# LAB INTR-1) Install the software

## 1.1 Install node

Find the file named something like `node-v6.4.0` for either Windows or Mac in the `Software setup` folder. Run the installation procedure.

Verify that it works by running:
`node -v`
Should report something like:
`v6.4.0`

## 1.2 Install Visual Studio Code

Find the file named something like `VSCodeSetup-stable` for either Windows or Mac in the `Software setup` folder. Run the installation procedure.

Verify that you can open the editor

**When opened press Ctrl+P and write `ext install auto import` and press enter**. When visible click the install button. (The plug-in may already have been installed automatically). This will help us import exported classes, function etc, when working with javascript modules (More on that later).
Install the **Angular Language Service**. **Ctrl+P and write** `ext install angular language service.` When visible click the install button.

## 1.3 Install git

Find the file named something like `Git-2.10.1-64-bit` for either Windows or Mac in the `Software setup` folder. Run the installation procedure.

Verify that it works by running:
`git --version`
It should report something like:
`git version 2.10.1`

## 1.4 Copy labs

Inside the Installation kit find the `labs.zip` file. Copy this file to a place of your choosing and unzip the file. All your work will be done inside the newly created `labs` directory.

**Avoid using directory names with spaces**

# LAB TS-1) Simple TypeScript

## 1.1 Install the TypeScript compiler

Install the TypeScript compiler globally by issuing:

```
npm install -g typescript
```

Verify the compiler works by running
`tsc -v`
Should report something like:
`Version 3.x.x`

If the installation succeeds, but the `tsc` is not found, you might need to add `npm` to the path:

```
C:/Users/xxx/AppData/roaming/npm
```

## 1.2 Create a file called simple-typescript.ts

In the directory `01-simple-typescript` create a file called simple-typescript.ts

Start the TypeScript compiler and let the compiler watch the file

## 1.3 Create an interface

Create an interface called `IVATCalculator`, and create a single method in this interface called `calculate`.

This method should have a single parameter called `amount` of type `number`. It should return a `number` as well.

If you open the `simple-typescript.js` in Visual Studio Code using the Split Editor, notice that the JavaScript file is empty. This is because interfaces only exist in TypeScript.

## 1.4 Create a class

In the same file create a class. Name the class VATCalculator, and let the class implement the interface `IVATCalculator`.

Create an implementation of the method `calculate`. This method should return the amount with VAT of 25% added to it.

Create an instance of this class and assign it to a constant called `vat` of type `IVATCalculator`.

Lastly create two log statements like this:

```
console.log(vat.calculate(100));
console.log(vat.calculate(120));
```

## 1.5 Execute your JS file

Execute your JS file like this:

```
node simple-typescript.js
```

This should printout two lines like this:

```
125
187.5
```

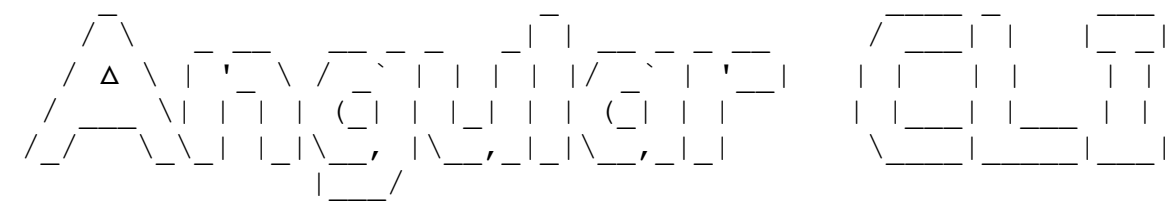
# LAB TOOL-1) Create a new project

## 1.1 Install Angular CLI

Install Angular CLI globally by issuing:

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

Verify Angular CLI works by running
`ng version`
Should report something like:



```
Angular CLI: 7.2.2
Node: 10.9.0
OS: linux x64
Angular:
...

Package                           Version
-----------------------------------------------------------
@angular-devkit/architect         0.12.2
@angular-devkit/core              7.2.2
@angular-devkit/schematics        7.2.2
@schematics/angular               7.2.2
@schematics/update                0.12.2
rxjs                              6.3.3
typescript                        3.2.2
```

## 1.2 Create a new project

In the directory `02-create-new-project` create a new Angular project called `playgrounds` using the CLI. Answer any question the CLI prompts you for.

