



Angular

Instructor : Youssef RAFII
January 2021

Training plan

First Day

Introduction &
Versions

Installation

Basics

Directives

Second Day

Databinding

Pipes

Services &
Dependency
Injection

Third Day

Routing

Observables

Forms: Template
driven

Fourth Day

Forms: Reactive

Making Http
Requests

Authentication

First Day

Introduction & Versions



Installation



Basics



Directives

Introduction & Versions

Introduction & Versions



Angular is a typescript-based front-end platform that makes it easy to build applications with.

Angular is Open Source and primarily maintained by Google.

Introduction & Versions

2

- ❖ May 2016
- ❖ Complete Re-Write of AngularJs
- ❖ Architecture is component based
- ❖ Supports ES6 and TS 1 to 1.8

Introduction & Versions

4

- ❖ Released on March 23, 2017.
- ❖ Angular 2 compatible
- ❖ Skipping V3 to avoid a confusion due to the misalignment of the router package's version
- ❖ Lot of performance improvement
- ❖ Added supports for email validation pattern
- ❖ Supports TS 2.1 & 2.2
- ❖ Instead of writing 2 ngIf, else block is introduced.
- ❖ Introduced **HttpClient**, a smaller, easier to use, and more powerful library for making HTTP Requests.

Introduction & Versions

5

- ❖ Released on November 1, 2017
- ❖ `@angular/http` is replaced with `@angular/common/http` library
- ❖ Add supports for Number, Date and Currency pipes
- ❖ Build Optimizer and improvements of Material Desing.
- ❖ Build optimizations
- ❖ Improvements of the perfs
- ❖ Supports TS 2.3

Introduction & Versions

6

- ❖ Released on May 4, 2018
- ❖ No major breaking changes
- ❖ I18N introduced (No requirement to build one application by locale)
- ❖ Angular CLI Changes: Two new commands have been introduced
 - ng update <package>
 - ng add <package>

Introduction & Versions

7

- ❖ Released on October 18, 2018
- ❖ Various improvements in the performance
- ❖ TypeScript 3.1 Support
- ❖ Angular CLI prompt user, to help him to discover the in-built SCSS support or routing.

Introduction & Versions

8

- ❖ Released on *May 28, 2019*
- ❖ Internal changes
- ❖ `@angular/http` is no longer supported, use `@Angular/common/http` instead
- ❖ `ViewChild` changed temporary
- ❖ Supports TS 3.4
- ❖ Use a new compiler(IVY)

Introduction & Versions

9

- ❖ was released on February 6, 2020
- ❖ Smaller builds
- ❖ Internal changes
- ❖ Speed and performance
- ❖ Faster testing
- ❖ Improved CSS class and style binding
- ❖ Improved Debugging.
- ❖ ViewChild returned as before

Introduction & Versions

10

- ❖ was released on June 24, 2020
- ❖ Internal changes
- ❖ Keeping Up to Date with the Ecosystem(TypeScript 3.9, TSLint v6)
- ❖ made several new deprecations and removals from Angular.

Introduction & Versions

11

- ❖ was released on Nov 11, 2020
- ❖ Faster Builds
- ❖ The only IE version now still being supported is 11
- ❖ Improved Logging and Reporting
- ❖ Supports TS 4.0

What will be done during this training

 Fleet Management

Sign Up

Log In

Email *

yra@gmail.com



Password *



login

Switch to Sign Up

Code in a git repository by Steps

Manage topics

3 commits 5 branches 0 packages 0 releases 1 contributor




Branch: Step01 New pull request Create new file Upload files Find file Clone or download

youma85 Step 01 creation of needed component Latest commit 6e1327c 2 days ago

e2e	initial commit	9 days ago
src	Step 01 creation of needed component	2 days ago
.editorconfig	initial commit	9 days ago
.gitignore	initial commit	9 days ago
README.md	initial commit	9 days ago
Step01.md	Step 01 creation of needed component	2 days ago
angular.json	Step 01 creation of needed component	2 days ago
browserslist	initial commit	9 days ago
karma.conf.js	initial commit	9 days ago
package-lock.json	Initial Commit	9 days ago
package.json	Initial Commit	9 days ago
tsconfig.app.json	initial commit	9 days ago
tsconfig.json	initial commit	9 days ago
tsconfig.spec.json	initial commit	9 days ago
tslint.json	initial commit	9 days ago
README.md		

Installation

Setup Environment

	Node JS	<u>https://nodejs.org/en/download/releases/</u>
	VsCode	<u>https://code.visualstudio.com/</u>
	Angular Cli	<u>https://cli.angular.io/</u>

Node Js



Download the current version from here: <https://nodejs.org/en/download/current/>

To check your version, run:

`node -v` in a terminal/console window.

Angular Cli



Command Line Interface

Installation:

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

Update to the latest version:

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

```
npm cache clean -force
```

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


CLI Overview

`ng add`: Adds support for an external library to your project.

`ng build`(`ng b`): Compiles an Angular app into an output directory named `dist/` at the given output path.

`ng deploy`: Invokes the deploy builder for a specified project or for the default project in the workspace.

`ng doc` (`ng d`): Opens the official Angular documentation (angular.io) in a browser, and searches for a given keyword.

`ng generate`(`ng g`): Generates files based on a schematic.

For more informations: <https://angular.io/cli>

Basics



Angular

“Angular is a platform and framework for building single-page client applications using HTML and TypeScript. Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your apps.”: From the Official WebSite.

The basic building blocks are NgModules, which provide a compilation context for components.

An app always has at least a root module that enables bootstrapping, and typically has many more feature modules.

Create new Angular App

- Run the CLI command `ng new` and provide the name as shown here:

```
ng new my-app
```

The `ng new` command prompts you for information about features to include in the initial app. Accept the defaults by pressing the Enter or Return key.

- Go to the workspace folder (`my-app`).
- Launch the server by using the CLI command `ng serve`, with the `--open` option:

```
cd my-app
```

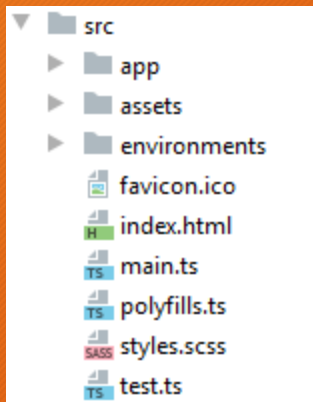
```
ng serve
```

The `ng serve` command launches the server, watches your files, and rebuilds the app as you make changes to those files.

The root content

Asset	Role
e2e	Contains the end to end tests scripts.
node_modules	Contains all external modules and libraries used in the application
src	The main workarea/app code resides inside this folder
angular.json	Configuration file that contains all the app configurations
karma.conf.js	Configuration file of karma runner of unit test
tsconfig.json	Build and compilation of the application
tslint.json	Linting or coding standards
package.json	Manifest of the project that includes the dependencies, information about its unique source control, project's name, description, and author, etc
package-lock.json	describe the exact dependency tree currently installed

The contents src folder



Asset	Role
app	Contains the primary App components, modules, directives,...
assets	Store static assets like images, styles, ...
environments	Environment configurations to allow you to build for different targets, like dev, qualification or production
favicon.ico	Image displayed as browser favorite icon
index.html	Root HTML file for the application
main.ts	Booting the web application
polyfills.ts	Imports some common polyfills required to run Angular properly on some outdated browsers
styles.css	Global stylesheet
test.ts	Unit test entry point, not part of application

App component

- App component is the root of the application

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.scss']  
})  
export class AppComponent {  
  
}
```

Import the component annotation

Define the component
and its properties

Create the component controller,

This component is used in the src/index.html file which is the HTML page hosting the Angular application.

App module

The App module is the packaging that tells Angular what's available to render

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
```

Imports Angular dependencies needed

```
import { AppComponent } from './app.component';
```

Imports the App component

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Uses the NgModule annotation to define a module by passing an object

Declarations are to list any components and directives used in the app.

Imports are other modules that are used in the app.

Providers are any services used in the app.

Bootstrap declares which component to use as the first to bootstrap the application.

The "root module" is a classic module whose particularity is to define the "root component" of the application via the bootstrap property.

Bootstrapping the app

main.ts

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

← If production is enabled, turn off Angular developer mode.

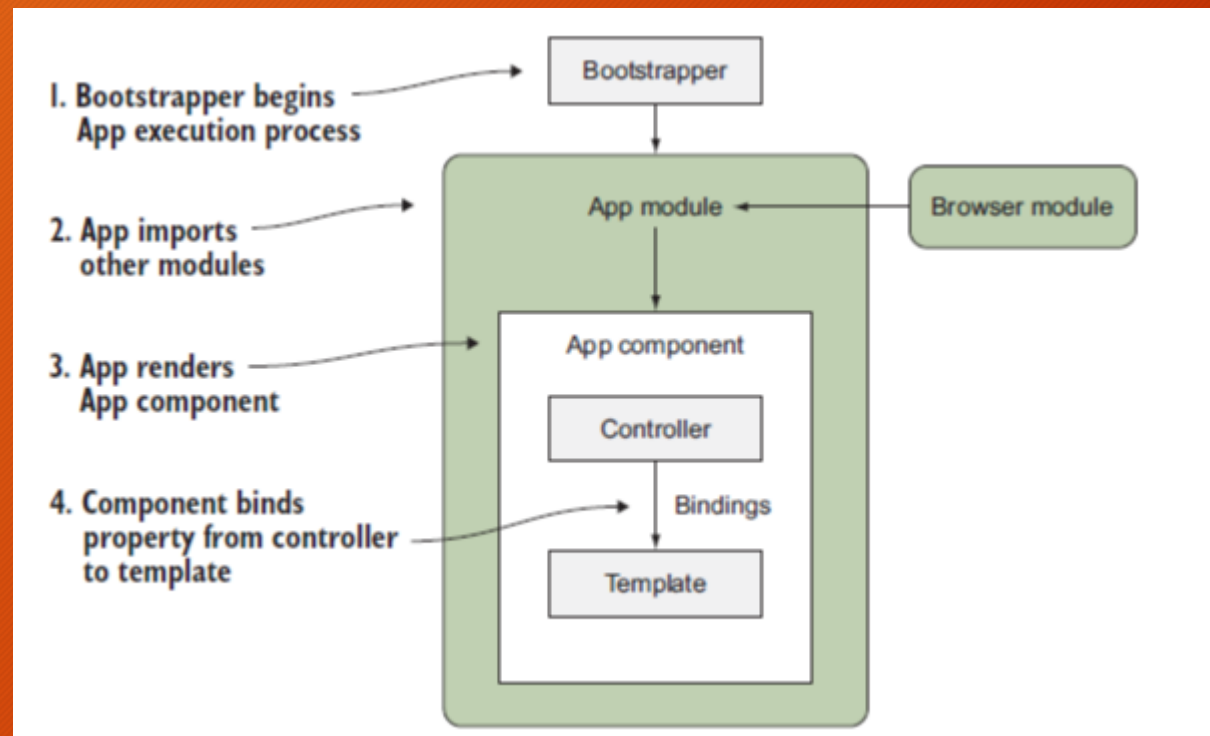
← Bootstraps the App module

index.html

```
<body>
  <app-root></app-root>
</body>
```

← Angular will look for the app-root element and replace it with the rendered component.

