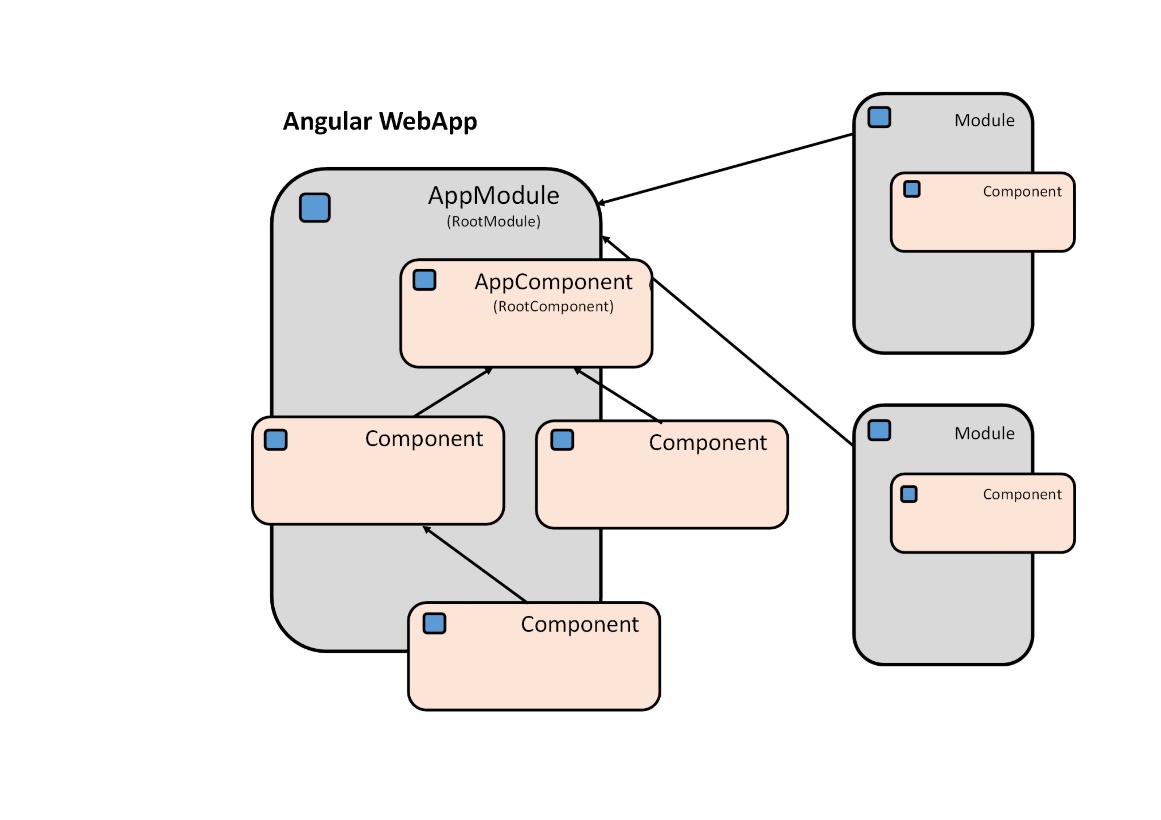
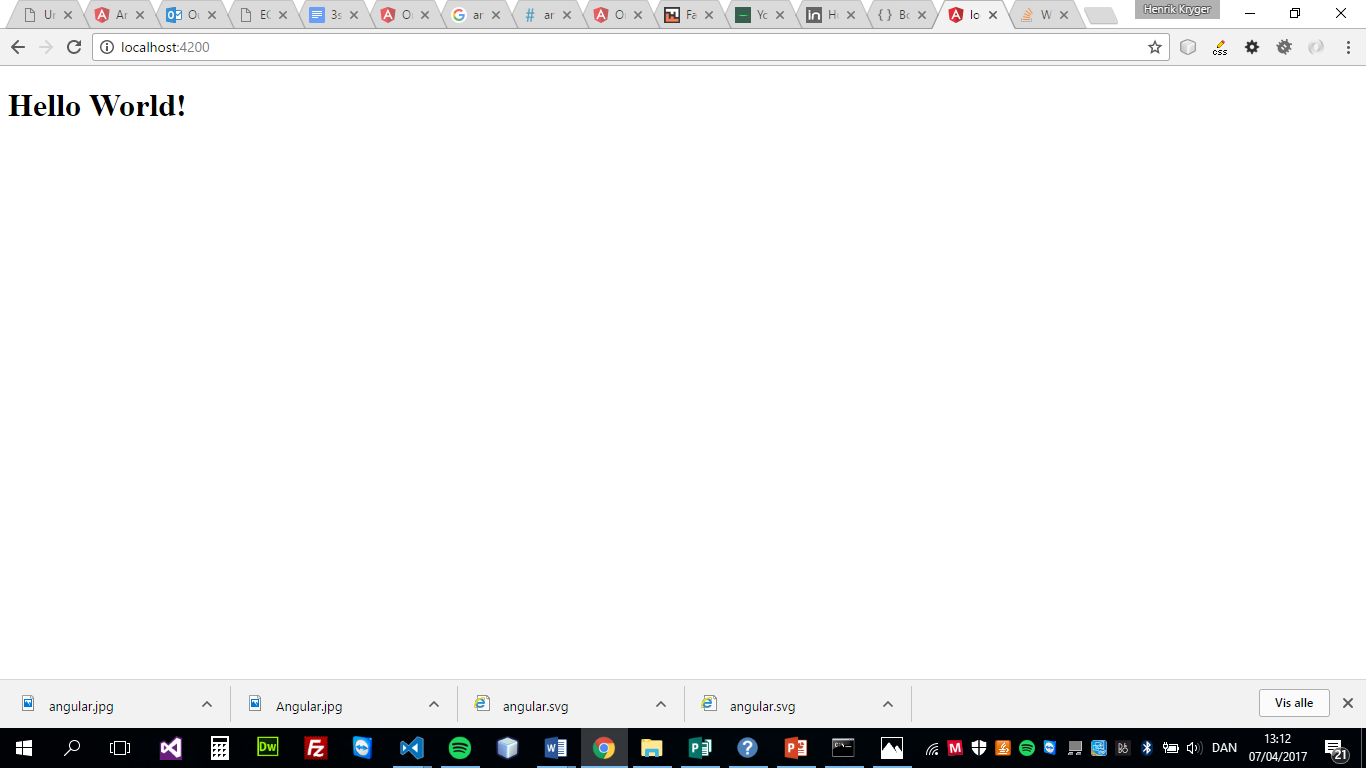
 **ngular – the PIXI Book  *Henrik Kryger Høltzer, Zealand-Roskilde October 2017***

**Overview**

Angular WebApps is based on modules and Angular has its own module system called Angular modules (or NgModules). All Angular app has at least one module, the *root module*, and one component, the *root component* - by convention called the *AppModule* and the *AppComponent*.   
 *Fig.1: Angular WebApp overview*

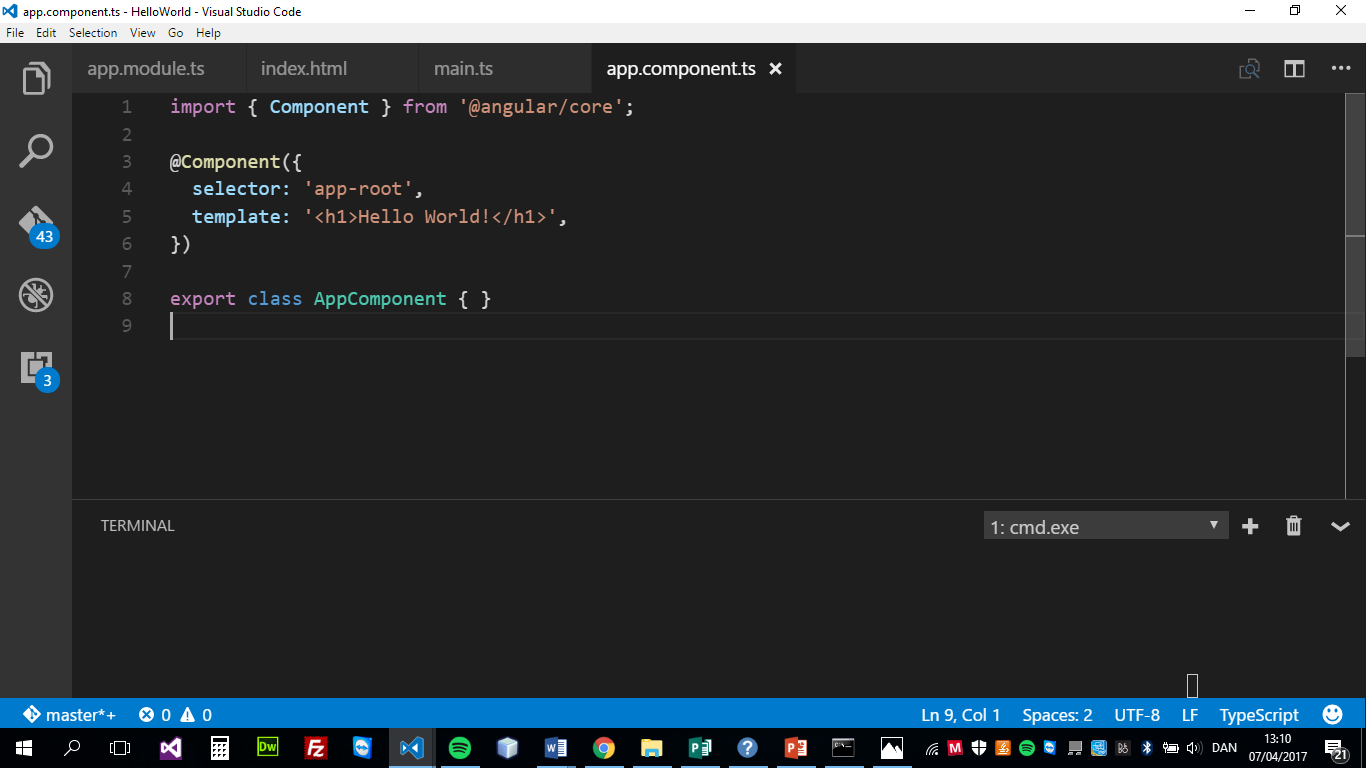
The Angular WebApp is launched by bootstrapping the *root module,* then Angular takes over and presents the app content defined in the components. Let us have a look at the simplest Angular app with the famous Hello-world example:



*Fig.2: HelloWorld – launched in the Browser*

In this simple example, we have four files: main.ts, index.html, app.module.ts and app.component.ts. Angular is written in TypeScript therefore the file-extension .ts. The filenames app.module.ts and app.component.ts are given by convention since the files contain a module and a component with the name App.