When the creating has completed, and this may take a while, verify that the application works by running in the project directory (You might need to change directory for this):
`ng serve`

It should say something like:
`...`

```
Serving on http://localhost:4200/
...
```

It might say that the port is already being used, in that case use `ng serve -port 4201`. Open a browser and verify you get something like this:

**Welcome to app!**

# LAB COMP-1) Create first component

## 1.1 Install dependencies

Open a command line in the `03-create-first-component` directory and run the command

```
npm install
```

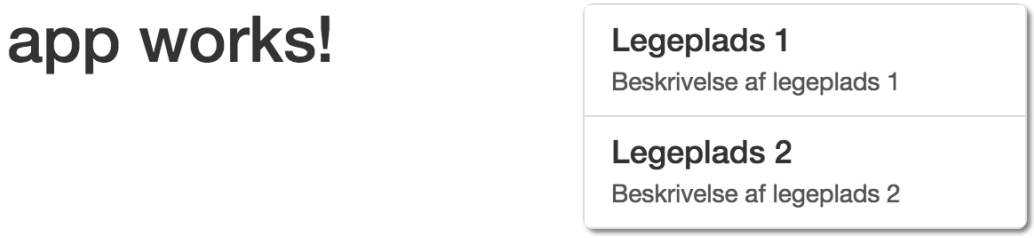
This will install all the dependencies for the Angular project. This happened automatically before, when we created a new project using `ng new playgrounds`

You may be wondering why you cannot use the project you created before? First of all we need some extra CSS, but also a feature module has already been created in the solution.

Besides that, a small chunk of HTML has been added to the `index.html`.

When the command has completed you should be able to run `ng serve`.

You should see something like this:




## 1.2 Create a new component

Using the skills, we got from the tooling primer lets create a new component called `sidebar`, using the Angular CLI.

When created you should find the source here:

```
src/app/sidebar
```

## 1.3 Add some HTML to the component

Before we include the component in the Angular application let's move the `aside` HTML element and all its children from the `index.html` to the `sidebar.component.html`

When we save the files, the browser should automatically refresh, and the sidebar has disappeared.

## 1.4 Add the sidebar component to Angular application

The root component template needs to be updated as well. We need to add the sidebar as a child to the root component. (Remember the component tree from before?)

When we save the files, the sidebar will reappear.

## 1.5 Create model

Before we proceed, we need to define what makes a playground? Just like we used to do in Java or C#.

So, let's create an interface called Playground. Use the Angular CLI to create the interface:

```
ng g interface shared/playground
```

This creates the playground interface in the shared folder, which makes sense, since the playground interface will be used throughout the application.

Add the following properties to the interface.

- id – string
- name – string
- addressDescription – **optional string**
- description – **optional string**
- position: Coordinate

The coordinate is already part of the solution and just needs to be imported into the `playground.ts` file using:

```
import { Coordinate } from './coordinate';
```

## 1.6 Adding mock data

In the `shared` folder create a file called `mock-playgrounds.ts`.

