Alexander Ferring
mail@aferring.de

A Progressive Web App (PWA) sample showcasing the techniques described in this document can be found here:
www.aferring.de

Angular and Typescript techniques

This document shall serve as a synopsis of relevant and modern Angular related topics. Ideally one would consider this a display of some of the skills I obtained during my studies of modern Progressive Web App (PWA) technology. These articles are not meant to be stand-alone guides to implementation, but rather to supplement existing literature and documentation, and to draw your attention to topics and common pitfalls.

Table of contents

1. Angular Universal Server Side Rendering (SSR).

1.1 Enhancing page paint times and providing crawler support and social-media integrations

According to Google DoubleClick, 53% of page visits are abandoned if a mobile site takes more than three seconds to load.[1] This holds true for modern devices with fairly stable and fast network connections but becomes even more pronounced on flaky networks and devices that have a harder time loading and evaluating javascript.

One technique to mitigate this problem is to employ Server Side Rendering (SSR), which can be achieved using the "Angular Universal" package: The website will bootstrap on the server and an Html string representation of the page will be transmitted to the client. This greatly enhances the time to first render, especially for slow devices limited by processing constraints. While the user can now read and interact with the server render, the app will be bootstrapped on the users' device in the background.

SSR has even more advantages: Universal renders play an important role in providing content to web-crawlers (SEO), search engines, bots and social media sites that may or may not be able to process Javascript themselves and often rely on meta tags to generate social-media previews. These meta tags would typically not be available during the initial page load, as they are usually evaluated after the Angular framework bootstrap, not least because authors need more precise control over page-by-page meta as opposed to sending static tags. Take Facebook as an example: To generate their preview, they use a specific meta tag called Open Graph. A typical Angular implementation might look like this:

```
<meta property="og:title" content="{{ngMeta.title}}" />
```

The ngMeta.title property will not be evaluated before the page finishes bootstrapping and will usually rely on additional data that may arrive asynchronously from an API call. Using SSR, these meta tags will already be evaluated for the requested page on the server-side and hence available to the crawler or site.

In practice, integrating Universal into new and existing applications is fairly straightforward.[2] A major advantage of Universal, is that you will be using Javascript for both the frontend and the backend stack. However, utilizing SSR techniques will not free the developer of carefully monitoring and reducing the overall bundle size, for instance by lazy-loading routes and bundles.

[1] "The need for mobile speed": https://www.doubleclickbygoogle.com/articles/mobile-speed-matters/
[2] For Universal integration, I suggest you consult the Angular.io guide at https://angular.io/guide/universal. A nice sample starter-repository can be found here: https://github.com/angular/universal-starter.

1.2 Common pitfalls when integrating Angular Universal

I will now briefly go over some issues that I came across when integrating Universal. In more complex apps you will likely rely on many external npm packages. These externals may:

A. Not be bundled /formatted in a way compatible with the Node.js Express bundle format, which is CommonJS.

   You will likely encounter this issue in form of a 'Unexpected token import' error. The Universal server implementation is based on the Express-engine[3], although there are other universal server modules available for aspnetcore- and hapi-engines. If you want the ease of use of having the complete stack Javascript based, you will want to employ the express-engine which will make you susceptible to these bundling issues. To mitigate this you can employ Webpack to re-bundle the server bundle whilst manually marking certain packages as 'externals' within the **webpack.server.config.js**.[4]

```
    externals: [
       nodeExternals({
           whitelist: [
               /ngx-bootstrap/,
               /canvas/,
               [...] // exclude any non-CommonJS packages
           ]
       }),
   ],
```

(requires 'nodeExternals' package from 'webpack-node-externals')

B. Access browser-only DOM properties such as window, document or navigator that are not accessible on the server

   You will find that some service level Angular code (could be component or directive based as well) should not be run when bootstrapping the app on the server. You will want to keep most services state-less and implement platform checks for edge-cases of state or DOM access using `isPlatformBrowser(platformId) (from '@angular/common')` where platformId is injected as `@Inject(PLATFORM_ID) private platformId: Object` from '@angular/core'.

In closing, I would like to mention that for transferring state (e.g. clicks, requests) from the server render to the browser you may want to take a look at **'Preboot'**: github.com/angular/preboot.

[3] Express-engine: https://github.com/angular/universal/blob/master/modules/express-engine/README.md
[4] See also: https://github.com/angular/universal-starter/issues/428#issuecomment-331558400

2. Modular component and service architecture.

2.1 Extending typescript classes to reuse Single Page Application code

Bread and butter of Angular's instance management and dependency injection system is the ability to define components, directives and services (henceforth referred to as elements), that can be injected into other Angular element constructors and modules. These elements are generally managed throughout their lifecycle by the Angular core framework automatically.

At the same time, separating component code from services makes these services reusable for multiple components and directives and allows for caching and other data and lifecycle management strategies and reactive patterns. The more complex an application grows, the more you will find yourself reusing very similar if not the exact same service and component code. In this chapter I would like to present my findings and how I tackled reusability and modularity to avoid rewriting similar functionality.

Consider for instance this extract of a typical (http) service tasked with making backend calls:

```
# BASE SERVICE (some imports excluded)

import { HttpClient } from '@angular/common/http';
import { Inject } from '@angular/core';
[...]
import { Observable, combineLatest, ReplaySubject } from 'rxjs';
import { catchError, map, tap } from 'rxjs/operators';
import { pick as _pick } from 'lodash';

export interface IBaseData<T extends BaseTypes, U extends BaseTypes = any> {
   data: T[] | T;
   additionalData?: U[];
}

export abstract class BaseService<T extends BaseTypes, U extends BaseTypes = any> {
   protected store: IBaseData<T, U> = {data: []};

   protected data: ReplaySubject<IBaseData<T, U>>
                = new ReplaySubject<IBaseData<T, U>>(1);
   public data$: Observable<IBaseData<T, U>> = this.data.asObservable();
```

```
    public getServiceIdentifier(): string { return 'base_service'; }
    public getBaseUrl(): string { return this.appConfig.backendUrl + 'api/'; }

    constructor(
        @Inject(APP_CONFIG) protected appConfig: AppConfig,
        protected errorHandler: AppErrorHandler,
        protected http: HttpClient
    ) { }

    public getAll(): Observable<T[]> {
        return this.http
            .get<T[]>(this.getBaseUrl() + this.getServiceIdentifier() + '/')
            .pipe(
                tap(d => {
                    this.store.data = d;
                    this.data.next(d);
                }),
                catchError(this.errorHandler.handleObservableError)
            );
    }

    [...]
}
```

Looking at this base.service.ts you will find both an internal data `store` as well as an Observable and a rxjs ReplaySubject.

The `data` ReplaySubject really is at the core of the service functionality: ReplaySubjects are special kinds of Subjects that reemit (replay) their contents when subscribed to later on. But what are Subjects you might ask?! Simply put, Subjects have an Input and Output side. The input side is accessible by calling the `.next()` method and allows us to push data to the Subject, whereas the Output side is equivalent to a regular Observable, in that we can subscribe to a stream of replayable asynchronous values. Note that the ReplaySubject is marked as protected, which means we will not be able to manipulate it outside of the service and it's child classes.

Lastly, the service will provide a public Observable named `data$` that we use throughout the Angular application to access the services' data. This `data$` Observable is defined as `data.toObservable()`, which effectively makes it a publicly accessible copy of the Observable part of the ReplaySubject.

2.2 Service inheritance - extending from an abstract BaseService class

As an example, imagine that I was going to write two components: a blog component tasked with handling blog articles and a setting component that can retrieve or manipulate a page-wide settings object. The services providing the data for those components share a lot of similar functionality: they handle backend calls and cache relevant data via observables.

```
# BLOG SERVICE (imports excluded)

@Injectable({ providedIn: 'root' })
export class BlogService extends BaseService<Blog> {

   public getServiceIdentifier(): string { return 'blogs'; }

   constructor(
       @Inject(APP_CONFIG) protected appConfig: AppConfig,
       protected errorHandler: AppErrorHandler,
       protected http: HttpClient
   ) {
       super(appConfig, errorHandler, http);
       this.getAll().subscribe(result => this.data.next({data: result}));
   }
}

// https://angular.io/api/core/InjectionToken#tree-shakable-injectiontoken
export const BLOG_SERVICE = new InjectionToken<BlogService>(
   'BLOG_SERVICE shakable token',
   {
       providedIn: 'root',
       factory: () =>
           new BlogService(
               inject(<any>AppConfig),
               inject(AppErrorHandler),
               inject(HttpClient)
           )
   }
);
```

and

```
# SETTING SERVICE (imports excluded)

@Injectable({ providedIn: 'root' })
export class SettingService extends BaseService<Setting, Blog> {
   store: IBaseData<Setting, Blog> = { data: [], additionalData: [] };

   public getServiceIdentifier(): string { return 'settings'; }

   constructor(
       @Inject(APP_CONFIG) protected appConfig: AppConfig,
       protected errorHandler: AppErrorHandler,
       protected http: HttpClient,
       protected apollo: Apollo,
       protected router: Router
   ) {
       super(appConfig, errorHandler, http);
       // use a different default data provider than this.getAll()
       this.getHome().subscribe(r => this.data.next(r));
   }

   [...]

}

export const SETTING_SERVICE = new InjectionToken<SettingService>(
   'SETTING_SERVICE shakable token',
   {
       providedIn: 'root',
       factory: () =>
           new SettingService(
               inject(APP_CONFIG),
               inject(AppErrorHandler),
               inject(HttpClient),
               inject(Apollo),
               inject(Router)
           )
   }
);
```

You will find that both **SettingService** and **BlogService** extend from the **BaseService** (first code snippet), which allows them to inherit all functions and parameters that this BaseService defines.

You will find that all services make use of Typescript generics, which allow the class to infer the (data) type it will handle by the reference passed to it during instantiation of the child class.

We define an abstract generic class with the generic type operators **T** and **U**:

```
export abstract class BaseService<T extends BaseTypes, U extends BaseTypes = any> {
```

Both **T** and **U** extend from **BaseTypes**, which is a type union of all potential module-level classes, in this case **Setting** and **Blog**, defined as: `type BaseTypes = Blog | CV | Setting | [...]`. This **BaseTypes** definition resides in a file called **module-types.ts**, which holds more type definitions that we will come back to in chapter 2.5.

Furthermore, the type **U** is defined with a default of `= any`. By providing generic parameter defaults[5], we can make the parameter optional.

We make use of the same generics structure for the data stored in our service `store`, defined via the **IBaseData** interface:

```
export interface IBaseData<T extends BaseTypes, U extends BaseTypes = any> {
   data: T[] | T;
   additionalData?: U[];
}
```

Now that we have the basic structure of the handled data and the BaseService, let's get back to our actual child implementations.

We fulfill the class inheritance requirements by calling the `super( )` class constructor from each extending service and passing the necessary dependency injections as parameters.

In terms of application data streams, we will also want to define a default value for the `data` ReplaySubject, a default data retrieval strategy if you will. In the child service, we call a service method like `getAll( )` and subscribe to its return value, which is an observable. When the observable returns, we can pass that result to the ReplaySubject via `data.next( { data: result } )`.