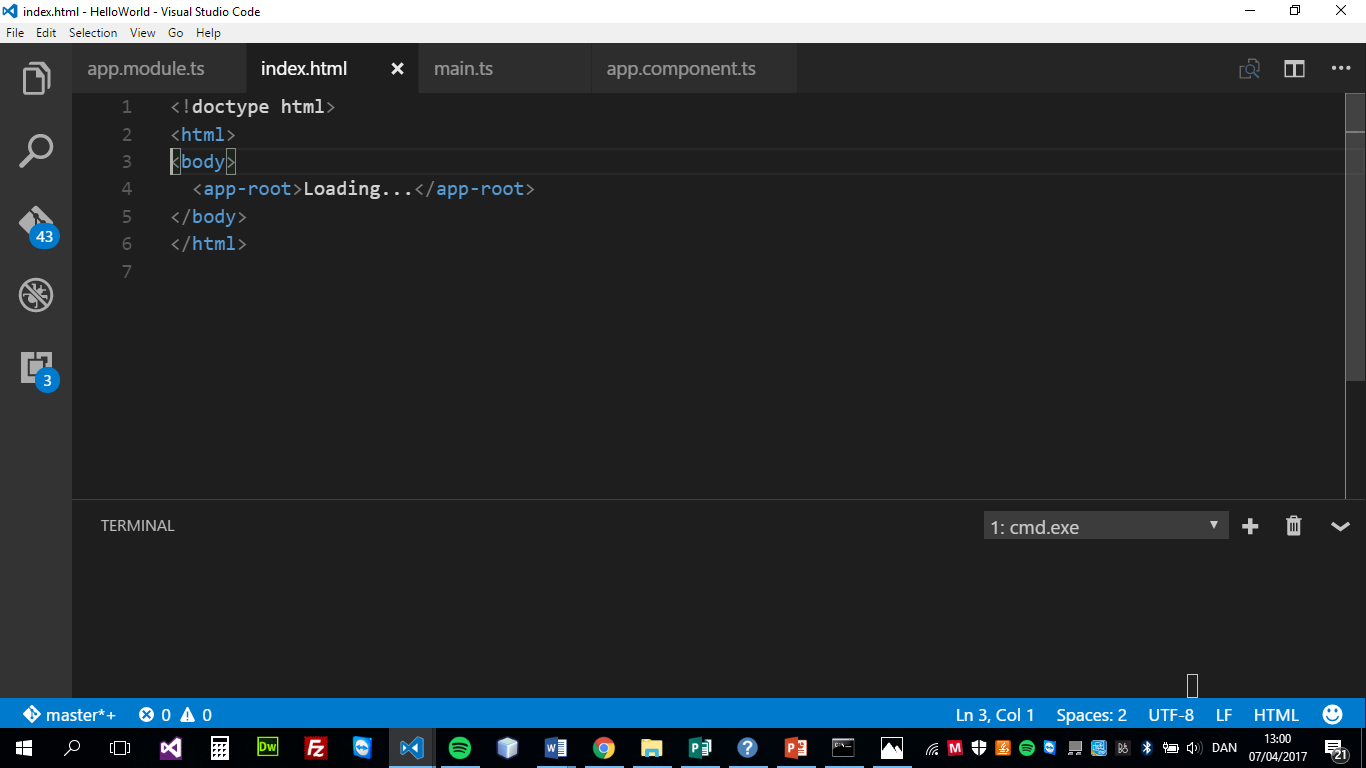
**app.component.ts**In this file we find the root component. All components in Angular are classes annotated with the decorator @Component. The component decorator itself is a function defined in the Angular core module, and we have to import it before use.



*Fig. 3 HelloWorld – app.component.ts*

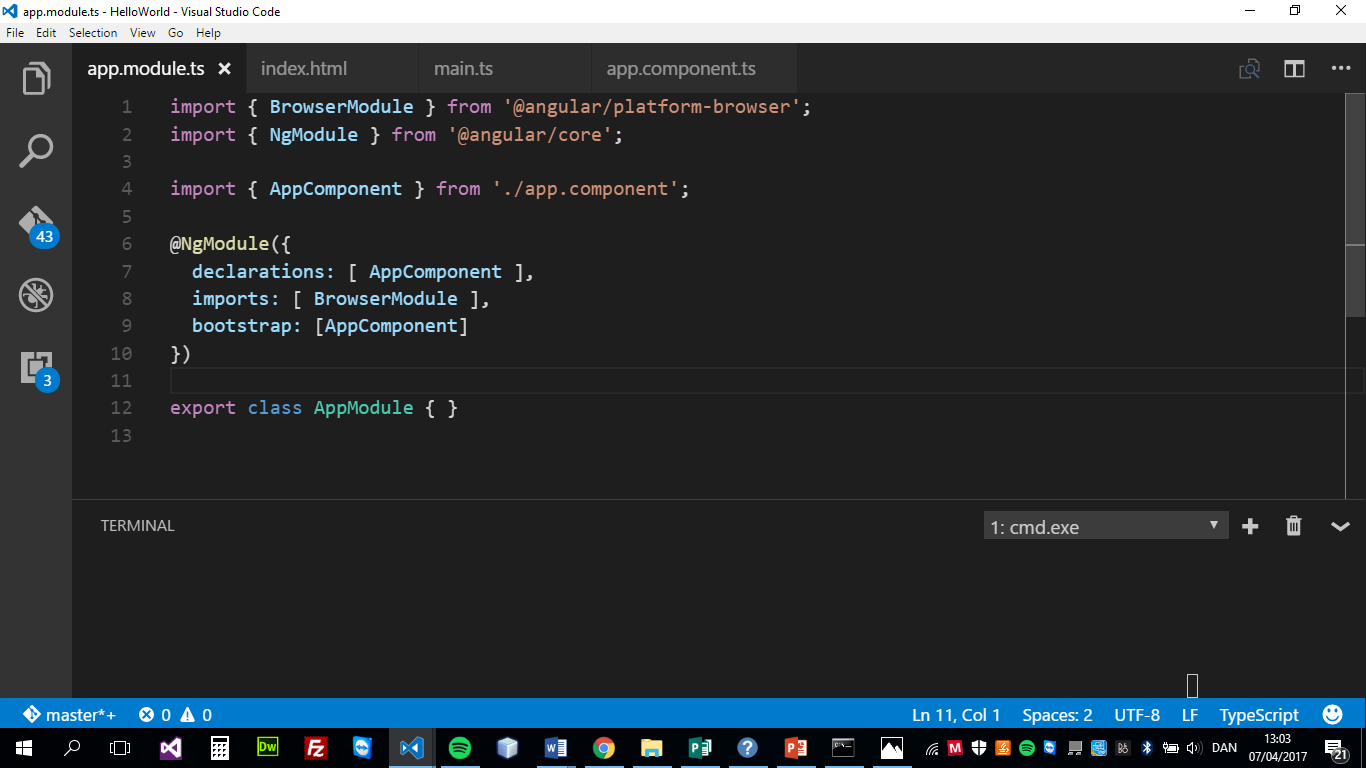
Selector and template are configuration-properties. Selector is a CSS selector that tells Angular to create an instance of the component and insert it into the DOM where it finds the <app-root> tag. The Template contains the html-code that tells Angular how to render the component, in this example <h1>Hello World!</h1> .

**index.html**In the index file we find our html-document with the <app-root>, the tag that Angular substitutes with the template from the root component.



*Fig. 4 HelloWorld – index.html*

**app.module.ts**The root component lies within a module implemented as a class annotated with the @ngModule decorator. The root component has to be imported into the module file together with NgModule (the decorator function defined in Angular core) and the Browser module since the app is a WebApp.



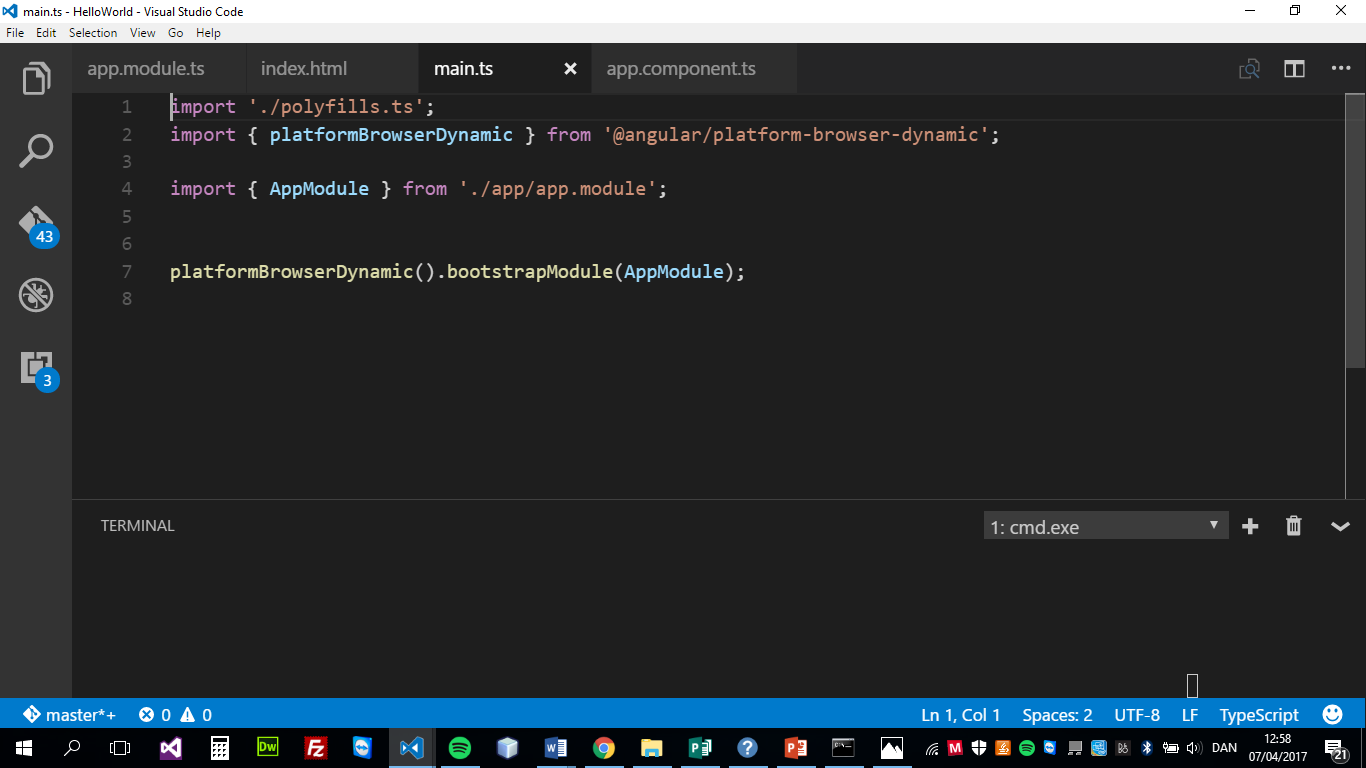
*Fig. 5 HelloWorld – app.module.ts*

The @ngModule decorator function also has some configuration-properties:

* declarations: all the view classes in the module (components, directives and pipes)
* imports: other modules that are needed by component templates declared in this module
* bootstrap: the root component, the main app view that host all the other views.

Finally, we need to export the root module class so it can be imported into the main.ts file and can tell Angular to bootstrap this module.

**main.ts**As mentioned above, we launch the Angular app by bootstrapping the *root module*. Since the app is a WebApp and we bootstrap it into a Browser, we import the Module: *platformBrowserDynamic .* The only function we use from the module is platformBrowserDynamic( ) that allows us to instantiate our app by passing our root module into the bootstrapModule( ).



*Fig. 6 HelloWorld – main.ts*

The import of the polyfills.ts is to compensate for the fact that Angular is built on the latest standards of the web platform and does not support all features of modern browsers.