You can either manually enter the following into this file:

```
import { Playground } from './playground';

export const MOCK_PLAYGROUNDS: Playground[] = [
    {
        id: 'legeplads.1',
        name: 'Hauser Plads',
        addressDescription: 'overfor Hauser Plads 16',
        description: 'Godt sted til en legepause på byturen.',
        position: {
            lat: 55.682711143117565,
            lng: 12.575818079682959
        }
    },
    {
        id: 'legeplads.2',
        name: 'Israels Plads',
        addressDescription: 'ved Ahlefeldtsgade',
        description: 'Legeplads med New Yorker-stemning.',
        position: {
            lat: 55.682657692613766,
            lng: 12.568528509584322
        }
    },
    {
        id: 'legeplads.3',
        name: 'Kastellet',
        addressDescription: 'På Smedelinien ved Gustafkirken, udfor Princessens Bastion',
        position: {
            lat: 55.69272449969055,
            lng: 12.591519460476988
        }
    },
    {
        id: 'legeplads.4',
        name: 'Nikolaj Plads',
        addressDescription: 'udfor Nikolaj Plads 5-11',
        description: 'Kunstlegeplads på rolig plads ved Strøget.',
        position: {
            lat: 55.67876337291623,
            lng: 12.582071325958921
        }
    }
];
```

Or you can copy paste it here: http://bit.ly/2cmhdiH

## 1.7 Make the sidebar dynamic

Right now, we have a model, we have some mock data and we have a sidebar component. But the sidebar component is still static.

Assign the MOCK_PLAYGROUNDS to an instance variable called `playgrounds` in the sidebar component. Next update the template to iterate over these playgrounds using `*ngFor`. Use one of the `<a>` element as a template on how to render a playground. Remove the other `<a>` element.

When you save the files and the browser refreshes you should see something like this:

app works!

# LAB MODU-1) Import the leaflet module

In this exercise you can choose to, either stay in the previous directory and continue there, or you can navigate to the `04-include-feature-module` directory and continue there. (You need to run npm install, if you choose to change directory)

## *1.1 Import feature module*

The solution already contains a feature module to display a map. We'll not be using this feature module right away, but let's make it part of the solution.

The feature module is placed in the directory `leaflet`. Add the `LeafletModule` to the `imports` array of the root module.

Save the root module and reload the application. If everything works, and there are no errors in the developer tools you are all done!

# LAB IO-1) Create footer component

In this exercise you can choose to, either stay in the previous directory and continue there, or you can navigate to the `05-create-footer-component` directory and continue there. (You need to run npm install, if you choose to change directory)

## *1.1 Make a presentation component of the sidebar*

Right now, even though it is a bit simple, the sidebar knows how to get a reference to the playgrounds to display. This makes it a smart component. Again, currently this is kind of simple.

Instead let's make the `AppComponent` the smart component. Copy the `playgrounds` property from the sidebar component to the app component. Next delete the assignment of the `playgrounds` property in the sidebar component and prefix the property with `@Input()`, remember to import it from `@angular/core` module. If you save now, no playgrounds are shown.

Next step is the assign the sidebars playgrounds property using the property data binding syntax, used in the parent → child scenario.

Now the sidebar has no knowledge of how the playgrounds were fetched.

## *1.2 Mark playground as selected*

Before we create an output from the `sidebar` component, mark a playground that is clicked on, using CSS as `active`.

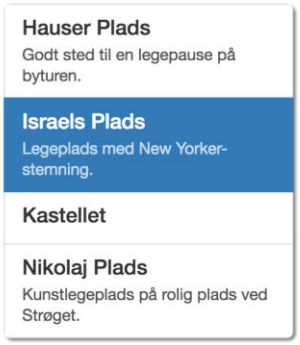
To do this we could add the `click` event listener to the `<a>` element. When clicked, it should invoke a method called `selectPlayground`. This method set a public property called `selectedPlayground` with the given playground.

Using a property data binding set the `<a>` element CSS class to active, when the selected playground is equal the playground in the `*ngFor` loop.

Hint: `[class.active] = "???"`

When it works, you should have something that looks like this:

app works!




## 1.3 Create output

Let's create some output using the `EventEmitter`!

Start by adding a property called `select`. Assign it to an instance of `EventEmitter`. Remember to use generics! Our event emitter emits a `Playground`.

