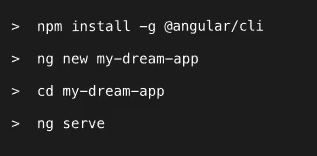
Angular

## Getting started

**Angular CLI** is really important to manage and write Angular applications. The Angular CLI is a toolset, for creating, building and managing Angular applications. It quickly creates Angular projects. 

We write our own applications mostly in the *app* folder.

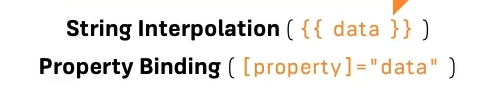
What is TypeScript? It’s a superset to JS, it offers more features than vanilla JS (like classes, interfaces and **types**). TS doesn’t run in the browser. It is always compiled to JS. This compilation is also done by the CLI. Angular is meant to be used together with the TypeScript.

To add BootStrap to our Angular application. We can install it with npm: **npm install –save bootstrap**  
then go to the angular-cli.json file and at the **styles** array you can define it’s path ((../)node\_modules/bootstrap/dist/css/bootstrap.min.css)

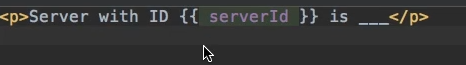
By using the selector in the TypeScript file we can call the different components. We just have to assign a name to the component for example **app-root**, which will be called in the html file like  **<app-root></app-root>.**

In Angular you build your whole application by composing from a couple of components, which you create on your own. Obviously, you can nest components into one another.

**Creating components** – Using CLI you can create components. You just have to type: **ng g(enerate) c(omponent) *name***

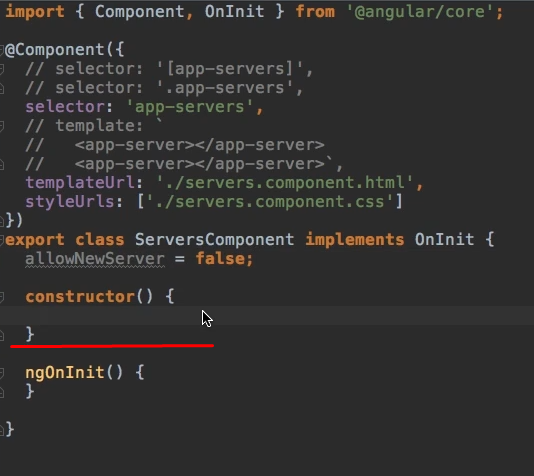
**Databinding -**  It’s a communication between your TypeScript Code (business logic) and the template code (HTML). There are different communications. For example we want to output data from our TS code in the template. We can use **String Interpolation** or **Property Binding.**There is another direction of the communication, when we want to record something (some event) from our template, we want to react to user events (for example a click event etc.). It is done with **Event Binding (event)=’’expression’’.**And then there is the third mode of databinding – the **two-way-binding** which is done with **[(ngModel)] = “data”.**

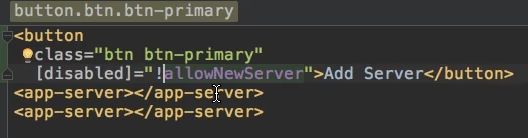
## String Interpolation





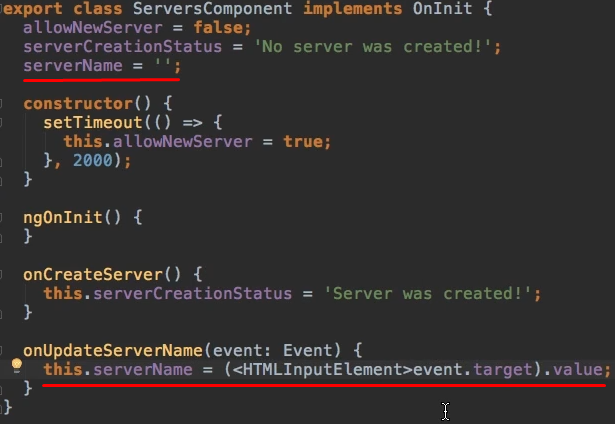
## Property Binding



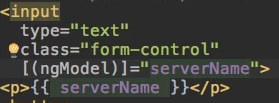
We are in a component’s TS file. The *allowNewServer* is a property, which has a boolean value. The constructor is simply a method executed at the point of time this component is created by Angular.  
  
Square brackets indicate, that we are using property binding, that we want to dynamically bind some property.