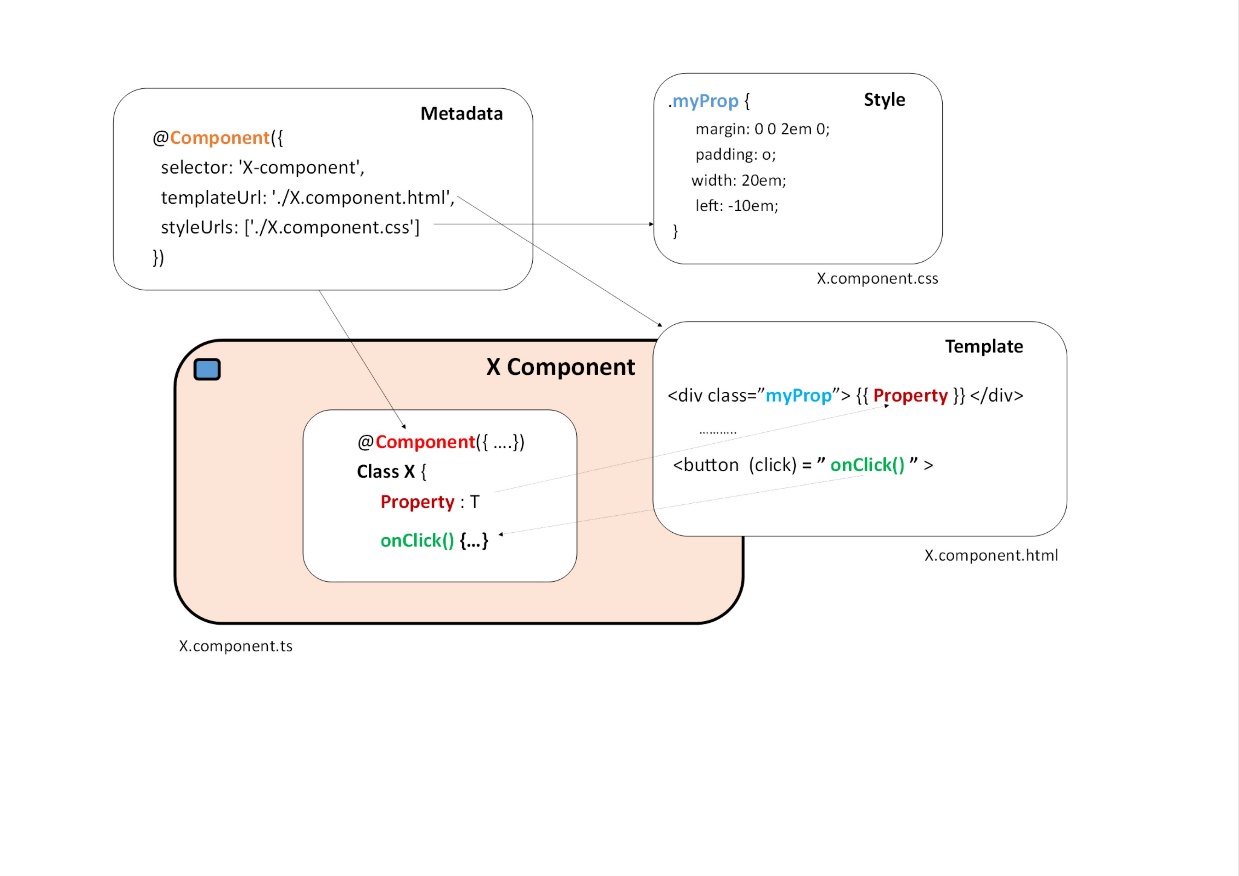
Angular modules vs. JavaScript modules

In Angular we use module’s to combine components (and directives and pipes) together into a single unified unit, a cohesive block of functionality.

JavaScript has its own module system and it is completely different from and unrelated to the module system in Angular. In JavaScript each file is a module and all objects defined in that file belong to the module. Objects in a JavaScript module can be declared to be public, by marking them with the *export* keyword. Other modules can then access the public objects by using the *import* statement. This is very similar to the *import* statement in Java and the *using* statement in C#. All these imports and exports give a lot of dependence between the files and to manage this dependence Angular use WebPack. WebPack solves all the dependency and builds a single file named: *bundle.js*.

The View – Component, metadata and template

A view is a part of the screen. The component class, the metadata and the template with its styling compose it. The metadata tells Angular how to process the class. The CSS-selector makes it possible to select and insert the view in the DOM. The TemplateUrl is a module-relative address of the components HTML template (notice we prefer to put the template in a separate file before the inline style we saw above). The styleUrls is an array of module-relative addresses of the components CSS-files. It is the component class that controls the view, it is in the class we define the application’s logic and make bindings between the template and properties and methods in the class.

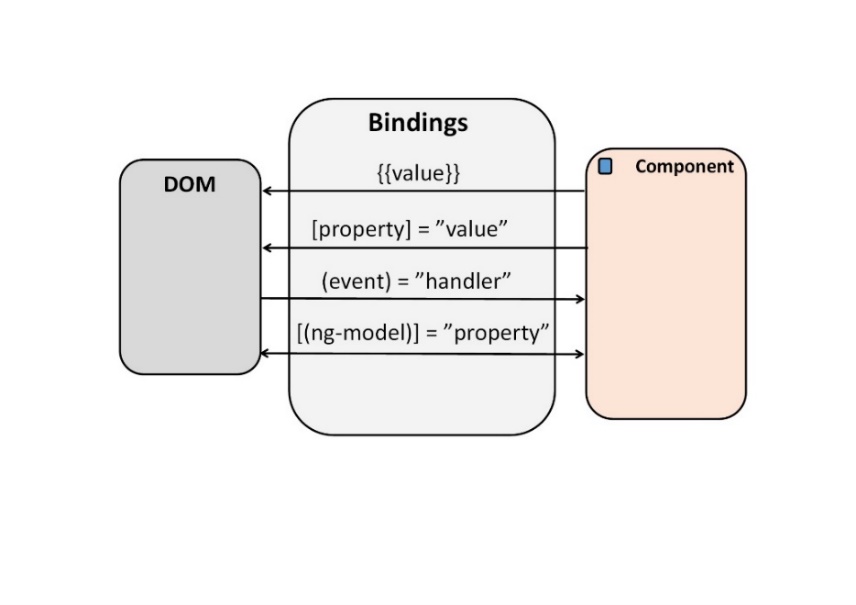


*Fig. 7 The View – Component, metadata and template*

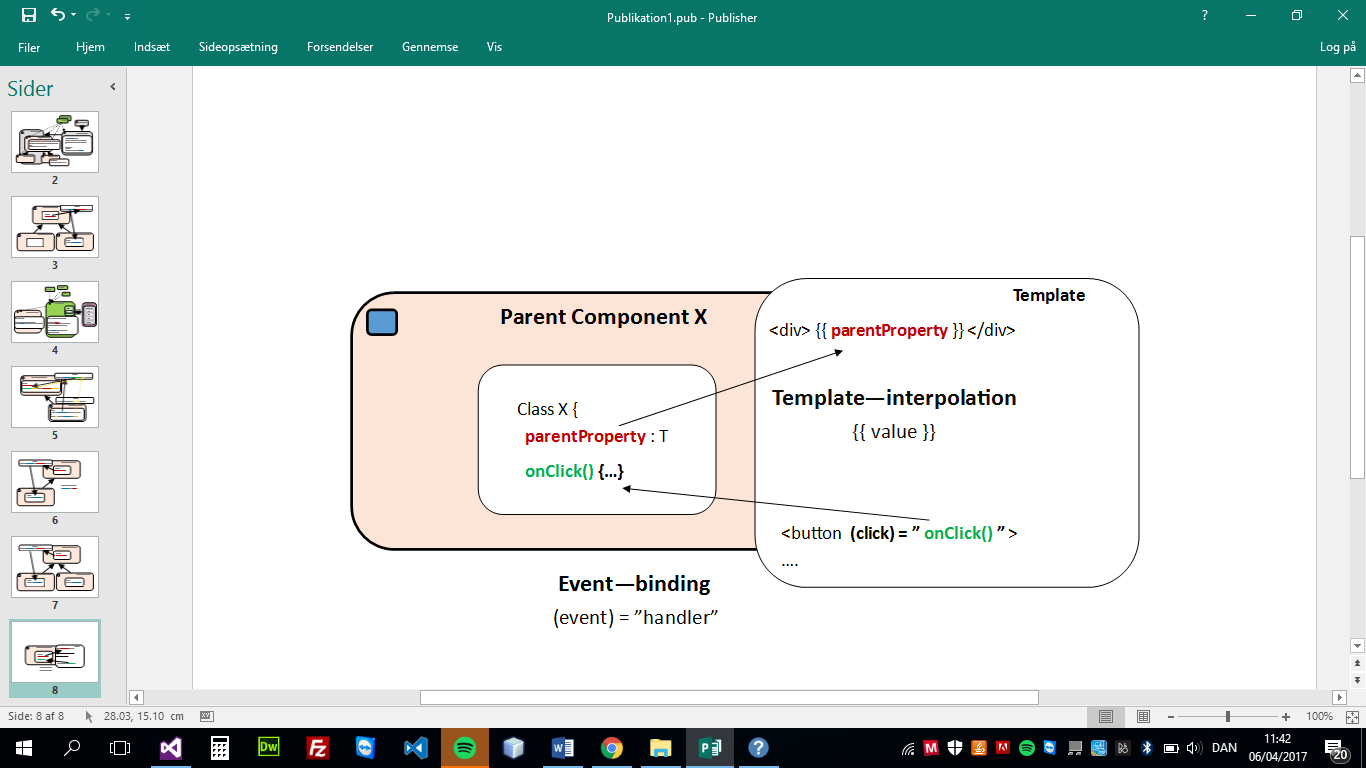
More to come: Something about tree-structure and components…

Bindings  
Data binding is one of the fundamentals in Angular.   
We use bindings in the communication between a view’s template and its component and in the communication between parent and child components. The binding can be either one-way, when we want to display a component property in the view, or two-way, when we need to retrieve values typed in by the user in a form. We also use bindings when we want to make a coupling between events in the view and a components method (event-handlers).

In Angular we have four forms of data binding syntax, illustrated in the following diagram:

*  
 Fig. 8 Bindings, one-way and two-way*

Template (string) interpolation and event-binding  
The easiest way to display a component’s property, is to bind the property using string interpolation – just put the property name in the view’s template in double curly brackets: {{parentProperty}}.

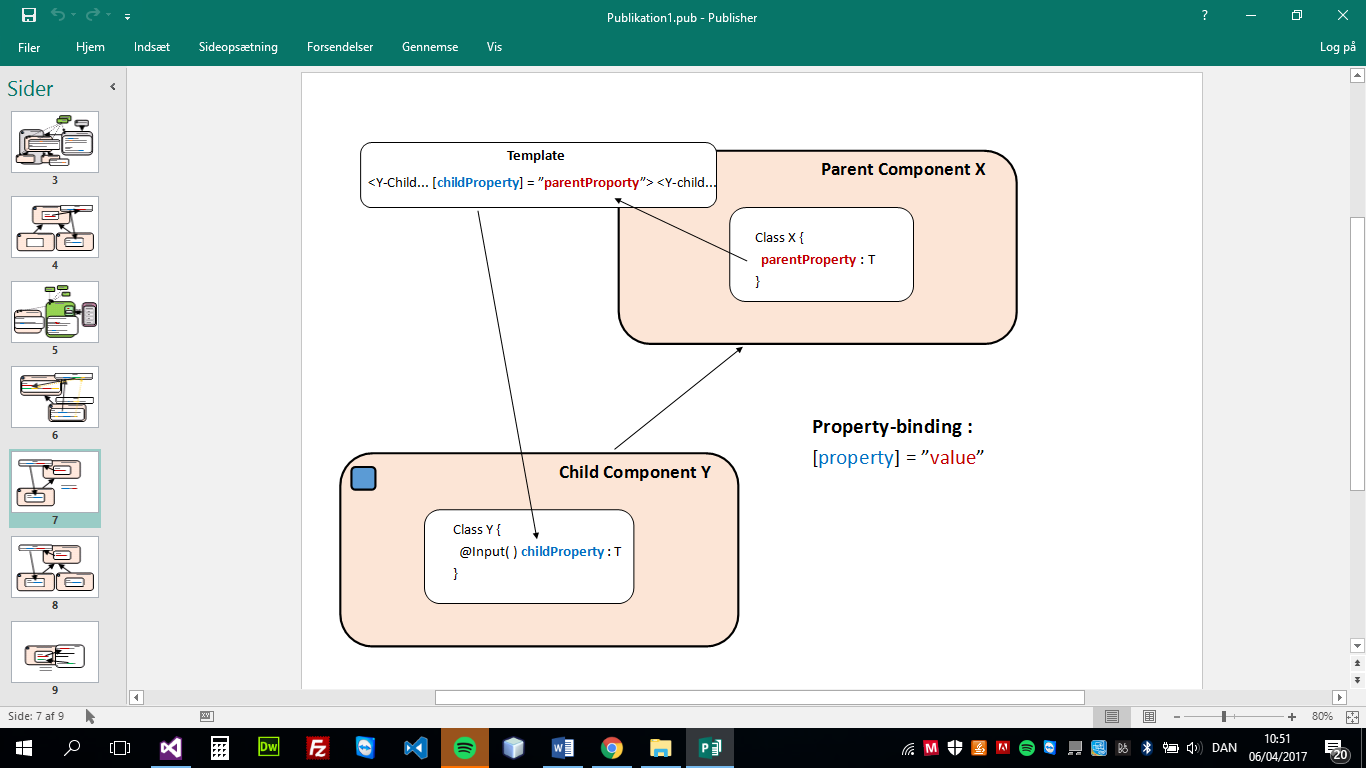


*Fig. 9 Template interpolation and event-binding*

The simplest way to add an event-handler to handle an event in the view is to use event binding. In the sample above we bind the component’s onClick( ) method to the button’s click-event simply by adding the assignment: *(click)=”onClick( )”* in the button-tag in the template to the component.

