UNIVERSITY OF CRETE
COMPUTER SCIENCE DEPARTMENT

COURSE CS-469 (OPTIONAL)
MODERN TOPICS
IN HUMAN – COMPUTER INTERACTION

Full Stack Template

Full Stack?

- Full Stack development refers to the development of an application for both front-end and back-end
- A full stack developer usually knows how to program a webpage, a server and a database
- **Popular Stacks:**
  - LAMP stack: JavaScript - Linux - Apache - MySQL - PHP
  - LEMP stack: JavaScript - Linux - Nginx - MySQL - PHP
  - **MEAN stack: JavaScript - MongoDB - Express - AngularJS - Node.js**
  - Django stack: JavaScript - Python - Django - MySQL
  - Ruby on Rails: JavaScript - Ruby - SQLite - PHP

MEAN stack

- We recommend to use our given full stack template that is based on the MEAN stack
- Mean stack stands for **M**ongoDB, **E**xpress.js, **A**ngular, and **N**ode.js
  - **Front-end:** Angular
  - **Back-end:** Node.js with express framework
  - **Database:** MongoDB

Fullstack template variants

- Verbose
- Docker-based (requires Win10 Professional)

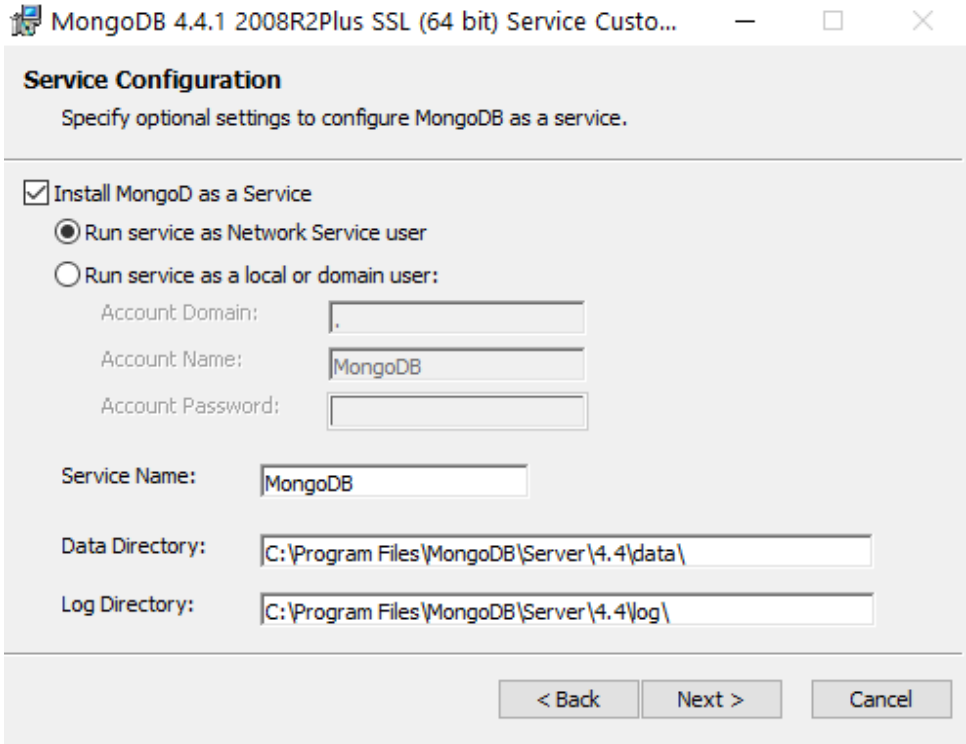
Verbose variant (1/6)

- Install LTS version of **Node.js** https://nodejs.org/en/
- Install the **Angular CLI** running the command 'npm install -g @angular/cli' on terminal (node.js should be installed first)
- Download **Redis** for Windows from https://github.com/microsoftarchive/redis/releases/tag/win-3.0.504
- Download **Minio** server from https://min.io/download#/windows