Bootstrapping the app



Basics: Modules (definition)

Angular offers a concept of modules to better structure the code and facilitate reuse and sharing.

An Angular module is a mechanism allowing to:

- group together components (but also services, directives, pipes, etc.),
- define their dependencies,
- and define their visibility.

An Angular module is defined simply with a class (usually empty) and the NgModule decorator.

Basics: Modules(definition)

declarations:

Defines the list of components (or directives, pipes etc ...) contained in this module.

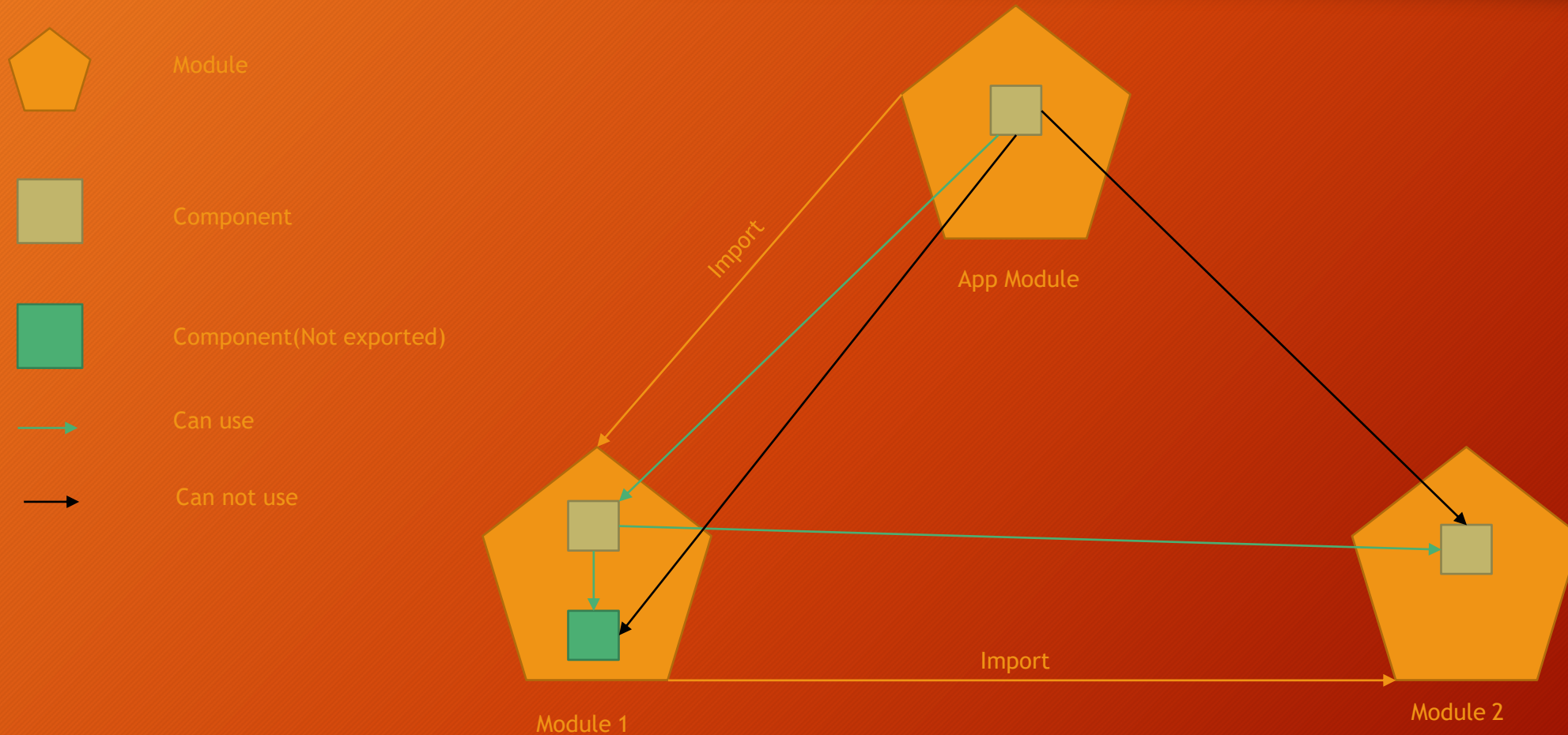
exports:

Defines the list of components that can be used by modules that import this one.

imports:

Defines the list of module dependencies. This is usually the list of modules containing the components used by the components in the declarations section

Basics: Modules(imports)



Basics: Modules(Types)

Root Module

- The App Module can also be called the root module. Every app must contain at least one module and that is the App Module. We launch our applications by bootstrapping the root module.

Core Module

- The Core Module is where we want to put our shared singleton services.

Feature Module

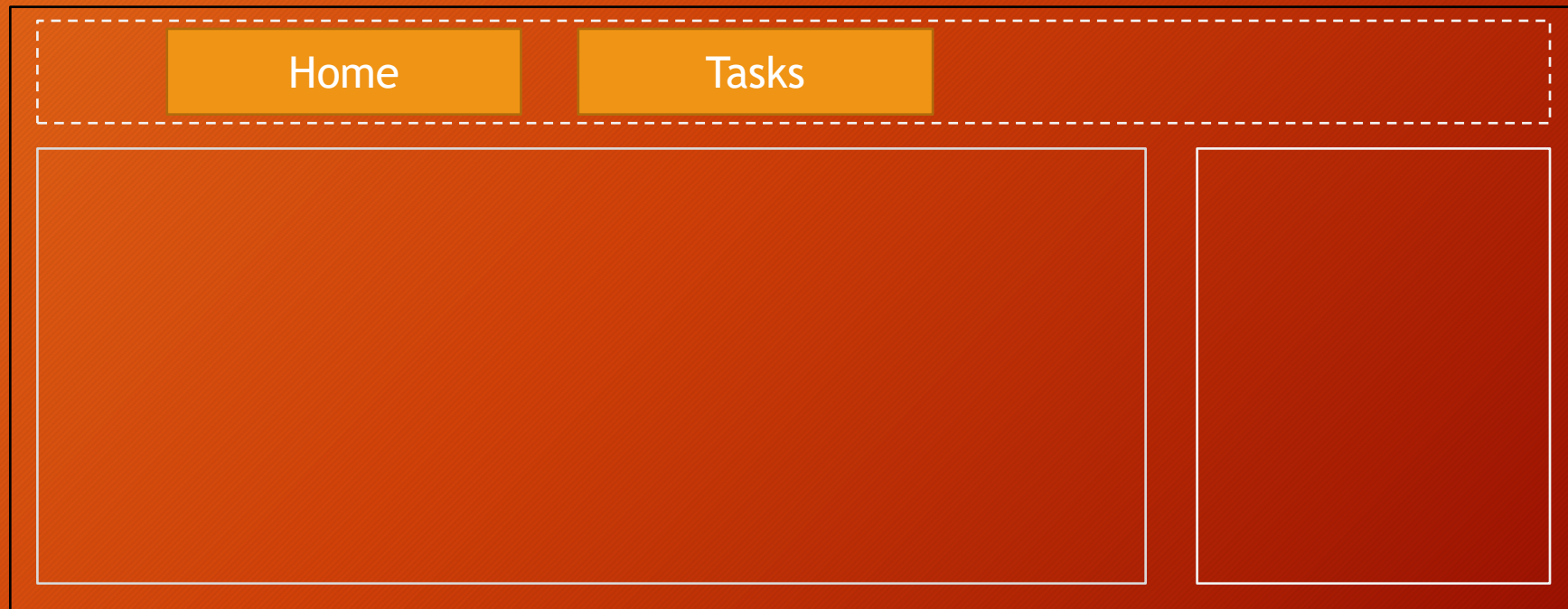
- The components (as well as the directives, pipes, etc.) are grouped together in modules by functionality. These modules are then called "feature module".

Shared Module

- The Shared Module is where we want to store common modules, components, etc. Everything is shared throughout the application.

Basics: Components

- Components are the most basic UI building block of an Angular app. An Angular app contains a tree of Angular components.



Basics: @Component

A component is a class decorated with @Component,

This decorator takes as parameter a configuration which contains at least:

- selector: the CSS selector which will link the HTML tag of the element and the code of the component.
- template: the HTML template used by Angular to generate the element's content in the DOM.