Property binding and @Input annotation

Very often, we need to send information from a parent-component to its child-component. We do it in two steps – first, we insert a property binding in the child-tag in the parent template, where we bind the child-property to the parent-property like: *[childProperty]=”ParentProperty”* – second, we add the @Input annotation to the child-property to tell that this property gets its value from the parent component.

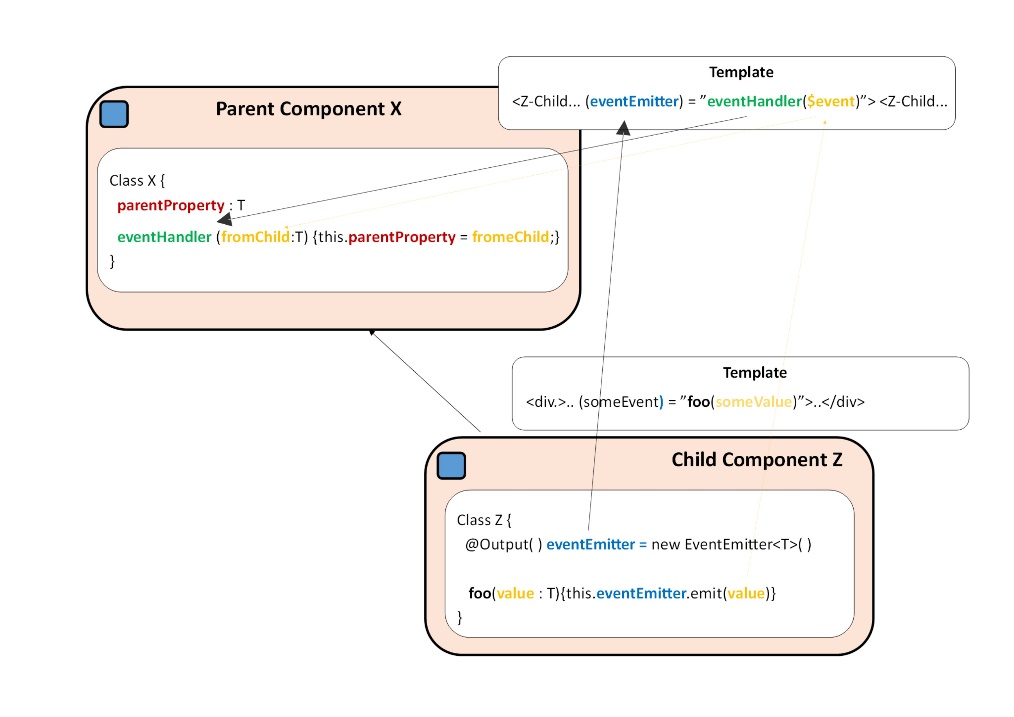


*Fig. 10 Property binding and @Input annotation (Parent -> Child)*

Event binding, @Output annotation and EventEmitter

Communication from a child component to a parent component or between child’s is a bit more complicated than the simple property binding from a parent to a child we saw above.

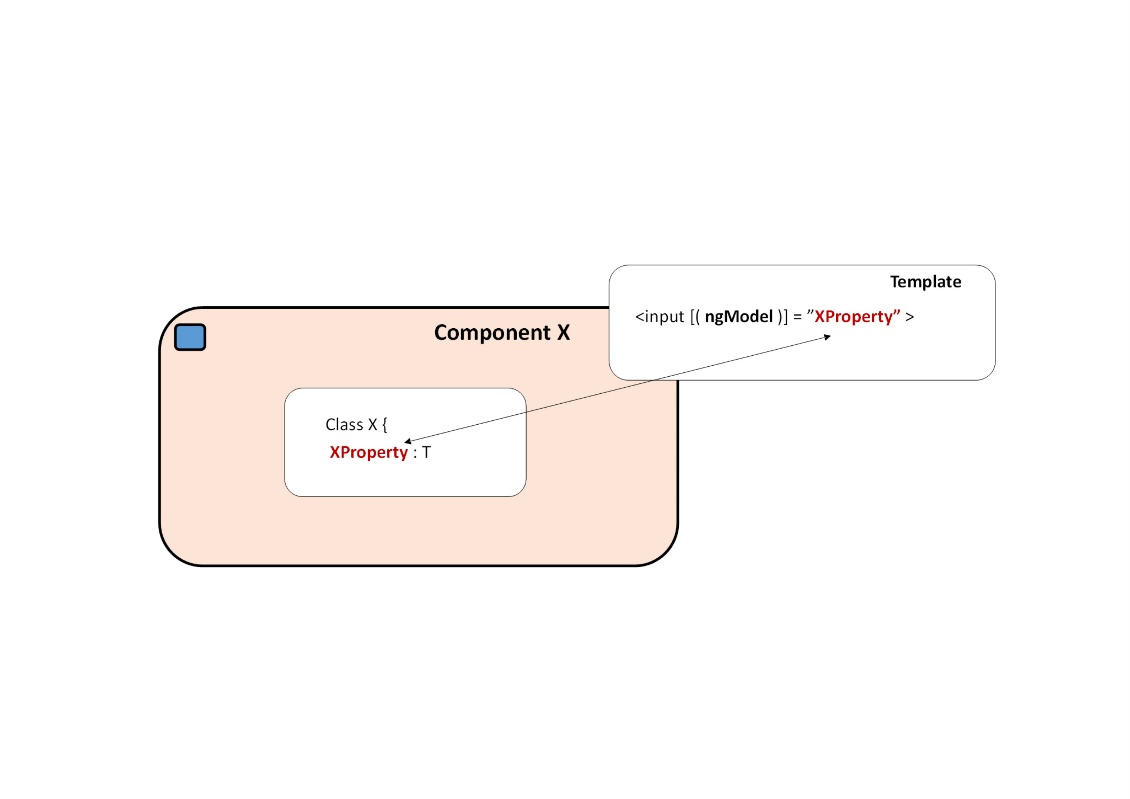
The scenario is: if a child’s property has been updated, then the parent is notified by a call to a handler-method with the new value passed as an argument. It is the old observable-design pattern and it is similar to the INotifyPropertyChanged implementation in C#. In Angular we use the class EventEmitter and its emit( ) method to do the notification. The registration is done using an event binding to the EventEmitter annotated with the @Output( ) in the child.



*Fig. 11 Event binding, @Output annotation and EventEmitter (Child -> Parent)*

// Full list of Angular Events  
  
(click)="myFunction()"   
(dblclick)="myFunction()"   
  
(submit)="myFunction()"  
(blur)="myFunction()"   
(focus)="myFunction()"   
(scroll)="myFunction()"  
  
(cut)="myFunction()"  
(copy)="myFunction()"  
(paste)="myFunction()"  
  
(keyup)="myFunction()"  
(keypress)="myFunction()"  
(keydown)="myFunction()"  
  
(mouseup)="myFunction()"  
(mousedown)="myFunction()"  
(mouseenter)="myFunction()"  
  
(drag)="myFunction()"  
(drop)="myFunction()"  
(dragover)="myFunction()"

Two-way binding  
The cool notation for a two-way binding with the ngModel directive in “a banana in a box”: *[(ngModel)] = “XProperty”,* is just a short and handsome notation for a combination of a property binding: *[value]=”XProperty”* and an event binding: *(input)=”XProperty=$event.target.value”.* Therefore, behind the scenes – if a user type in some value in the input-field, an input-event is fired and the property is assigned the typed in value (and yes, if we dig in deeper there is an EventEmitter involved).



*Fig. 12 Two-way binding with “a banana in a box” syntax*

Notice:   
The combination of forms and two-way binding is generally a bad idea since, if the user regrets during the entry and push the back button instead of submit, then the domain object will be contaminated – therefor use one-way binding instead and update the domain object with submit.

Directives

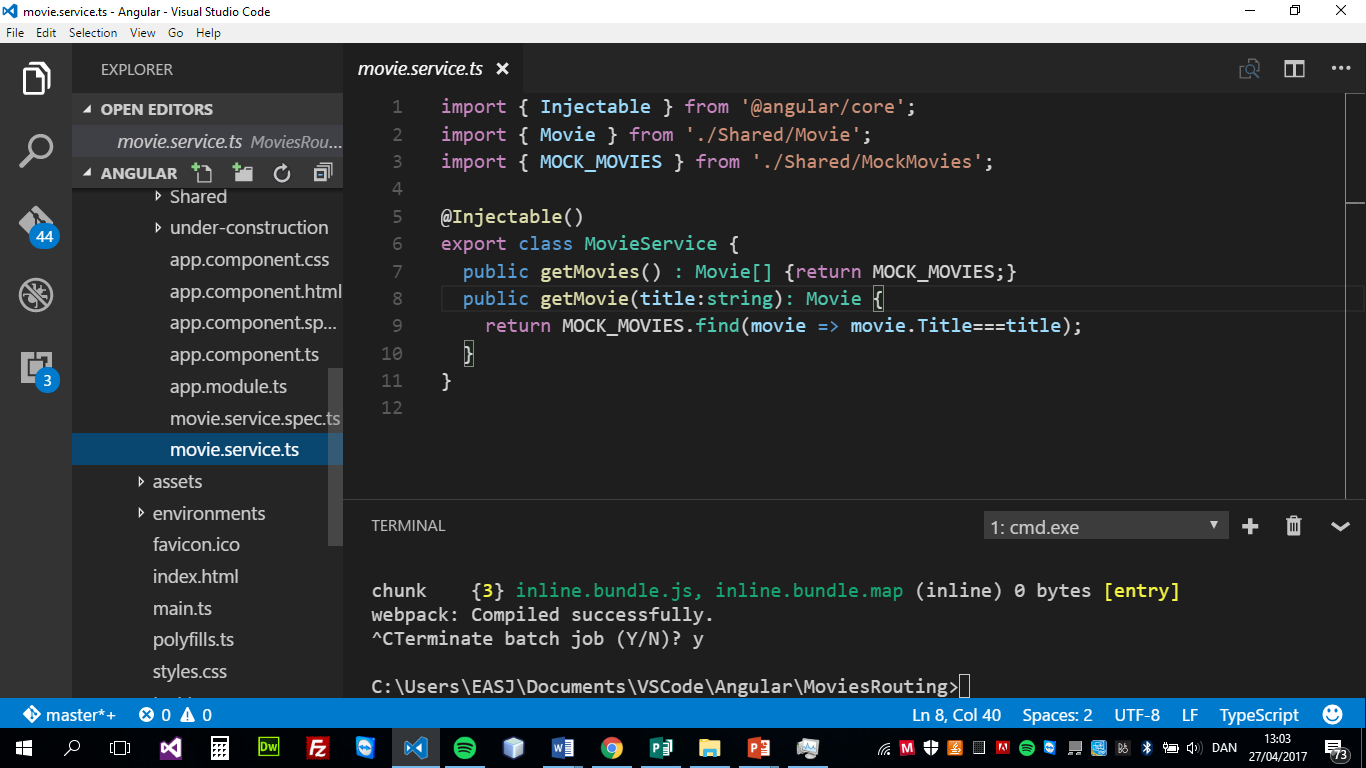
More to come …….