Verbose variant (2/6)

- **MongoDB**

Download MongoDB community server https://www.mongodb.com/try/download/community and install it using the default settings



Verbose variant (3/6)

- **MongoDB**

In order to run it, you have to create 2 folders that are used for storage space:

- create a folder in C:\ named **data**
- inside C:\data folder you created, create another folder named **db**

Verbose variant (4/6)

To be able to start the backend and frontend, you need to first install the dependencies of each subproject

- open a terminal window inside the **backend** folder and run **npm install**
- open a terminal window inside the **frontend** folder and run **npm install**

Verbose variant (5/6)

Running the project

- Start MongoDB
  - run **mongod.exe** which is located under the bin folder of mongoDB's installation folder (default path: C:\Program Files \MongoDB\Server\4.4\bin)
- Start Redis server
  - run **redis-server.exe** which is located under the installation folder of redis (default path: C:\Program Files \Redis)

Verbose variant (6/6)

Running the project

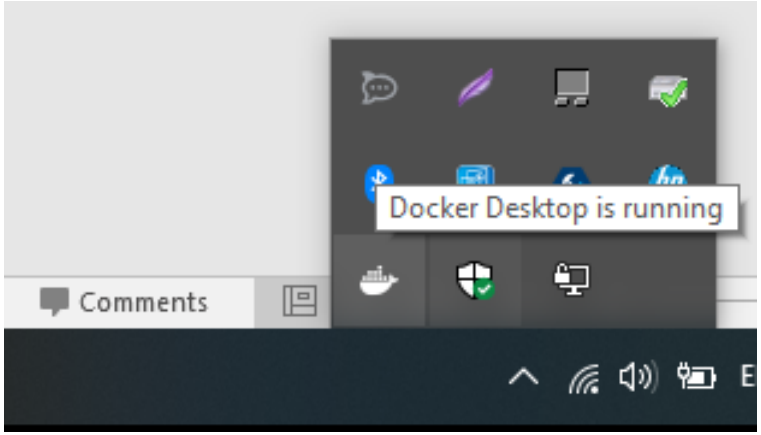
- Start MinIO server
  - open a terminal inside the folder where minio.exe is and run the command: **Minio.exe server C:\minio**
- Start backend
  - open a terminal inside the backend folder and run the command: **gulp serve**
- Start frontend
  - open a terminal inside the frontend folder and run the command: **ng serve**

Docker-based variant (1/6)

- Docker is a tool designed to make it easier to create, deploy, and run applications by using containers
- **Containers** allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package
- In addition, Docker **containers wrap up software and its dependencies into a standardized unit**. This guarantees that your application will always run the same and makes collaboration as simple as sharing a container image

Docker-based variant (2/6)