For the **SettingService**, we wish to use a different data source and methodology than the **BlogServices'** `getAll( )` that we call `getHome( )`, hence pointed to in the constructor. In fact, the

[5] This is a Typescript feature since v. 2.3, see: https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-3.html#generic-parameter-defaults

getHome() method uses GraphQL to return results from multiple endpoints on the API and storing the results in the frontend store on both the `data` as well as the `addtitionalData` properties. Therefore, we will want to instantiate the **SettingService** with both **T** and **U** generic types defined as follows:

```
export class SettingService extends BaseService<Setting, Blog> {
```

Similarly, for the **BlogService**, we pass the **'Blog'** as the first generic type. No need to supply a second type, as the **BlogService** will not be storing any `additionalData: U[ ]`.[6]

By structuring our services this way, we won't need to reimplement each method (like `getAll()`) and point it to a specific endpoint for each service. This is because the BaseService provides a `getServiceIdentifier()` method returning a string element specific to each child: basically a string representation of the backend-url-segment for that service. (In the **SettingService** case 'https://api.aferring.de/api/setting/')

Lastly, all the services provide a tree-shakable injection token used to inject a singleton instance of the service throughout the application, while at the same time being tree-shakable in case they remain unused. The token goes hand-in-hand with its definition in the `providers[ ]` array, such as the one in the **AppModule**:

```
# APP MODULE (imports excluded)

@NgModule({
  imports: [...],
  declarations: [...],
  providers: [
    [...]
    { provide: BLOG_SERVICE, useClass: BlogService },
    { provide: SETTING_SERVICE, useClass: SettingService },
  ],
  bootstrap: [AppComponent]
})
export class AppModule { [...] }
```

[6] For further information on Generic classes, union and intersection types, please consult the TypeScript docs https://www.typescriptlang.org/docs/handbook/advanced-types.html.

2.3 Component inheritance - extending from a BaseComponent class and dynamically injecting data and services via router and resolver

When writing components that are similar in structure, one can utilize Typescripts class extension techniques to reduce the amount of duplicate code in components. This is especially helpful for apps that have a structure similar to content management systems (CMS) like Drupal or Wordpress or wherever your app components share many common traits. The patterns are similar to extending a service such as the ones I described in the previous chapter, with one major difference: We will be dynamically assigning a specific service at runtime, hence building a dynamic modular shell that can be used for any data-type /-service we like. Let's begin by defining a base component to provide a baseline of functions and service access like so:

```
# ADMIN MODULE BASE COMPONENT (imports excluded)

@Component({
    templateUrl: 'admin-module-base.component.html',
})
export class AdminModuleBaseComponent {
   protected dataService: ModuleServices;
   public data: ModuleClasses[];
   public dataType: string;
   public loading = false;
   [...]

   constructor(
       @Inject(PLATFORM_ID) protected platformId: Object, // used in children
       protected errorHandler: AppErrorHandler,
       protected activatedRoute: ActivatedRoute,
       protected injector: Injector
   ) {
       this.data = this.activatedRoute.snapshot.data.resolvedData;
       this.dataService = this.injector.get(
           this.activatedRoute.snapshot.data['serviceToken']
       );
       this.dataType = this.dataService.getServiceIdentifier();
   }

   [...] // common functionality for all ModuleComponents would follow e.g.:
   onItemDrop(event: CdkDragDrop<string[]>) { moveItemInArray(...) }
}
```

In this example, we defined the **AdminModuleBaseComponent.** It's core functionality is mostly embedded in the constructor.

For one, it defines a local `this.data` property as equal to the resolvedData property on `this.activatedRoute.snapshot.data`. This resolvedData is defined /located on the router and populated by a special Angular class, a 'resolver':

```
# DATA RESOLVER (imports excluded)

@Injectable({
   providedIn: 'root'
})
export class DataResolver implements Resolve<BaseTypes[] | BaseTypes> {
   dataService: ModuleServices;

   constructor(
       protected errorHandler: AppErrorHandler,
       private injector: Injector,
       private router: Router
   ) {}

   resolve(
       routeSnapshot: ActivatedRouteSnapshot,
       state: RouterStateSnapshot
   ): Observable<BaseTypes[] | BaseTypes> | Observable<never> {
       this.dataService = this.injector.get(routeSnapshot.data.serviceToken);

       return this.dataService.data$.pipe(
            // tap(l => console.log('[DataResolver] data$: ', l)),
            take(1),
            map(r => r.data),
            catchError(err => {
                  this.router.navigate([this.router.url]);
                  this.errorHandler.handleErrorOutput(err);
                  return EMPTY;
               })
           );
     }
}
```

Such a resolver, in this case the modular **DataResolver** can be reused for all (admin) modules and is tasked with querying a specified service and then populating the routers data property with the resulting data on the `resolvedData` field. Just like the **AdminModuleBaseComponent**, it uses the router property (ActivatedRouteSnapshot.data.) `serviceToken` to `this.injector.get()` a service instance that is assigned to `this.dataService`. Which particular service is used for both the data resolver and the component can be controlled from the router configuration. Let us take a glance at an exemplary **AdminRoutingModule** definition:

```
# admin-routing.module (extract, imports excluded)

const adminRoutes: Routes = [
   {
       path: '',
       component: AdminComponent,
       children: [
           [...]
           {
               path: 'push',
               component: AdminPushModuleComponent,
               canActivate: [AuthGuardService],
               resolve: { resolvedData: DataResolver },
               data: {
                   title: 'Admin Push',
                   serviceToken: PUSH_SERVICE,
                   // resolvedData   ← will be populated by the DataResolver
                   depth: 1
               }
           },
     [...]
}
@NgModule({
   imports: [RouterModule.forChild(adminRoutes)],
   exports: [RouterModule]
})
export class AdminRoutingModule {}
```

This admin routing module lists a ‘push’ path, defining a **AdminPushModuleComponent**. Crucially, this route also declares the `data: { serviceToken: PUSH_SERVICE, [...] }` property and assigns a `resolve: { resolvedData: DataResolver }`, property as well.

We want to use this **AdminPushModuleComponent** to manage push endpoints. We also want the push component to inherit common functionality that all admin components share by extending from the **AdminModuleBaseComponent.** Its code might look like this:

```
# ADMIN MODULE PUSH COMPONENT

import { Component } from '@angular/core';
import { AdminModuleBaseComponent } from '../admin-module-base/admin-module-base.component';


@Component({
   templateUrl: 'admin-push-module.component.html',
})
export class AdminPushModuleComponent extends AdminModuleBaseComponent { }
```

If we need functionality in addition to the base provided by the **AdminModuleBaseComponent,** this **AdminPushModuleComponent** would be the place to implement it. Furthermore, in the child class we may optionally define a specific template different to the one on the base component.

2.4 Modularity, error tracing and extendability

We now have a reliable and modular system to dynamically assign services and templates for a multitude of components thereby greatly reducing the amount of duplicate code.

You might wonder if an approach like this introduces debugging and error tracing issues beyond what we would have to handle when explicitly defining each component without extending from a modular base class. In my experience, once one understands the structural approach, this is not an issue. While the errors we observe may look slightly different from a regular pattern, the stack traces will usually clearly reveal the extending component that caused the issue. If for whatever reason that does not suffice, it is possible to enhance the error logging from within the extending classes by (re)implementing the `ngOnInit()` lifecycle hook and referencing the parent hook by a `super.ngOnInit()` call:

```
    ngOnInit() {
        super.ngOnInit();
        # any additional child code
    }
```

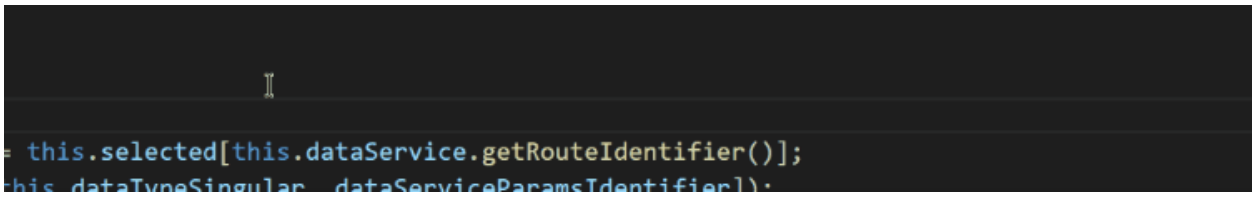
2.5 Using Intersection Types for better type safety and intellisense

Using Typescript Intersection Types[7] we can combine multiple types into one, providing us with stronger type safety, intellisense and documentation annotations than if we were to use the very broad type 'any'.

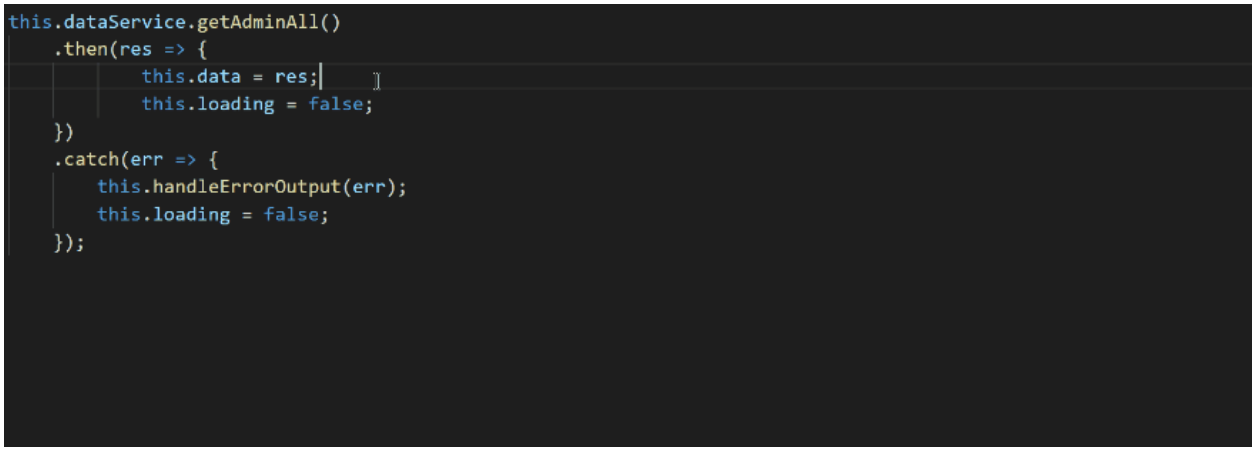
The careful observer may have noticed that in the **DataResolver** as well as the **BaseService** and it's child classes, we defined the (API-)data types and generics as ModuleClasses and BaseTypes, respectively.

```
# BASE SERVICE (extract)

export abstract class BaseService<T extends BaseTypes> {
   protected data: ReplaySubject<IBaseData> = new ReplaySubject<IBaseData>(1);
   data$: Observable<IBaseData> = this.data.asObservable();
   [...]
```



Intellisense will make it possible to distinguish which properties are available for all modules (eg. '_id), and which ones are only available for specific modules (eg. 'filename' for File class):



[7] See Intersection Types here: https://www.typescriptlang.org/docs/handbook/advanced-types.html

To leverage Intersection Types, we define the module types as follows:

```
# module-types.ts (imports excluded)

export type BaseTypes = Blog | CV | Message | File | Setting | Push;

export type ModuleClasses = Blog & CV & Message & File & Setting & Notification & Push;

export type ModuleServices = BlogService & CVService & MessageService & FileService
                              & SettingService & NotificationService & PushService;

export type FileArrayTypes = BlogFiles | CVFiles;
```

Implementing this allows us to have one centralized file that defines all possible module types that our dynamic data components could end up handling.