Basics: Components creation

We can create component in 2 ways:

- Using angular cli:

ng generate component tasks or shortly (*ng g c tasks*)

- Manually by creating the class and declare it in the app module file:

```
@Component({  
  selector: 'app-student',  
  templateUrl: './student.component.html',  
  styleUrls: ['./student.component.scss']  
})  
export class StudentComponent {  
  
}
```

```
@NgModule({  
  declarations: [  
    AppComponent,  
    StudentComponent,  
  ]  
})
```

Basics: Component selector type

- `app-root`: used as `<app-root>`
- `[app-root]`: used as `<div app-root>`
- `.app-root`: used as `<div class="app-root">`

Basics: Databinding

Databinding = communication between ts code and what you see (html)

String Interpolation

Allow us to display information from typescript in the html code like variables or functions. E.g.: `{{student.firstName}}`

Property binding

[property]: indicate to angular that we want to dynamically bind some property of the tag. E.g.:

```
<button [disabled]="false"></button>
```


Basics: Databinding

Event binding

Event binding allows you to listen for certain events such as keystrokes, mouse movements, clicks, and touches.

```
<button (click)="onNewStudent()">New Student</button>
```

```
onNewStudent(){  
  ...  
}
```

Two-Way-databinding

For Two-Way-Binding to work, you need to enable the `ngModel` directive by adding the **FormsModule** to the `imports[]` array in the `AppModule`.

With two way databinding we combine property and event binding.

```
<input name="lastName" [(ngModel)]="student.lastName" >
```

Directives

Directives: Definition

Directives are classes which allow you to enrich and modify the view by simply adding HTML attributes to the template.

There are three kinds of directives in Angular:

- 1.Components: directives with a template.
- 2.Structural directives: change the DOM layout by adding and removing DOM elements, for example `*ngFor` and `*ngIf`.
- 3.Attribute directives: change the appearance or behavior of an element, component, or another directive, for example `ngStyle`

Directives: NgIf

```
<p *ngIf="boolean"></p>
```

With else

```
<p *ngIf="form.isValid; else elseBlock ">  
  Form is Valid  
</p>
```

```
<ng-template #elseBlock>  
  Form is not Valid  
</ng-template>
```


Directives: NgStyle

with a Object:

```
<button [ngStyle]="{background: 'red'}">Click Me!</button>
```

with a function:

```
<p [ngStyle]="{backgroundColor:getColor()}"></p>
```

```
getColor(){  
  return this.status==='done'? 'green': 'red';  
}
```

Directives: NgClass

with a class name:

```
<p [ngClass]="online">paragraph 1</p>
```

with a multiple class name:

```
<p [ngClass]="online active">paragraph 1</p>
```

Conditionnal application of classes

```
<p [ngClass]="{ online: false, active: false }">paragraph 1</p>
```

With databinding

```
<p [ngClass]="{ online: !isValid }">paragraph 1</p>
```

```
.online {
  color: blue;
}

.active {
  background-color: yellow;
}
```

Directives : NgFor

```
*ngFor="let std of students"
```

Get the index

```
*ngFor="let std of students; let i= index"
```


Directives: NgSwitch

```
<div [ngSwitch]="value">  
  <p *ngSwitchCase="5">Value is 5</p>  
  <p *ngSwitchCase="10">Value is 10</p>  
  <p *ngSwitchCase="100">Value is 100</p>  
  <p *ngSwitchDefault>Value is default</p>  
</div>
```

Directives: Custom directives

Creation with CLI: `ng g d directive_name`

```
@Directive({
  selector: '[appShadow]'
})
export class ShadowDirective{

  constructor(elem: ElementRef, renderer: Renderer2) {
    renderer.setStyle(elem.nativeElement, 'box-shadow', '2px 2px 12px #602593');
  }
}
```

Must be declared in the app.module

```
@NgModule({
  declarations: [
    AppComponent,
    StudentComponent,
    ShadowDirective
  ],
```

```
<div appShadow>
  ...
</div>
```

ElementRef : The "Dependency Injection" allows to retrieve via the ElementRef class, a reference to the object allowing to handle the associated DOM element

Directives Listener

```
@HostListener('mouseenter')
mouseOver(eventData:Event){
  this.elementRef.nativeElement.style.backgroundColor='yellow';
}

@HostListener('mouseleave')
mouseLeave(eventData:Event){
  this.elementRef.nativeElement.style.backgroundColor='transparent';
}
```

@HostListener () is a decorator allowing to add a "listener" on the element on which the directive is applied ("host element").

Using HostBinding to Bind to Host Properties

Bind `style.backgroundColor` to the property `backgroundColor`:

```
@HostBinding('style.backgroundColor') backgroundColor:string='transparent';

@HostListener('mouseenter') mouseOver(eventData:Event){
  this.backgroundColor='yellow';
}

@HostListener('mouseleave') mouseLeave(eventData:Event){
  this.backgroundColor='transparent';
}
```

Second Day

Components & Databinding



Pipes



Services & Dependency Injection

Components & Databinding

Input: Property Binding

To transmit data to a "child component", we will communicate with it in the same way as we control the properties of a native element, that is to say using the Property Binding:

```
<app-student-item  
  *ngFor="let std of students"  
  [student]="std">  
</app-student-item>
```

This block of code will call an implicit "set" of the «student» property of the instance of the child component.

Input: @Input Decorator

By default, no component property can be changed by Property Binding.

It is therefore necessary to define the properties that can serve as "input" to the component by simply adding the decorator @Input().

```
export class StudentItemComponent implements OnInit {  
  @Input() student: Student;  
}
```

Input: Binding To custom property

```
@Component({
  selector: 'app-student-list',
  template: `
<app-student-item
  *ngFor="let std of students"
  [student]="std">
</app-student-item>`
})
export class StudentListComponent implements OnInit {
  students: Student[] = [];
}
```

```
@Component({
  selector: 'app-student-item',
  template: `
<h3>Name: {{student.name}}</h3>
` })
export class StudentItemComponent {
  @Input() student: Student;
}
```


Input: Binding To custom property(alias)

```
@Component({
  selector: 'app-student-list',
  template: `
<app-student-item
  *ngFor="let std of students" [stdElm]="std">
</app-student-item>`
})
export class StudentListComponent implements OnInit {
  students: Student[] = [];
}
```

```
@Component({
  selector: 'app-student-item',
  template: `
<h3>Name: {{student.name}}</h3>
` })
export class StudentItemComponent {
  @Input('stdElm') student: Student;
}
```

Output: Event Binding

In the same way that Inputs allow you to communicate data to a "child component", Output can transmit data to the "parent component" via an "Output" mechanism similar to the Event Binding used previously to capture native events.

```
<app-student-item  
  *ngFor="let std of students"  
  (studentSelected)="onSelectingStudent(std)">  
  
</app-student-item>
```

The expression `onSelectingStudent(std)` allows to register a listener for the «studentSelected» event.

Note the similarity with the Event Binding on click event(or Other DOM events).

```
<button (click)="onSave()">Save</button>
```

Output: Declaration of property and decorator @Output ()

By simply declaring the studentSelected property on the app-student-item component:

```
export class StudentItemComponent{  
  studentSelected= new EventEmitter<void>();  
}
```

.. nothing happens, you must add the decorator @Output():

```
export class StudentItemComponent{  
  @Output() studentSelected= new EventEmitter<void>();  
}
```

@Output allows you to make the eventEmitter listenable from the outside

Output: Sending values

As indicated by his name, an EventEmitter allows you to emit values. It can therefore be used anywhere in the StudentItemComponent class to pass values to the parent component via the emit method.

```
<button mat-raised-button (click)="onClick()">Show</button>
```

```
onClick() {  
  this.studentSelected.emit();  
}
```

Output: Binding To custom event

```
@Component({
  selector: 'app-student-list',
  template: `
    <app-student-item
      *ngFor="let std of students"
      [student]="std"
      (studentSelected)="onStudentSelected($event)"></app-student-
item>
  `
})
export class StudentListComponent implements OnInit {
  students: Student[] = [
  ];

  onStudentSelected(student: Student) {
    console.log(student);
  }
}
```

```
@Component({
  selector: 'app-student-item',
  template: `
    {{student.firstName}} {{student.lastName}}
    <button mat-raised-button (click)="onClick()">Show</button>
  `
})
export class StudentItemComponent {
  @Input() student: Student;
  @Output() studentSelected = new EventEmitter<Student>();

  onClick() {
    this.studentSelected.emit(this.student);
  }
}
```

Local reference

```
<input
  type="text"
  class="form-control"
  #studentName>

<button
  class="btn btn-primary"
  (click)="onAddServer(studentName)">Add Server</button>
```

```
onAddServer(name:HtmlInputElement){
  console.log(name.value);
}
```

Or, we can manipulate the local reference from typeScript via ViewChild:

```
@ViewChild('studentName') studentName: ElementRef;
```

Pipes

Pipe

Pipes are filters that can be used directly from the view in order to transform the values during the binding.

The Pipes syntax is simply inspired by UNIX shell Pipes found in many templating systems.

Pipes are added into template expressions using the pipe character (|).

```
<div>{{ user.firstName | lowercase }}</div>
```

```
{{ server.started | date }}
```

Angular has several native "pipes": <https://angular.io/api?type=pipe>

Pipe with parameters

Pipes can take parameters that must be putted after the Pipe and separated with the symbol ":".

```
<div>{{ user.firstName | slice:0:10 }}</div>
```

```
{{ student.birthDate | date:'short' }}
```

Chaining

The "pipes" can be chained.

```
<div>{{ student.firstName | slice:0:10 | lowercase }}</div>
```

```
{{ server.started | date:'fullDate' | uppercase }}
```


Creating a Custom Pipe

To create a custom Pipe, you need:

1. Create a class that implements the PipeTransform interface,
2. decorate this class with the decorator `@Pipe ()` by indicating the name of the Pipe.
3. add the class to the declarations (and exports) of the associated module.

Creation with angular CLI: `ng g p namePipe`

Creating a Custom Pipe

```
import { PipeTransform } from "@angular/core";

@Pipe({
  name:'shorten'
})
export class ShortenPipe implements PipeTransform{
  transform(value:any){
    if(value.length>10){
      return value.substr(0,10)+'...';
    }
    return value;
  }
}
```

```
{{ student.firstName | shorten }}
```

parametrizing a custom pipe

```
import {PipeTransform} from "@angular/core";

@Pipe({
  name: 'shorten'
})
export class ShortenPipe implements PipeTransform {
  transform(value: any, limit: number) {
    if (value.length > limit) {
      return value.substr(0, limit) + '...';
    }
    return value;
  }
}
```

```
{{ student.firstName | shorten:5 }}
```

```
{{ student.firstName | shorten:5:arg2:arg3 }}
```

Services & Dependency Injection



What is "Dependency Injection"?