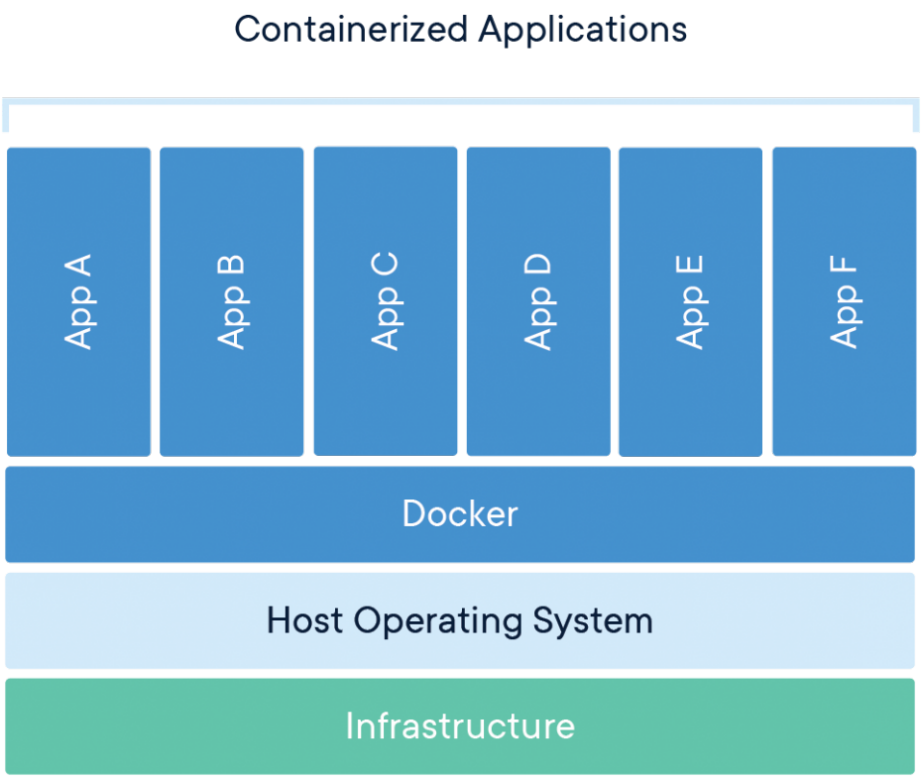
- Getting Started with Docker by creating an account (https://hub.docker.com/signup)
- Download Docker Desktop and Install it (https://hub.docker.com/)
- Start Docker Desktop and wait for it to start running



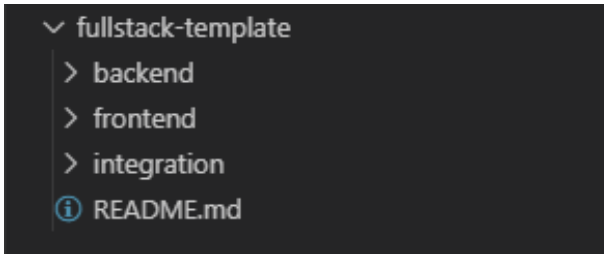
- That's it, you can now start using Docker!

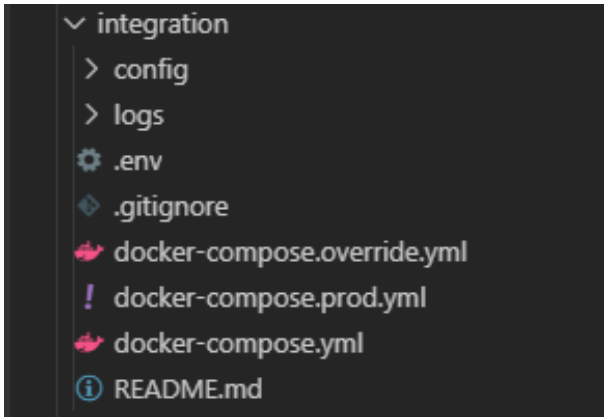
Docker-based variant (3/6)

- The template is a MEAN stack integrated with Docker
- Each part of the stack is a separate Docker container
- Everything is ready to use and its purpose is to separate the process of setting up an environment and configuring from the development process.
- A developer can focus only on writing the business logic of their application.



Docker-based variant (4/6)





- The template consists of **three** main parts
  - backend
  - frontend
  - integration
- The **integration** part is a **blackbox** to you, **<u>except</u>** that you have to change the APP_NAME in the .env file *(contains the necessary docker files to build your project)*
- It is the part where you will run a command from the terminal in order for the application to start, close, debug and build
- You can find the commands (and documentation) on how to start at README.md
- Do not change anything from this folder unless you know what you are doing

Docker-based variant (5/6)

- How to **start** the application

```
PS C:\Users\          \Downloads\fullstack-template\fullstack-template\integration> docker-compose up -d
```

- How to **close** the docker

```
PS C:\Users\          \Downloads\fullstack-template\fullstack-template\integration> docker-compose down
```