## Event Binding

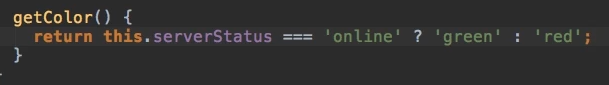
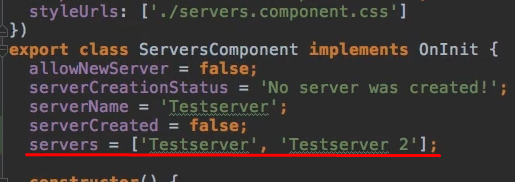
  
When the users clicks this button, the *onCreateServer* method will be executed, which is defined in the TS file.

  
This is how we can pass data to a method by using Event Binding.   
  
And this is how we can catch and use this value.

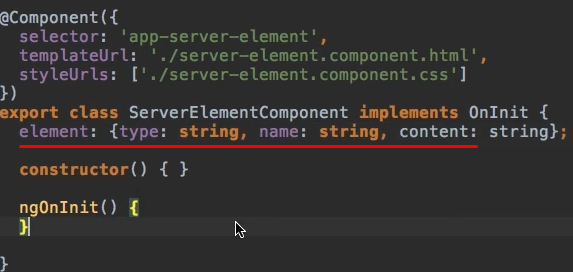
## Two-way-Binding

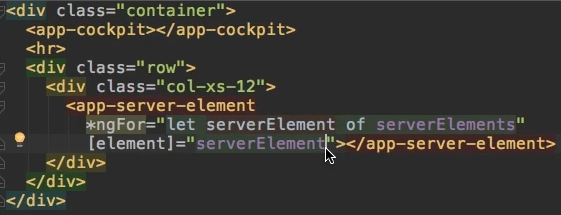
It is an even easier way to bind to some data. With two-way-bindig we can bind property and event binding. We use it by combining their syntaxes (square brackets and parantheses within) and by using a special directive **ngModel**.  
   
It will trigger on the input event and update the value of serverName in our component automatically. On the other hand, since it is two way binding it will also update the value of the input element if we change serverName somewhere else.

## What are Directives?

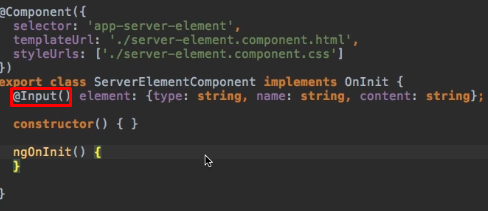
Directives are Instructions in the DOM.   
Screenshot_14.png  
ngIf is a frequently used directive in Angular. In this case we used the ngIf directive to output data conditionally, we just have to pass a boolean value to it.   
  
This is the case when we enhance the ngIf with an else condition.  
We can style elements dynamically with ngStyle. Screenshot_16.png  
  
We can output list by using the ngFor directive.  
Screenshot_18.png  


## Binding to Custom Properties



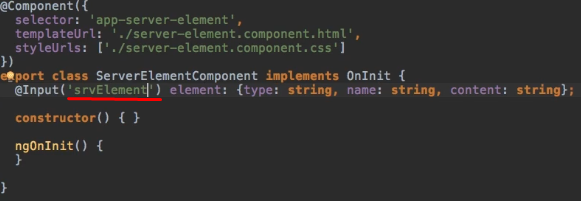
This is how we create our own property. This is also how you define types in TS. This property is part of this component only (of the server-element component). We can’t access it outside. By default every property is accessible only inside it’s component, not from outside. 

This is how you can call the **element** property in another component’s html file. Though this is not enough, you need to do one more thing to succeed with the custom property binding. All you need is a **Decorator**.