The "Dependency Injection" is a "design pattern" which consists in separating the instantiation (and therefore the implementation) of a dependency and its use.

It decreases coupling between a class and its dependency.
Extending the application becomes easier.
Helps in Unit testing.

Injection of an Angular Service

With Angular, a dependency is generally the instance of a class allowing to factorize certain functionalities or to access a state thus allowing the components to communicate with each other.

In the Angular vocabulary, these classes are called "services".

An Angular service can be injected by any Angular class (i.e.: component, Directive, Service or Pipe) via the parameters of its constructor.

```
export class StudentListComponent {  
  
  constructor(private studentService: StudentService) {  
  }  
}
```

Declaration of a Service

To declare an Angular service, just create a TypeScript class and decorate it with the decorator `@Injectable()`.

```
@Injectable()
export class StudentService {
}
```

`@Injectable()`: Decorator that marks a class as available to be provided and injected as a dependency.

```
@NgModule({
  providers: [StudentService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

In order to instantiate a service, Angular needs a "provider" telling it how to produce the instance of this service.

This is generally done via the providers property in the app module.

Create Service : `ng g s ServiceName`

Scope of Services

AppModule

Same instance of
Service is
available
**Application-
wide**

AppComponent

Same instance of
Service is
available for all
Components(but
Not for other
services)

Any Other Component

Same instance of
Service is
available for the
**Component and
all its child
components**

Services in Angular 6+

Since Angular 6, it is no longer necessary to define "providers" at the module level.

Instead of adding a service class to the `providers[]` array in `AppModule`, you can set the following config in `@Injectable()`:

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


Third Day

Routing



Observables



Forms: Template driven

Routing



Routing definition

Routing is the mechanism that allows you to navigate from one page to another on a website.

The Angular router is a core part of the Angular platform.

It enables developers to build Single Page Applications with multiple views and allow navigation between these views.

Setting up of "Routing"

The "Routing" configuration is transmitted to the RouterModule module when it is imported by the AppModule "root module".

it is recommended to place this configuration in a dedicated module AppRoutingModule imported by the "root module" AppModule.

```
const appRoutes: Routes = [  
  {path: '', redirectTo: '/students', pathMatch: 'full'},  
  {path: 'students', component: StudentComponent},  
  {path: 'classrooms', component: ClassroomComponent}  
];  
  
@NgModule({  
  imports: [RouterModule.forRoot(appRoutes)],  
  exports: [RouterModule]  
})  
export class AppRoutingModule {  
}
```

ng generate module app-routing --flat

Setting up routes with parameters

The path of a "route" can define mandatory parameters with the prefix «:»

```
{path: 'students/:id', component: StudentDetailsComponent}
```

<router-outlet>

The configuration of "Routing" allows you to define which component to display according to the route but this does not tell Angular where to inject the component in the page.

To indicate the the loading location of the routes, use the <router-outlet> directive directly in the "root component" AppComponent (or another one).

```
<router-outlet></router-outlet>
```

Navigating with Router Links

Using native links ``, the "browser" will generate an HTTP GET request to the server and reload the entire application.

To avoid this problem, the Angular Routing module provides the `routerLink` directive which allows to intercept the click event on the links and to change the "route" without reloading the whole application.

```
<a mat-button routerLink="/students">Students</a>
```


Navigating Programmatically

The navigation can be done programmatically with the Router object

```
import {Router} from "@angular/router";  
...  
constructor(private router: Router) { }  
  
...  
  
this.router.navigate(['/students']);
```


Styling Active Router Links

```
<a mat-button routerLink="/" routerLinkActive="active" [routerLinkActiveOptions]="{exact:true}">Home</a>  
<a mat-button routerLink="/students" routerLinkActive="active">Students</a>  
<a mat-button routerLink="/classrooms" routerLinkActive="active">Classrooms</a>
```

`[routerLinkActiveOptions]="{exact:true}"`

Do not activate the routerLinkActive only if the url is exactly '/'

Child(Nested) Route

A child route is like any other route, in that it needs both a path and a component.
The one difference is that you place child routes in a children array within the parent route

```
{path: 'students', component: StudentComponent, children: [  
  {path: '', component: StudentListComponent},  
  {path: 'new', component: CreateStudentComponent},  
  {path: ':id', component: UpdateStudentComponent}  
]},
```

Must add <router-outlet></router-outlet> in the Html of the Student component

Passing Parameters to Routes

Configuration:

```
...  
{path: ':id', component: StudentDetailsComponent}  
...
```

Calling the route:

```
<a [routerLink]="['/students',5]" ></a>
```

Or programmatically

```
this.router.navigate(['/students', idStudent]);
```

Fetching Route Parameters

```
import {ActivatedRoute} from "@angular/router";

constructor(private route: ActivatedRoute) {
  this.id = this.route.snapshot.params['id'];
}
```


Fetching Route Parameters Reactively

```
constructor(private route: ActivatedRoute) {  
  
  this.id = this.route.snapshot.params['id'];  
  
  this.route.params  
    .subscribe(  
      (params: Params) => {  
        this.id = params['id'];  
      });  
}
```

QueryParams

```
<a [routerLink]="['/students']" [queryParams]="{allowEdit:'1'}"></a>
```

Or programmatically

```
this.router.navigate(['students'],{queryParams:{allowEdit:'1'}});
```

How to access these values:

```
this.route.snapshot.queryParams
```

With observable:

```
this.route.queryParams.subscribe();
```

Redirecting and Wildcard Routes

```
{path:'not-found',component:PageNotFoundComponent},  
{path:'**',redirectTo:'/not-found'}
```

The path:'**' must be the last route in the appRoutes

Guards

"Guards" are used to control access to a "route" (e.g. authorization) or departure from a "route".

Configuration:

The "Guards" are added in "Routing" configuration:

```
{path: 'classrooms', component: ClassroomComponent, canActivate:[IsSignedInGuard]}  
{path: 'student-details', component: DetailsComponent, canDeactivate:[IsNotDirtyGuard]}
```


CanActivate

An activation guard is a service that implements the CanActivate interface.

This service must therefore implement the canActivate method which is called on each request for access to the "route"; it must then return a value of type boolean or Promise <boolean> or Observable <boolean> indicating whether access to the "route" is authorized or not.

```
@Injectable({
  providedIn: 'root'
})
export class IsSignedInGuard implements CanActivate {

  constructor(private authService: AuthService) {
  }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    return this.authService.isAuthenticated();
  }
}
```

CanDeactivate

A deactivation guard is a service that implements the CanDeactivate interface.

This service must implement the canDeactivate method.

This method is called whenever the user wishes to leave the route. it must then return a value of type boolean or Promise <boolean> or Observable <boolean> indicating whether access to the "route" is authorized or not.

```
@Injectable({
  providedIn: 'root'
})
export class IsNotDirtyGuard implements CanDeactivate<ProfileViewComponent> {
  canDeactivate(component: ProfileViewComponent,
    currentRoute: ActivatedRouteSnapshot,
    currentState: RouterStateSnapshot,
    nextState?: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
    return component.isDirty();
  }
}
```

```
export class ProfileViewComponent {
  isDirty() {
    return false;
  }
}
```

Lazy Loading

When the application starts, Angular loads all the modules and all the entities of the application.

If the application is very large then the startup will be slower.

To avoid these scalability problems, Angular allows modules to be loaded on demand (i.e. "Lazy Loading") in order to ease the initial load of the application.

Lazy Loading configuration

The configuration of "Lazy Loading" is done in the "Routing" configuration.

The AppRoutingModule "Routing" module can delegate the "Routing" management of a part of the application to another module. This "Lazy Loaded" module will therefore be loaded asynchronously when visiting the "routes" for which it is responsible.