- How to **build** the application
  - (do this every time you do npm i <module> or if you change something at angular.json file. **You will need to run <u>docker-compose down</u> first, before you build**)

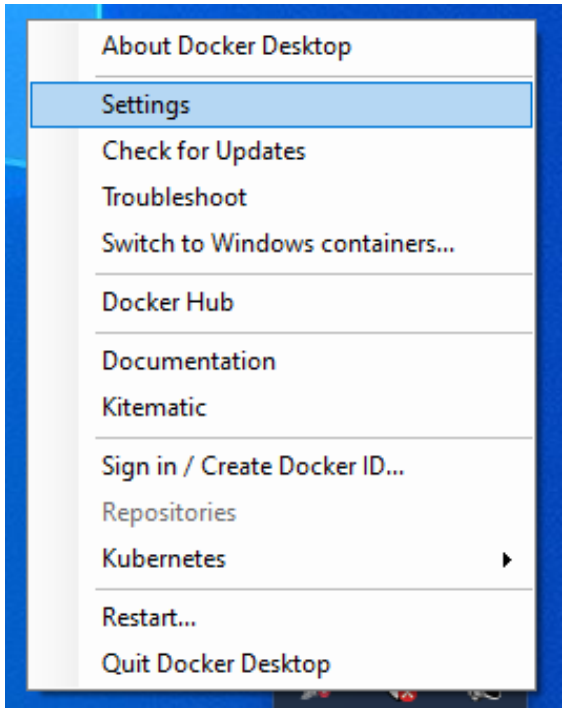
```
PS C:\Users\          Downloads\fullstack-template\fullstack-template\integration> docker-compose up --build
```

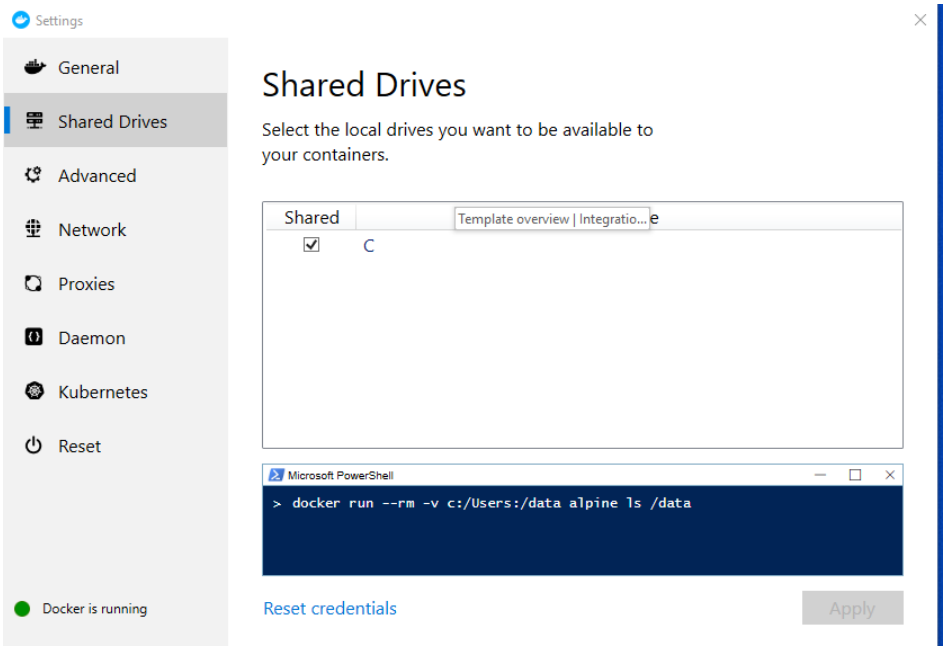
- How to **debug** your backend

```
PS C:\Users\          ownloads\fullstack-template\fullstack-template\integration> docker-compose logs -f backend
```

Docker-based variant (6/6)

