# Task 1. Hello World Angular application

1) Open initial application, analyze it

2) Install necessary dependencies with npm install

3) Create file app/app.module.ts with this contents:

import { NgModule } from '@angular/core';  
import { BrowserModule } from '@angular/platform-browser';  
import { FormsModule } from '@angular/forms';  
import { AppComponent } from './app.component';  
  
@NgModule({  
 imports: [ BrowserModule, FormsModule ],  
 declarations: [ AppComponent ],  
 bootstrap: [ AppComponent ]  
})  
export class AppModule { }

4) Create app/main.ts with this contents:

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  
import { AppModule } from './app.module';  
const platform = platformBrowserDynamic();  
platform.bootstrapModule(AppModule);

5) Create file app/app.component.ts with this contents:

import { Component } from '@angular/core';  
@Component({  
 selector: 'my-app',  
 template: '<h1>My First Angular App</h1>'  
})  
export class AppComponent { }

6) Execute application with npm start, look at <http://localhost:3000>

7) Update file app/app.component.ts with this contents:

import {*Component*} from '@angular/core';  
  
@*Component*({  
 selector: 'my-app',  
 template: `  
 <h1>Hello Angular2 App</h1>  
 <br>Your name: <input type="text" [(ngModel)]="name">  
 <p></p>Hello, {{name}} from Angular! `  
})  
export class AppComponent {  
 name="John";  
}

8) Look at the automatically updated result in browser

# Task 2. Notes application with Angular

Take initial application, create folder app.

### 1. Create and use NotesComponent

1) Create component AppComponent in app/app.component.ts. Define this template:

template: `  
 <h1>Notes Angular App</h1>  
 <notes></notes>`

2) Create file app/notes.component.ts with this contents:

import {*Component*} from '@angular/core';

@*Component*({  
 selector: 'notes',  
 template: `Notes list:`

})  
export class NotesComponent { }

3) Import NotesComponent: add to the import block in app.module.ts:

import { NotesComponent } from './notes.component';

4) Define required directive by adding this to declarations of @NgModule in app.module.ts:

declarations: [ …, NotesComponent ]

5) Execute and check that «Notes list» is shown in the application

### 2. Show notes in NotesComponent

1) Define interface for Note in notes.component.ts:

interface Note {  
 text: string;  
}

2) Add this code to the template:

<ul>  
 <li \*ngFor="let note of notes ">  
 {{note.text}}   
 </li>  
</ul>

3) Define initial notes list in NotesComponent class:

export class NotesComponent {  
 notes: Note[] = [  
 {text:"Note one"},  
 {text:"Note two"}  
 ]

}

4) Execute and check that notes list is shown

### 3. Add possibility to add the note to the list

1) Add these lines to the template in notes.component.ts:

<textarea [(ngModel)]="text" ></textarea>  
<button (click)="add()">Add</button>

2) Define text and add() method in NotesComponent class:

text: string  
  
add() {  
 let note = { text: this.text }  
 this.notes.push(note);  
 this.text = "";  
}

3) Execute and look how it's working

### 4. Add possibility to remove the note from the list

1) Change the template to show notes this way:

<ul>  
 <li \*ngFor="let note of notes; let i=index">  
 {{note.text}} <button (click)="remove(i)">remove</button>  
 </li>  
</ul>

2) Define method remove:

remove(idx) {  
 this.notes.splice(idx,1);  
}

3) Check the possibility to remove notes

### 5. Retrievieng data from the server

1) Change app.module.ts:

Add import

import { HttpModule } from '@angular/http';

Add HttpModule to imports in @NgModule:

imports: [ BrowserModule, HttpModule ]

2) Create server folder and put server.js to it.

In server/server.js allow cross-origin requests (because lite-server and Node are running on different servers):

*app*.get("/notes", function(req,res) {  
 res.header("Access-Control-Allow-Origin", "\*");  
 res.header("Access-Control-Allow-Headers", "Content-Type, X-Requested-With");  
 var notes = [  
 {text: "First note"},  
 {text: "Second note"},  
 {text: "Third note"}  
 ]  
 res.send(notes);  
});

Install Express by typing in server folder:

npm install express

Otherwise you can create package.json by typing **npm init**, and then execute

npm install express --save

It will add express to package.json.