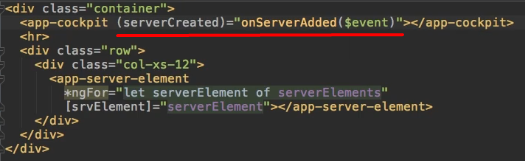
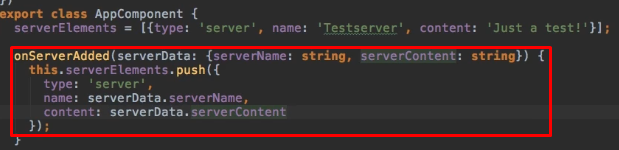
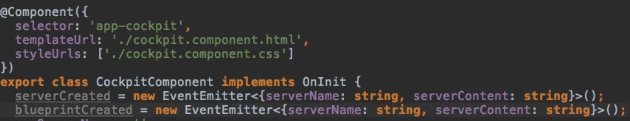
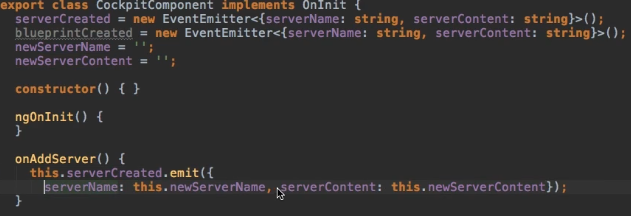


Now we can reach this property from outside.. from another component. Learning how Custom Property Binding works is crucial is Angular.. it’s super-important.

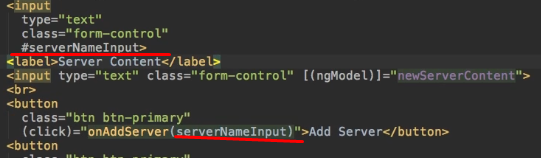
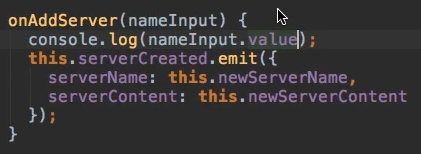
## Assigning an Alias to Custom Properties

Sometimes you don’t want to use the same property name outside of the component as you use inside of it.  


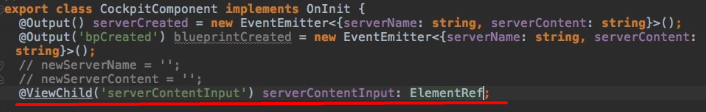
## Binding to Custom Events

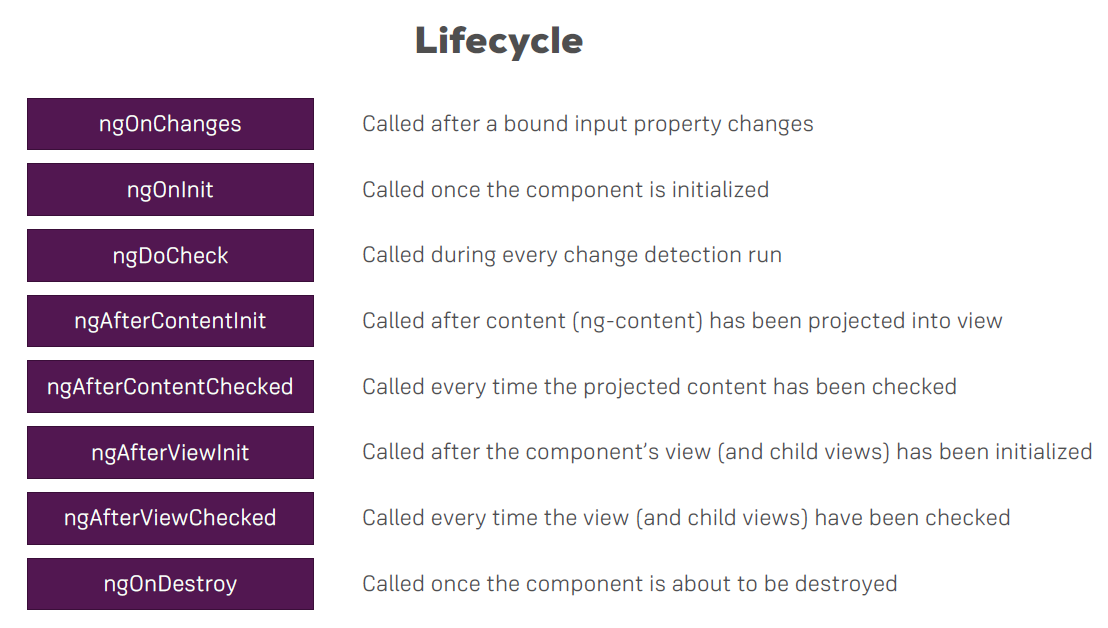
What if we have a component and something changes in there, and we want to inform our parent component (so the component, which implements the other component).   
  