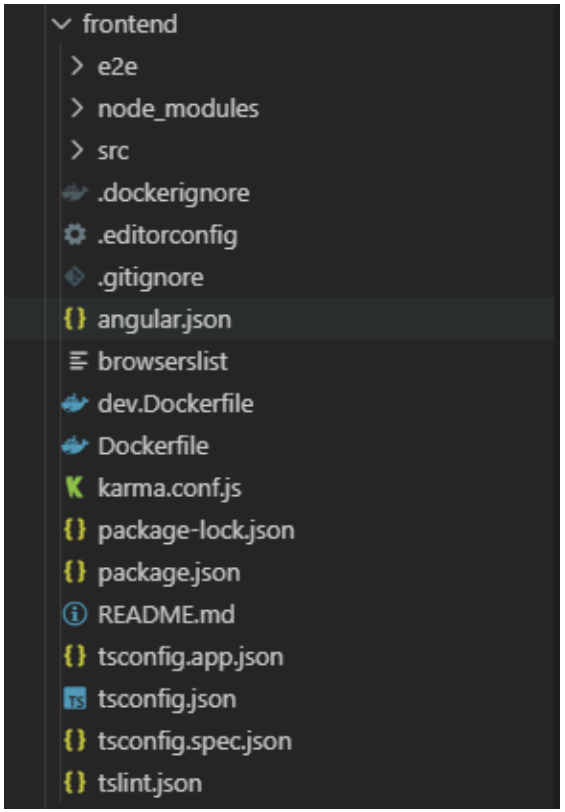
- There is possibility that you will face some errors with disk privileges when you first do docker-compose up –d. If you do so please apply the following
  - Right click on the docker to the bottom right of your windows desktop
  - Go to Settings -> shared drives and check the shared drive checkbox





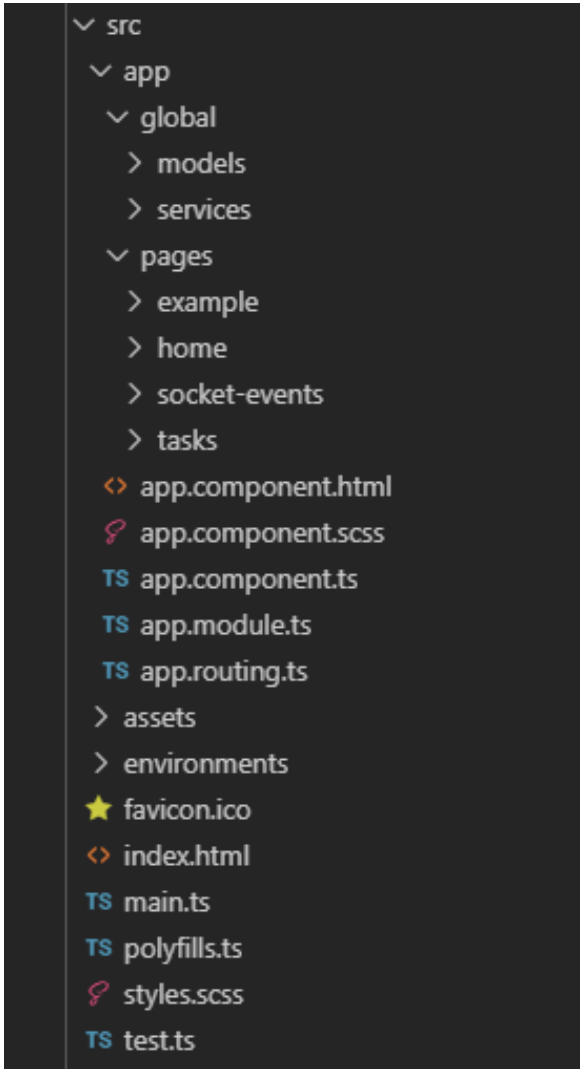
Template overview

Template overview | Frontend (1/2)