Once added, decorate it with the `@Output()` decorator.

When all this is done, emit the selected playground in the `selectPlayground` method we just created.

## 1.4 Listen to the event

Going back to the app component, we need to listen for this event.

We can only listen for events on the child elements that emits them, so the two siblings cannot communicate using event emitters.

So create a method called playgroundSelected in the `app` component. When called it should set a property called `playground` to the given value.

Lastly update the `app` template to listen to the event from the `sidebar` component.

Before proceeding to next exercise try to log out the selected playground using console.log()!

## 1.5 Create the footer component

Using the Angular CLI, create a new component called `footer`. Include this new component inside the `app.component` template.

When you save the files, you can see that the `footer` component has been added to the view:


## 1.6 Add `playground` input

Right now, the footer component knows nothing about any playground. So, let's add a `playground: Playground` property to the `footer` component class.

Using the parent → child style, we add the `@Input` decorator to our `playground` property.

## 1.7 Update the footer template

Let's change the footer component template from "footer works" to something more useful.

The footer must show a playground. It should show the playgrounds name, address description and description.

The root element of the template must be a `<footer>` HTML 5 element. Within the footer element place a `<h3>` element and two `<p>` elements. Place the playground name in the `<h3>`, the address description in the first `<p>` and the general description in the second `<p>`. All using interpolation.

When we save out files now, the application stills loads, but we should see a lot of errors in the developer tool.

Why is that? Well, unless we used the Elvis notation, accessing the properties: name, address description and description, of the playground property result in a:
`Cannot read property '<property-name>' of undefined`

But even if we fixed this by adding the `?` to all our interpolations, and thereby fixing the errors, we would still have the footer element in the view, even when no playground is selected. That's not very pretty!

So instead, let's make the entire footer element disappear when no playground is selected. Remember a certain build-in structural directive to achieve that?

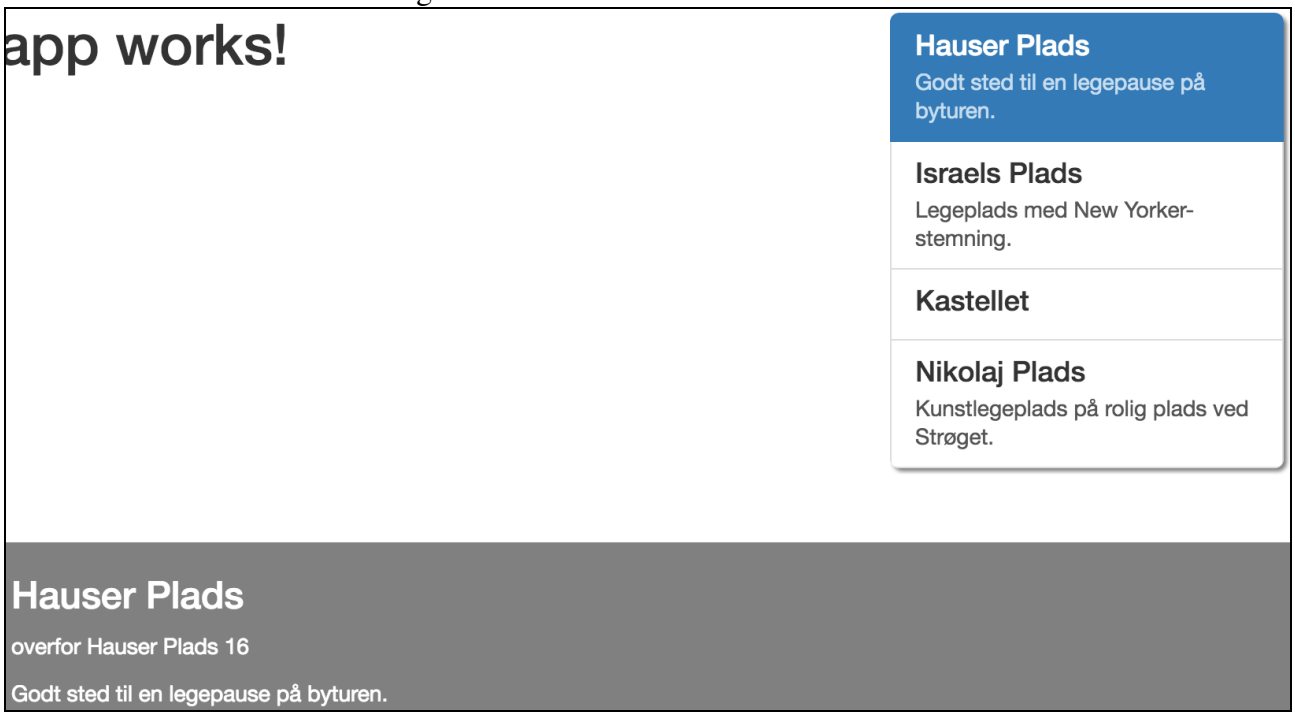
Update the template, save the files, and verify the errors have disappeared.