3. How adding GraphQL to your stack can reduce implementation and roundtrip costs

3.1 GraphQL can reduce roundtrips and avoid over- or under-fetching data

During Facebook's evolution and iteration towards modernized mobile applications, it quickly became apparent that the added complexity of new Front-End features could not readily be supported by the Back-End APIs. In order to provide the required data - often nested and of complex structure - the Facebook team developed a new technology that eventually became known as GraphQL. Using GraphQL, which was open-sourced in 2015, a schema is provided by the API that specifically defines which data can be retrieved via 'queries' and in what manner data can be manipulated via 'mutations'. Additionally, GraphQL supports subscriptions, such that client applications are notified of and pushed new data.

Let's look at the GraphQL syntax for defining a very basic data structure:

```
type Blog {
  name: String!
  pages: [Page!]!
}
```

As you can readily see, GraphQL uses its very own syntax that differs from Typescript or Javascript. For existing projects, in order to avoid having to fully copy and adapt all existing model definitions, we will be using the graphql-compose framework to generate the schema on the following pages.

The final schema is then provided by a GraphQL endpoint, such that any potential client application will have a reliable idea of the structure of the data and possible interactions with that endpoint. As a result, not only can GraphQL clients manipulate or retrieve data using one method per request, but also combine multiple interactions (queries and manipulations) into one roundtrip. Additionally, one query for data must not necessarily request all properties on an existing model, but can select exactly which properties it wishes to retrieve on a per-field basis. This is what is famously referred to as 'avoiding over- and under-fetching of data'.

As an additional result, any Frontend application can now iterate over new designs much quicker, as there is no strict need to implement a new API-endpoint for every new client view.

A few years after the initial adoption of GraphQL by Facebook and other well known companies such as Github, plenty of helpful helper libraries exist that greatly ease the implementation of GraphQL on both the client and server.

3.2 Building a GraphQL schema on top of a REST-API and Mongodb

3.2.1 From an existing Mongoose model to a GraphQL schema

Let's take a look at what a server implementation may look like. This builds on top of an existing Node.js REST-API with Mongodb database. Using a github project called graphql-compose[8] (and the graphql-compose-mongoose[9] additions), we can use our existing Mongodb (Mongoose) models to more readily create a GraphQL schema out of these data-structures.

Let's assume we have a Blog model that looks something like this:

```
# BLOG MODEL

'use strict';
var mongoose = require('mongoose');
var mSchema = mongoose.Schema;
var blogSchema = new mSchema({
   name: { type: String, required: true },
   desc: { type: String, required: true },
   active: { type: Boolean, default: false },
   content: { type: String, required: true },
   created_at: Date,
   order: Number,
   changed_by: { type: String, default: 'System' }
});
[...]
var Blog = module.exports = mongoose.model('Blog', blogSchema);
```

We can now build the GraphQL schema like so:

```
# GRAPHQL SCHEMA

var { schemaComposer } = require('graphql-compose');
var { composeWithMongoose } = require('graphql-compose-mongoose');


var jwt = require('jsonwebtoken'); // used with auth helpers later on
```

[8] GraphQL-compose: https://github.com/graphql-compose/graphql-compose
[9] GraphQL-compose-mongoose: https://github.com/graphql-compose/graphql-compose-mongoose

```
/////////// BLOGS ///////////

var Blog = require('models/blog.model');
var BlogService = require('services/blog.service');
const globalOptions = {
   fields: {
       // Remove potentially sensitive data from GraphQL Schema
       remove: ['hash', 'changed_by', 'vapidPrivateKey'],
   }
};

// generate Blog TypeComposer with global options object
const BlogTC = composeWithMongoose(Blog, globalOptions);

// define custom resolver based on findMany (inherits findMany properties)
const defaultFindMany = BlogTC.getResolver('findMany')
   .addFilterArg({ // add filter arg for active (adds visible default on client)
       name: 'active',
       type: 'Boolean',
       description: 'property active is true (enforced by default)',
       defaultValue: true, // ! enforce active true (can not be overwritten from client)
       query: (query, value, resolveParams) => { query.active = true; }
   }).addSortArg({ // add sort arg for order descending
       name: '_ORDER_DESC',
       description: 'sorting by order descending (enforced by default)',
       value: { order: 'desc' },
   }).addSortArg({ // add sort arg for order descending
       name: '_ORDER_ASC',
       description: 'sorting by order ascending',
       value: { order: 'asc' },
   }).wrapResolve(next => (rp) => {
       // console.log('rp: ', rp);
       if (rp.args.sort === undefined) { // set default order sorting (not enforced)
           rp.args.sort = { order: 'desc' };
       }
       return next(rp);
   });
// store for future usage
BlogTC.setResolver('defaultFindMany', defaultFindMany);
```

```
// add a customFindOne resolver from scratch (does not inherit findOne)
BlogTC.addResolver({
   name: 'customFindOne',
   args: { name: BlogTC.getFieldType('name') },
   type: BlogTC,
   resolve: async ({ source, args, context, info }) => {
       res = await BlogService.getByName(args);
       return res;
   }
});

// define Blog Queries on schemaComposer
schemaComposer.Query.addFields({
   blog: BlogTC.getResolver('customFindOne'),

   blogs: BlogTC.getResolver('defaultFindMany'),

   blogsCount: BlogTC.getResolver('count')
});

////////////// SETTINGS ////////////

var Setting = require('../models/setting.model');
const SettingTC = composeWithMongoose(Setting, globalOptions);

schemaComposer.Query.addFields({
   setting: SettingTC.getResolver('findOne')
});

schemaComposer.Mutation.addFields({
   update: SettingTC.getResolver('updateOne', [authMiddleware])
});


////////////// FINALIZE GQL SCHEMA ////////////

const graphqlSchema = schemaComposer.buildSchema();
module.exports = graphqlSchema;
```

```
/////////// AUTH HELPERS ///////////
async function authMiddleware(resolve, source, args, context, info) {
   // could differentiate user permissions here
   return await verifyJwt(context).then(
       ()=> { return resolve(source, args, context, info); },
       (err) => { throw new Error(err) }
   );
}

var verifyJwt = context => {
   let token;
   if ( context.headers.authorization &&
        context.headers.authorization.includes('Bearer') ) {
       token = context.headers.authorization.split(' ')[1];
   }
   return new Promise((resolve, reject) => {
       if (!token) reject('[GraphQLSchema Auth] JWT missing');
       jwt.verify(token, 'SHHHHHHH-SECRET-STRING', (error, decoded) => {
           if (error) reject('[GraphQLSchema Auth] JWT invalid');
           resolve();
       });
   });
};
```

When building a schema, we begin by importing the existing Mongoose model(s). You will find that for the Blog section, in addition to the **BlogModel** I also imported the **BlogService**. This is because we will be showing examples not only of the native Mongoose methods like 'findOne' or 'findMany' as queries or 'updateOne' and 'updateMany' as mutations, but also hook existing REST service methods that provide more in-depth functionality.

Let's first look at an example, where we make due with the Mongoose 'findOne' method: Generally speaking, we use the schemaComposer. Query.addFields( ) and Mutation.addFields( ) methods respectively to define query and mutation fields like so:

```
schemaComposer.Query.addFields({
   setting: SettingTC.getResolver('findOne')
});
```

This will provide the setting query based on the default Mongoose 'findOne' function. Without further additions, this uses an '_id' to search for that specific document.

3.2.2 Building custom resolvers to change query, filter or sort arguments

If we wanted to use different mongo query params, we could instead define a custom resolver:

```
BlogTC.addResolver({
   name: 'customFindOne',
   args: { name: BlogTC.getFieldType('name') },
   type: BlogTC,
   resolve: async ({ source, args, context, info }) => {
       return await BlogService.getByName(args);
       // return await Blog.findOne(
       //     {
       //         name: args.name,
       //         active: true
       //     }
       // )
   }
});

schemaComposer.Query.addFields({
   blog: BlogTC.getResolver('customFindOne'),
});
```

In this example, we define the 'customFindOne' resolver with custom resolve method returning from the BlogService.getByName(args);. This uses a string (BlogTC.getFieldType('name') args property), instead of the default '_id' (MongoID) 'findOne' uses by default. In the part of the code that is commented out, you will find another way of approaching this by making use of 'findOne' without having to reference the BlogService, but similarly passing the 'name' argument and an additional filter 'active: true'.

Whether you wish to use the methods on the Mongodb schema and Mongoose wrapper directly or additionally employ a Node.js service like the BlogService as shown in this example, depends a lot on your existing infrastructure. Assuming you already have further API functionality like advanced filtering, trimming or safety checks implemented in services, the graphql-compose-mongoose flexibility allows you to freely choose how you build the GraphQL layer and endpoint on top of your services.

If you do not yet have an existing service-esque architecture or if you wish to allow the GraphQL endpoint more flexibility in constructing queries and as to which filters and sorting will be applied optionally and or enforced, you may also implement fairly involved custom resolvers like so:

```
const defaultFindMany = BlogTC.getResolver('findMany')
   .addFilterArg({ // add filter arg for active (adds visible default on client)
       name: 'active',
       type: 'Boolean',
       description: 'property active is true (enforced by default)',
       defaultValue: true, // ! enforce active true (can not be overwritten from client)
       query: (query, value, resolveParams) => { query.active = true; }
   }).addSortArg({ // add sort arg for order descending
       name: '_ORDER_DESC',
       description: 'sorting by order descending (enforced by default)',
       value: { order: 'desc' },
   }).addSortArg({ // add sort arg for order descending
       name: '_ORDER_ASC',
       description: 'sorting by order ascending',
       value: { order: 'asc' },
   }).wrapResolve(next => (rp) => {
       if (rp.args.sort === undefined) { // set default order sorting (not enforced)
           rp.args.sort = { order: 'desc' };
       }
       return next(rp);
   });

BlogTC.setResolver('defaultFindMany', defaultFindMany);

schemaComposer.Query.addFields({
   blogs: BlogTC.getResolver('defaultFindMany'),
});
```

Here, we first import schema 'findMany' baseline-properties via BlogTC.getResolver('findMany') and extend that resolver by adding sort and filter arguments.

Since we are using the blogs query as a public endpoint, we want it only deliver documents that have the property 'active' set to true. To enforce this, we provide a defaultValue of true for that addFilterArg().

Additionally we want to apply one of the two custom sortings added via addSortArg() to be applied by default by wrapping the resolve function and setting `rp.args.sort = { order: 'desc' }` as shown above. This will only set the default sorting, but not enforce or overwrite any ordering passed from clients.

3.2.3 Restricting access to private fields and adding authentication functions

When building a GraphQL endpoint, there may well be fields on our models that we do not wish to expose to the schema. In this example, that is the 'changed_by' property on the Blog model and the 'changed_by' and 'vapidPrivateKey' fields of the Setting model. We can exclude these fields from the schema by defining a (global) options object that we pass to the composeWithMongoose() :

```
const globalOptions = {
   fields: {
       remove: ['changed_by', 'vapidPrivateKey'],
   }
};
const BlogTC = composeWithMongoose(Blog, globalOptions); // and similar for other TCs
```

Lastly, let's take a quick look at how we implement authentication in the schema: At the end of the main code-sample above you will find the `authMiddleware()` and `verifyJwt()` helper functions. (See full code sample above). In the authMiddleware function we can control which user permissions a user needs to have for a specific request (not shown) and that a JSON web token (jwt) is transferred with the request to ensure the user is authenticated. These implementations may vary depending on your use-case.

We can reference this authentication middleware in our schema by adding the [authMiddleware]:

```
   update: SettingTC.getResolver('updateOne', [authMiddleware])
```

3.3 Universal SSR ready GraphQL on the Front-End using Apollo-client

On the Angular client side, I have used the Apollo client implementation with in-memory cache and graphql-tag query builder helper.