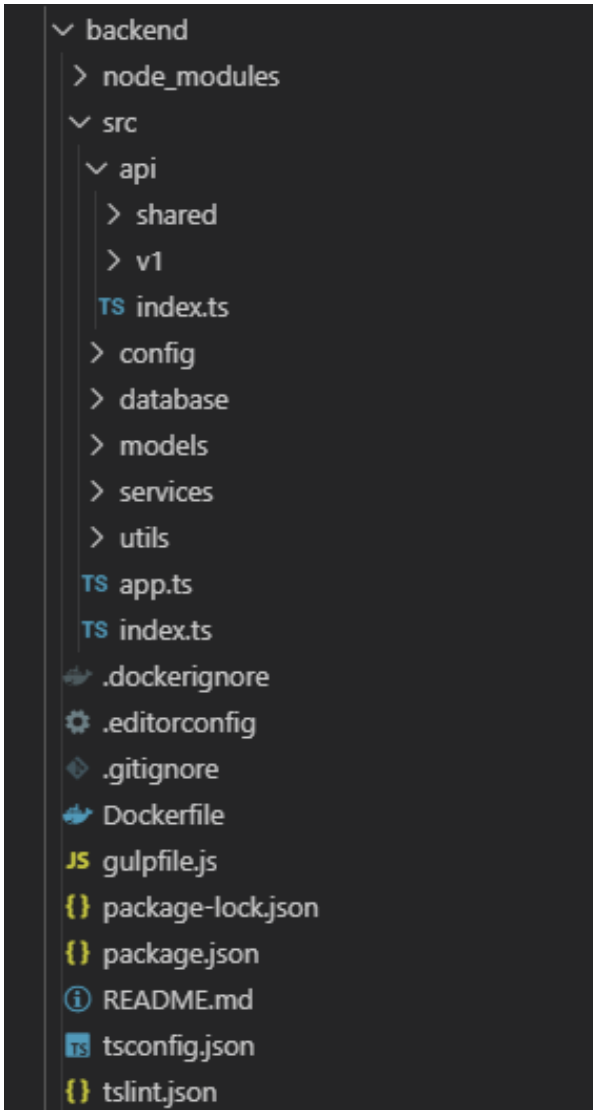
- The **frontend** is an Angular project where you will manipulate the appearance of your application
- All files **except** the **src folder** and the **angular.json file** will be a **blackbox** to you

Template overview | Frontend (2/2)



- The **global folder** contains 2 subfolders, the models and the services
  - You will **not need the models folder** for the purposes of this project
  - The **services folder** is the place where you will implement your services, as we will see later in the example
- The **pages folder** consists all of the components for the project. Each subfolder is a component where you will use it either for a page or a template. We recommend to add you new routes/pages there.
- **The rest** is known from the basic angular architecture

Template overview | Backend



- For the purposes of this project, you will not be using the backend a lot
- The idea is to build a demo application and not a fully functional one (e.g. admin functionality etc..)
- In general you will use **static json files** and not MongoDB
- The subfolder of api, **v1** folder and the **index.ts**, are the only parts of the backend that you will use. **Everything else is again a blackbox to you**

Asynchronous messaging

Sockets

- Sockets allow us to communicate (broadcast messages) from the server to the connected devices (clients) of our application
- By using our coding skills, we can adapt the information to apply only to the devices that we want to, and not to all of them
- We have the option to broadcast all the messages to our SocketService or choose a subset of events that we want to broadcast
- An example is shown in the next slides

Demo | Treat Someone Example (1/7) | backend

```
v1
  example
    TS example.controller.ts
    > files
    > socket-events
  TS index.ts
```

```
// Example routes
.use(
  '/example',
  new ExampleController().applyRoutes()
);
```

```
v1
  example
    TS example.controller.ts
    > files
    > socket-events
  TS index.ts
```

```
/**
 * Apply all routes for example
 *
 * @returns {Router}
 */
public applyRoutes(): Router {
    const router = Router();
    router.post('/treatSomeone', this.treatSomeone)

    return router;
}
```