## 1.8 Bind playground to footer

Final thing we need to do is to bind the `playground` property from the `app` component to the input property of `footer` component.

Update the `app` template with this property data binding!

The result should look something like this:



# LAB SERV-1) Create playground service

In this exercise you can choose to, either stay in the previous directory and continue there, or you can navigate to the `06-create-playground-service` directory and continue there. (You need to run npm install, if you choose to change directory)

## 1.1 Create the playground service

Let's start out by creating a simple service with a single method. Let's call the service `PlaygroundService` and the method `getPlaygrounds`. The method must return an array of `Playgrounds`.

Normally a method calling a backend service would return a promise or an observable, but for now, since we have not talked about any of them, the `getPlaygrounds` method should just return the mock array from earlier.

Create the service using the Angular CLI and place the service in the `shared` folder.

Create the public `getPlaygrounds` method and return the mock array.

## 1.2 Registering the provider

Before Angular 6 we would need to provide this service in the app module. But now the service will automatically be registered as a service, since we have `providedIn: 'root'`. If we were to delete that statement, the service would need to beregistered manually in a module.

## 1.3 Inject the service into the `app` component

Let's finish off by injecting the service into the app component.

We do that by adding one parameter to the `app` constructor. Like this:

```
playgroundService: PlaygroundService
```

Remove any and all reference to the `MOCK_PLAYGROUNDS` array in the `app` component.

Lastly assign the result of the `getPlaygrounds` method to the instance variable `playgrounds`. Consider doing this in the `ngOnInit` method.

When you save the files, you should see no difference between using the `MOCK_PLAYGROUNDS` and using the service. But this will change very soon!

# LAB AJAX-1) Add ajax to the playground service

In this exercise you can choose to, either stay in the previous directory and continue there, or you can navigate to the `07-add-ajax-to-playground-service` directory and continue there. (You need to run `npm install`, if you choose to change directory)

## *1.1 Provide the http service*

Before we start using the http service in our `playground` service, we need to import the http module.

We will do so by importing the `HttpClientModule` module into out app module. `HttpClinetModule` is placed in `@angular/common/http`

## *1.2 Update the PlaygroundService*

Now that the http client service is accessible throughout the application, let's inject the `HttpClient` service into the `playground` service constructor.

We can get the playgrounds from `'assets/copenhagen.json'`

Since we know an observable is only a blueprint until subscribed, let's setup the http, with the multicast function in the constructor. Assign the observable returned from the http `get` to a instance variable called `request$` of type `Observable<Playground[]>`.

Change the `getPlaygrounds` to return `request$`, and of course the return value of the `getPlaygrounds` method.

And since the playgrounds rarely changes, use this combination:
`publishLast(),`
`refCount(),`

Delete any trace of MOCK_PLAYGROUNDS in the playground service. You could even delete the file, if you felt like it.