```
{  
  path: 'classrooms',  
  loadChildren: './classrooms/classroom-routing.module#ClassroomRoutingModule'  
},
```

This configuration delegates the "Routing" of all the /classrooms/... part of the application to the ClassroomRoutingModule module.

Observables

Observables

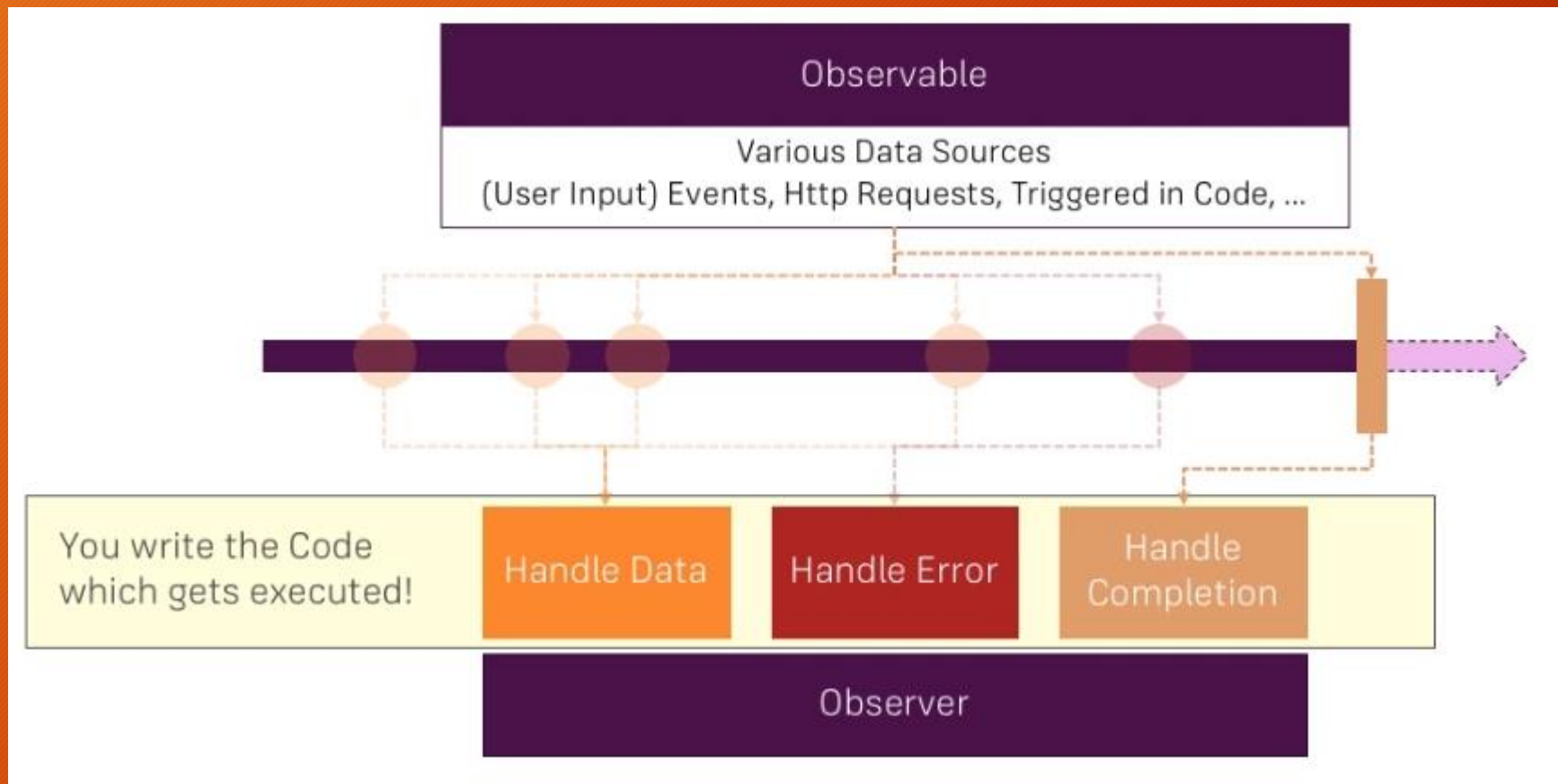
An Observable is an object allowing to make the “link” between publishers and subscribers.

An Observable is basically a function that can return a stream of values to an observer over time.

Angular makes use of observables as an interface to handle a variety of common asynchronous operations.

Observables will emit events which will be intercepted by observers.

Observables



Observers

A handler for receiving observable notifications implements the Observer interface.

NOTIFICATION TYPE	DESCRIPTION
next	Required. A handler for each delivered value. Called zero or more times after execution starts.
error	Optional. A handler for an error notification. An error halts execution of the observable instance.
complete	Optional. A handler for the execution-complete notification.

Promises(1/2)

Promises are objects that promise they will have value in the near future - either a success or failure.

Promises have their own methods which are then and catch. `.then()` is called when success comes, else the `catch()` method calls.

Promises are created using the promise constructor:

```
new Promise( /* executor */ function(resolve, reject) { ... } );
```

If the asynchronous operations are successful then the expected result is returned by calling the resolve function by the creator of the promise. Similarly if there was some unexpected error the reasons is passed on by calling the reject function.

Promises(2/2)

```
const promiseToReader = new Promise((resolve, reject) => {  
  setTimeout(function() {  
    if (userLikedTheArticle) {  
      resolve('This article is awesome!')  
    } else {  
      reject('Not very good')  
    }  
  }, enoughToReadArticle)  
})
```

```
promiseToReader.then(result => {  
  console.log('result', result)  
})
```

```
promiseToReader.catch(error => {  
  console.error('error', error)  
})
```

Promises vs observables

Observables are lazy whereas promises are not:

observables are lazy, that is we have to subscribe observables to get the results. In the case of promises, they execute immediately.

Observables handle multiple values unlike promises:

Promises can only provide a single value whereas observables can give you multiple values.

Observables are cancelable:

You can cancel observables by unsubscribing it using the unsubscribe method whereas promises don't have such a feature.

Observables provide many operators:

There are many operators like map, forEach, filter etc. Observables provide these whereas promises does not have any operators in their bucket.

Subscribe to an observable

In the example below we will create an observable using the interval function which produces a self-incrementing value on a regular basis.

```
import { interval } from 'rxjs';  
  
const data$ = interval(1000);
```

As long as you don't subscribe to the "observable", nothing happens because this observable is "lazy".

The common point between all "observables" is the subscribe method which allows you to subscribe to an Observable and be notified of new values, errors or the end of the "stream".

```
const subscription = data$.subscribe({  
  next: value => console.log(value),  
  error: err => console.error(err),  
  complete: () => console.log('DONE!')  
});
```


Unsubscription

The subscribe method returns an object of type Subscription.

This object is mainly used to unsubscribe from an Observable via its unsubscribe method.

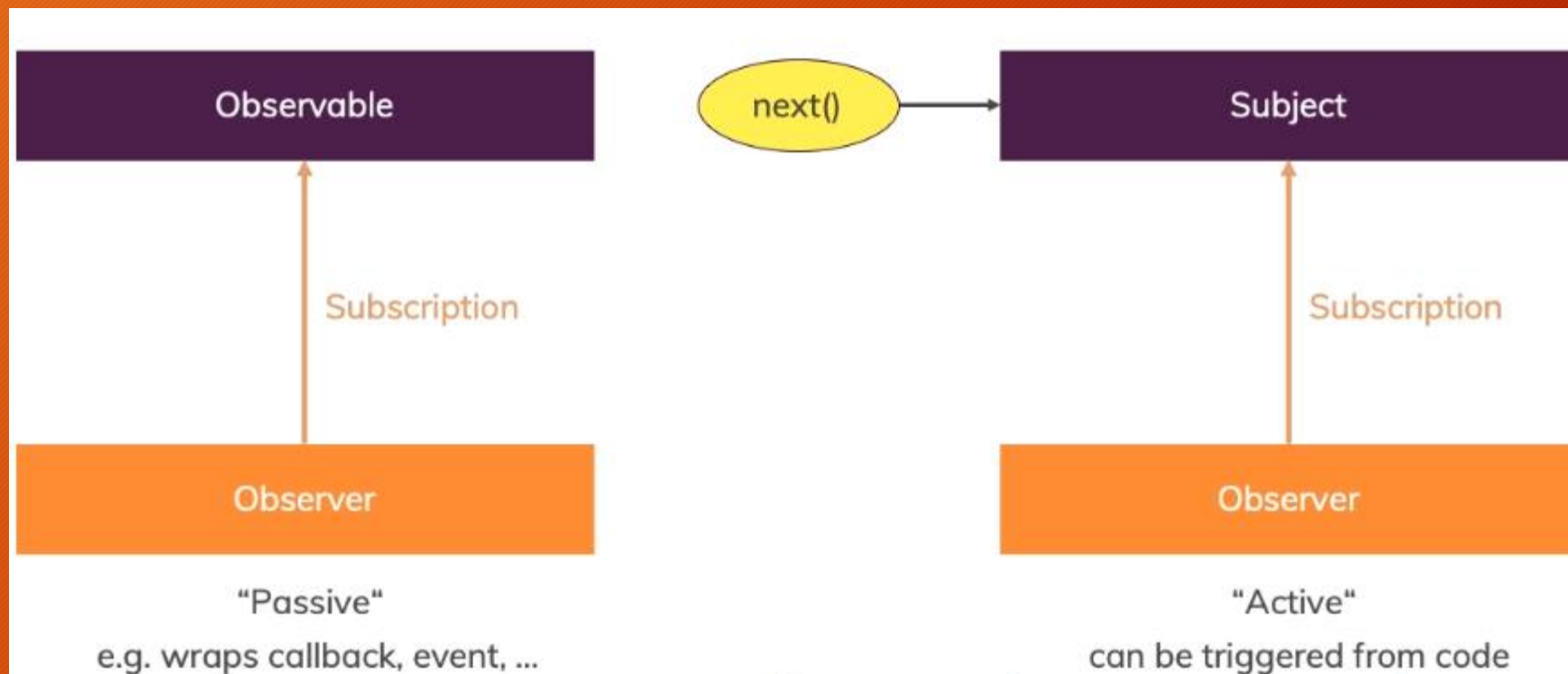
```
subscription.unsubscribe ();
```

The unsubscribe method allows you to:

- unsubscribe the "callbacks": next, error and complete;
- destroy the Observable (interrupt the processing carried out by the Observable)
- possibly free the memory because by unregistering the "callbacks" .

Subject

A Subject is a special kind of Observable



Subject

A Subject is both an observable AND an observer. We can therefore subscribe to it, but also send it values with next:

```
const subject = new Subject<number>();

subject.subscribe((number) => {
  console.log( number);
});

subject.next(1); // send value
subject.next(2); // send another
```

A subscriber will only get published values that were emitted *after* the subscription.

BehaviourSubject

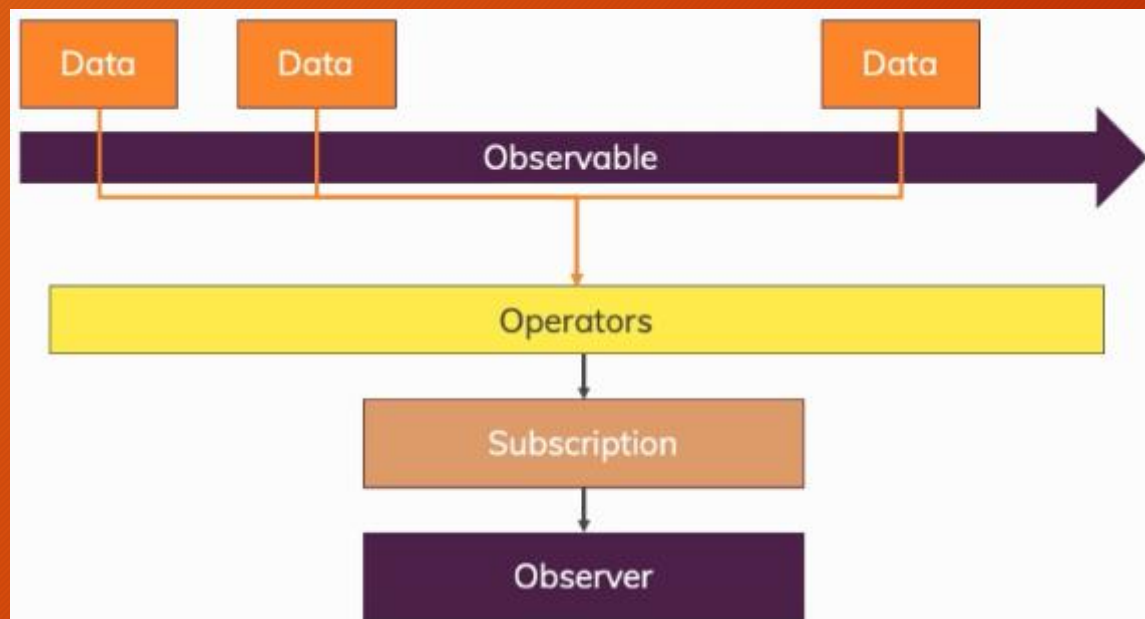
A Behaviour Subject is like a Subject, the difference is that behaviour subject also gives subscribers immediate access to the previously emitted value, even if they haven't subscribed at the point of time that value was emitted.

So we can fetch data, even if it was emitted before we subscribe to the observable.

The BehaviorSubject is initialized with an initial value.

Operators

An operator makes it possible to define an Observable from another by applying some transformations to it. Operators are functions that can be applied to an Observable via the pipe method.



map

The map operator allows you to create a new Observable from the original Observable by transforming each of its values.

```
import { from } from 'rxjs';  
import { map } from 'rxjs/operators';  
  
//emit (1,2,3,4,5)  
const source = from([1, 2, 3, 4, 5]);  
//add 10 to each value  
const example = source.pipe(map(val => val + 10));  
//output: 11,12,13,14,15  
const subscribe = example.subscribe(val => console.log(val));
```

ExhaustMap

ExhaustMap, as well as other ****Map** operators, will substitute value on the source stream with a stream of values, returned by inner function.

It waits for the first observable to complete, and replace it with the inner observable returned instead the function of exhaustMap.

Example:

We have a login page with a login button, where we *map* each click to an login ajax request.

If the user clicks more than once on the login button, it will cause multiple calls to the server.

So we can use exhaustMap to temporarily “disable” the mapping while the first http request is still on the go — this makes sure you never call the server while the current request is running.

filter

The filter operator allows you to keep only the elements for which the predicate function returns true.

```
import { from } from 'rxjs';
import { filter } from 'rxjs/operators';

//emit (1,2,3,4,5)
const source = from([1, 2, 3, 4, 5]);
//filter out non-even numbers
const example = source.pipe(filter(num => num % 2 === 0));
//output: "Even number: 2", "Even number: 4"
const subscribe = example.subscribe(val => console.log(`Even number: ${val}`));
```


Take

Emits only the first specified number of values emitted by the source Observable.

```
import { of } from 'rxjs';
import { take } from 'rxjs/operators';

//emit 1,2,3,4,5
const source = of(1, 2, 3, 4, 5);
//take the first emitted value then complete
const example = source.pipe(take(2));
//output: 1 2
const subscribe = example.subscribe(val => console.log(val));
```

Forms: Template driven



Forms

Two Approaches

Template-Driven

Angular infers the Form Object from the DOM

Reactive

Form is created programmatically and synchronized with the DOM

TD vs Reactive

Template Driven Forms Features	Reactive Forms Features
Easy to use	More flexible, but needs a lot of practice
Suitable for simple scenarios	Handles any complex scenarios
Two way data binding(using [(NgModel)] syntax)	No data binding is done
Minimal component code	More component code and less HTML markup
Automatic track of the form and its data(handled by Angular)	Adding elements dynamically

TD: Registering the Controls

First of all the module FormsModule must be imported in the app.module

To register a control we have to add NgModel and a name for this control

```
<form>  
  <input type="text" id="username" class="form-control" ngModel name="username">  
</form >
```

TD: Submitting and Using the Form

```
<form (ngSubmit)="onSubmit(f)" #f="ngForm">  
...  
</form >
```

```
import { NgForm } from '@angular/forms';  
...  
  
onSubmit(form:NgForm){  
  console.log(form);  
}
```

TD: Adding Validation to check User Input

```
<input type="text"  
id="username"  
class="form-control"  
ngModel  
name="username"  
required minlength="4">
```


TD: Using form state

```
<button type="submit" [disabled]="!f.valid">Submit</button>
```

```
input.ng-invalid, select.ng-invalid {  
  border: 1px solid red;  
}
```

TD: Outputting Validation Error Messages

```
<input type="email" id="email"  
  class="form-control" ngModel name="email"  
  required email #email="ngModel">
```

```
<span class="help-block" *ngIf="!email.valid && email.touched">  
  Please enter a valid email  
</span>
```

TD: Grouping Form Controls

```
<div id="user-data" ngModelGroup="userData" #userData="ngModelGroup">
  ... control s here
</div>
```

```
<p *ngIf="!userData.valid && userData.touched">user Data is invalid</p>
```


TD: Handling Radio Buttons

```
<div class="radio" *ngFor="let g of genders">  
  <label>  
    <input type="radio" name="gender" ngModel [value]="g">{{g}}  
  </label>  
</div>
```

```
genders=['male','female'];
```

TD: Using Form Data

```
user={
  username:'',
  email:'',
  secretQuestion:'',
  answer:'',
  gender:''
};

...
onSubmit(myForm:NgForm){
  this.user.username=myForm.value.userData.username;
  this.user.email=myForm.value.userData.email;
  this.user.gender=myForm.value.gender;
  this.user.secretQuestion=myForm.value.secret;
  this.user.answer=myForm.value.questionAnswer;
}
```

TD: Reset a form

```
this.myForm.reset();
```

Fourth Day

Forms: Reactive



Making Http Requests



Authentication

Forms: Reactive



Reactive: Adding a basic form control

There are three steps to using form controls.

1. Register the reactive forms module(`ReactiveFormsModule`) in your app. This module declares the reactive-form directives that you need to use reactive forms.
2. Generate a new `FormControl` instance and save it in the component.
3. Register the `FormControl` in the template.

Reactive: Creation of the form

```
signupFrom: FormGroup;

ngOnInit(){
  this.signupFrom= new FormGroup({
    'username': new FormControl(null),
    'email': new FormControl(null),
    'gender': new FormControl('male')
  });
}
```

Reactive: Syncing HTML and Form

```
<form [formGroup]="signupForm">
  <div class="form-group">
    <label for="username">Username</label>
    <input type="text" id="username" class="form-control"
      formControlName="username"
    >
  </div>
</form>
```


Reactive: submit the form

The FormGroup directive listens for the submit event emitted by the form element and emits an ngSubmit event that you can bind to a callback function.

```
<form [formGroup]="signupForm" (ngSubmit)="onSubmit()">  
  ...  
  <button type="submit">Submit</button>  
</form>
```

```
onSubmit(){  
  console.log(this.signupForm.value);  
}
```

Reactive: Validation

```
import { Validators } from '@angular/forms';  
  
...  
  
this.signupForm = new FormGroup({  
  'username': new FormControl(null, Validators.required),  
  'email': new FormControl(null, [Validators.required, Validators.email]),  
  'gender': new FormControl('male')  
});
```

Reactive: Getting Access to Controls

```
<span
  *ngIf="!signupFrom.get('username').valid && signupFrom.get('username').touched"
  class="help-block">
  Please enter a valid username
</span>
```

```
input.ng-invalid.ng-touched {
  border: 1px solid red;
}
```

Reactive: Creating nested form groups(1/2)

```
this.signupForm= new FormGroup({  
  'gender': new FormControl('male'),  
  'userData': new FormGroup({  
    'username': new FormControl(null, Validators.required),  
    'email': new FormControl(null, [Validators.required,Validators.email])  
  })  
});
```


Reactive: Creating nested form groups(2/2)

```
<form [formGroup]="signupForm">
  <div class="form-group">
    <label for="username">Username</label>
    <input type="text" id="username" formControlName="username" class="form-control">
  </div>
  ...
  <div formGroupName="userData">
    <div class="form-group">
      <label for="username">Username</label>
      <input type="text" id="username" formControlName="username" class="form-control">
    </div>
    <div class="form-group">
      <label for="email">email</label>
      <input type="text" id="email" formControlName="email" class="form-control">
    </div>
  </div>
</form>
```

Reactive: Creating dynamic forms

In order to declare controls dynamically we use `FormArray`

You can initialize a form array with any number of controls, from zero to many, by defining them in an array.

```
this.signupForm= new FormGroup({  
  ...,  
  'hobbies':new FormArray([])  
});
```

Reactive: Access the FormArray control

We can create a get method that return the controls of the array

```
getControls() {  
  return (<FormArray>this.signupForm.get('hobbies')).controls;  
}
```

Define a method to dynamically insert an alias control into the alias's form array. The FormArray.push() method inserts the control as a new item in the array.