We begin by adding the ApolloModule to the **AppModule** imports and initialize both the Apollo client and in-cache memory, for instance in the **AppModule** class:

```
# APP MODULE

import { ApolloModule, Apollo } from 'apollo-angular';
import { HttpLinkModule, HttpLink, HttpLinkHandler } from 'apollo-angular-link-http';
import { InMemoryCache, NormalizedCacheObject } from 'apollo-cache-inmemory';

const APOLLO_KEY = makeStateKey<any>('apollo.state');

@NgModule({
   imports: [
      [...]
      ApolloModule
   ],
   [...]
   bootstrap: [AppComponent]
})
export class AppModule {
   cache: InMemoryCache;
   link: HttpLinkHandler;
   constructor(
       @Inject(PLATFORM_ID) readonly platformId: Object,
       private apollo: Apollo,
       private readonly transferState: TransferState,
       private readonly httpLink: HttpLink
   ) { // Set up Apollo Cache, Link and perform cache state transfer for SSR bundles
       const isBrowser = isPlatformBrowser(platformId);

       this.cache = new InMemoryCache();
       this.link = this.httpLink.create({ uri: environment.backendUrl + 'graphql/' });
```

```
        this.apollo.create({
            link: this.link,
            cache: this.cache,
            ...(isBrowser
                    ? { ssrForceFetchDelay: 200 }
                    : { ssrMode: true, }
                )
        });

        if (isBrowser) { // Browser - read the serialized state
            const state = this.transferState.get<NormalizedCacheObject>(APOLLO_KEY, null);
            this.cache.restore(state);
            // console.log('[AppModule] apollo.state from server:', state);
        } else { // Server - serialize the cache and put it to transferState under a key
            this.transferState.onSerialize(APOLLO_KEY, () => this.cache.extract());
        }
    }
}
```

This implementation is **Universal** Server Side Rendering (SSR) ready as it transfers the state from the server render by serializing it in a NormalizedCacheObject called 'APOLLO_KEY' to later de-serialize after the client bootstrap.

We can now build a service, e.g. the **HomeService** and get it ready to use GraphQL by injecting the `protected apollo: Apollo` service. You will also find that we use the graphql-tag package to convert a string representation of the query to a GraphQL query object using `gql(queryString)`:

```
# HOME SERVICE

import { Inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';

import { Apollo } from 'apollo-angular';
import gql from 'graphql-tag';

export class HomeService {
    constructor(
        protected http: HttpClient,
        protected apollo: Apollo,  // <-- inject apollo service
    ) { }
```

```
    public getHome(): Observable<any> {
        const queryString = `
            query {
                setting {
                    pageTitle
                    additionalOptions
                }
                blogs(limit: 3) {
                    name
                    desc
                    created_at
                }
            }
        `;

        return this.apollo.query<any>({ query: gql(queryString) })
            .pipe(
                // tap(console.log),
                shareReplay(1)
            );
    }
}
```

The `getHome( )` method should now readily provide the results of the combined setting and blog query, with a limit of 3 blogs. To verify that you can not retrieve any private fields that we have excluded from the schema in the previous chapter, you may try to add `created_by` to either the blogs or setting query parameters, which should result in the request failing.

Additionally, if you wish to test the authorization middleware you will want to send a jwt token as a header to the endpoint. The default jwt structure is: 'Authorization': 'Bearer ' + tokenObject.[10] You can add headers either from within your Angular app or your browser plugin of choice. I use the 'ModHeader' plugin.[11] This plugin will allow you to add headers to browser interfaces like GraphiQL - which is used by many server and client packages by default - instead of having to use apps like Postfix which may or may not provide GraphQL syntax highlighting and intellisense-esque completion.

[10] Additional info on JSON Web Tokens (jwt): https://en.wikipedia.org/wiki/JSON_Web_Token
[11] Modheader: https://chrome.google.com/webstore/detail/modheader/idgpnmonknjnojddfkpgkljpfnnfcklj

4. Angular stateful router animations

4.1 An introduction to animations with the Angular framework

Modern progressive web apps would not be the same without the capability to transition elements and pages using animations, for they can greatly improve not only the visual experience but also make the application easier to use. For instance, they can provide another level of context /feedback on where the user is currently located or being relocated to.

You may have noticed that on this page I use two kinds of animations globally throughout the app:

1) Pages that are on the same (hierarchy) level, which are all elements in the navbar (Home, Blogs, CV and Contact) are transitioned between by sliding the pages from side to side and
2) Pages that are nested deeper into the page tree, use a zoom-in and –out type animation.

In this chapter I will give a brief overview over the @angular/animations usage and capabilities and show in particular, how I achieved the slightly advanced animations that are used to provide location context to the user.

Angulars' animation package @angular/animations provides a variety of animations capable of running with the same kind of performance that pure CSS animations would. These animations are built on top of the Web Animations API and run natively on the following supported browsers: Firefox, Chrome, Opera, Android Browser and Chrome for Android.

(For Internet Explorer, Edge, Safari, iOS Safari and Opera Mini, the web-animations-js polyfills are required for the animations capability)

Implementing animations is fairly trivial, after importing the appropriate packages from @angular/animations, for this example

```
import { trigger, state, style, animate, transition } from '@angular/animations';
```

they can be defined as part of Angulars' component definition

```
@Component({
    [...]
    animations: [ ]
})
```

Now, there are different ways of defining animations, all of which need to first define a trigger that will be used to bind the animation to some component template element, for instance:

```
    animations:[
        trigger('triggerName', [
            state('inactive', style({
                transform: 'scale(0)', height: '0', color: '*'
            })),
            state('active', style({
                transform: 'scale(1)', height: '*', color: '#cfd8dc'
            })),
            transition('inactive => active', animate('500ms ease-in')),
            transition('active => inactive', animate('400ms ease-out'))
        ])
    ]
```

In this example, the trigger 'triggerName' will define which animations (and stylings) are applied to the element, and in which manner the animation system shall transition between these states. In this case, the element will be easing into active state over a time of 500ms and ease-out into its inactive state over 400ms. As the name suggests, this animation can be used to ease in and out an element. Let's apply it to a element containing some text. Additionally, we'll drop in a button that will be calling a 'toggleAnimation()' function:

```
<button (click)="toggleAnimation()"> Click me </button>
<div [@triggerName]="triggerVariable">Text</div>
```

Lastly, within the **SomeComponent**, the function 'toggleAnimation()' is used to handle the variable 'triggerVariable':

```
# some.component.ts (extract)
[...]
export class SomeComponent {
    triggerVar = 'inactive';

    constructor() {}

    toggleAnimation() {
        this.triggerVariable = this.triggerVariable === 'inactive' ? 'active'
                                                                   : 'inactive';
    }
}
```

And with that, we're all ready to animate. This is probably the most basic kind of animation that can be defined using @angular/animations, however there are many more capabilities that the animation package offers. For example, the developer can utilize * wildcard operators to define animations and states that apply regardless of which state the animation is currently in. Furthermore, the package allows the definition of groups which effectively runs animations in parallel, animation callbacks as well as multi-step animations with keyframes.

Now that we know how animations are generally defined and used, let's take a look at improving our router outlet by animating page / outlet changes for our app on the next page, shall we?

4.2 Utilizing additional router data to enhance transitions and convey location

Let's examine how we can provide context to our app and hence the animations as to where the user is located, which is the depth of the app tree.

The @angular/router exposes an interface called RouterState that contains a snapshot of the currently active route. (https://angular.io/api/router/RouterState)

This RouterStateSnapshot in turn contains a root element of type ActivatedRoute and is equally accessible through the ActivatedRoute interface[12]:

```
import { ActivatedRoute } from '@angular/router'
```

[12] ActivatedRoute interface: https://angular.io/api/router/ActivatedRoute

Let's take a look at the interface ActivatedRoute as defined by the Angular framework:

```
interface ActivatedRoute {
    snapshot:    ActivatedRouteSnapshot
    url:         Observable<UrlSegment[]>
    params:      Observable<Params>
    queryParams: Observable<Params>
    fragment:    Observable<string>
    data:        Observable<Data>
    outlet:      string
    component:   Type<any>|string|null
    [...]
}
```

What we are interested in is the data: Observable<Data> field of the interface, which is 'An observable of the static and resolved data of this route'. This observable can be used to store additional data on the router itself like so:

```
const appRoutes: Routes = [
    {
        path: 'home',
        component: HomeComponent,
        data: { title: 'Home', depth: 1 }
    },
    {
        path: 'lazy',
        loadChildren: 'app/lazy/lazy.module#LazyModule',
        data: { title: 'Lazy Route', depth: 0 }
    },
    {
        path: 'blogs',
        children: [
            {
                path: '',
                component: BlogComponent,
                data: { title: 'Blogs', depth: 1 }
            },
            {
                path: ':name',
                component: BlogDetailComponent,
                data: { title: 'Blog detail', depth: 2 }
            },
        ]
    },
    [...]
];
```

This extract of my app-routing.module.ts shows 4 possible routes: home, lazy, blogs and blogs/:name, whereas the lazy route is a lazy loaded module as indicated by the loadChildren property. The lazy loaded module will itself contain its own route definition module with several routes.

The data: Observable<Data>[13] represents the static data associated with each route.

```
type Data = { [name: string]: any };
```

In this example, the data property contains a title and depth variable that we can use to determine which animation we want to spool on our router outlet element. Whenever we navigate to a new route with a higher depth, we will play a zoom in animation and vice versa. For routes from depth 1 -> 1 we will be using a sliding animation. As usual, we need to define the animations in the component and reference them by their trigger in the template and then provide a mechanism to determine which type of animation we want to use.

Let's begin by defining the animations, shall we? To avoid overly bloating this chapter with code even further I will show only one example with code: the animation for navigating to a route with a higher depth (=deeper):

```
    animations: [
        trigger('routerOutletAnims', [
            [...]
            transition('* => deeper', [
                style({ height: '!' }),
                query(':enter', [
                    style({ transform: 'scale(0.5)' })
                ], { optional: true }),
                query(':enter, :leave', [
                    style({ position: 'absolute', top: 0, left: 0, right: 0 })
                ], { optional: true }),
                group([
                    query(':leave', [
                        animate('0.2s', style({ transform: 'scale(1.5)', opacity: 0 }))
                    ], { optional: true }),
                    query(':enter', [
                        animate('0.2s', style({ transform: 'scale(1)', opacity: 1 }))
                    ], { optional: true }),
                ]),
            ]),
    ]
```

(Note: The use of the 'group([])' field to parallelize the enter and exit animations needs the group import from @angular/animations!).

[13] Router data observable property: https://angular.io/api/router/Data

In the next step, we bind the animation to the template using the ‘routerOutletAnims’ trigger:

```
    animations: [...],
    styles: [`
        .routeContainer {
            position:relative;
            overflow-x:hidden;
        }
        .routeContainer>* {
            display:block;
            overflow-x:hidden;
        }
    `],
    template: `
        <div class="routeContainer" [@routerOutletAnims]="getAnimationState()">
            <router-outlet></router-outlet>
        </div>
    `
```

You may have noticed that the animations are defined on a transition from any (wildcard) to a string of ‘deeper’ (transition('* => deeper', ...). Conversely, the depth data stored on the routes is a number (0 to 2 in this case) instead. Why don’t we store depth as a string ‘higher’ or ‘deeper’ per route you might ask. That is because we want to allow more fine grained control over which animations play and allow the component to handle animations for multiple levels of depth. (Depth in this context may refer to both horizontal and vertical depth, although not part of the examples I’m showing here).

```
    lastDepth: number;
    animationState: string;

    getAnimationState() {
        return this.animationState;
    }

    determineAnimations(depth: number) {
        [...]
        if (depth === 2) {
            this.animationState = 'deeper';
        } else if (this.lastDepth === 2 && depth === 1) {
            this.animationState = 'higher';
        }
        this.lastDepth = depth;
    }
```

We will be using the `determineAnimations()` function to decide the `animationState` depending on the depth property we embedded in the router data Observable. When navigating to a different route, the routeContainer containing our <router-outlet> will be calling getAnimationProps(), which will return the `animationState` we have just determined. Now Angular will apply the appropriate animation properties, the styles that are defined for both entering (:enter) and leaving (:leave) elements within that routeContainer. We're all set, the router outlet should be animated now. If you want to adapt this example within your Angular app, in order to have another animation transition ('*=>higher') defined for route navigations back to the lower depth route, it's as simple as swapping the transform: 'scale(0.5)' with transform: 'scale(1.5)' value for that transition.