- By adding these lines of code at backend/src/api/v1/index.ts you are creating the route http://localhost:8080/api/example which listens to the backend of your project
- By adding this function in src/api/v1/example/example.controller.ts you are creating the endpoint http://localhost:8080/api/example/treatSomeone which the **this.treatSomeone** function (implemented in the next slide)

Demo | Treat Someone Example (2/7) | backend

```
v1
  example
    example.controller.ts
    files
    socket-events
  index.ts
```

```
/**
 * Apply all routes for example
 *
 * @returns {Router}
 */
public applyRoutes(): Router {
    const router = Router();
    router.post('/treatSomeone', this.treatSomeone)

    return router;
}
```

```
/**
 * Broadcasts a received message to all connected clients
 */
public treatSomeone(req: Request, res: Response) {
    const message: string = req.body.message;
    const event: string = req.body.event;

    //Sending a broadcast message to all clients
    const socketService = DIContainer.get(SocketsService);
    socketService.broadcast(event, message);
}
```

- When we will make a post request to the route http://localhost:8080/api/example/treatSomeone from a service (we will see later how) the function treatSomeone will get the **message** and the **event** body parameters from the post request (An example can be found in the next slide)
- **The line socketService = DIContainer.. is also a blackbox**
- The socketService.broadcast, is broadcasting the event with the given message to all connected clients (devices in use)

Demo | Treat Someone Example (3/7) | frontend

```
global
  models
  services
    core
    example
      example.service.ts
    tasks
  index.ts
```

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { environment } from 'src/environments/environment';

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

  private hostURI: string;

  constructor(private http: HttpClient) {
    this.hostURI = environment.host;
  }

  public treatSomeone(foodToTreat, toUserID, fromUserID){

    console.log("hereee");

    return this.http.post(`${this.hostURI}/api/example/treatSomeone`,
    {
      message: {
        food: foodToTreat,
        userID: toUserID,
        fromUserID: fromUserID
      }
      ,event: "treating"
    })
  }
}
```

- Great! Let's see how to call the endpoint we created from our frontend
- The environment.host is the http://localhost:8080 that is hidden in the template, you don't care about that, you just use it (we will discuss further details later)
- The treatSomeone function uses the HttpClient to make a post request to the endpoint we created previously
- The first argument is the endpoint url
- The second argument is the body of the request, i.e. the **message** and the **event** 'variables' which are consumed from the backend using the r**eq.body** we used in order to use them

Demo | Treat Someone Example (4/7) | frontend

- Create in frontend/src/app/pages an example component with the command: ng g c example
- Load it in app.routing by adding this line of code:
  `{ path: 'example/:id', component: ExampleComponent}`
- You can access this component from http://localhost:4200/example/xx where xx will be a number of your choice that will be the id of the user

Demo | Treat Someone Example (5/7) | frontend

```
import { Component, OnInit } from '@angular/core';
import { ExampleService } from 'src/app/global/example/example.service';
import { SocketsService } from 'src/app/global/services';
import { ActivatedRoute, Router } from '@angular/router';

@Component({
  selector: 'ami-fullstack-example',
  templateUrl: './example.component.html',
  styleUrls: ['./example.component.scss']

})
export class ExampleComponent implements OnInit {

  public myUserID;
  public userIDtoTreat;
  public foodToTreat;
  public socketEvents: {event: string, message: any}[];

  constructor(private route:ActivatedRoute, private exampleService: ExampleService,
      private socketService: SocketsService) {
    this.socketEvents = [];
  }
```