We create our custom Event **serverCreated** and we assign a method to it. So when this event happens, the **onServerAdded()** method will be executed which is defined in the parent component’s TS file. And we even expect to get some data with **$event** – this is how you could catch data, that could be some object, which gives us the required information about server (like the name or the content).  
Screenshot_10.png  
This would be the type of object we expect to get in this method.  
  
And this is how you create Custom Events.   
  
And this is how you can emit the event.  
  
Now there is one more thing left we have to manage. To make the event listenable from outside. It is done with another **Decorator** – yes yes yes, with the **@Output()** decorator. To assign an alias to the event, you have to write the assigned name into the paranthesis like **@Output(‘srvCreated’).**

## Using local references in templates

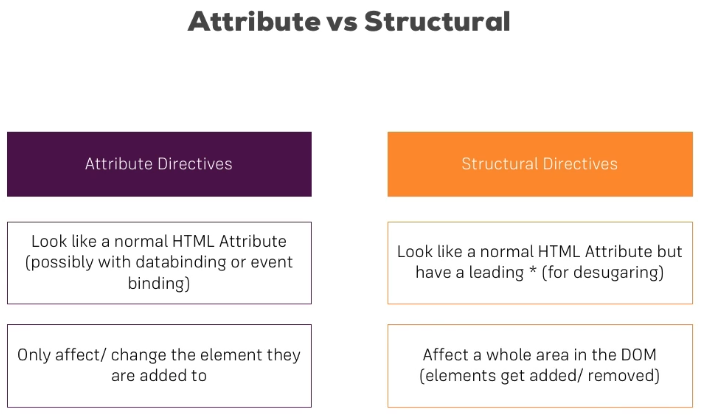
A local reference can be placed on any HTML element and you add it with hashtag like **#serverNameInput.** The reference will hold a reference to the given element. To the whole HTML element, with all it’s properties. You can use the references everywhere in your template, but (important) ONLY there. Local reference is a very nice feature to get access to some elements in your template and then use it even directly in the template.  
  


## Getting access to the template and DOM with @ViewChild

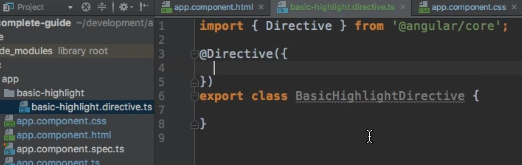
There is an alternative way to access templates.   
  
Here the **serverContentInput** is the local reference name itself, which is of course implemented in the template.

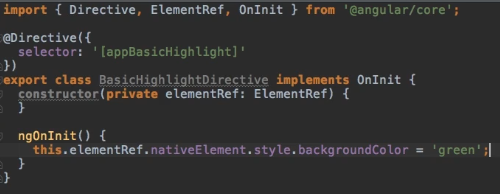


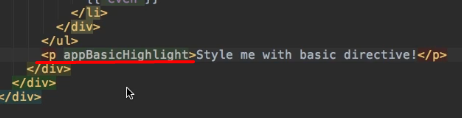
## Understanding Directives

There are 2 types of directives. **Attribute directives** and **Structural directives.** Attribute directives sit on elements, just like attributes. Structural directives are basically the same, but they also change the structure of the DOM around this element. If you have ng-if on a paragraph – if that condition is false – this paragraph is removed from the DOM, so the overall viewcontainer is affected. Whereas at the attribute directive you never destroy an element from the DOM, you only change properties of that element (for example the background color). 

# Creating a Basic Attribute Directive

It’s good to create a new folder for your custom directives. 

This is how you create a directive. It’s important to take care about the filename. Within the brackets (where the cursor is on the picture) you can configure the directive. The one thing, that the directive absolutely needs is a **selector**. 

This is how your directive looks, after it’s configuration is done. To use this directive, we have to do 2 things. First of all we have to inform Angular, that we have a new directive. We do it in the **app.module.ts** file, in the **declarations** part. 

That’s how we use it. And this is the reason it’s called **attribute directive**, because we use it just like an attribute (that means without any brackets). The reason behind this is we don’t need a selector (square brackets), because the square brackets in the selector’s name (on the 2nd picture) tell Angular ’’please select it as an attribute element’’.