Sidenote: Whether the user accesses the ActivatedRoute in a component by importing it directly or accessing it through the *router.routerState.snapshot.root* does not matter for this example, however I found that if lazy loading is utilized in the app, the .data property will be nested deeper inside the RouterState snapshot for child lazy-routes (sometimes by multiple levels). Therefore I recommend using the router to traverse the snapshot instead of the ActivatedRoute in the following manner:

```
    constructor( router: Router ) {}

    ngOnInit() {
        this.router.events
            .filter((event: any) => event instanceof NavigationEnd)
            .subscribe(() => {
                let root = this.router.routerState.snapshot.root;
                while (root) {
                    if (root.children && root.children.length) {
                        root = root.children[0];
                    } else if (root.data && root.data['title']) {
                        this.setTitle(root.data['title']);
                        this.determineAnimations(root.data['depth']);
                        return;
                    } else {
                        return;
                    }
                }
            });
    }
```

Source: Answer to '[Retrieving a data property on an Angular2 route regardless of the level of route nesting using ActivatedRoute]' by Robba.

This chapter was motivated particularly by the blog-entry '[Hierarchical Route Animations in Angular]' by Steven Fluin and the ng-conf talk on '[Animations in Angular 4.0.0]' by Matias Niemela.

5. Docker setups and linking Docker networks

5.1 Using Docker to deploy a performant and isolated stack to production

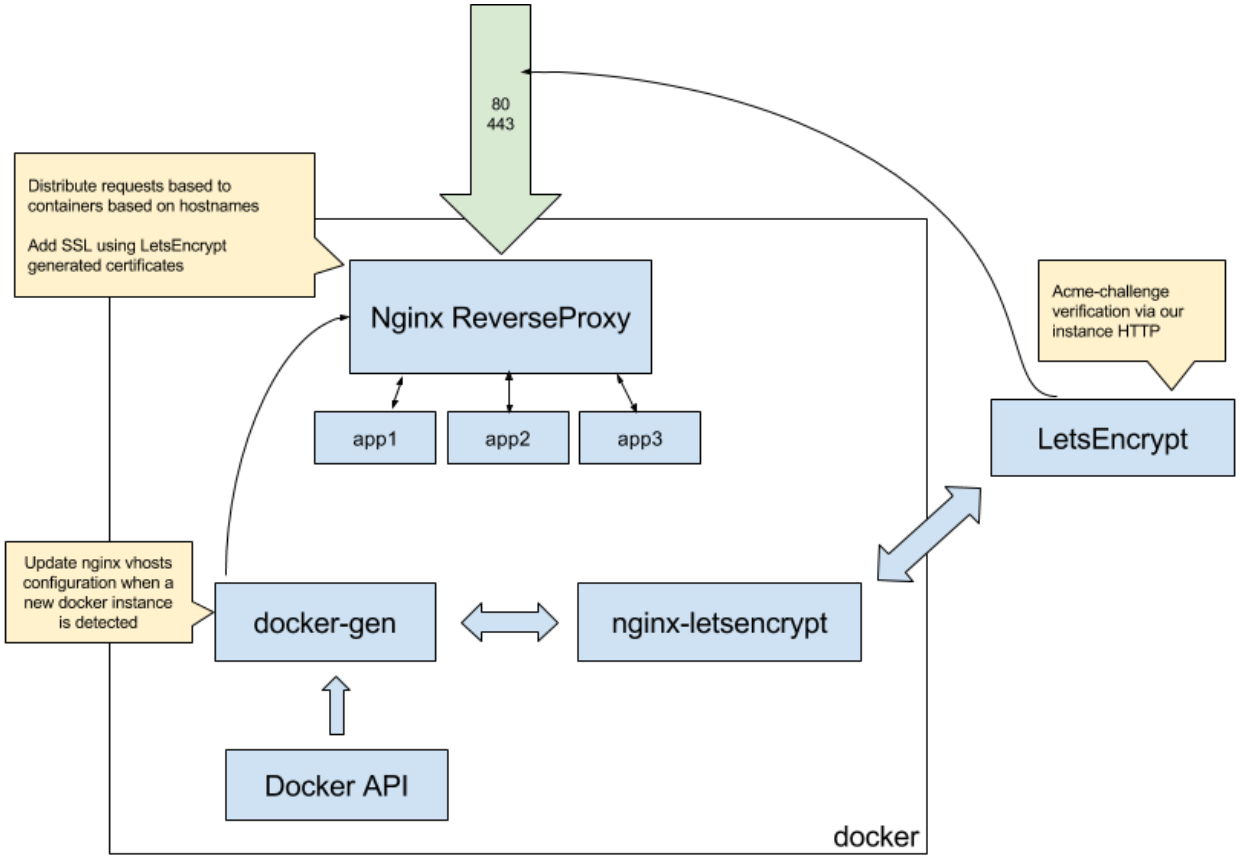
> Docker is a software technology providing containers, [...] an additional layer of abstraction and automation of operating-system-level virtualization on Windows and Linux. Docker uses the resource isolation features of the Linux kernel [...] to allow independent "containers" to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines (VMs).
>
> *(Extract from Wikipedia: 'Docker')*

Docker can be an amazing utility to test and deploy a variety of technology-stack setups. In my case this stack consists mostly of what would be considered a MEAN stack: Mongoose, Express and Angular on the basis of Nodejs. Furthermore I used mailcow-dockerized as a mail server and mail transfer agent (MTA) as well as nginx as a reverse proxy.

In this chapter I will show how to utilize Docker to combine all these elements into one functional network and technology stack. In particular I will show an advanced docker-compose.yml configuration that leverages the very neat Docker "docker-letsencrypt-nginx-proxy-companion" by JrCs[14] to automagically request, verify and handshake Let's Encrypt SSL certificates and generate proxy configurations for a multitude of domains and their corresponding Docker containers.

Throughout the text I will be referring to the docker-letsencrypt-nginx-proxy-companion as **DLENPC**.

[14] Docker-LE-proxy-companion sources https://github.com/JrCs/docker-letsencrypt-nginx-proxy-companion



DLENPC functionality. Source: [docker-letsencrypt-nginx-proxy-companion](#) Github

In practice, what I wanted my app to provide is basically 3 gateways or domains that are able to operate independently of each other: A website frontend, an API for the backend and lastly a mail transfer agent.

In order to configure the DLENPC, one can use Dockers command line utilities to start and maintain containers following the github examples, but to effectively bundle and deploy a stack setup it is much more convenient to define a docker-compose file (.yml extension, YAML file).

5.2 Docker setup stack overview

The docker-compose file consists of 7 containers as defined under the services array:

DLENPC functionality:	MEAN stack +:
• Nginx, • Nginx-gen, • Nginx-le-companion	• Ngxhome-api • Ngxhome-website • Ngxhome-db • Ngxhome-mta

5.3 The docker-compose.yml configuration

```
# docker-compose.yml - ngxhome container deploy

version: "3"
services:
  nginx:
    image: nginx
    labels:
        com.github.jrcs.letsencrypt_nginx_proxy_companion.nginx_proxy: "true"
    container_name: nginx
    restart: unless-stopped
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./volumes/nginx/conf.d:/etc/nginx/conf.d
      - ./volumes/nginx/vhost.d:/etc/nginx/vhost.d
      - ./volumes/nginx/html:/usr/share/nginx/html
      - ./volumes/nginx/certs:/etc/nginx/certs:ro
      - ./volumes/nginx/nginx.conf:/etc/nginx/nginx.conf
    networks:
      - main

  nginx-gen:
    image: jwilder/docker-gen
    labels:
        com.github.jrcs.letsencrypt_nginx_proxy_companion.docker_gen: "true"
    container_name: nginx-gen
    restart: unless-stopped
    volumes:
      - ./volumes/nginx/conf.d:/etc/nginx/conf.d
      - ./volumes/nginx/vhost.d:/etc/nginx/vhost.d
      - ./volumes/nginx/html:/usr/share/nginx/html
      - ./volumes/nginx/certs:/etc/nginx/certs:ro
      - /var/run/docker.sock:/tmp/docker.sock:ro
      - ./volumes/nginx/templates:/etc/docker-gen/templates:ro
    links:
      - "nginx:nginx"
    networks:
      - main
    command: -notify-sighup nginx -watch -wait 5s:30s
/etc/docker-gen/templates/nginx.tmpl /etc/nginx/conf.d/default.conf

  nginx-le-companion:
    image: jrcs/letsencrypt-nginx-proxy-companion
    container_name: nginx-le-companion
    restart: unless-stopped
    volumes:
      - ./volumes/nginx/conf.d:/etc/nginx/conf.d
```

```
      - ./volumes/nginx/vhost.d:/etc/nginx/vhost.d
      - ./volumes/nginx/html:/usr/share/nginx/html
      - ./volumes/nginx/certs:/etc/nginx/certs:rw
      - /var/run/docker.sock:/var/run/docker.sock:ro
    environment:
      - NGINX_DOCKER_GEN_CONTAINER=nginx-gen
      - NGINX_PROXY_CONTAINER=nginx
    networks:
      - main

  ngxhome-db:
    build: ./volumes/ngxhome-db/
    restart: unless-stopped
    image: ngxhome-db
    container_name: ${PROJECT:-ngxhome}-db
    volumes:
      - ./volumes/ngxhome-db/data/:/data/db/
    # pass user - see run command docs:
https://docs.docker.com/engine/reference/run/#user
    user: "${MONGO_USERID:-2005}:${MONGO_USERID:-2005}"
    environment:
      MONGO_INITDB_ROOT_USERNAME: ${MONGO_USER:?MONGO_USER env var required!} # Required
env var
      MONGO_INITDB_ROOT_PASSWORD: ${MONGO_PASS:?MONGO_PASS env var required!}
    networks:
      - backend

  ngxhome-api:
    build: ./server/
    restart: unless-stopped
    image: ngxhome-api
    container_name: ${PROJECT:-ngxhome}-api
    volumes:
      - ./server/config.json:/home/app/src/config.json
      - ./server/uploads:/home/app/src/uploads/:rw
    environment:
      - VIRTUAL_HOST=api.${BASE_DOMAIN:?BASE_DOMAIN env var required!} # require base
domain env var
      - VIRTUAL_PORT=4000
      - LETSENCRYPT_HOST=api.${BASE_DOMAIN}
      - LETSENCRYPT_EMAIL=admin@${BASE_DOMAIN}
      - NODE_ENV=${ENV}
    links:
      - "${PROJECT:-ngxhome}-db:${PROJECT:-ngxhome}-db"
    depends_on:
      - ${PROJECT:-ngxhome}-db
    networks:
      - main
      - backend
```

```
  ngxhome-website:
    build: ./client/
    restart: unless-stopped
    image: ngxhome-website
    container_name: ${PROJECT:-ngxhome}-website
    environment:
      - VIRTUAL_HOST=www.${BASE_DOMAIN}, ${BASE_DOMAIN}
      - VIRTUAL_PORT=4200
      - LETSENCRYPT_HOST=www.${BASE_DOMAIN}, ${BASE_DOMAIN}
      - LETSENCRYPT_EMAIL=admin@${BASE_DOMAIN}
    depends_on:
      - ${PROJECT:-ngxhome}-api
    networks:
      - main

  ngxhome-mta:
    build: ./volumes/ngxhome-mta/
    restart: unless-stopped
    image: ngxhome-mta
    container_name: ${PROJECT:-ngxhome}-mta
    volumes:
      - ./volumes/ngxhome-mta/conf.d/:/etc/nginx/conf.d
    environment:
      - VIRTUAL_HOST=mail.${BASE_DOMAIN}, autodiscover.${BASE_DOMAIN},
autoconfig.${BASE_DOMAIN}, smtp.${BASE_DOMAIN}
      - VIRTUAL_PORT=80
      - LETSENCRYPT_HOST=mail.${BASE_DOMAIN}, smtp.${BASE_DOMAIN}
      - LETSENCRYPT_EMAIL=admin@${BASE_DOMAIN}
    external_links:
      - mailcowdockerized_nginx-mailcow_1
    networks:
      - main
      - mailcow-network-ref


networks:
  # default / main network
  main:
  # db <-> api
  backend:

  # reference to mailcow network (external) for the mail proxy container
  mailcow-network-ref:
      external:
          name: mailcowdockerized_mailcow-network
```