Execute server on port 8080:

node server.js

3) Import Http to notes.component.ts:

import { Http } from '@angular/http';

4) Define notesUrl:

private notesUrl = 'http://localhost:8080/notes'; *// URL to web api*

5) Define getNotes() method:



getNotes(): Promise<Note[]> {  
 return this.http.get(this.notesUrl)  
 .toPromise()  
 .then(response => response.json() as Note[]);  
}



Also add this import to use toPromise operator (it’s not added automatically):

import 'rxjs/add/operator/toPromise';

This is needed for every rxjs operator you want to use

6) Add constructor to inject http and retrieve from server:

constructor(private http: Http) {   
 this.getNotes().then(notes=>{  
 this.notes=notes  
 *console*.log(notes);  
 });  
}

7) Start server.js and see the notes loaded from the server

## Additional tasks

1) Add «Send to top» button

# Task 3. Keeping notes in session

### 1. Execute node instead of lite-server

To run node, edit package.json in root application folder by changing

"start": "tsc && concurrently \"tsc -w\" \"lite-server\" ",

to

"start": "tsc && concurrently \"tsc -w\" \"node server/server.js\" ",

This way we will start node server instead of lite-server.

Otherwise you can install nodemon which will restart node server every time you when you are changing server.js script:

npm install nodemon

And then starting script will be looking this way:

"start": "tsc && concurrently \"tsc -w\" \"nodemon server/server.js\" ",

Now we can restart server with npm start. Since now Node is responsible for both client and server side. However, page autoreloding is not working anymore, so you will need to do it manually.

Then add this line to server.js:

app.use(express.static(path.join(*\_\_dirname*, '..')));

This will make the root folder as static folder, i.e. index.html will be static resource served by Node server.

Change **notesUrl** value in notes.component.ts to 'notes' because now we are working on the same server, so we don't need the external server address.

### 2. Load notes from session

Add session middleware to server.js:

app.use(*session*({  
 secret: 'angular\_tutorial',  
 resave: true,  
 saveUninitialized: true  
}));

Also we need to install express-session to Node server. Type this in server folder:

npm install express-session --save

And add session variable to the top of server.js:

var session = require('express-session');

It will install express-session and add it to package.json.

Change reading of notes to this:

app.get("/notes", function(req,res) {  
 *console*.log("reading notes", req.session.notes);  
 if (!req.session.notes) {  
 req.session.notes = notes\_init;  
 }  
 res.send(req.session.notes);  
});

Now notes are loaded from the session or initialized by notes\_init variable.

Define notes\_init like this:

var notes\_init = [  
 {text: "First note"},  
 {text: "Second note"},  
 {text: "Third note"}  
];

### 2. Add possibility to add notes to the session

Add this route to server.js:

app.post("/notes", function(req,res) {  
 var note = req.body;  
 *console*.log("adding note", req.session.notes);  
 req.session.notes.push(note);  
 res.end();  
});

To retrieve values from HTTP body, you need body-parser middleware. To install it, use

npm install body-parser --save

in server folder. Also add these lines to the beginning of server.js:

var bodyParser = require('body-parser')

app.use(bodyParser.urlencoded({extended: true}));

app.use(bodyParser.json());

Add method which sends the added note to the server to notes.component.ts:

addNote(note:Note) {  
 this.http.post(this.notesUrl, note).toPromise()  
 .then(response => *console*.log("note sent, response", response) );  
}

And add call to addNote() to add() method of NotesComponent.

## Additional tasks

1) Implement remove note from session

2) Implement «Send to top» button in session

# Task 4. Use mongodb to store notes

## Task descrition

In this task we will start to use mongodb to work with data.

## Time

2 hours

## Detailed desctiption

### 4.1 Working with mongodb from NodeJS

1) Edit package.json

Add dependency on last mongodb package:

"mongodb": "^2.2.0"

Then execute **npm install** to install necessary modules.