Services and delegation

A typical Service in Angular is a class that encapsulates some functionality and provide it as a service.  
The purpose of services in Angular is to make components more lean.

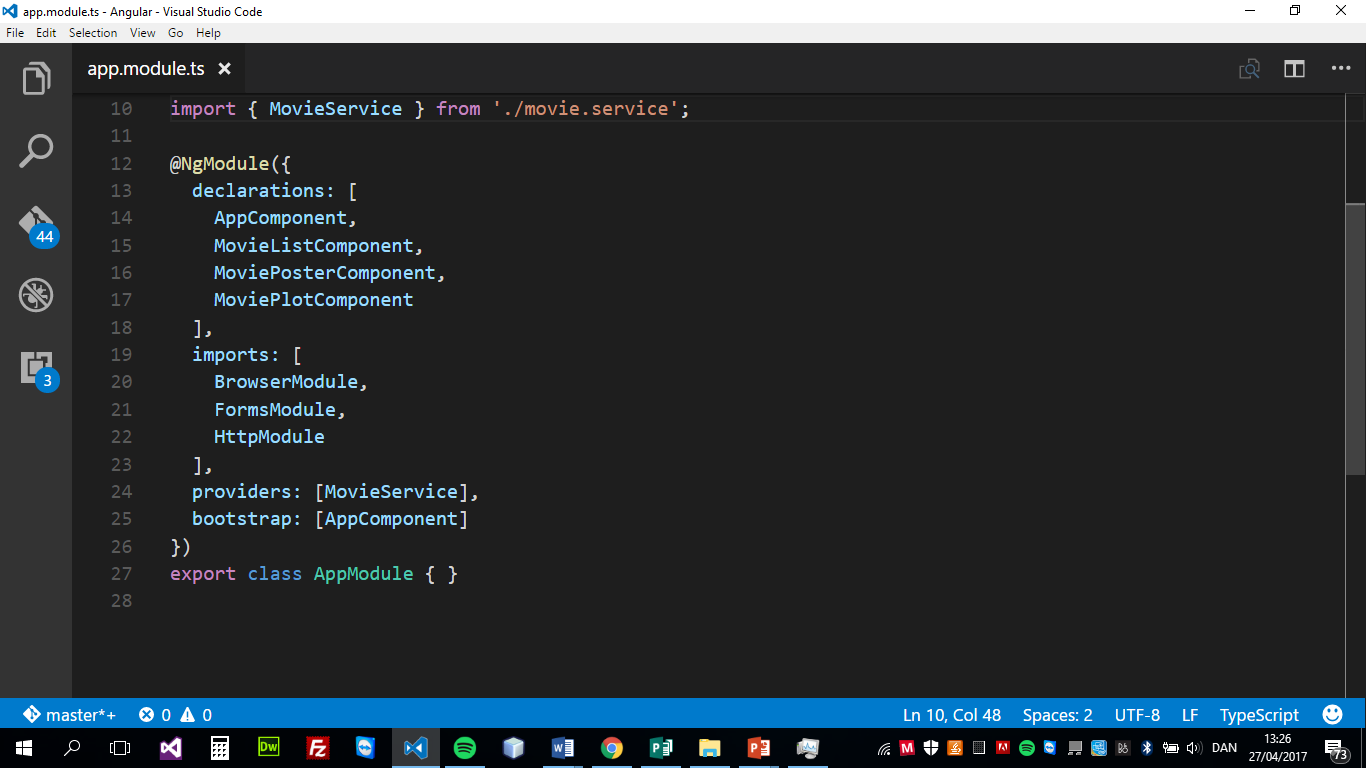
A component class should be simple, there job is to enable user experience and nothing more. A Component in Angular is like the ViewModel in the MVVM-architecture. The component acts as a data- and service-provider for the View - it presents properties and methods for data binding. All the nontrivial tasks in the component, like validating user input or communication with the server, should be delegated to a service.

Notice: There is nothing specifically *Angular* about services. Angular has no definition of a service. There is no service base class, and no place to register a service, but we still have to provide the service and annotate the service class with @Injectable.



*Fig. 13 Service with @Injectable*

Fig. 13 shows an example of a service class, *MovieService*. Notice the annotation with @Injectable, it enables the class to be injected into other components, so they can use the services *getMovies( )* and *getMovie(title:string)* .



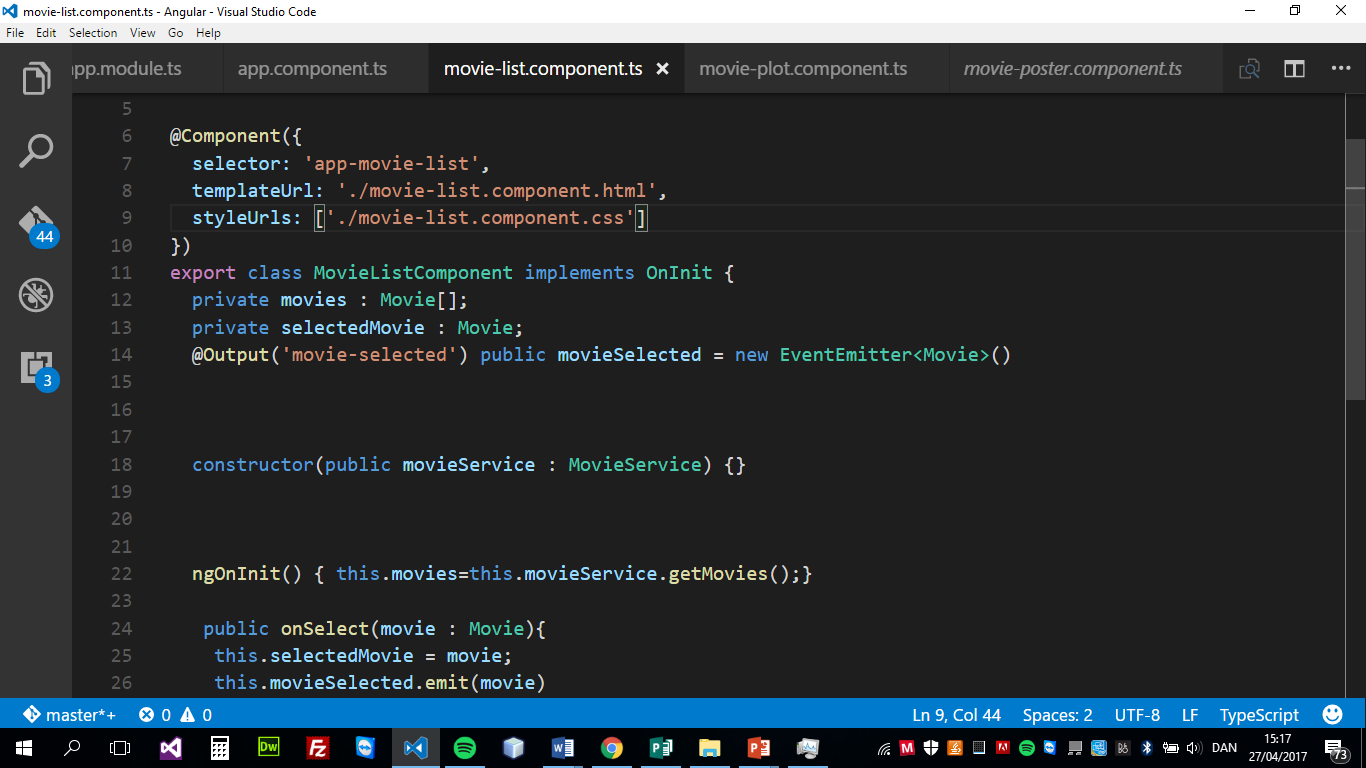
*Fig. 14 Registration of the service as a provider*

Fig. 14 shows the registration of the MovieService as provider in the AppModule.

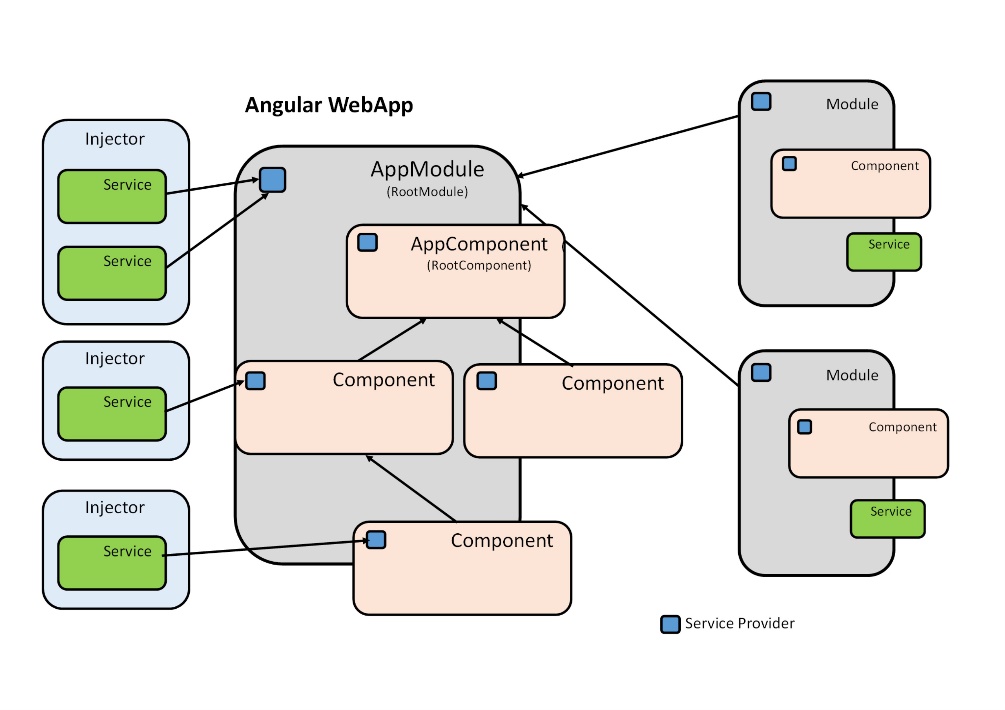
Dependency injection

Most dependence in Angular are for services and the dependency injection is the magic that gives a new instance of a class all the services it needs.

If a component needs a service, we simple add the service to the parameter list of the component’s constructor, eg the MovieService in the following constructor:

  
*Fig. 15 Dependencies specified in a constructor*

When Angular create a new component, it sends a request for all the services specified in the component’s constructor to the module’s injector (or the component’s injector if the component’s have its own local provider). An injector maintains a container of service instances (services specified in the providers-property of the component’s or module’s metadata). It will return a reference to the requested service if the service is in the container, otherwise the injector will make a new instance of the service and add it to the container, before it returns the service. When all requested services have been resolved and returned, Angular call the component’s constructor with the services as arguments. This is *dependency injection*.