Note the use of the mailcow-network-ref at the bottom of the docker-compose.yml that is defined as an external network. This way, one can effectively link to another docker-compose file by referencing its defined network “mailcowdockerized-mailcow-network”. Be advised that this

requires a running instance of said network in order for the docker-compose up (-d) to go through.

Additionally, despite the docker-compose file generally using version 3 syntax and functionality, I do use some syntax that would commonly be used under versions < 3, e.g. the volumes definitions:

```
services:
  ngxhome-api:
    volumes:
      - ./server/config.json:/home/app/src/config.json
```

Personally, I find that defining them this way makes it easier to observe the volume-to-container relation per service (container) as well as host-container relation and allows for more fine-grained container level differences.

Since version 3, another way of defining volumes[15] is to define them in a global "named volumes" array and then reference the volume(s) per service:

```
# An alternative way of defining volumes is to list them in the global volumes section
services: [...] # the containers
  ngxhome-api:
    volumes:
      - conf
volumes:
  shared_volume:
    conf:./server/config.json:/home/app/src/config.json
```

5.4 Data persistence and file ownership with Docker containers

I would like to draw your attention to a point of concern regarding file ownership: Under the ngxhome-db you will find that I defined one volume for the "data" folder ("/data/db" within the container) and then passed a user "1004:1004" to the container.

```
  ngxhome-db:
    build: ./volumes/ngxhome-db/
    volumes:
      - ./volumes/ngxhome-db/data/:/data/db/
    # pass user to enforce ownership on data volume
    user: "1004:1004" # corresponds to user docker_mongodb on host
    [...]
```

[15] Docker docs on volumes: https://docs.docker.com/compose/compose-file/#volumes

Working with docker images you will notice, that when making changes to a containers definition and structure, the old image becomes obsolete and will be overridden when booting up the docker setup the next time. This may be useful for self-contained structures such as an API that does not store data, however for our database, we want to keep any data stored within the database in case we make changes to the container, hence we bind our data folder on the host to the mongo database within the container.

In order for that “ngxhome-db” container to be able to access and write to the data folder, one could set the volumes definition to be of access-type read-write (rw) instead of the default read-only (ro). This won’t work in the ngxhome-db case unfortunately, as the data folder that will be referenced as a volume has one specific user on the host machine that is not the root user. We will instead pass a “user” parameter to the container of type “1004:1004” which corresponds to the host-machine owner of the mongodb data directory “docker_mongodb”. This is the user on the host:

```
root@ngxhome:/home/ngxhome/volumes/ngxhome-db# ls -al
drwxr-xr-x 3 root           root           4096 Jul 25 17:42 .
drwxr-xr-x 8 root           root           4096 Aug 22 22:00 ..
drwxr-xr-x 4 docker_mongodb docker_mongodb 4096 Nov 25 17:15 data
-rw-r--r-- 1 root           root            121 Jul 25 17:42 Dockerfile
```

Now let me close my thoughts by showing the resultant container permissions. Using the docker execute (exec) command we can run arbitrary commands within the running container. Note how the db folder which contains the database files within the container is owned by “1004:1004”:

```
root@ngxhome:/home/ngxhome# docker exec -it ngxhome-db ls -al data
drwxr-xr-x 4  root    root    4096 Jul 6 2017 .
drwxr-xr-x 49 root    root    4096 Feb 19 17:36 ..
drwxr-xr-x 2  mongodb mongodb 4096 Feb 19 17:36 configdb
drwxr-xr-x 4  1004    1004    4096 Feb 19 17:38 db
```

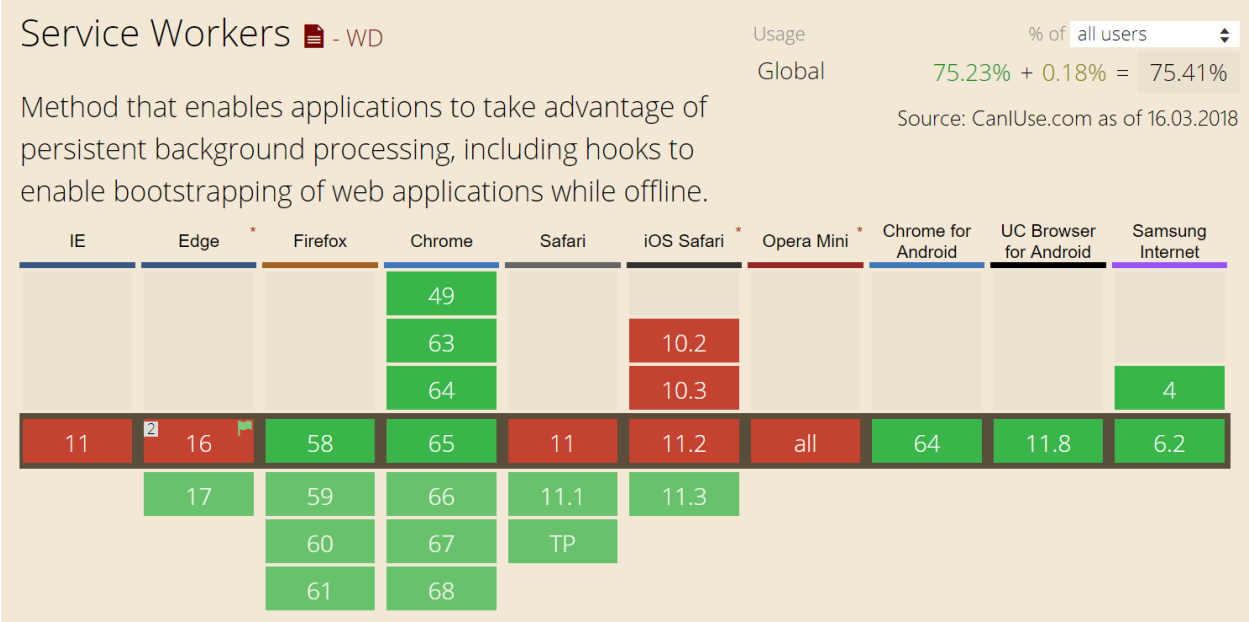
Of course, this is only necessary if you need to persist the data within the database container over several container iterations and want the host to have access to the mongodb data (db), which is particularly helpful during development. Alternatively, for production setups, one could keep the database data sealed to the outside (host) and perform manual backups and when updates to the container were to be done, re-import the mongodb data into the new container iteration.

6. Service Workers, Notifications and Push API.

6.1 Providing offline-support through intelligent caching techniques

Modern websites face a multitude of challenges: From flaky network connections to device and processing constraints. Additionally, user devices vary, sometimes greatly, as regards to their screen space and browser functionality. Users expect websites to function on mobile devices and tablets just like they should in a desktop environment. Progressive Web Apps (PWA) seek to bridge that gap between desktop and mobile (and even native) by rapidly introducing new frameworks, technologies and browser capabilities such as Web Workers[16], Push[17] and Service Worker APIs[18].

Take for instance the case of Service Worker (SW), which seeks to enhance the user experience by providing methods for persistent background processing. In other words, a SW sits in between the user's browser and the content and provides functionality such as push-notifications and offline-support /cache. If for some reason the user's network connection degrades or is lost altogether, the service-worker is capable of providing content to the application from it's cache. Service-workers are now supported by most modern browsers, with some in ongoing development or close to integration (Edge >=17, Safari >11 and iOS Safari >11.2):



[16] Caniuse.com - Web Workers: https://caniuse.com/#feat=webworkers
[17] Caniuse.com - Push API: https://caniuse.com/#feat=push-api
[18] Caniuse.com - Service Workers: https://caniuse.com/#feat=serviceworkers

Integrating Serviceworkers into your application is greatly eased by development and build tools such as the Angular-CLI[19] or Workbox[20]. Personally, I have been building a custom service-worker myself, but I also heavily rely on the Angular-CLI as a bundler and would recommend you start there with your SW integration. Using SWs can be as easy as enabling them in the **.angular-cli.json** configuration file as follows:

```
{
   "apps": [
       {
           "main": "main.ts",
           [ ... ]
           "serviceWorker": true,
       }
   ],
   [...]
}
```

Additionally, you will want to supply a **ngsw-config.json** configuration file:

```
{
   "index": "/index.html",
   "assetGroups": [
       {
           "name": "app",
           "installMode": "prefetch",
           "resources": {
               "files": [
                   "/favicon.ico",
                   "/index.html"
               ],
               "versionedFiles": [
                   "/*.bundle.css",
                   "/*.bundle.js",
                   "/*.chunk.js"
               ]
           }
       },
       {
           "name": "assets",
           "installMode": "lazy",
           "updateMode": "prefetch",
           "resources": {
               "files": [
                   "/assets/**"
               ],
```

[19] Angular-CLI on Github: https://github.com/angular/angular-cli
[20] Workbox - tooling for caching and offline-support: https://developers.google.com/web/tools/workbox/

```
            "urls": [
                "https://fonts.googleapis.com/**",
                "https://fonts.gstatic.com/**"
            ]
        }
    }
  ],
  "dataGroups": [
    {
      "name": "from-api",
      "urls": [
        "/blogs",
        "/cvs"
      ],
      "cacheConfig": {
        "strategy": "freshness",
        "maxSize": 15,
        "maxAge": "1h",
        "timeout": "5s"
      }
    }
  ]
}
```

This configuration will enable the CLI to dynamically generate a service-worker configuration file from your specific bundle setup. There are many possible configuration parameters, but let's look at the 3 most important elements to configure:

1. **Asset groups to prefetch**
   - ➔ These are the files required for your app shell and installed during the initial load.
   - ➔ They are defined under "assetGroups" with the "**prefetch**" "installMode"
2. **Asset groups to lazy-load**
   - ➔ These files may be needed anywhere in your bundle, for instance any (external) fonts ("urls" mapping with the ** wildcard operator) or assets ("files" mapping).
   - ➔ They are also defined under "assetGroups" but with the "**lazy**" "installMode"
3. **Data groups (API cache)**
   - ➔ This is the category that we use to define external data requests such as the ones made to our backend API or third-party data providers.
   - ➔ They are defined as "**urls**" under "dataGroups"

That's it! During (production) build, the Angular-CLI will now generate a configuration file from the options we gave in the ngsw-config and integrate the actual service-worker file (ngsw-worker.js) into the bundle.

