-add-edit',
templateUrl: './blog-post-add-edit.component.html',
styleUrls: ['./blog-post-add-edit.component.scss']
})
export class BlogPostAddEditComponent implements OnInit {
form: FormGroup;
actionType: string;
formTitle: string;
formBody: string;
postId: number;
errorMessage: any;
existingBlogPost: BlogPost;

constructor(private blogPostService: BlogPostService,
private formBuilder: FormBuilder, private avRoute:
ActivatedRoute, private router: Router) {
const idParam = 'id';
this.actionType = 'Add';
this.formTitle = 'title';
this.formBody = 'body';
if (this.avRoute.snapshot.params[idParam]) {
this.postId = this.avRoute.snapshot.params[idParam];
}
```

```
this.form = this.formBuilder.group(
{
postId: 0,
title: ['', [Validators.required]],
body: ['', [Validators.required]],
}
)
}

ngOnInit() {

if (this.postId > 0) {
this.actionType = 'Edit';
this.blogPostService.getBlogPost(this.postId)
.subscribe(data => (
this.existingBlogPost = data,
this.form.controls[this.formTitle].setValue(data.title),
this.form.controls[this.formBody].setValue(data.body)
));
}
}

save() {
if (!this.form.valid) {
return;
}
```

```

if (this.actionType === 'Add') {
let blogPost: BlogPost = {
dt: new Date(),
creator: 'Martin',
title: this.form.get(this.formTitle).value,
body: this.form.get(this.formBody).value
};

this.blogPostService.saveBlogPost(blogPost)
.subscribe((data) => {
this.router.navigate(['/blogpost', data.postId]);
});
}

if (this.actionType === 'Edit') {
let blogPost: BlogPost = {
postId: this.existingBlogPost.postId,
dt: this.existingBlogPost.dt,
creator: this.existingBlogPost.creator,
title: this.form.get(this.formTitle).value,
body: this.form.get(this.formBody).value
};
this.blogPostService.updateBlogPost(blogPost.postId,
blogPost)
.subscribe((data) => {
```

```
this.router.navigate([this.router.url]);
});
}
}

cancel() {
this.router.navigate(['/']);
}

get title() { return this.form.get(this.formTitle); }
get body() { return this.form.get(this.formBody); }
}
```

Here we're introducing Angular forms: FormBuilder, FormGroup and also Validators.

Depending on if we're creating a new blog post or editing an existing one, we use actionType to show the correct form view with or without data. When we save or update a blog post, we create a new BlogPost object which we then fill with correct form data and then post to our service.

Let's open the **blog-post-add-edit.component.html** and edit it to look like this:

```
<h1>{{actionType}} blog post</h1>
<form [formGroup]="form" (ngSubmit)="save()"
#formDir="ngForm" novalidate>
<div class="form-group row">
```

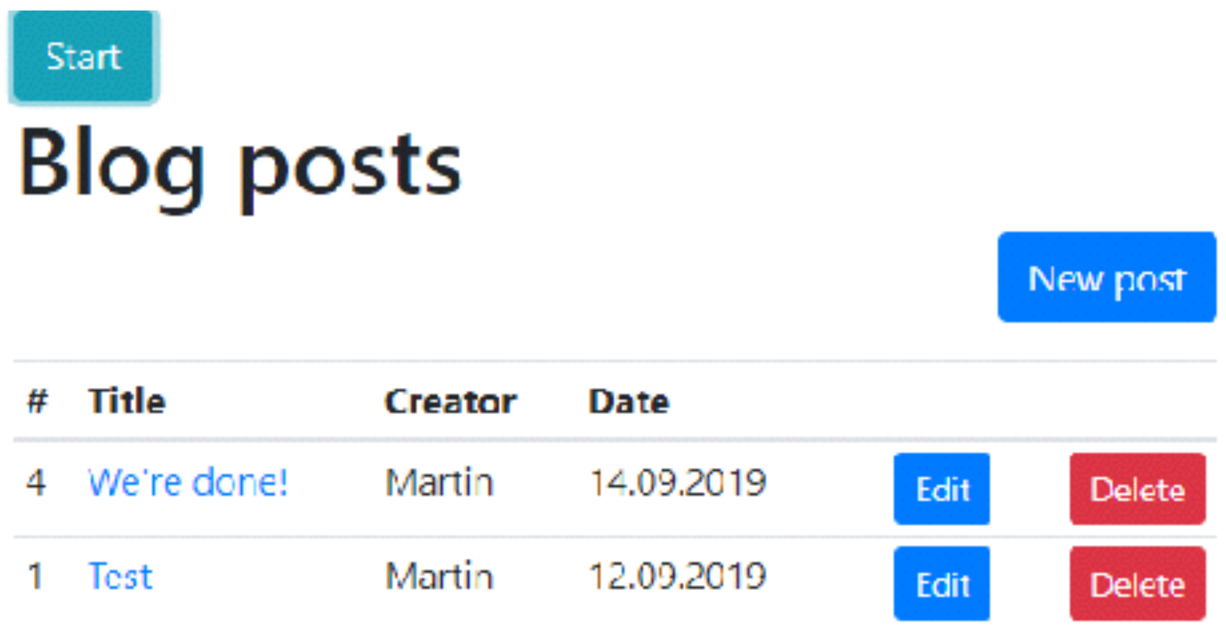
```html
<label class=" control-label col-md-12">Title</label>
<div class="col-md-12">
<input class="form-control" type="text" formControlName="title">
</div>
<span class="text-danger ml-3" *ngIf="title.invalid && formDir.submitted">
Title is required.
</span>
</div>
<div class="form-group row">
<label class="control-label col-md-12" for="Body">Body text</label>
<div class="col-md-12">
<textarea class="form-control" rows="15" formControlName="body"></textarea>
</div>
<span class="text-danger ml-3" *ngIf="body.invalid && formDir.submitted">
Body is required.
</span>
</div>
<div class="form-group">
<button type="submit" class="btn btn-success float-right">Save</button>
<button class="btn btn-secondary float-left" (click)="cancel()">Cancel</button>
```