```
  ngOnInit() {
    this.myUserID = this.route.snapshot.paramMap.get("id");;
    this.userIDtoTreat = "userToTreat";
    this.foodToTreat = "afoodToTreat";
    this.socketService.syncMessages("treating").subscribe(msg => {
      this.socketEvents.push(msg);
    })
    this.treatSomeone();
  }

  public treatSomeone(){
    this.exampleService.treatSomeone(this.foodToTreat, this.userIDtoTreat,
                                     "browser"+this.myUserID).subscribe();
  }

}
```

- Here we use the syncMessages(arg) method in example.component.ts
  - With this method we can choose the **events** that we are broadcasting with **the name "treating"** (at previous slide in our service, at the http.post request, we passed two objects inside the request body (second part after the url) the **message** and the **event**.
    <u>This is the purpose of the event object, to help us broadcast only the events that we want)</u>
  - There is also the syncAllMessages method where we are broadcasting all the events

Demo | Treat Someone Example (6/7) | frontend

In example.component.html file we add this

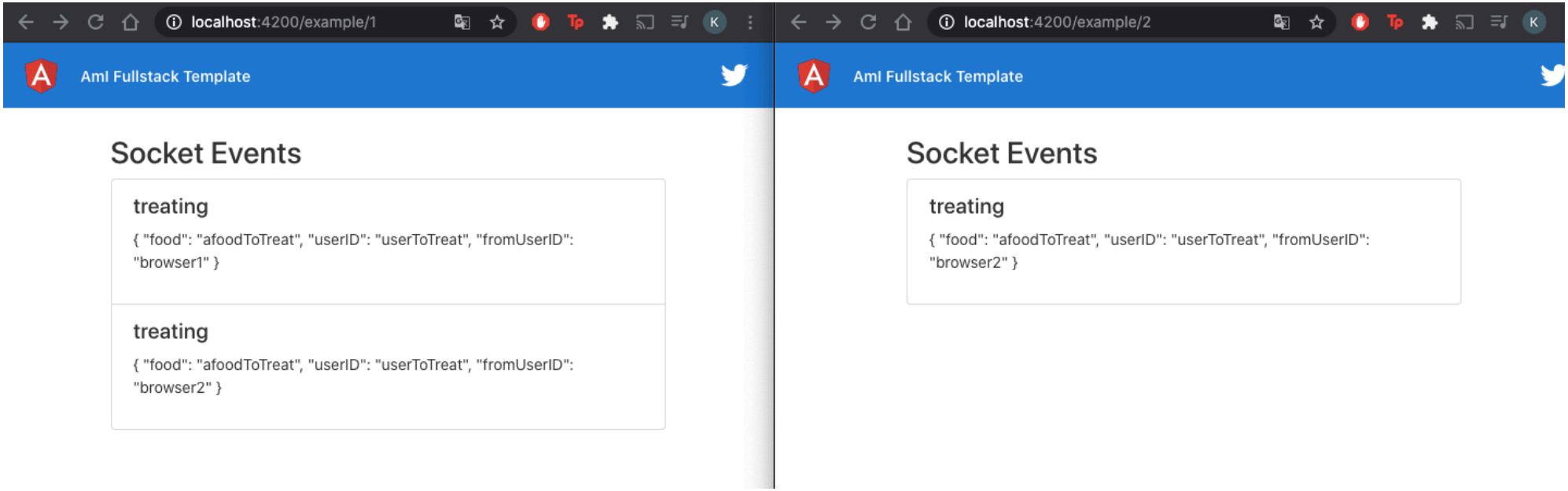
```
<div class="container mt-4">
    <h3>Socket Events</h3>

    <ul class="list-group">
      <li *ngFor="let event of socketEvents" class="list-group-item">
        <h5>{{event.event}}</h5>
        <p>{{event.message | json}}</p>
      </li>
    </ul>

  </div>
```

Demo | Treat Someone Example (7/7)

- If you open http://localhost:4200/example/1 and then http://localhost:4200/example/2 this is what you'll see



- With each new reload, ngOnInit will run and a new request will be sent so the list will grow.