6.2 Notifications with the Push API.

Web-push enables a server to push notifications and data to the client application, even if it is not currently open on the users device. Notifications are closely tied to the aforementioned service-workers, in that the service-worker is responsible for handling the data that is being sent from a push service.

In the following I will describe how I implemented push notifications into both Angular and my Express server based backend. The implementation is slightly more complicated, as it consists of not only the Push API, but also the Notifications API and the ServiceWorker (SW). On the backend we will be working with the web-push library from: https://github.com/web-push-libs/web-push.

6.2.1 Enhancing the ngsw-worker.js to handle notification clicks

Following through with our Angular-CLI SW front-end implementation, it is important to note that, as of the time of writing this, the ngsw-worker.js that is being generated from the CLI does not include the event listeners of type “notificationclick” and “notificationclose” that are required to facilitate the handling of notification clicks on the clients’ end. If we deliver a push notification to a user, we obviously want him to be able to actually click on the notifications as well, which will usually contain not only a text message but also some link or set of actions (effectively links). Fortunately it is fairly trivial to manually adapt the SW to our push use-case with notification clicks:

1. You will want to make a copy of the **ngsw-worker.js**, either from dist after performing a build or the @angular/service-worker node_module source.
2. In this ngsw-worker.js, search for the term: `this.scope.addEventListener('push'`. Now, in the next line we add our custom event listener code:

```
// Custom event listeners to add to the ngsw-worker.js to handle notification clicks
this.scope.addEventListener('notificationclick', function (event) {
    console.log('[SW] notificationclick: ', event)
    event.notification.close()

    function onNotificationClick(event) {
        // See if the current url is already open and if so, focus it
        var urlToOpen = '';
        if (event.notification.data
            && event.notification.data.url
            && event.notification.data.target_urls) {
            // data, url and target_urls objects exist
            if (event.action) { // an action event was clicked
                urlToOpen = event.notification.data['url']
                            + event.notification.data.target_urls[event.action]
            } else { // the notification itself (default action) was clicked
                urlToOpen = event.notification.data['url']
                            + event.notification.data.target_urls['default-action'];
            }
```

```
        } else { // when data is unavailable, use the SW's origin instead
            urlToOpen = self.location.origin;
        }

        event.waitUntil(
            clients.matchAll({
                includeUncontrolled: true,
                type: 'window'
            }).then(function (clientList) {
                for (var i = 0; i < clientList.length; i++) {
                    var client = clientList[i]
                    if (client.url == urlToOpen && 'focus' in client)
                        return client.focus()
                }
                if (clients.openWindow)
                    return clients.openWindow(urlToOpen)
            })
        )
    }

    // handle different types of clicks
    if (!event.action) {
        // regular notification click (no action)
        onNotificationClick(event);
        return;
    }
    switch (event.action) {
        case 'yes-action':
            console.log('User clicked yes.');
            onNotificationClick(event);
            break;
        case 'no-action':
            console.log('User clicked no.');
            onNotificationClick(event);
            break;
        default:
            console.log(`An undefined action was clicked: '${event.action}'`);
            break;
    }
})

this.scope.addEventListener('notificationclose', function (event) {
    console.log('Notification was closed without click or action.')
})
// end of custom notification handlers
```

This set of ‘notificationclick’ and ‘notificationclose’ event listeners is mildly opinionated and you may find that a different setup suits your needs. Be advised that the implementation also depends on which push service you use and how the data your push service delivers is structured. In the following paragraph I will show an exemplary backend push service.

6.2.2 Setting up a push service

Depending on your setup, it may be easier to use firebase or other third party services as your push service. Generally, a custom integration consists of 2 steps:

1. Generating a VAPID keypair which is used to identify and encrypt the push data you will send to your clients.
   Wherever you initialize your database or database models, you will want to generate this keypair once.[21] As I am using Mongoose, I do that after the database initialization / provisioning **settings.model.js** file, by running:

   ```
   var webpush = require('web-push'); // https://github.com/web-push-libs/web-push
   const vapidKeys = webpush.generateVAPIDKeys();
   ```

   and then storing the `vapidKeys.publicKey` and `vapidKeys.privateKey` in the database.

2. Implementing your push service into the API.
   We begin by defining a test payload (json) structure: **pushPayloadExample.json**

   ```
   {
      "notification": {
          "title": "Serviceworker notification",
          "actions": [
              { "action": "yes-action", "title": "Yes!" },
              { "action": "no-action", "title": "No!" }
          ],
          "body": "This is a test notification text body",
          "dir": "auto",
          "icon": "/icons/android-chrome-72x72.png",
          "badge": "/icons/android-chrome-36x36.png",
          "lang": "en",
          "renotify": true,
          "requireInteraction": true,
          "tag": "test-tag-1",
          "vibrate": [ 300, 100, 400 ],
          "data": {
              "url": "https://www.aferring.de",
              "created_at": "Mo Jan 01 01:01:01 +0001 2018",
              "target_urls": {
                  "default-action": "/",
                  "yes-action": "/blogs",
                  "no-action": "/no"
              }
          }
      }
   }
   ```

[21] With the 'web-push' package installed globally ('npm i -g web-push') you can also run the 'web-push generate-vapid-keys' command in your shell to get a VAPID keypair.

Now, somewhere in your API backend code, for instance in a **push.service.js** you will want to import the sample payload and set up your push functionality as follows:

```
# push.service.js (extract)

var webpush = require('web-push');
var pushPayloadJSON = require('../commons/pushPayloadExample.json');

const pushSubscription = {
    endpoint: push.endpoint,
    TTL: 60,
    keys: {
        auth: push.keys.auth,
        p256dh: push.keys.p256dh
    }
};

pushPayloadJSON.notification.data['created_at'] = new Date();
const pushPayload = JSON.stringify(pushPayloadJSON);

const pushOptions = {
    vapidDetails: {
        subject: 'mailto:push@aferring.de',
        publicKey: vapidPublicKey,
        privateKey: vapidPrivateKey,
    },
    TTL: 60
};

webpush.sendNotification(pushSubscription, pushPayload, pushOptions)
    .then(() => { pushPromise.resolve('[a-push] Tickle() done'); })
    .catch(error => {
        console.warn('[push] sendNotification() error: \n' + error);
        pushPromise.reject('[push] sendNotification() error: \n' + error);
    });
```

This **push.service.js** is tasked with sending a test notification to the **push.endpoint**. You will have to supply this endpoint as well as the **push.keys.auth** and **push.keys.p256dh** from the front-end. (More on this in the next chapter!) Additionally, the push.service will need the **vapidPublicKey** and **vapidPrivateKey** we stored in our database (see step 1). Lastly, note that throughout the Express server we are using a promise library to resolve or reject backend requests asynchronously. Whichever one you utilize should not affect your push.service implementation.

6.2.3 Integrating the Push and Notification API in an Angular application

Finally, we will now integrate the Push and Notifications API in our front-end Angular application to be able to create a subscription object that our backend needs to deliver the push messages (using the push.endpoint and push.keys as described above). I will show only an extract of my service code that may not include all necessary “progressive enhancement” techniques that are required to detect whether the client’s browser is actually capable of utilizing these APIs. Please consult, for instance the google developers documentation[22] for a more thorough explanation of the push process and involved libraries.

We will begin by setting up a **serviceworker.service.ts** that is used to request a subscription object from the SW and handle the Notification API state. You want to make sure that this service has access to the VAPID public key. Establish some means of access, e.g. via environment variables, or ideally transferring the VAPID public key from the server to your front-end via http request. In this example we will set an exemplary **vapidPublicKey** property on the service statically. Note that for the sake of brevity I will not show unsubscription functionality:

```
# serviceworker.service.ts

import { Injectable } from '@angular/core';
import { isPlatformBrowser } from '@angular/common';
import { PushService } from './push.service';
import { ServiceLocator } from '../../shared/service-locator';

declare var Notification;

@Injectable()
export class ServiceWorkerService {
    private vapidPublicKey =
'BImM-XJR9Dyoy5udQ3gzAWxXV9bs76x4NBdjLdTiDSLRkhOMaaobqOoJxxZ3hE71UHj5XqxUkUpDhCILQiswBEY';

    private registration: ServiceWorkerRegistration = undefined;
    private pushSubscription: PushSubscription = undefined;
    public pushEndpoint: PushSubscription['endpoint'] = undefined;
    public isNotificationAllowed = false;
    public isSubscribedSW = false;

    constructor( private pushService: PushService ) {
        if (this.getNotificationPermission() === 'granted') {
            this.isNotificationAllowed = true;
        };
    }
```

[22] Google push101: https://developers.google.com/web/fundamentals/push-notifications/how-push-works
Subscription: https://developers.google.com/web/fundamentals/push-notifications/subscribing-a-user

```
public initializePush(): Promise<boolean> {
    return this.getSWRegistration().then(() => {
        return this.getSWSubscription().then(subscription => {
                    if (subscription === null) {
                        return Promise.resolve(false);
                    } else {
                        // check API for existence of subscription
                    }
                }).catch(err => {
                    console.log('[SW.service] initializePush() Error: ', err);
                    return false;
                });
        }).catch(err => {
            console.log('[SW.service] initializePush() Error: ', err);
            return false;
        });
}

public getNotificationPermission(): string {
    return ('Notification' in window) ? Notification.permission
                                      : 'Notifications are not supported';
}

public doDisplayNotification(title: string, options: any): Promise<void> {
    if (this.registration !== undefined
        && this.getNotificationPermission() === 'granted') {
        return this.registration.showNotification(title, options)
            .catch(err => {
                console.log('[SW.service] Error displaying notification: ', err);
            });
    }
}

public doCheckNotificiationPermission() {
    if (Notification.permission === 'granted') {
        this.isNotificationAllowed = true;
    } else if (Notification.permission === 'blocked') {
        this.isNotificationAllowed = false;
    } else { // prompt for permission
        Notification.requestPermission(function (permission) {
            permission === 'granted' ? this.isNotificationAllowed = true
                                     : this.isNotificationAllowed = false;
        });
    }
}
```

```
public doSubscribePush(): Promise<boolean> {
    if (this.registration !== undefined) {
        if (this.isSubscribedSW) {
            // already have a subscription, only sub on server
            return this.serverSubscribeEndpoint(this.pushSubscription);
        } else {
            // no sub object, get a new subscription on front and then send to back
            const applicationServerKey = this.urlB64ToUint8Array(this.vapidPublicKey);
            return this.registration.pushManager
                .subscribe({
                    userVisibleOnly: true,
                    applicationServerKey: applicationServerKey
                })
                .then(subscription => {
                    this.pushSubscription = subscription;
                    this.isSubscribedSW = true;
                    this.pushEndpoint = subscription.endpoint;
                    console.log('[SW.service] User is now subscribed!');
                    return this.serverSubscribeEndpoint(subscription);
                })
                .catch(err => {
                    return Promise.reject('Failed to subscribe the user!');
                });
        }
    } else { return Promise.reject('doSubscribePush(): '); }
}

private urlB64ToUint8Array(base64String): Uint8Array {
    const padding = '='.repeat((4 - base64String.length % 4) % 4);
    const base64 = (base64String + padding).replace(/\-/g, '+').replace(/_/g, '/');
    const rawData = window.atob(base64);
    const outputArray = new Uint8Array(rawData.length);
    for (let i = 0; i < rawData.length; ++i) {
        outputArray[i] = rawData.charCodeAt(i);
    }
    return outputArray;
}

private getSWRegistration(): Promise<ServiceWorkerRegistration> {
    if ('serviceWorker' in navigator) {
        return navigator.serviceWorker.getRegistration().then(registration => {
                    if (registration !== null && registration !== undefined) {
                        console.log('[SW.service] SW registration: ', registration);
                        this.registration = registration;
                    } else {
                        return Promise.reject('SW registration undefined');
                    }
                });
    } else { return Promise.reject('No \'serviceWorker\' in navigator'); }
}
```

```
    private getSWSubscription(): Promise<PushSubscription> {
        if ('PushManager' in window && this.registration !== undefined) {
            return this.registration.pushManager.getSubscription()
                .then(subscription => {
                    if (subscription === null) {
                        this.pushSubscription = subscription;
                        this.isSubscribedSW = false;
                        return Promise.resolve(null);
                    } else if (subscription !== undefined) {
                        this.pushSubscription = subscription;
                        this.isSubscribedSW = true;
                        this.pushEndpoint = subscription.endpoint;
                        return Promise.resolve(this.pushSubscription);
                    } else {
                        return Promise.reject('Subscription is undefined');
                    }
                }).catch(err => {
                    return Promise.reject('getSWSubscription(): ' + err);
                });
        } else { return Promise.reject('No \'PushManager\' in window'); }
    }

    private serverSubscribeEndpoint(subscription: PushSubscription): Promise<boolean> {
        return this.pushService.subscribe(subscription, userid)
            .then(res => {
                // console.log('[SW.service] Push subscription succeeded: \n', res);
                this.isSubscribedServer = true;
                return true;
            })
            .catch(err => {
                console.log('[SW.service] Push subscription request failed', err);
                this.isSubscribedServer = false;
                return false;
            });
    }
}
```