## *1.3 Update the app component*

The `app` component is still using the old version that returned an array directly.

Let's update the app component to use the stream (Remember `subscribe`?).

Save all files and check that you now have more than the 4 mock playgrounds.

# LAB RXJS-1) Provide the location service

In this exercise you can choose to, either stay in the previous directory and continue there, or you can navigate to the `08-create-location-service` directory and continue there. (You need to run `npm install`, if you choose to change directory)

## 2.1 Provide the location service

Inside the `shared` directory a file called `location.service.ts` contains the `LocationService`.

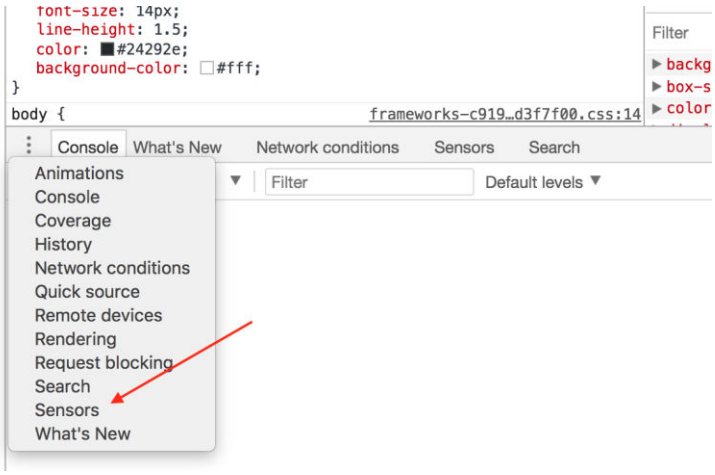
To provide this service to the application open up the `app.module.ts` and add the `LocationService` to the list of providers. Or use the `providedIn` syntax. You decide.

Make sure the application stills load without errors.

## 2.2 Inject the service

Just to make sure our service is available; let's inject the service into the `app` component. Subscribe to the location services `current` property and write a log message to the console, when to location has been obtained. (`console.log(…)`)

If you select the sensor tab in developer tools in Chrome, you should be able to emulate geolocation coordinates. Try to do that, and notice how the subscription logs out new messages.




When done close the browser tab and open a new tab on the same address. Chrome seems to have problems resetting the geo location back to normal.

# LAB RXJS-2) Include the map!

In this exercise you can choose to, either stay in the previous directory and continue there, or you can navigate to the `09-include-the-map` directory and continue there. (You need to run `npm install`, if you choose to change directory)

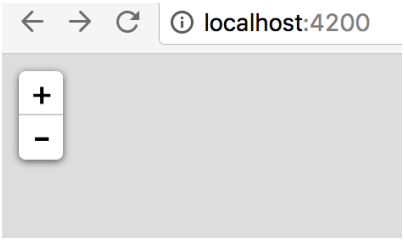
## 2.3 Add the map

Finally, we are ready to add the map to the application.

Remember that we included the leaflet feature module? It's time to use it.

All you need to do is to add the `<leaflet>` element to the app components template.

Try to reload to page to verify that no errors occurred. If no errors occurred you should notice the zoom in & out button that has appeared in the top left corner.




So even though the map is not shown at least something has happened.

## 2.4 Center the map

The reason the map is not shown is because no center has been set. So, let's add a public `center: Center` property to the app component class. You can import the `Center` class from './leaflet'. Notice `Center` is a class and should be created using the `new` operator

Next assign this property to an instance of a center. Use these values:

- latitude: 56.360029
- longitude: 10.746635

Next step is to bind the center attribute on the <leaflet> element to your center property in the app component.

Reload the application and watch the map being centered on Denmark!

## 2.5 Add your current position to the map

Final step is to add your position to the map.

The leaflet component has one more attribute – markers. The markers attribute takes an observable with the type Marker (`Observable<Marker>`). You can import the `Marker` class from ‘./leaflet’.