2) Define variables in server.js

var Db = require('mongodb').Db;

var Server = require('mongodb').Server;

3) Open mongodb connection

var db = new Db('tutor',

new Server("localhost", 27017, {safe: true},

{auto\_reconnect: true}, {}));

db.open(function(err){

if (err) console.log(err);

else console.log("mongo db is opened!");

});

4) Add collection notes

Add these lines inside db.open callback function

db.collection('notes', function(error, notes) {

db.notes = notes;

});

5) Make notes to be loaded from the database

app.get("/notes", function(req,res) {

db.notes.find(req.query).toArray(function(err, items) {

res.send(items);

});

});

6) Make notes to be saved in database

app.post("/notes", function(req,res) {

db.notes.insert(req.body).then(function() {

res.end();

});

});

7) Extract reading notes from the constructor to separate method readNotes():

**constructor(private http: Http) {**  
 **this.readNotes();**  
 **}**  
  
**readNotes() {**  
 **this.getNotes().then(notes=>{**  
 **this.notes=notes;**  
 **});**  
 **}**

8) Reread list of notes after adding new note:

**addNote(note:Note) {**  
 **this.http.post(this.notesUrl, note).toPromise()**  
 **.then(response => {**  
 **this.readNotes();**  
 **} );**  
 **}**

9) Go to db folder and execute mongodb:

<MONGODB\_PATH>/bin/mongod --dbpath .

Now you can restart server and add some notes.

### 4.2 Using mongo console to find/update/insert notes

1) Start mongo client:

<MONGODB\_PATH>/bin/mongo



2) Change database to tutor:

use tutor



3) Find all notes:

db.notes.find()

4) Insert note:

db.notes.insert({text:"my note"})

5) Find note by query:

db.notes.find({text:"my note"})



6) Change note text:

db.notes.update({text:"my note"}, { $set:{text:"his note"} })

7) Change or insert note:

db.notes.update({text:"my note"}, { $set:{text:"her note"}})

This command will not do anything because there's no note with text "my note".

db.notes.update({text:"my note"}, { $set:{text:"her note"}},

{upsert:true})

This will find for note "my note", and if it was not found, insert the new note with the text "her note".

8) Change multiple notes

Usually update changes only one record. If we need to change all records, we have to add parameter **{multi:true}**.

Let’s add property lastUpdated to the notes.

Execute this command:

db.notes.update({}, {$set:{lastUpdated:new Date().getTime()}})

You will see that only one item is updated. To update all items, you need to add {multi:true} parameter:

db.notes.update({}, {$set:{lastUpdated:new Date().getTime()}}, {multi:true})

It will add lastUpdated to all notes.

9) Ordering items

By default items in collection are ordered as they were added. You can reorder it by using orderBy. For example to order by text (alphabetically) use this command:

db.notes.find().sort( { text: 1 } )

This command will order by text in decreasing order:

db.notes.find().sort( { text: -1 } )

You also can use several fields for ordering:

db.notes.find().sort( { lastUpdate:1, text: 1 } )

It will order by lastUpdate, and then by text.

9.1) Removing item from collection

db.notes.remove({text:"my text"})

Remove collection with indexes

db.notes.drop();

10) Remove field from the items

To remove field lastUpdated from notes, execute this command:

db.notes.update({}, {$unset: {lastUpdated:""}}, {multi:true})

### 4.3 Delete notes from database by id

1) Add ObjectID variable to server.js:

var ObjectID = require('mongodb').ObjectID;

2) Implement delete in server.js

app.delete("/notes", function(req,res) {

var id = new ObjectID(req.query.id);

db.notes.remove({\_id: id}, function(err){

if (err) {

console.log(err);

res.send("Failed");

} else {

res.send("Success");

}

})

});

3) Add remove button to NotesComponent template

<ul>  
 <li \*ngFor="let note of notes">  
 {{note.text}} <button (click)="remove(note.\_id)">remove</button>  
 </li>  
</ul>

4) Define remove function in NotesComponent

remove(id:string) {  
 this.http.delete(this.notesUrl, { params: {id} })  
 .toPromise()  
 .then(response => {  
 *console*.log(  
 `note with id ${id} removed, response`, response);  
 this.readNotes();  
 });  
}

Now you can restart server and refresh page, and delete notes by clicking remove button.