Using this serviceworker.service you will want to set up a component to facilitate the notification process. One way would be for the user to manually click a button that prompts him to grant notification permissions and shows his willingness to receive push notifications via SW.

In a production setup, you should find both an unintrusive way and point at which to poll the user for his approval, which is generally not the landing page of your app. It is advisable to implement a mechanism of gathering existing SW subscriptions for repeat visits (in this service example the initialize() method). If possible, i.e. when you have user accounts and login states that allow you to keep track of approval for a given user, you will probably want to enable push on multiple devices for that user. This may require you to re-prompt for notification permissions on any new device, but will usually allow you to programmatically gather ServiceWorker subscriptions.

Lastly, we will define a **push.service.ts** that is tasked with sending the push subscription object to our backend server and triggering a test via the 'ticklePush()' method (see also 5.2.2):

```
# push.service.ts
import { Injectable, Injector, InjectionToken } from '@angular/core';
import { Http } from '@angular/';
import { BaseService } from '../../shared/base.service';
import { Push } from './push';
import { isPlatformServer } from '@angular/common';

@Injectable()
export class PushService extends CommonHandler {
   /**
    * Subscribe a push endpoint.
    * @param subscription containing: endpoint, expirationTime and keys{ p256dh,auth }.
    */
   public subscribe(subscription: PushSubscription): Promise<Push> {
       return this.http
           .post(this.getBaseUrl() + this.getServiceIdentifier() + '/subscribe/',
               {
                 subscription: subscription,
                 navigator: { userAgent: navigator.userAgent, vendor: navigator.vendor }
               }
           ).toPromise()
           .then(response => response.json() as Push)
           .catch(this.handlePromiseError);
   }

   /**
    * Tickle (=test) the push service by requesting a push notification to an endpoint.
    * @param endpoint reqbody: the endpoint to tickle to.
    */
   public ticklePush(endpoint: string): Promise<boolean> {
       return this.authHttp
           .post(this.getAdminUrl() + this.getServiceIdentifier() + '/tickle/',
              { endpoint: endpoint })
           .toPromise()
           .then(this.handleResponseStatusCode)
           .catch(this.handlePromiseError);
   }
}
```

I discussed the mechanics on the front-end of requesting notification permissions, requesting a subscription object from the ServiceWorker and sending it to the back-end via a push.service. For the API, I showed an example of generating the required VAPID keys and sending a push notification to an endpoint. In closing, I do hope this brief example will aid you in implementing push notifications in your application. If you need help or if you want to give feedback related to content or errors on my part, please feel free to contact me via email or using the contact form.

7. Using the IntersectionObserver API to detect element-view intersections

An essential tool to detect which elements are about to enter the view. Useful for pre- /lazy-loading mechanics.

I recently saw the latest 2017 Chrome Dev Summit, more particularly a presentation about "Progressively Improving E-Commerce" which sparked my interest in regards to detecting page elements coming into view. So far, to detect whether an element was visible (or close to becoming visible) I utilized a directive to detect the elements distance from the top as well as the views scroll position.

If you have ever worked with different browsers you are sure to know the pain that is making accurate assessments as to dimensions and positions in a multi-device environment. My hacky solution used to work well in all common desktop browsers, but as soon as smartphones and tablets came into play, the results were a lot less reliable. I used to extend my directive to detect the device from the *navigator.userAgent* string using regular expressions, for instance:

```
    if (/(android|bb\d+|meego).+mobile|avantgo|bada\/|blackberry|blazer|compal|
        [...] |xiino/i.test(navString) || [...]) {
            // perform downgrade
    }
```

While this certainly works to gradually downgrade app functionality, it is sometimes unreliable and needs to constantly be revisited for changes as new devices /exceptions emerge in the market. A standardized API is both more capable, performant and platform-agnostic (provided that all browser vendors had time to implement and agreed on the integration).

The following directive utilizes the **Intersection Observer API**[23] to solve our problem of detecting whether an element enters or is about to enter the viewport:

```
# IntersectionDirective

import { ElementRef, Output, Directive, OnDestroy, EventEmitter, Input, Inject,
PLATFORM_ID } from '@angular/core';
import { isPlatformBrowser } from '@angular/common';

@Directive({
    // tslint:disable-next-line:directive-selector
```

[23] See also: https://developer.mozilla.org/en-US/docs/Web/API/Intersection_Observer_API

```
    selector: '[intersection]'
})
export class IntersectionDirective implements OnDestroy {
    @Output() intersection: EventEmitter<void>;
    @Input() rootMargin = 0;
    iO: IntersectionObserver;
    element: Element;

    constructor(
        private elementRef: ElementRef,
        @Inject(PLATFORM_ID) private platformId: Object
    ) {
        this.intersection = new EventEmitter<void>(false);
        this.element = this.elementRef.nativeElement;
        if (isPlatformBrowser(this.platformId)) {
            this.registerIO();
        } else { // for server renders bypass the iO
            this.intersection.emit();
        }
    }

    ngOnDestroy() {
        if (this.iO) {
            this.iO.unobserve(this.element);
            this.iO = null;
        }
        this.intersection = null;
    }

    registerIO() {
        const iOOptions = {
            rootMargin: `0px 0px ${this.rootMargin}px 0px`,
        }
        this.iO = new IntersectionObserver(this.handleIntersectionUpdate, iOOptions);
        this.iO.observe(this.element);
    }

    private handleIntersectionUpdate = (entries: IntersectionObserverEntry[]): void => {
        for (const entry of entries) {
            if (entry['isIntersecting']) {
                this.intersection.emit();
            }
        }
    }

}
```

This directive will emit an event (intersection: EventEmitter<void>) to its' host component when the IntersectionObserver observes the elementRef.nativeElement (=this.element) to be visible (or with a "rootMargin" set, distant by that set amount).

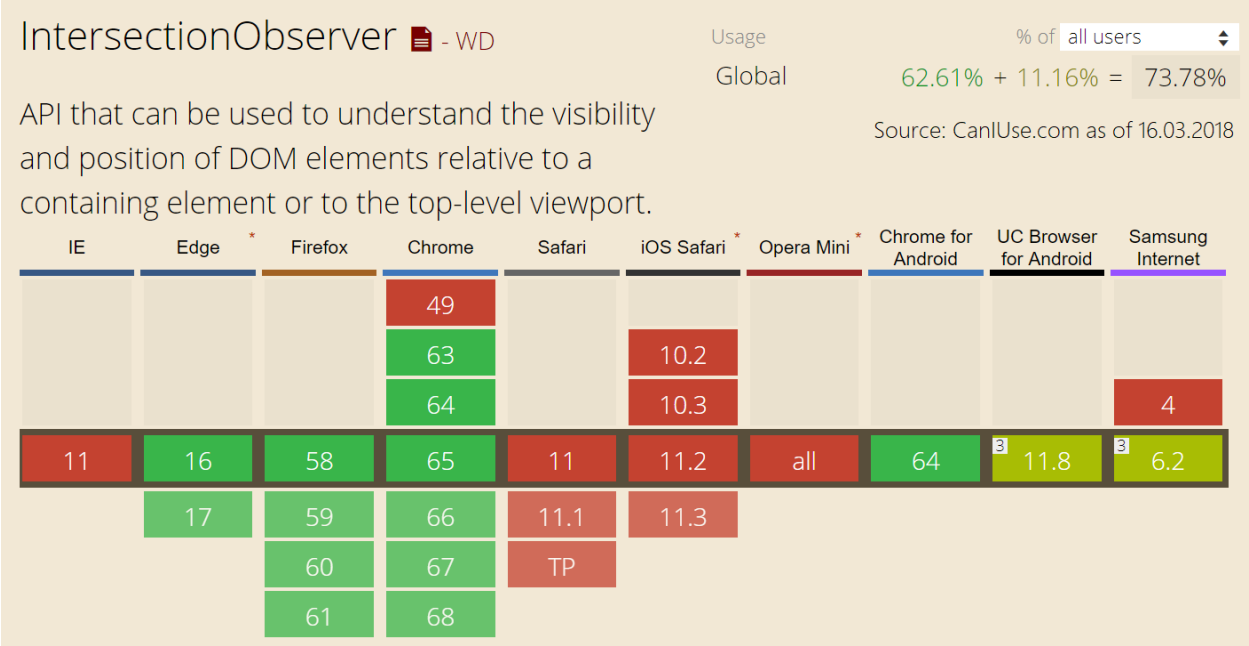
Within the host component you can now reference this directive like so:

```
<div class="col" (intersection)="onColAppear()"></div>
```

The event-binding (“intersection”) will trigger when the directive emits and call the onColAppear() method within the hosting component, that you can use e.g. to trigger animations. The API is much more capable than what I could show you in this chapter. For instance, it can be used to dynamically assign a source property to images that are about to enter the view. This can be very useful especially for huge lists of images /elements that you want to lazy-load in and or that you want to be more performant by streaming into view (and vice versa removing elements that leave the viewport). For an implementation of said use-case, please also see the following video and blog by Ben Nadel: [Lazy Loading Images With The IntersectionObserver API In Angular 5.0.0]

Please be advised that the Intersection Observer is not yet implemented in all browsers, hence requiring a polyfill[24] that can be obtained from npm: `npm install intersection-observer`

Now (with Angular-cli) import it in polyfills.ts or otherwise, depending on your setup, import it as an Html script or ‘require it’ for your module loader of choice. Fyi: The current adaption of IntersectionObserver[25] by the major browsers is as follows:



[24] IntersectionObserver polyfill: https://github.com/w3c/IntersectionObserver/tree/master/polyfill
[25] Caniuse.com - InterSectionObserver: https://caniuse.com/#feat=intersectionobserver