Accessing elements directly like this is not a good practice though. We should use a different tool. The **Renderer.** The Renderer is a better approach of accessing the DOM.A screenshot of a cell phone

Description automatically generated

We should use the renderer for any DOM-manipulations. The HostBinding and HostListener are great ways for working with elements inside of a directive. A screenshot of a cell phone

Description automatically generated

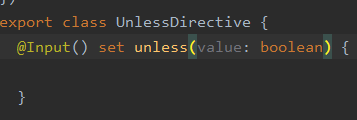
# Structural Directives

Structural directives are indicated by asterisk before their name (e.g. \*ngFor). To create a new directive using the CLI type: **ng g d *name****.*

A screenshot of a cell phone

Description automatically generated

The **unless** is still a property. The set is a method, that gets executed whenever the property changes outside of this directive.



**Unless** needs to receive the value the property would normally get as an input, and this will be a Boolean because it will be our condition at the end. So we can name it as condition instead of *value*.

A screenshot of a cell phone

Description automatically generated

This is how our Structural directive looks.

# ngSwitch

This is an example how **ngSwitch** works.

A close up of a screen

Description automatically generated

The **value** in this case is a variable which is declared in the TS file.

# Services and Dependency Injection

What are services? Service is another piece in your Angular app, another class you can add, which acts as a central repository, as a central business unit.. where you can store and centralize your code in. We could create a **LogService** to centralize our log statements. Or a **UserService** to store user data.

What is a Dependency Injector? A dependency is something a class of ours will depend on. The dependency injector injects an instance of this class into our component automatically. All we need to do is – we need to inform Angular, that we require such an instance. We add to the constructor of the dependent class the following types.

A screenshot of a cell phone

Description automatically generated

To use a service within another service, we need to use the **@Injectable** decorator. This decorator tells Angular, that something can be injected into this service. You always add this decorator to a service **WHERE** you want to inject something (so the receiving service). A screenshot of a cell phone

Description automatically generated

## Routing

Angular has it’s own router, which allows you to change the URL in the URL bar and still only use one page, but then exchange major parts or a lot of parts of that page, so that to the user it really looks like a new page was loaded, but behind the scenes it is still JS changing a lot of parts in your DOM, in your page. Routing should be configured first of all in the app.module.ts file like this:

A képen képernyőkép, szöveg látható

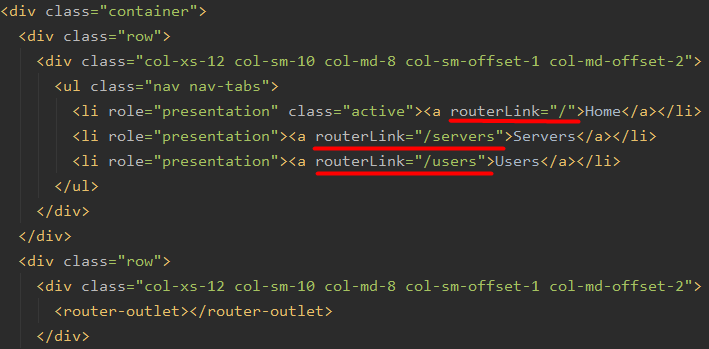
Automatikusan generált leírás

We also have to inform the HTML file of the app.component, to add the routing.

A képen képernyőkép, szöveg látható

Automatikusan generált leírás

This is the way how we should navigate around with **routerLink**. It allows as to reach other subpages without refreshing the pages, so it gives us a better user experience, it doesn’t restart the app, therefore it keeps the app’s state and it’s much faster, than reloading the page all the time.

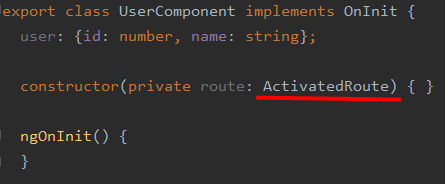


# Passing parameters to Routes

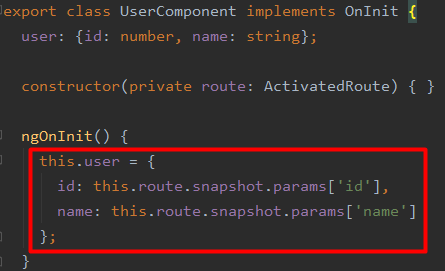
A képen képernyőkép látható

Automatikusan generált leírás

By adding the column, we define, that any given pathname (or parameter) would fit in there. That is a dynamic path segment.

The **ActivatedRoute** type is a reference to the currently loaded route. The **ActivatedRoute** object we injected will give us access to the id passed in the URL => Selected User. 