```
</div>
</form>
```

Here's the form with validation.

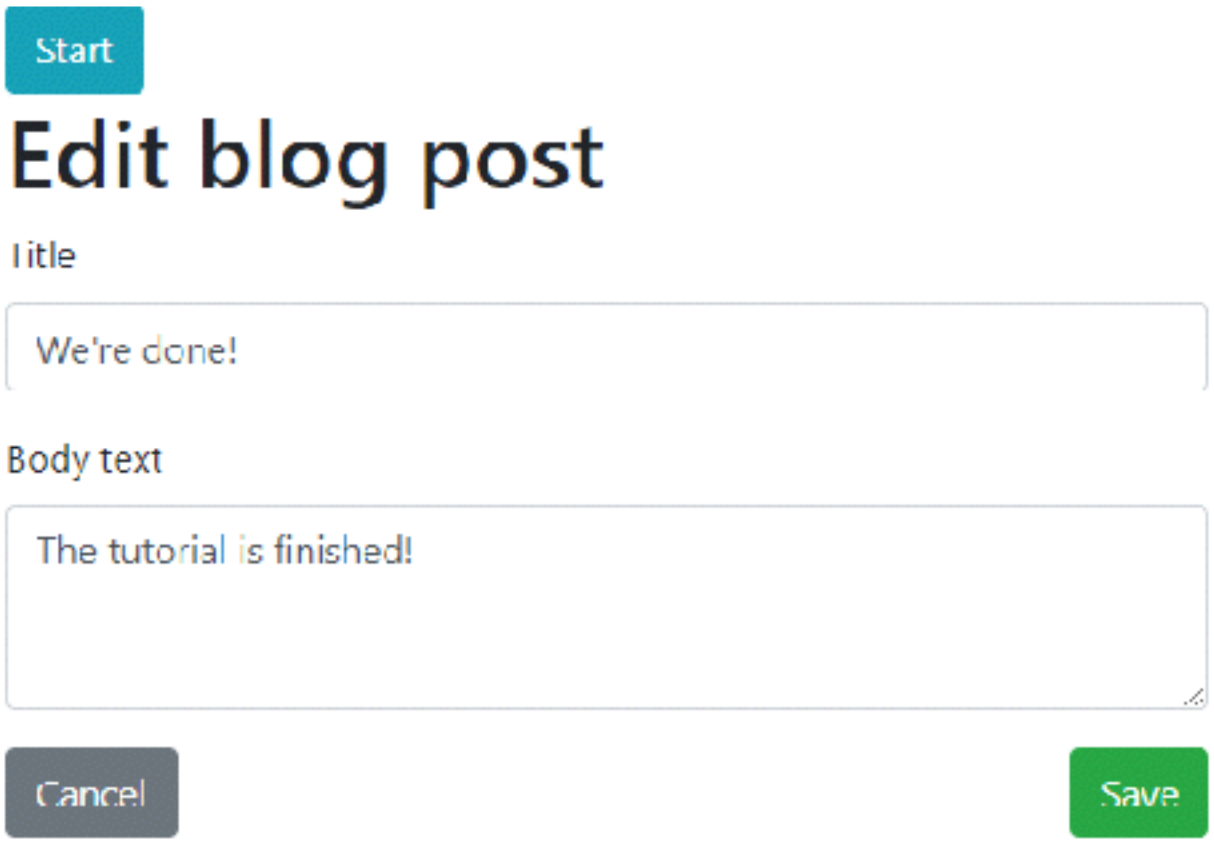
We're done!

Press F5 in Visual Studio 2019 or use the Node.js command prompt and ng serve to browse our final app. (If you use Node.js to launch the app, make sure the back-end is launched as well in the background (using Visual Studio F5 command))



Blog posts

#	Title	Creator	Date
4	We're done!	Martin	14.09.2019
1	Test	Martin	12.09.2019



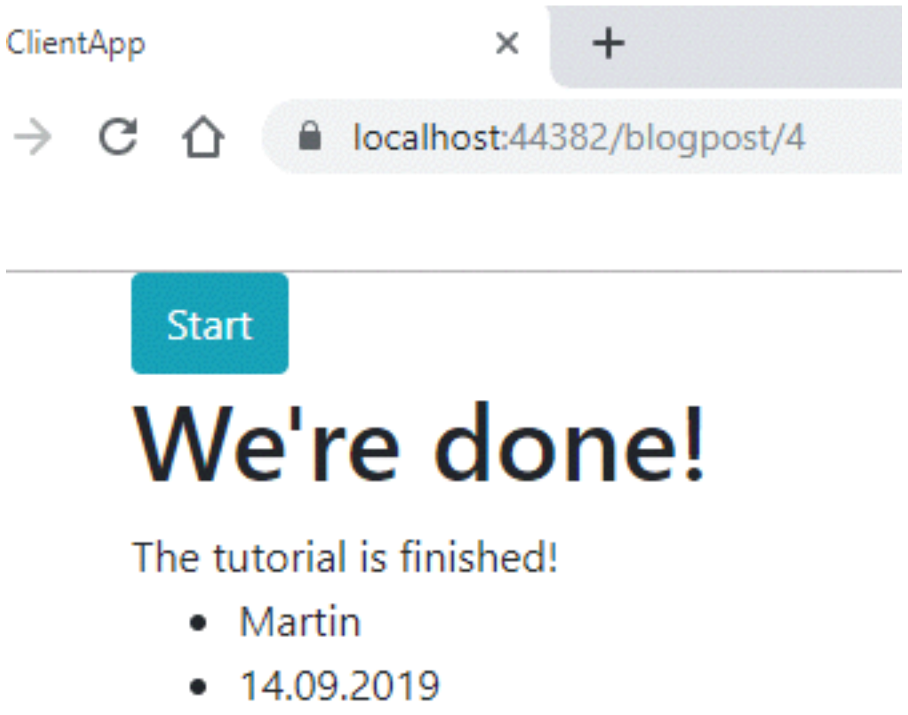
Edit blog post

Title

We're done!

Body text

The tutorial is finished!



localhost:44382/blogpost/4

We're done!

The tutorial is finished!

- Martin
- 14.09.2019