```
onAddHobby(){  
  const control=new FormControl(null,Validators.required);  
  (<FormArray>this.signupForm.get('hobbies')).push(control);  
}
```

Reactive: Display the form array in the template

```
<form [formGroup]="signupForm">
...
<div formArrayName="hobbies">
  <h4>Your hobbies</h4>

  <button class="btn btn-default" type="button" (click)="onAddHobby()">Add Hobby</button>

  <div class="form-group" *ngFor="let hobby of getControls(); let i=index">
    <input type="text" class="form-control" [formControlName]="i">
  </div>
</div>
</form>
```


Updating parts of the data model

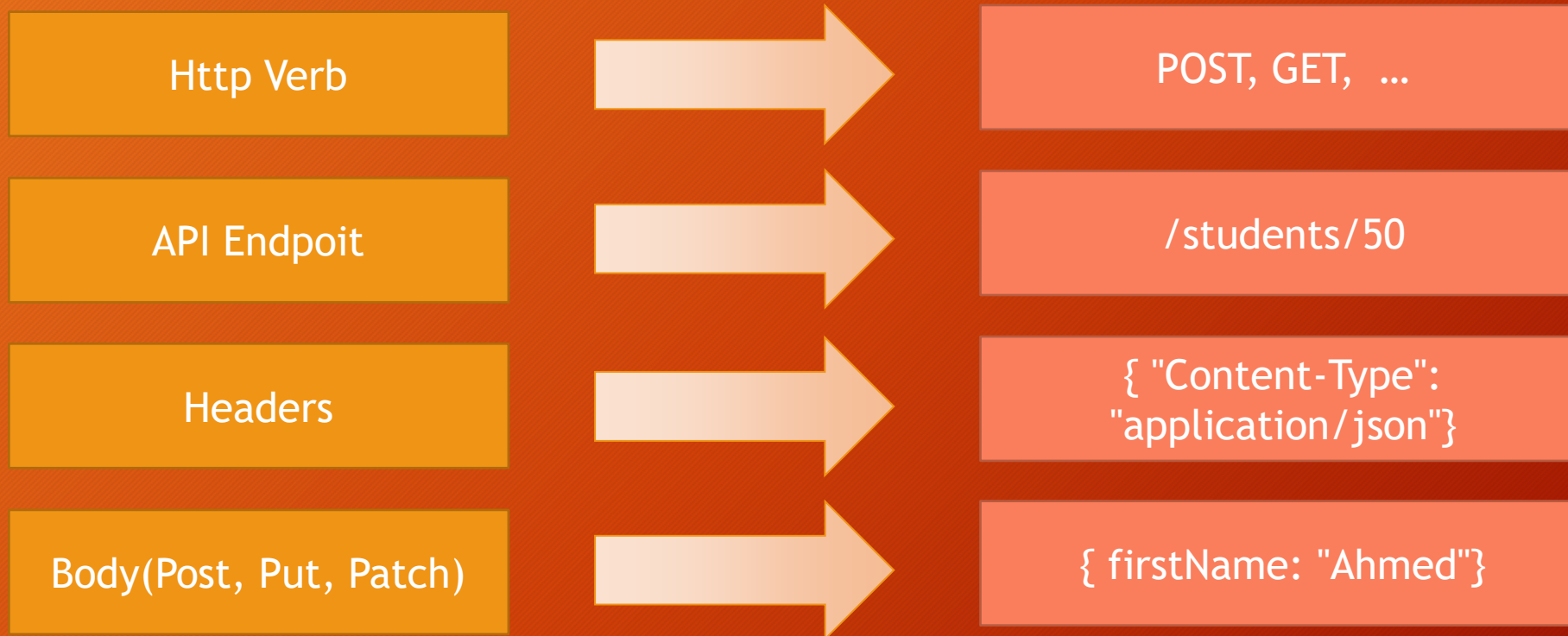
There are two ways to update the model value:

1. Use the `setValue()` method to set a new value for an individual control. The `setValue()` method strictly adheres to the structure of the form group and replaces the entire value for the control.
2. Use the `patchValue()` method to replace any properties defined in the object that have changed in the form model.

Http Requests



Anatomy of HTTP Requests



Use of HttpClient

must import the module HttpClientModule in app.module

```
import { HttpClientModule } from '@angular/common/http'
```

HttpClient is an Angular service; we can therefore recover it with the Dependency Injection.

```
constructor(private httpClient: HttpClient) {  
}
```


Sending a POST Request

```
export class StudentService {  
  constructor(private http: HttpClient) {}  
  
  addStudent (student: Student): Observable<Student> {  
    return this.http.post<Student>(this.studentUrl, student);  
  }  
}
```

```
this.studentService.addStudent(newStudent)  
  .subscribe(  
    ...  
  );
```

Sending a PUT Request

```
export class StudentService {  
  constructor(private http: HttpClient) {}  
  
  editStudent(student: Student): Observable<Student> {  
    return this.http.put<Student>(`${this.studentUrl}${student.id}`, student);  
  }  
}
```

```
this.studentService.editStudent(student)  
  .subscribe(  
    ...  
  );
```

Getting Data

```
export class StudentService {  
  constructor(private http: HttpClient) {}  
  
  getStudents(): Observable<Student> {  
    return this.http.get<Student>(this.studentUrl);  
  }  
}
```

```
private fetchStudents() {  
  this.studentService.getStudents()  
    .subscribe(students => {  
    this.students= students;  
  });  
}
```

Delete request

```
deleteStudent (id: number): Observable<{}> {  
  const url = `${this.studentsUrl}/${id}`; // DELETE api/students/42  
  return this.http.delete(url);  
}
```

```
this.studentService .deleteStudent(hero.id).subscribe(...);
```


Handling Errors

```
private fetchStudents() {  
  this.studentService..getStudents()  
    .subscribe(  
    students => {  
      this.students= students;  
    } , error => {  
      this.error = error.message;  
    });  
}
```

```
<div class="alert alert-danger" *ngIf="error">  
  <h1>An error has occured</h1>  
  <p>{{error}}</p>  
</div>
```

Setting Headers

```
return this.http
  .get<{ [key: string]: Post }>(
    url,
    {
      headers: new HttpHeaders({'Custom-header' : 'hello'})
    }
  )
```

Adding Query Params

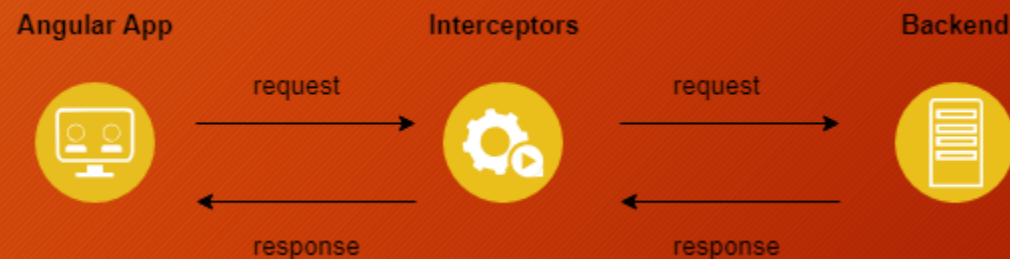
```
let searchParams = new HttpParams();
searchParams = searchParams.append('print', 'pretty');

return this.http
  .get<{ [key: string]: Post }>(
    url,
    {
      headers: new HttpHeaders({'Custom-header' : 'hello'}),
      params: searchParams
    }
  )
```

Introducing Interceptors

Interceptors are a unique type of Angular Service that we can implement.

Interceptors allow us to intercept incoming or outgoing HTTP requests using the HttpClient. By intercepting the HTTP request, we can modify or change the value of the request.



Use of Interceptors

First, create a service that implements `HttpInterceptor`

```
export class AuthInterceptorService implements HttpInterceptor {  
  intercept(req: HttpRequest<any>, next: HttpHandler) {  
    const modifiedReq = req.clone({  
      headers: req.headers.append('Auth', 'xyz')  
    });  
    return next.handle(modifiedReq);  
  }  
}
```

It has to be added to the list of all **HTTP_INTERCEPTORS**, which can be done that way in **app.module.ts**:

```
providers: [{provide : HTTP_INTERCEPTORS, useClass: AuthInterceptorService, multi : true}],
```

How we can use multiple interceptors?

```
providers: [  
  { provide: HTTP_INTERCEPTORS, useClass: MyInterceptor, multi: true },  
  { provide: HTTP_INTERCEPTORS, useClass: MySecondInterceptor, multi: true }],
```

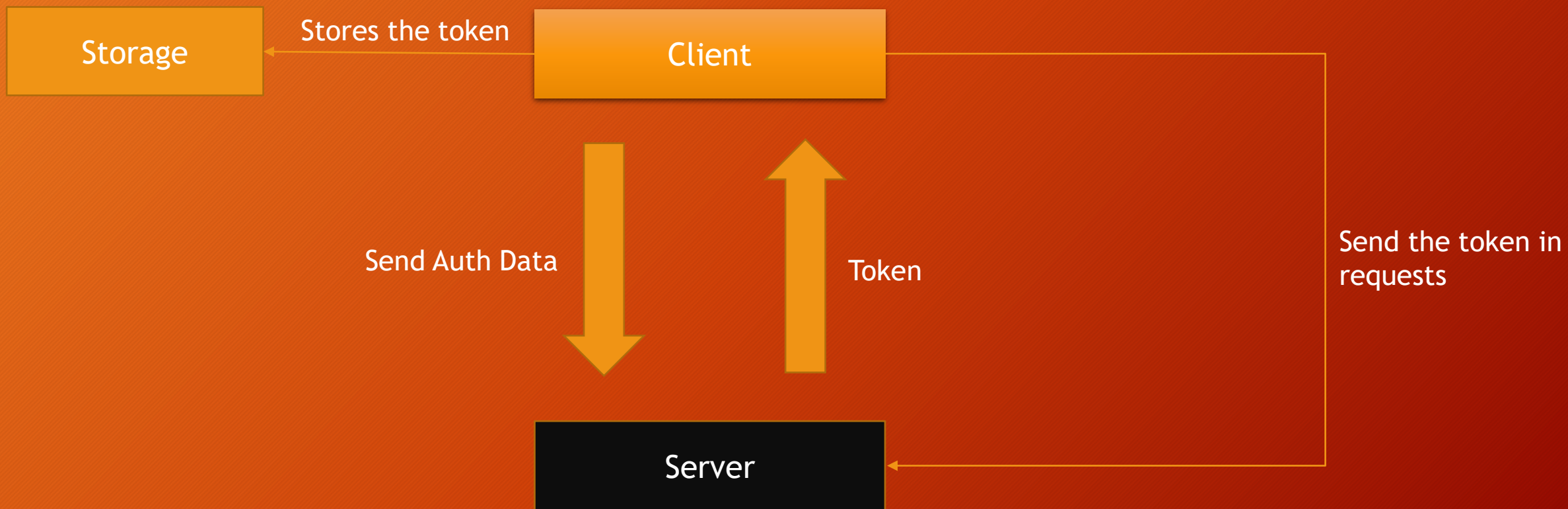
The interceptors will be **called in the order** in which they were provided.

Using `multi: true` tells Angular that the provider is a multi provider.

With multi providers, we can provide multiple values for a single token in DI.

Authentication

How authentication works



QUESTIONS



SESSION END

THANK YOU
for
your
ATTENTION

ANNEX

Source code

Source code can be found in this github Link:

<https://github.com/youma85/FleetManagement>

Testing Fundamental Concepts

Angular supports two types of testing

- Unit Tests
- End To End Tests

Unit tests are written in spec.ts files.

The e2e file test are stored in an e2e folder, and end with e2e-spec.ts.

Testing Framework in Angular

Angular supports two main framework:

- Jasmine/Karma: Used for unit tests
- Protractor: Used for E2E tests

These frameworks are configured in the following config file:

- Karma: `karma.config.js`
- Protractor: `e2e/protractor.conf.js`

Angular Unit Testing

Unit Testing is a type of testing where the focus of the test is to test a particular piece of the application.

The test can be for components, pipes, directives, services,...

Tests are written in Jasmine framework.

To test we run the command: `ng test`

This command will:

- Compile code
- Start a karma server and give it a port number
- Execute the unit tests.
- Open a new window of a browser with a report of tests.



Angular E2E Testing

E2E tests means automating the application's workflow for a functionality e2e.

To test we run the command: `ng e2e`

Jasmine(1/3)

Jasmine is an extensible framework dedicated to tests on "browser" and on NodeJS.

It includes everything you need for:

- define test suites (describe and it functions),
- implement assertions of all kinds (expect function),
- etc

Jasmine (2/3)

For example if we wanted to test this function:

```
function helloWorld() {  
  return 'Hello world!';  
}
```

We would write a Jasmine test spec like so:

```
describe('Hello world', () => { (1)  
  it('says hello', () => { (2)  
    expect(helloWorld()) (3)  
      .toEqual('Hello world!'); (4)  
  });  
});
```

1. **describe(string, function):** function defines what we call a Test Suite, a collection of individual Test Specs.
2. **The it(string, function):** function defines an individual Test Spec, this contains one or more Test Expectations.
3. **The expect(actual):** expression is what we call an Expectation. In conjunction with a Matcher it describes an expected piece of behaviour in the application.
4. **The matcher(expected):** expression is what we call a Matcher. It does a boolean comparison with the expected value passed in vs. the actual value passed to the expect function, if they are false the spec fails.

Jasmine (3/3): Setup and Teardown

beforeAll:

This function is called once, before all the specs in a test suite (describe function) are run.

afterAll:

This function is called once after all the specs in a test suite are finished.

beforeEach:

This function is called before each test specification (it function) is run.

afterEach:

This function is called after each test specification is run.

TypeScript(Types)

In typeScript, type System represents different types of datatypes which are supported by TypeScript.

BUILT-IN DATA TYPE	KEYWORD	DESCRIPTION
Number	number	It is used to represent both Integer as well as Floating-Point numbers
Boolean	boolean	Represents true and false
String	string	It is used to represent a sequence of characters
Void	void	Generally used on function return-types
Null	null	It is used when an object does not have any value
Undefined	undefined	Denotes value given to uninitialized variable
Any	any	If variable is declared with any data-type then any type of value can be assigned to that variable

TypeScript(Classess)

TypeScript is object oriented JavaScript. TypeScript supports object-oriented programming features like classes, interfaces, etc. A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object.

```
class Car {  
    //field  
    engine:string;  
  
    //constructor  
    constructor(engine:string) {  
        this.engine = engine  
    }  
  
    //function  
    disp():void {  
        console.log("Engine is : "+this.engine)  
    }  
}
```

TypeScript(Interfaces)

An interface is a syntactical contract that an entity should conform to. In other words, an interface defines the syntax that any entity must adhere to.

```
interface IPerson {  
  firstName:string,  
  lastName:string,  
  sayHi: ()=>string  
}
```

ES6+: var vs. let vs. const

The let statement allows you to declare a variable with block scope.
The const statement allows you to declare a constant.

Constants are similar to let variables, except that the value cannot be changed..

```
var x = 10;  
// Here x is 10  
{  
  let x = 2;  
  // Here x is 2  
}  
// Here x is 10
```

```
const user = {  
  firstName: 'Foo',  
  lastName: 'BAR'  
}  
user = user.firstName; // TypeError: Assignment to constant variable.
```


ES6+: Arrow Functions

Arrow functions allows a short syntax for writing function expressions.

You don't need the function keyword, the return keyword, and the curly brackets.

They must be defined **before** they are used.

```
// ES5  
var x = function(x, y) {  
  return x * y;  
}
```

```
// ES6  
const x = (x, y) => x * y;
```


ES6+: Classes

A class is a type of function, but instead of using the keyword function to initiate it, we use the keyword class, and the properties are assigned inside a constructor() method.

```
class Student {  
  constructor(name) {  
    this.name = name;  
  }  
}
```

ES6+: Template String

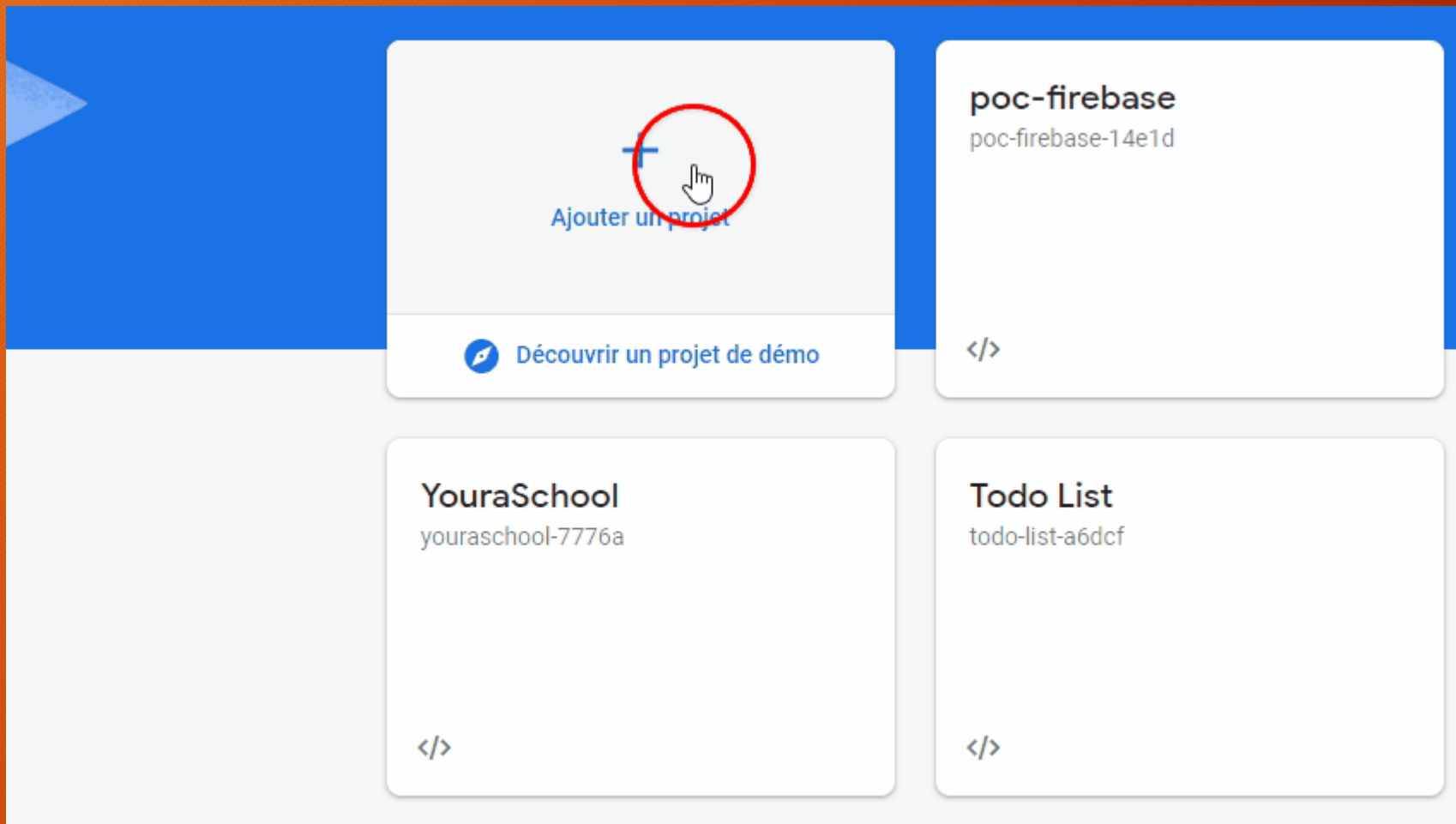
```
const appName = 'Wishtack';  
const userName = 'Foo';  
  
const greetings = `Hi ${userName},  
Welcome to ${appName}!`  
  
console.log(greetings);  
  
// Result:  
// Hi Foo,  
// Welcome to Wishtack!
```

ES6+: Default Parameter Values

ES6 allows function parameters to have default values.

```
function myFunction(x, y = 10) {  
  // y is 10 if not passed or undefined  
  return x + y;  
}  
myFunction(5); // will return 15
```

Create a firebase project



Create an application

The screenshot shows the Firebase console interface. On the left is a dark sidebar with the 'Firebase' logo and a navigation menu. The main content area has a blue header with the project name 'AngularSchool' and a 'Formule Spark' button. Below the header, the text 'Lancez-vous en ajoutant Firebase à votre application' is displayed, followed by icons for iOS, Android, and code. A red circle highlights a mouse cursor pointing at a button in the bottom right corner of the main content area.

AngularSchool Formule Spark

Lancez-vous en ajoutant
Firebase à votre
application

iOS Android </> |

Ajoutez une application pour démarrer

Stockez et synchronisez les données de votre application en quelques mi