## Additional tasks

1) Save date and time of adding note in date field. Print it in hh:mm dd.mm.yyyy format (use pipe to format the date):

{{date | date: 'HH:mm dd.MM.yyyy'}}

2) Add possibility to reorder notes (add "send to top" button)

For this:



* add order property to the note
* find the note with minimal order number by using query

db.notes.find().sort( { order: 1 } ).limit(1)

* read the minimal order value and decrement it



* update the note by setting order to decremented value
* use ordering by order when loading the notes from mongodb



3) Add possibility to edit notes

Task 7

# Task 5. Sections

## Task description

In this task you have to create sections for notes, with possibility for reordering. Also we will use Bootstrap to create a nice looking design of the application.

## Time

2 hours

## Detailed description

### 5.1 Add bootstrap css

We will be using Twitter bootstrap for design of our Notes application.

1) Create **css** folder in project root, copy **bootstrap.min.css** from **bootstrap-init** to **css** folder

2) Copy folder **fonts** from **bootstrap-init** to the project **root** folder

3) Add link to bootstrap to **index.html**:

<link rel="stylesheet" href="css/bootstrap.min.css">

4) Use **app.component.html** as a template for **AppComponent** (put template to app folder, add templateUrl to @Component, it should be "app/app.component.html")

5) Use **notes.component.html** as a template for **NotesComponent**

6) Add empty component SectionsComponent. Set selector to ‘sections’, leave template empty.

Refresh page to see how it is looking with the bootstrap design.

### 5.2 Show sections

1) Add the sections collection to server.js:

db.collection('sections', function(error, sections) {

db.sections = sections;

});

2) Add possibility to get sections:

app.get("/sections", function(req,res) {

db.sections.find(req.query).toArray(function(err, items) {

res.send(items);

});

});

3) Update component SectionsComponent which should use sections.component.html as a template (put template to app folder, add templateUrl to component).

4) Define Section interface:

interface Section {  
 \_id: string;  
 title: string;  
}

5) Define sections url:

private sectionsUrl = 'sections'; *// URL to web api*

6) Define list of sections:

sections: Section[];

7) Define readSections() and getSections() method in SectionsComponent:

readSections() {  
 this.getSections().subscribe(sections=>{  
 this.sections=sections;  
 });  
}

getSections(): Observable<Section[]> {  
 return this.http.get(this.sectionsUrl)  
 .map(response => response.json() as Section[]);  
}

Also add map operator to import:

import 'rxjs/add/operator/map';

Or you can import all Observable operators at once by using

import 'rxjs/Rx';

8) Add constructor which will show list of sections on startup:

constructor(private http: Http) {  
 this.readSections();  
}

9) Run **mongo, switch to tutor database (type use tutor)** and add some sections:

db.sections.insert({title:"Work"});

db.sections.insert({title:"Vacations"});

db.sections.insert({title:"Children"});

10) Also add test note to Work section (will be used later for testing):

db.notes.insert({section:"Work", text:"test work note"});

Now refresh the page and see the new design with the sections on the right.

### 5.3 Allow to select a section and show notes for the section

1) Update sections.component.html to react on section click:

<li \*ngFor="let section of sections"

[ngClass]="{active:section.title==activeSection }"  
 (click)="showSection(section)" class="list-group-item">  
 {{section.title}}  
</li>

2) Add activeSection field of type string. It will keep the currently selected section.

3) Add showSection() to SectionsComponent:

showSection(section:Section) {  
 this.activeSection = section.title;   
}

4) In readSections() set default activeSection if it was not set:

if (this.activeSection == null && this.sections.length>0) {  
 this.showSection(this.sections[0]);  
}

Add this inside subscribe callback.

5) Reload page and try to switch active section.

### 5.4 Show notes for the selected section

1) Add field section:string to NotesComponent. It will be the title of active section. Set its initial value to "Work".

2) Update add() to keep the section with the new note:

let note = { text: this.text, section: this.section };

3) Update getNotes() to retrieve notes only for the selected section:

params.set('section', this.section);  
return this.http.get(this.notesUrl, {params:{section:this.section}})  
 .map(response => response.json() as Note[]);

Also update getNotes signature from Promise to Observable,

and update readNotes to use subscribe instead of then.