*Fig. 16 Services and injectors*

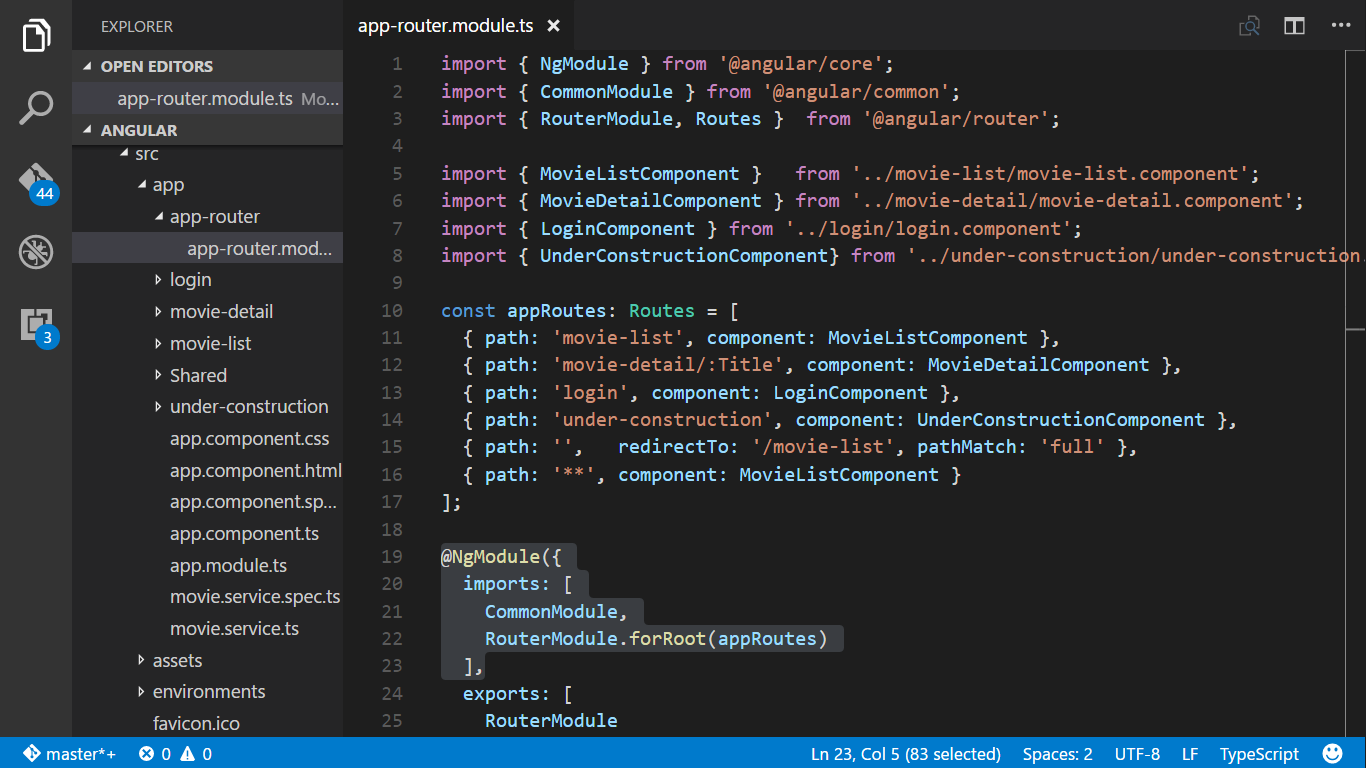
Fig 16 shows Injectors and the Service providers. The root module provider provides services globally to all the components in the App. The component’s provider provide services locally to a single component. In general, add providers to the root module, it only creates one instance of a service and that instance will be available everywhere. If we register a provider at component level, then it creates a new injector and a new instance of the service. The service will only be available for the component itself and its child-components and hidden for all other modules and components.

Routing & Navigating

Angular’s routing system gives us the possibility to divide our webApp into multiple views and to navigate between these views based on user interactions. The core in the routing system is the router component (*@angular/router*). It can interpret a URL as an instruction to navigate to a view. It can pass optional parameters to the view and support the component to find the specific content to present. It can navigate to a specific view if a user click at a router-bounded link. It can navigate programmatically (imperatively) when a user click on a button, selects from a drop box etc. It logs activity in the browsers history journal so we can use back and forward buttons as well.

Imports

The Angular Router is an optional service and is not a part of the *@angular/core* package. It has its own package *@angular/router*, and If we need anything from the package, we have to import it as usual:

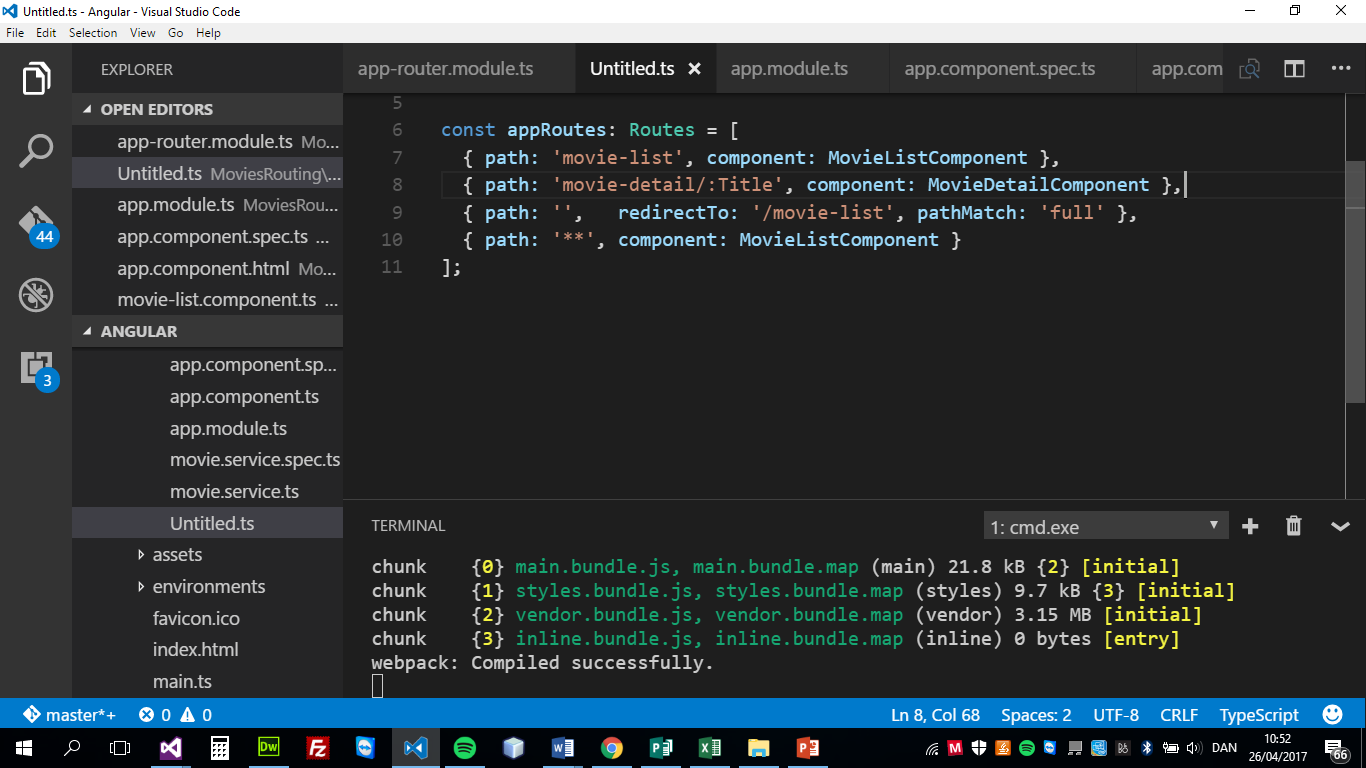


*Fig. 17 Router imports*

Configuration

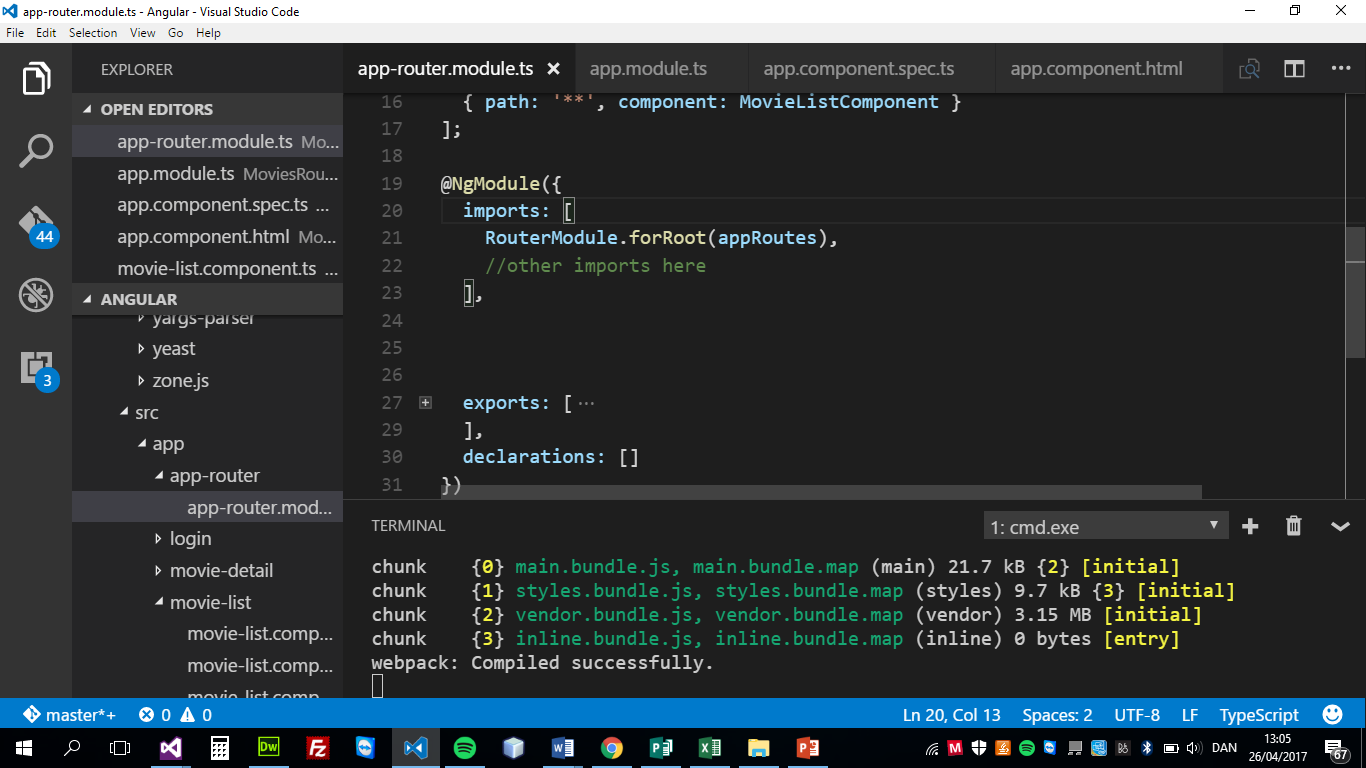
An Angular WebApp with routing has a single instance of the *Router* Service. If the browsers URL has changed or the *navigate( )* method has been called, then the *Router* service will look up for a corresponding Route to find the component to display. However, no routing before configuration, so we have to create some route definitions and make a configuration to tell the *Router* Service where to find the route definitions.   
  
Route definitions  
We use the *Routes* component from *@angular/router* to define our routes, each route has a path and a component property.

In the sample in fig. 18 we have four routes. The first gives a map between the URL: “movie-list” and the component: “MovieListComponent”. The second gives a map between “movie-detail/:Title” and “MovieDetailComponent”, where :/Title in the URL is a token for a route parameter, in a URL like: movie-detail/:Skyfall, will ”Skyfall” be the value of the Title parameter. The third route with the empty path ‘ ‘ represents the default path for the WebApp. The last route with the wildcard ‘\*\*’ is used represent a no-math path and can be used to redirect to a default route or to display a “404-not found” page.



*Fig. 18 Route definitions*

RouterModule.forRoot and RouterModule.forChild  
In fig. 18 we declare a const *appRoutes* of the type *Routes* and initialise it with an array of all our route definitions. Next step is to make the configuration, so the *Router* Service knows the *appRoutes,* via the *RouterModule*.*forRoot* method and add the result to the Moduls imports array.