Create an observable that maps the current position into a `Marker` object using the **map** composition function from the observable. Next step, bind this observable to the `markers` attribute. You should not subscribe to this observable, since the leaflet component will handle subscribe and unsubscribe.

# LAB ROUT-1) Add some routes to the playgrounds!

In this exercise you can choose to, either stay in the previous directory and continue there, or you can navigate to the `11-add-routes` directory and continue there. (You need to run `npm install`, if you choose to change directory)

## *1.1 Create map component*

Before we proceed to add routes to our solution, we should move code from the AppComponent. The app component will only be used as a placeholder for the different routes, going forward.

So, create a component called `map` using the Angular CLI.

Now move all the code inside the `AppComponent` class into the `MapComponent`.

Next step is to cut the html from the `app.component.html` and paste it in the `map.component.html` file.

Verify that we haven't broken anything by placing the `<app-map></app-map>` element inside the `app.component.html`.

Reload the page to verify that everything is still working fine!

## *1.2 Define the paths*

Right now, our app has two states. Either no playground is selected, or a playground is selected.

So let's create a file called `app-routing.module.ts` in the `app` directory. Inside this file we need to work with RouterModule & Routes, so go ahead and import those from the `@angular/router` library.

Next setup the two routes. Both should use the `MapComponent` as their component, but their paths should be something like this:

- `''`
- `':id'`

When you have created all paths, create a `AppRoutingModule` module by calling the `RouteModule.forRoot` method.

## *1.3 Import the module*

Back in the `app.module` we import the `AppRoutingModule` module we just created.

Before we can test that it works, replace the `<app-map>` with `<router-outlet>`. This is where our map components content will go.

Now we should be able to reload the application. Everything should work as before.

## 1.4 Updating the map component

It seems like the sidebar is working but notice that when we click a playground the URL does not change.

So, update the map component to navigate to the correct URL, when the sidebar emits a playground.

Start by injecting the `Router` service. Next use the `navigate` method

Try to click on a few playgrounds. Notice that the back and forward buttons works as expected!

But we are not done. Try to select a playground, now reload the page. Notice that the selected playground is no longer selected.

This is because the `selectedPlayground` in the sidebar component has not been initialized. Remember that the sidebar should not read from the URL, as it is a presentation component and should have no context.

Make the `selectedPlayground` property assignable from the parent. (Remember component communication?)

## 1.5 Assign the selected playground to the sidebar

Back in the map component we need to find the selected playground based on the URL.

So, let's start by adding a find method to the playground service. This method should take a single parameter `id: string`, and return an `Observable<Playground>`. This should be doable using the `map` method on the existing `request$`. This should be a one-liner!

Now we just need the id from the URL! Inject `ActivatedRoute`. Using the id from the URL and the new `find` method in the `playground` service, we should be able to set the components `playground` property. We are going to need `switchMap` to do this. (BTW – Remember we do not always have an id)

If you reload the page now the footer should have started working again.

This still didn't mark the playground as selected in the sidebar. But all there is let to do, is to assign the `selectedPlayground` property in the sidebar component using the property data binding syntax.

## 1.6 Add marker, center and zoom

Our application is almost complete!

But when we select a playground we need to center the map on the selected playground, we need to zoom in and we need to place a marker.

Let's us start with the "easy" part. Center and zoom! In the subscription where you set the property playground to the playground identified by the URL update the center property. Convert the `playground` object into a `center` object and assign it to the `center` property. Set the `zoom` level to 14.

**Next step markers**

We know leaflet uses one observable as input. And that we need two kinds of markers, one for playground and one for our current position.

But we already know about the `merge` function on an Observable. Use the observable from the location service and merge it with the observable from the find method on the playground service.

Use the `map` function to convert the `playground` into a `marker` instance.

**CONGRATULATIONS! YOU ARE NOW DONE!**

I know this is hard, so congratulation if you got this far!