4) Reload page. Now it should add notes to section Work and show notes only from this section(since it is hardcoded).



### 5.5 Implement communication of SectionsComponent with NotesComponent

To reflect change of the active section, we should update section in NotesComponent and reread sections on every change of active section in SectionsComponent. We will use AppComponent for communication.

To do this:

1) Pass section title to NotesComponent in app.component.html:

<notes [section]="'Work'"></notes>

2) Add @Input() decorator to section field in NotesComponent

@Input() section: string;

3) Since @Input property will be not available in component constructor, move reading notes from constructor to ngOnInit() – here we will already know the section passed from AppComponent. Also add OnInit interface:

class NotesComponent implements OnInit

4) Now reload page. It should work the same way, showing notes for "Work" section.

5) To pass section name from SectionsComponent, introduce variable section in AppComponent – it will behave as mediator:

section: string;

Also define this method in AppComponent:

setSection(section:string) {  
 this.section = section;

}

6) Update app.component.html by adding event handler to <sections>:

<sections (sectionChanged)="setSection(*$event*)"></sections>

Now on every change of section setSection() method will be fired with title of the section as a parameter.

7) Add @Output field to SectionsComponent to emit event:

@Output() sectionChanged: EventEmitter<string> =

new EventEmitter<string>();

Note that **EventEmitter** should be imported from **"@angular/core":**

**import {EventEmitter} from "@angular/core";**

8) Update showSection() so that it emits the event on every change of the section:

showSection(section:Section) {  
 this.activeSection = section.title;  
 this.sectionChanged.emit(this.activeSection);  
}

9) In app.component.html pass remove hardcoded "Work" section and pass section field instead:

<notes [section]="section"></notes>

10) Also change method name ngOnInit() to ngOnChanges() in NotesComponent and add **implements OnChanges**. It will allow execute this method and update notes list not only on component initialization, but also on every change of the section.

11) Now update all your notes to put it to some section, because notes with no sections are not supported anymore:

Run mongo and execute

db.notes.update({}, {$set: {section:"Old notes" }}, {multi:true});

Also add section "Old notes" to sections collection.

12) Reload page and check that now we can select section, see the notes of the selected section and add the note to the active section.

### 5.6 Add section

Now we should allow user to add a new section from the UI.

1) In sections.component.html update input box to enter new section name:

<input type="text" class="form-control" placeholder="New section name" #newSection>

2) Update button Add to process click:

<button class="btn btn-default" type="button" (click)="addSection(*newSection*)">Add</button>

3) Now that’s possible the section has no \_id (just added section has no id), so we should update Section interface:

export interface Section {  
 \_id?: string;  
 title: string;  
}

\_id is now optional. Also add **export** because we will need to use it from outside.

4) Add method addSection() to SectionsComponent:

addSection(newSection: HTMLInputElement) {

let title = newSection.value;  
 if (!title) return;  
  
 *// check for duplicates*  
if (this.sections.map(s=>s.title).find(t=>t===title)) return;  
  
 const section: Section = { title };  
 this.sections.unshift(section);  
 this.showSection(section);

*// write sections to server and clear add section input box*

this.writeSections().subscribe(res=>newSection.value = "");  
}

Method unshift will add section to the beginning of sections list.

Also we will switch to the newly added section.

5) Add writeSections() method which will send the list of sections to the server:

writeSections() {  
 return this.http.post(this.sectionsReplaceUrl, this.sections);   
}

Also define sectionsReplaceUrl in component:

sectionsReplaceUrl = "/sections/replace";

6) Add possibility to replace old section list by new list in server.js. It will be used for adding and reordering of the sections.

app.post("/sections/replace", function(req,resp) {

// do not clear the list

if (req.body.length==0) {

resp.end();

}

db.sections.remove({}, function(err, res) {

if (err) console.log(err);

db.sections.insert(req.body, function(err, res) {

if (err) console.log("err after insert",err);

resp.end();

});

});

});

Now you can refresh page and check the adding of the new section.