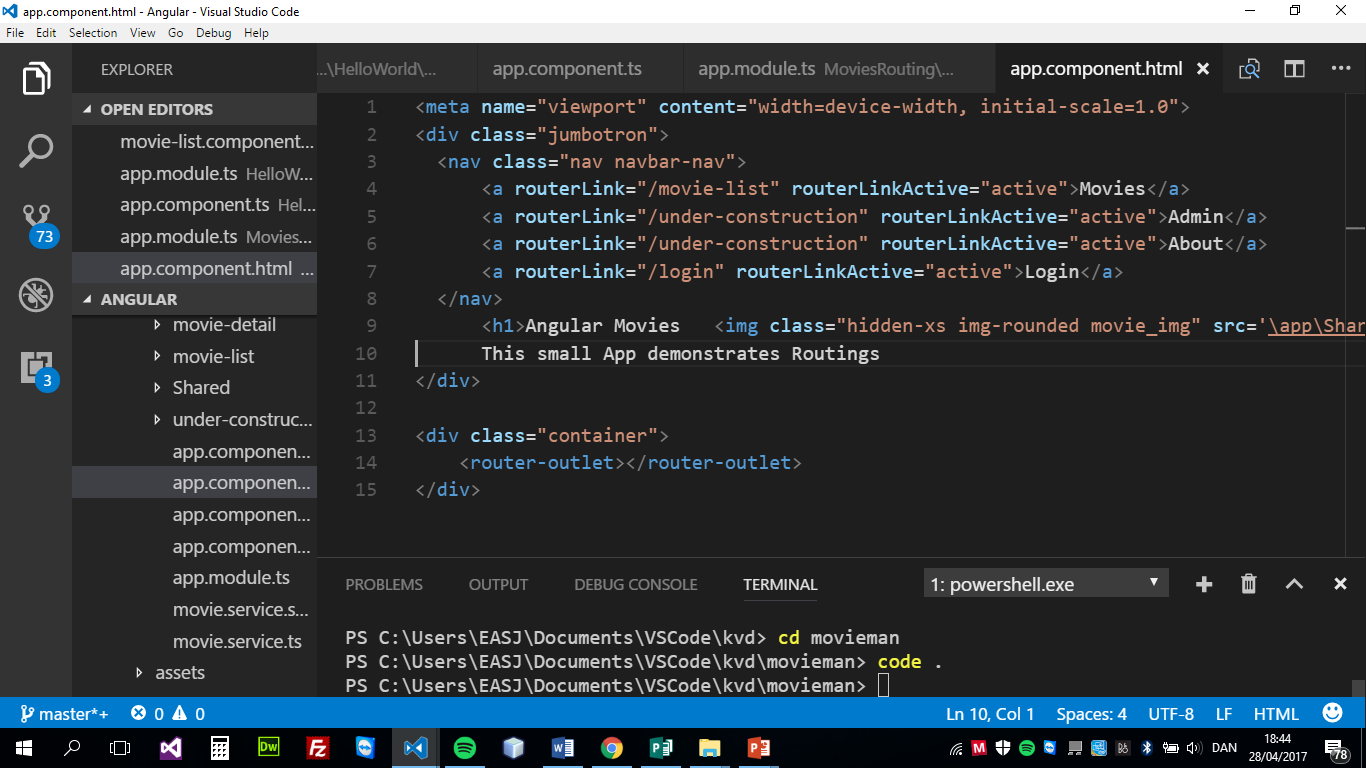
*Fig. 19 Router configuration with RouterModule.forRoot*

RouterModule.forRoot creates a module with all directives, the given routes and the route service itself. RouterModule.forChild is quite similar to the forRoot method, however it don’t creates an instance of the route service and is more likely to be used in a feature module - it makes it possible not to define all routes in the root module but delegate the routing to the child modules where the sub routes belongs to.

Router directives

Router outlet

The *RouterOutlet* directive has the selector: *router-outlet*, and the tag *<router-outlet></router-outlet>* is the markup for where to display the component identified by the route:

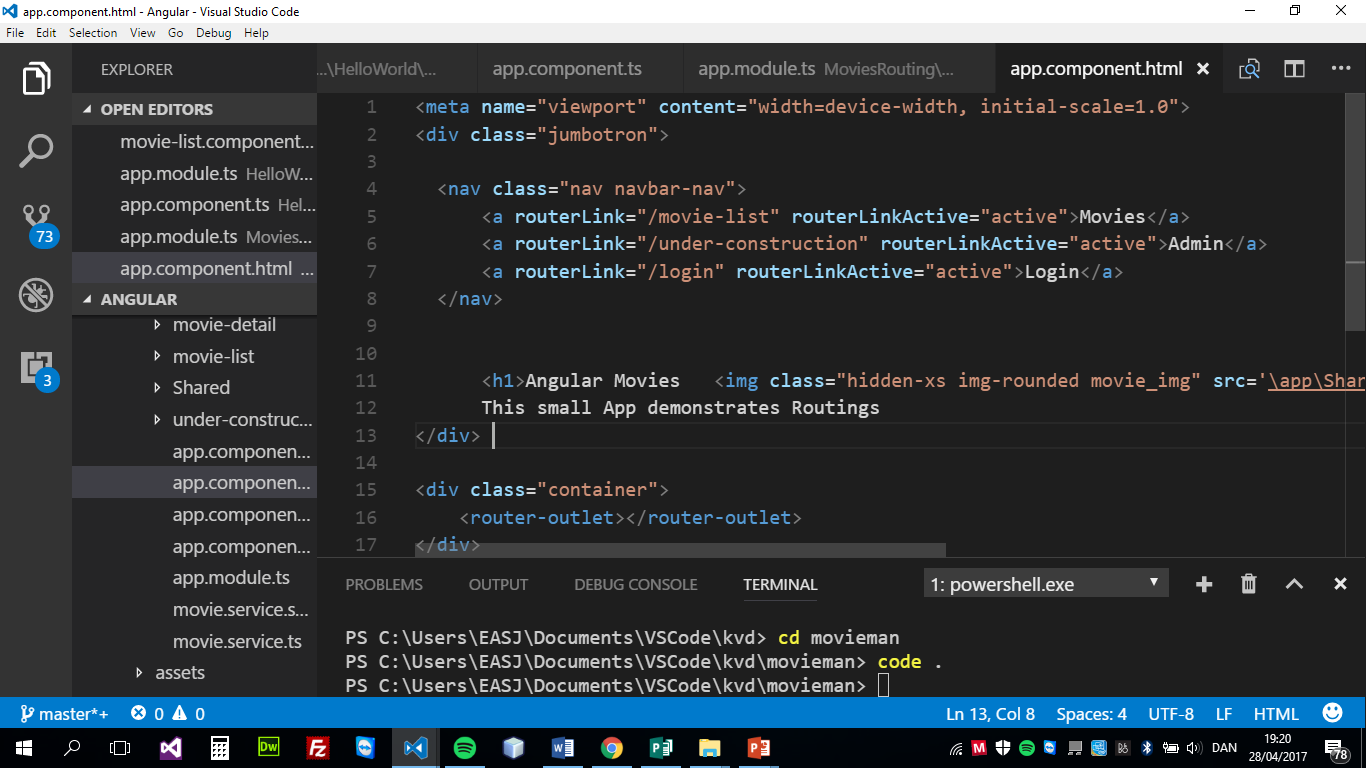


*Fig. 20 Router outlet markup*

When the router has resolved a route and identified the component(s) to load, it will create the component(s) and inject it as sibling(s) by the *router-outlet* element.

Router links

Now we know how to make route definitions and route configurations and we know how to insert placeholders for routed views, we just need to know how we do the navigation. The URL could come from the browser address bar, but most of the time the navigation is a result of some user action such as a click of an anchor tag.  
  
Adding a *RouterLink* directive to an anchor tag, gives the router control over the element. We specify where to navigate by assigning a navigation path to *routerLink*:

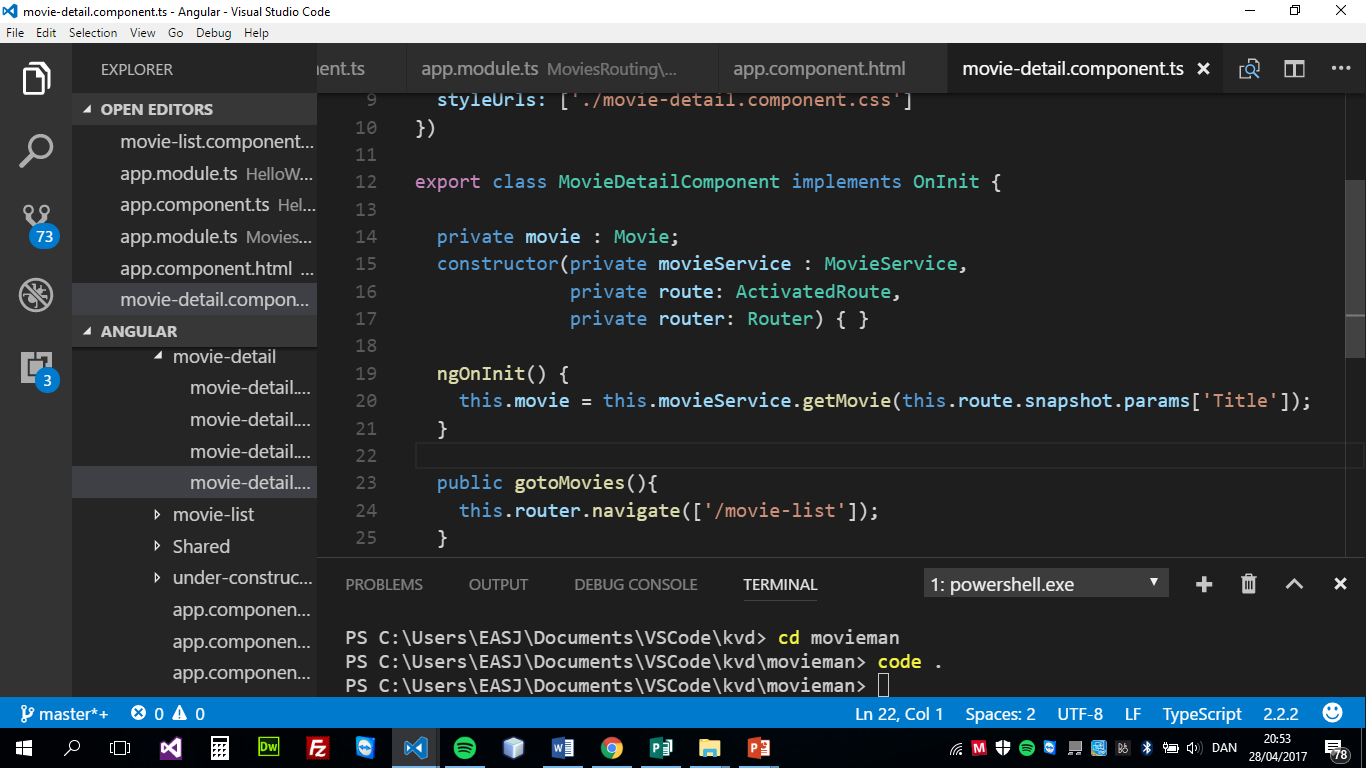


*Fig. 21 RouterLink*

The *RouterLinkActive* directive makes it possible to add/remove a CSS-class to the element when the links route becomes active/inactive.