This is the way how we can fetch user data into our URL:



A képen óra, objektum látható

Automatikusan generált leírás

A képen képernyőkép látható

Automatikusan generált leírás

# Passing Query Parameters and Fragments

Adding key value pairs and fragments

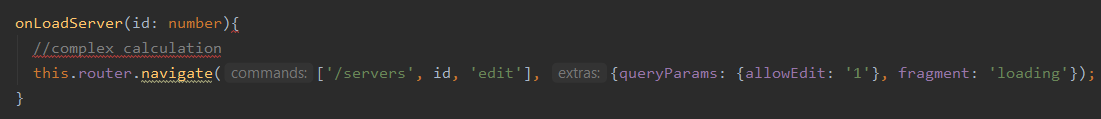
A képen képernyőkép, monitor látható

Automatikusan generált leírás

Doing the same programmatically:

A képen fal látható

Automatikusan generált leírás



The URL will be the same both times:

A képen óra, képernyőkép, út látható

Automatikusan generált leírás

# Retrieving Query Parameters and Fragments

A képen képernyőkép, szöveg látható

Automatikusan generált leírás

# Setting up child (nested) routes

It is done the following way:

A képen szöveg látható

Automatikusan generált leírás

But this is not enough, we need to change something more in the parent component’s HTML file.

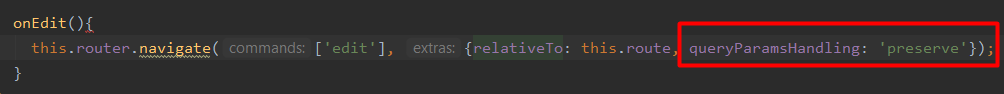
A képen képernyőkép, szöveg látható

Automatikusan generált leírás

Here where the child components selector was previously (**app-users**), we need to add the **router-outlet**.

# Configuring the handling of query parameters

Sometimes the following issue occurs: our query parameters are gone, whenever we navigate one level further for example. So we need to preserve them. So as an example solution, in a single component, where we navigate, we can use the **queryParamsHandling**, which takes a string as a value and this could be **merge** to merge our old query params with any new we might add. We can also use the **preserve** string. This option makes sure, that the old query params are kept, but if you want to add the new ones, use the merging.



# Redirecting and Wildcard Routes

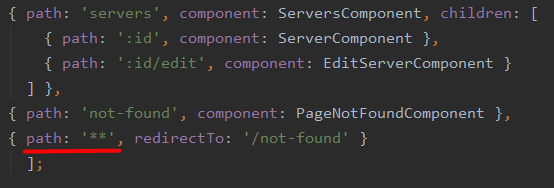
Sometimes maybe we want to redirect the user to a specifyc page, whenever he tries to visit a page we don’t have.

A képen fal, beltéri látható

Automatikusan generált leírás

This is how we use redirection.

The wildcard route can also be used, which catches all paths it doesn’t know about. We need to make sure the wildcard route is always the **last one** in our array of paths.



# Outsorcing the route configuration

Typically, if your have more than let’s say 2 or 3 routes, you don’t add it directly to the app module. Instead you add a new file, which is for the application as a whole, typically is called **app-routing.module.ts.** This module will handle all our routing-related tasks. We need to initialize it as a module by using **@NgModule.**

A képen szöveg látható

Automatikusan generált leírás

All we have left to do is just add the AppRoutingModule in our AppModule, at the imports section.

# Route Guards

It’s a functionality which is executed before a route is loaded, or once you want to leave a route. It allows us to run some code at a point of time defined by us. For this purpose we create a new service file which implements the **CanActivate** file, which forces us to use the canActivate method. This will make sure that servers are only accessible (and all its child routes) if the AuthGuard CanActivate method returns true, which will only happen if in the AuthService loggedIn is set to true.

export class AuthService {  
 loggedIn = false;  
  
 isAuthenticated(){  
 const promise = new ***Promise***(  
 (resolve, reject) => {  
 setTimeout(() => {  
 resolve(this.loggedIn);9  
 });  
 }  
 );  
 return promise;  
 }  
  
 login(){  
 this.loggedIn = true;  
 }  
  
 logout(){  
 this.loggedIn = false;  
 }  
}

A képen képernyőkép látható

Automatikusan generált leírás

A képen szöveg látható

Automatikusan generált leírás

The opposite of **canActivate** (controls access to a route) is the **canDeactivate**, which basically controls when you are allowed to leave a specific route. We create it’s own file because it is a service too.

A képen képernyőkép látható

Automatikusan generált leírás

A képen képernyőkép látható

Automatikusan generált leírás