### 5.7 Add sections reordering [optional]

We will be using library Dragula which implements the possibility to reorder list by drag&drop. Here you can find the full description of the library: <https://github.com/valor-software/ng2-dragula>

To add this, do the following:

1) Copy **dragula.css** from **bootstrap-init folder** to **css** folder.

Add link to dragula.css in index.html:

<**link rel="stylesheet" href="css/dragula.css"**>

2) Add dragon libraries to package.json:

"dragula": "^3.7.2",  
"ng2-dragula": "^1.3.1"

Then run npm install.

3) Configure systemjs.config.js:

Add to map {…} section these lines:

**'ng2-dragula'**: **'npm:ng2-dragula'**,  
**'dragula'**: **'npm:dragula/dist/dragula.js'**,

Also add this configuration to packages {…} section:

**'ng2-dragula'**: {  
 **main**: **'./index.js'**  
},

4) Edit app.module.ts: add DragulaModule to imports:

imports: [ …, DragulaModule ],

Also add this import to imports in app.module.ts:

import {DragulaModule} from "ng2-dragula";

5) Update sections list by adding dragula directive in sections.component.html:

<ul class="list-group" [dragula]='"sections"'>

6) Update constructor in SectionsComponent to subscribe to drop event:

constructor(private http: Http, private dragulaService: DragulaService) {  
 this.readSections();  
 dragulaService.drop.subscribe(this.onDrop.bind(this));  
}

Add this import to SectionsComponent (**NB!** WebStorm may suggest the wrong import path):

import {DragulaService} from "ng2-dragula";

7) Implement onDrop method in SectionsComp onent:

onDrop(value) {  
 let [bag, elementMoved, targetContainer, srcContainer] = value;  
 if (targetContainer.children) {  
 let arr = *Array*.from(targetContainer.children);  
 this.sections = arr.map((li:HTMLLIElement)=>

{ return {title: li.textContent.trim() } });  
 this.writeSections().subscribe();  
 }  
}

Now you can refresh the page and check how section drag&drop is working.

### 5.8 Add sections filter [optional]

We will add the input field to filter out sections which title starts with the entered value. For this we will create pipe.

1) Create pipe SectionFilterPipe with this code:

@Pipe({  
 name: 'sectionFilter'  
})  
export class SectionFilterPipe implements PipeTransform {  
 transform(sections: Section[], v: string):Section[] {  
 if (!sections) return [];  
 return sections.filter(  
 s => s.title.toLowerCase().startsWith(v. toLowerCase()));  
 }   
}

2) Add this pipe to AppModule declarations

3) Update sections.component.html: add input box for the section filter:

<input type="text" class="form-control" placeholder="Section filter" #filter>

just before <ul> block. Modify ngFor in <li>:

<li \*ngFor="let section of sections | sectionFilter: *filter*.value"

4) Reload page and check the result. When you will enter something in Section filter and move focus outside input box, sections list will be filtered.

5) To filter sections on every keypress, add keypress event binding to input box:

<input … #filter (keyup)="0">

This will fire keyup event and make Angular to update template.

6) However, if we will add new section that satisfies filter, we will not see it until we change filter value. That happens because pure pipe rerun only if its parameters change. In this case sections is checked as a reference, and reference doesn’t change on adding new section. To make pipe to be executing on any update, make it impure: set **pure:false** in **@Pipe** decorator of pipe. It will execute pipe on any change.

## Additional tasks

1) Implement filter for the notes. It should look for any substring in notes.

2) Implement tags for notes, with possibility to filter with tags and storing tags in database

# Task 6. Routes



## Task description



In this exercise we will add the support of the routing in application.

## Time

2 hours

## Detailed description

### 6.1 Adding routes

1) Update app.module.ts by adding routes:

const appRoutes: Routes = [  
 { path: '', component: NotesEditorComponent },  
 { path: '\*\*', component: PageNotFoundComponent }  
];

2) Add RouterModule to imports with definition of appRoutes:

imports: [ BrowserModule, RouterModule.*forRoot*(appRoutes), …],

3) Define NotesEditorComponent by moving bottom <div> container with notes and sections from app.component.html to NotesEditorComponent template and replacing it by

<router-outlet></router-outlet>

Also move logic from AppComponent to NotesEditorComponent (section field and setSection method).