Router API

Beside *RouterLinks* the navigation can be done by using the imperative API the *router* also provides. The ability to redirect the user in the code, rather than after a click at a link gives more flexibility. We simple inject the router in the constructor to a component and the call the navigate-method on the injected router object:



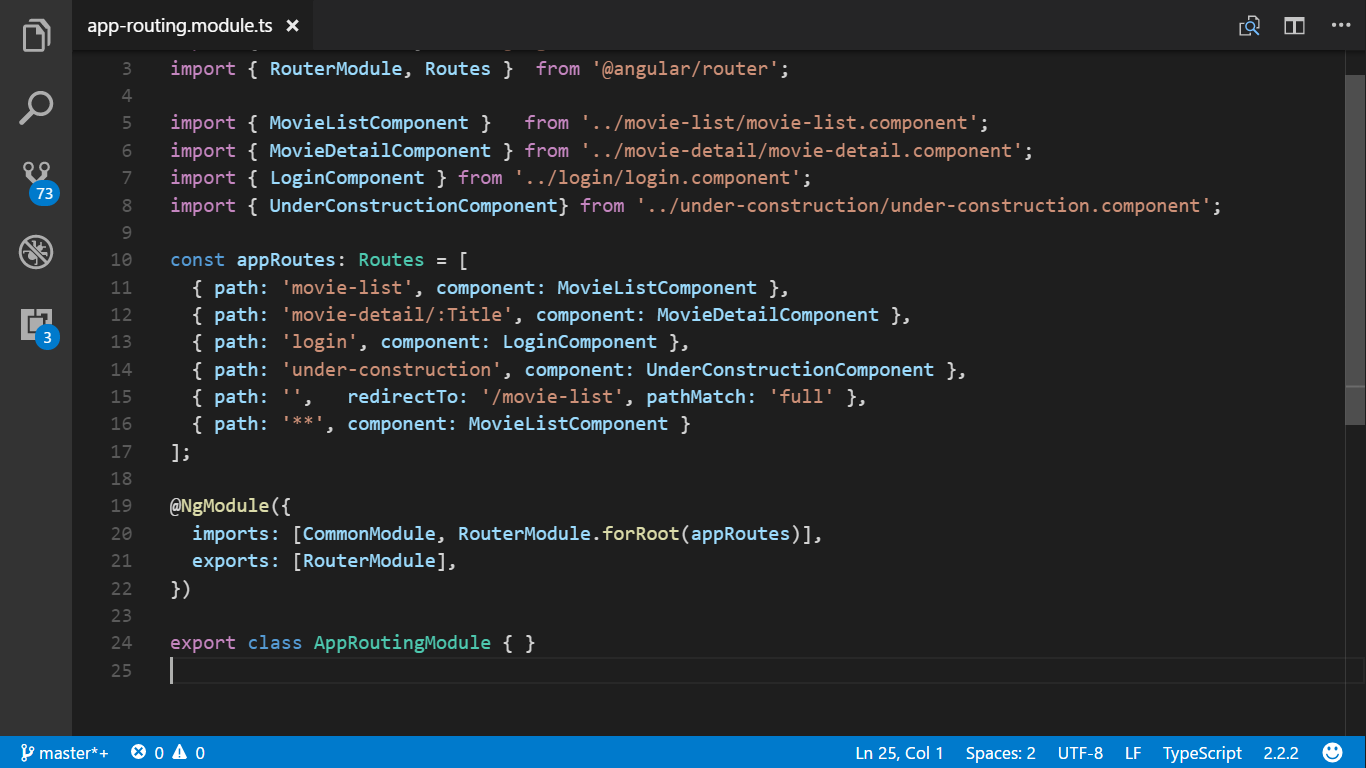
*Fig. 22 this.router.navigate( )*

Router state  
More to come …….

Router params  
More to come …….

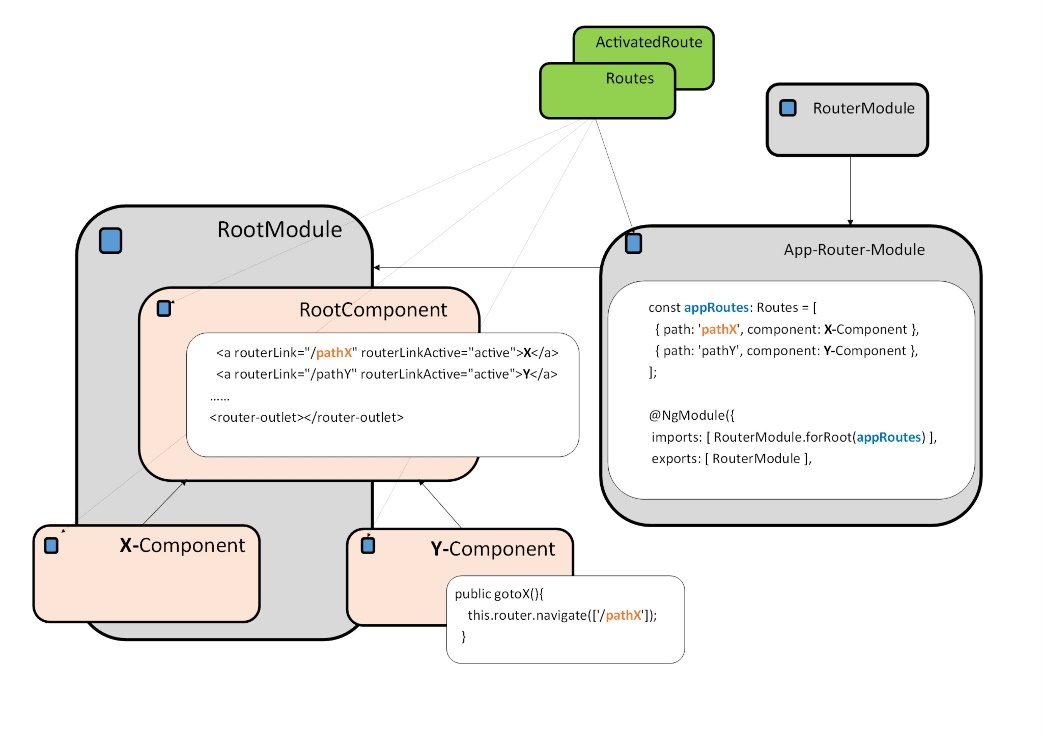
RouterModule

For small webApps it acceptable to have all the route definitions and configurations in the root component, but when the webApps grows and we need more routing features we prefer to move the route configuration into its own routing module:



*Fig. 23 AppRoutingModule in app-routing.module.ts*

By conventions we name the router module: *AppRoutingModule* and the file: *app-routing.module.ts* . Beside the router imports we need to import all components in the route definitions. Route definitions and the configuration (RouterModule.forRoot(appRoutes)) is moved from the root module to the routing module and finally by re-export the RouterModule and import the AppRoutingModule in AppModule, gives all components declared in the AppModule access to all the router directives like RouterLink and RouterOutlet.

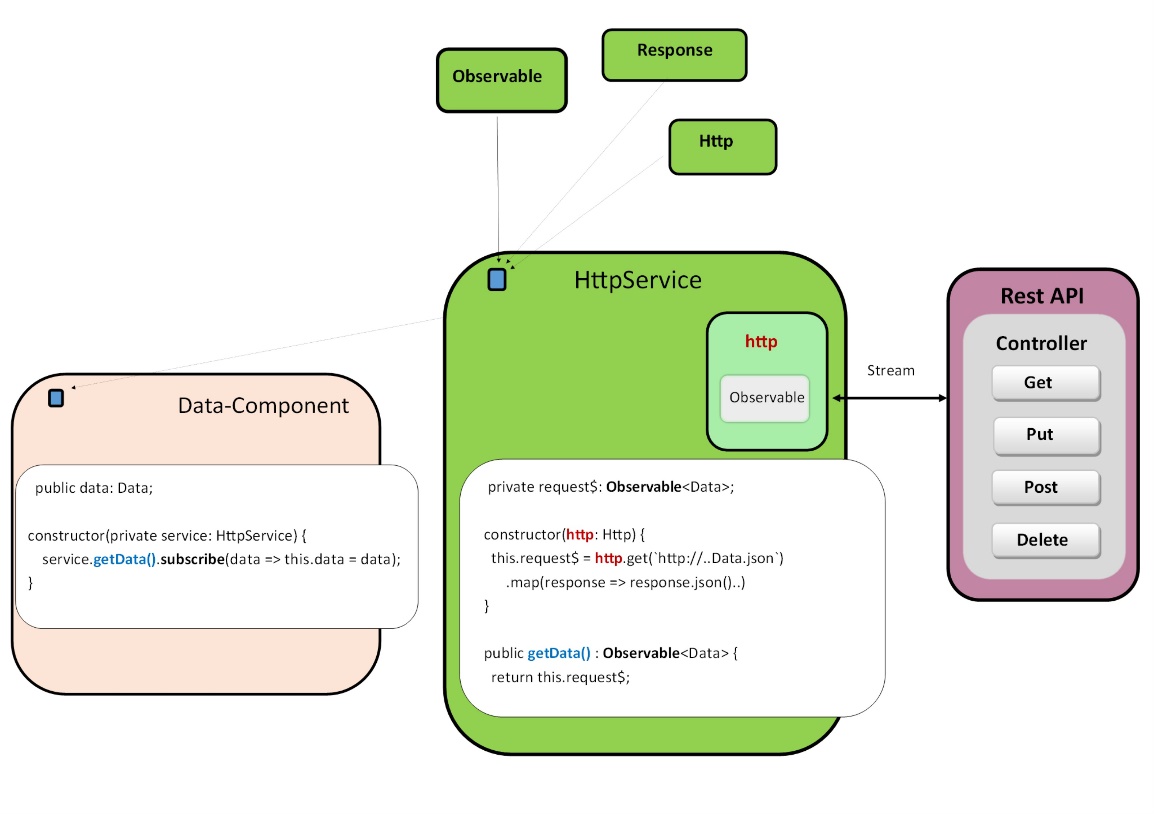


*Fig. 24 Routing and navigating – RouterModule overview*

Fig. 24 gives an overview of the structure after refactoring of the routings to its own module. Navigating via routerLink is typically in a navbar in the the RootComponent and programmatically navigating via router API is typically in child-components. All components with routing facilities needs injection of routings and our AppRoutingModule needs import of RouterModule.

HttpService

More to come …….



*Fig. 25 HttpService*

More to come …….

private request$: Observable<Data>;

constructor(http: Http) {

this.request$ = http.get(`http://..Data.json`)

.map(response => response.json()..)

}

public getData(): Observable<Data> {

return this.request$;

}

public data: Data;

constructor(private service: HttpService) {

service.getData().subscribe(data => this.data = data);

}

Pipes

More to come …….

Forms

More to come …….

Lifecycle Hooks

More to come …….

Testing

More to come …….

Sources:  
Basics:  
<https://toddmotto.com/bootstrap-angular-2-hello-world>  
<https://toddmotto.com/creating-your-first-angular-2-component>

Architecture:  
<https://angular.io/docs/ts/latest/guide/architecture.html>

Modules:  
<https://angular.io/docs/ts/latest/guide/ngmodule.html>

<http://exploringjs.com/es6/ch_modules.html>

<https://webpack.js.org/>

Services  
<https://www.barbarianmeetscoding.com/blog/2016/03/26/getting-started-with-angular-2-step-by-step-2-refactoring-to-services/>

Bindings  
<https://blog.thoughtram.io/angular/2016/10/13/two-way-data-binding-in-angular-2.html>

Forms

<https://angular.io/docs/ts/latest/guide/forms.html>  
https://angular.io/docs/ts/latest/guide/reactive-forms.html  
<https://blog.thoughtram.io/angular/2016/03/21/template-driven-forms-in-angular-2.html>

<https://blog.thoughtram.io/angular/2016/06/22/model-driven-forms-in-angular-2.html>

<https://blog.thoughtram.io/angular/2016/07/27/custom-form-controls-in-angular-2.html>

<https://blog.thoughtram.io/angular/2016/03/14/custom-validators-in-angular-2.html>

Routings

<https://angular.io/docs/ts/latest/guide/router.html>

<https://angular.io/docs/ts/latest/api/router/index/RouterOutlet-directive.html>

<https://angular.io/docs/ts/latest/api/router/index/RouterLinkActive-directive.html>

<https://toddmotto.com/angular-component-router#routermoduleforchild>  
<https://vsavkin.com/the-powerful-url-matching-engine-of-angular-router-775dad593b03>

<https://vsavkin.com/angular-2-router-d9e30599f9ea>

<https://angular.io/docs/ts/latest/api/router/index/RouterModule-class.html>  
<https://www.barbarianmeetscoding.com/blog/2016/03/28/getting-started-with-angular-2-step-by-step-4-routing/>