4) Define component PageNotFoundComponent with template having text “page not found”.

5) Add base href to index.html body:

<base href="/">

Now you can check if the application works correctly.

6) If you want to check PageNotFoundComponent, add link to the end of **app.component.html** (remove it after checking):

<a [routerLink]="'wrongpage'">Wrong Page</a>

### 6.2 View section

This feature will allow to preview only one section and send it as URL to someone, or add to bookmarks.

1) Create component ViewSectionComponent

2) Add viewSection.component.html should have this code:

<div class="col-md-8 col-md-offset-2">  
 <div class="panel panel-primary ">  
 <div class="panel-heading">  
 <h3 class="panel-title">{{section}}</h3>  
 </div>  
 <ul class="list-group">  
 <li \*ngFor="let note of notes"  
 class="list-group-item">{{note.text}}</li>  
 </ul>  
 </div>  
</div>

3) Create constructor of ViewSectionComponent this way:

constructor(private route: ActivatedRoute) {}

4) Define section property in ViewSectionComponent.

In ngOnInit() retrieve the section:

ngOnInit() {  
 this.section = this.route.snapshot.params["name"];

}

Route snapshot returns the values of the parameters at the moment of component initialization. If parameter will change thereafter, it will not be updated.

**http://localhost/viewSection/Vacation**

**http://localhost/viewSection/Work**

5) Add viewSection route to appRoutes in app.module.ts:

{ path: 'viewSection/:name', component: ViewSectionComponent }

Note: add this route before '**\*\***' path in appRoutes.

6) Add link to view section:

Edit notes.component.html and change {{section}} to make it a link:

<a [routerLink]="['/viewSection',section]">{{section}}</a>

Now you can open this link and see the title of the section (but without the notes).

7) Now we need to read the list of notes for the selected section. We already have this logic in NotesComponent, so let we extract this logic to the service.

8) Create services folder as subfolder of app. Add **NotesServerService** inside it:

@Injectable()  
export class NotesServerService {  
 private notesUrl = 'notes'; *// URL to web api*  
  
constructor(private http: Http) { }

}

Add NotesServerService to providers list is **AppModule**:

providers: [ NotesServerService ]

Move method **getNotes(): Observable<Note[]>** from **NotesComponent** to **NotesServerService**, retrieve section as method parameter:

getNotes(section): Observable<Note[]>

Also mark Note interface in notes.component.ts with **export** to make it available from outside.

Inject **NotesServerService** into **NotesComponent** constructor:

constructor(private http:Http, private notesServer: NotesServerService) {}

9) Also inject **NotesServerService** to **ViewSectionComponent**:

constructor(private route: ActivatedRoute,  
 private noteServer: NotesServerService) {  
}

10) Create method getNotes() in **ViewSectionComponent**:

getNotes() {  
 return this.noteServer.getNotes(this.section);  
}

11) Add field **notes: Note[]** and call to getNotes() in ngOnInit():

ngOnInit() {  
 this.section = this.route.snapshot.params["name"];  
 this.getNotes().subscribe(notes=>this.notes=notes);  
}

Now you can open notes and use the link to open view section.

### 6.3 Page reload: process virtual URL on server side

If you will go to view section and reload page, you will get 404 error from the server. Problem is that server thinks that it should load this URL and it doesn’t know that client side is responsible to process it.

To fix it, we should change server side. It should return index.html in case of URLs controlled from Angular application.

First, open server.js and define variable root in the beginning:

var root = \_\_dirname + '/..'

Also update express.static call to

app.use(express.static(root));

Also add this route to the end of server js (it has to be in the bottom to not override all other routes):

app.get("\*", function(req, res, next) {  
 res.sendFile('index.html', { root : root });  
});

Now you can reload page like <http://localhost:8080/viewSection/Work>

Note that this approach will allow to deploy application only to the server root.

### 6.4 Navigating sections

Now we want to provide section navigation using URL. It will allow using Back/Forward buttons in browser, as well as adding section to bookmarks.

So, if we go to Work section, browser should show this URL:

http://localhost:8080/Work

To implement this, we should update **NotesEditorComponent**.

1) In constructor inject router and route:

constructor(private route: ActivatedRoute, private router: Router) {}

2) In setSection() method, navigate to URL having name of the section:

setSection(section:string) {  
 this.section = section;  
 this.router.navigate([section]);  
}

3) Also add this route to appRoutes in app.module.ts:

{ path: ':name', component: NotesEditorComponent },

You have to insert it after path '**viewSection/:name**', but before path '\*\*', so that it wouldn’t override viewSection path.

4) If you will run application, you will see that URL changes, but if you will try to go forward or backward, nothing will change. This is because we need to handle the change of the URL by subscribing to the changes. To do it, update constructor in NotesEditorComponent and subscribe to the changes in route params:

constructor(private route: ActivatedRoute, private router: Router) {  
 this.route.params  
 .map(params=>params["name"])  
 .subscribe(section=>this.section=section);  
}

It will change section on every change of URL. Now, notes are loaded for the section in URL because section if transmitted to <notes> component with this:

<notes [section]="section"></notes>

However, <sections> component do not get updated from URL active section. To change it, you should update notes editor component by injecting updated section inside section component:

<sections [section]="section" (sectionChanged)="setSection($event)"></sections>

And in **SectionsComponent** introduce setter for section:

@Input()  
set section(section:string) {  
 if (section && section.length>0) {  
 this.activeSection = section;  
 }  
}

So when URL will be changed, this setter will be executed, and **activeSection** will be retrieved from URL and be set.

### 6.5 Using async pipe in ViewSection [optional]

We was retrieving list of notes in ViewSectionComponent by setting notes value. There’s the alternate way to keep notes as the Observable and resolve it in the view. To do it, you should:

1) Introduce field

notes$: Observable<Note[]>;

in ViewSectionComponent

2) Update ngOnInit(): instead of subscribing to notes:

this.getNotes().subscribe(notes=>this.notes=notes);

we just keep the Observable link:

this.notes$ = this.getNotes();

3) Update the template in ViewSectionComponent:

<li \*ngFor="let note of notes$ | async">

Now we resolve observable not in the component code, but in the view.

### 6.6 Prevent changing section if the note text was entered [optional]

We want to ask user if he really wants to go to another section in case if he has entered note text. To implement this, we will be using the mechanism CanDeactivate, which will check the possibility to change the route. To do this:

1) Create service CanDeactivateNote with this code:

@Injectable()  
export class CanDeactivateNote implements CanDeactivate<NotesEditorComponent> {  
  
 canDeactivate(  
 notesEditorComponent: NotesEditorComponent,  
 route: ActivatedRouteSnapshot,  
 state: RouterStateSnapshot  
 ): Observable<boolean>|Promise<boolean>|boolean {

return true;  
 }  
}

2) Now we should use CanDeactivate in our route change mechanism. To do that, change appRoutes configuration in app.module.ts:

{ path: '', component: NotesEditorComponent,

canDeactivate: [CanDeactivateNote] }

And

{ path: ':name', component: NotesEditorComponent,

canDeactivate: [CanDeactivateNote] },

Also add CanDeactivateNote to providers.

Now you can try the mechanism by returning false in CanDeactivateNote.

It’s still not working because we should rely solely on route change, otherwise we would not prevent to change the section.

3) We have to update setSection() in NotesEditorComponent:

setSection(section:string) {  
 *// this.section = section;*  
this.router.navigate([section]);  
}

You see that now we do not change section, we only navigate to another route. Section will be changed after router mechanism will notice the URL change and retrieve section name from URL (see route.params subscribe in constructor).

4) Also we need to update showSection method in SectionsComponent: now it should not change activeSection, but only emit the event of changing section (which will fire setSection method in NotesEditorComponent):

showSection(section:Section) {  
 *//this.activeSection = section;*  
this.sectionChanged.emit(section.title);  
}

Now if you will return false from CanDeactivateNote and restart page, you will see that section change is not happening.

5) Now in CanDeactivateNote we want to check if note text is entered and prevent change of the section. In CanDeactivateNote we have access only to NotesEditorComponent, but note text is defined in NotesComponent. To access NotesComponent from NotesEditorComponent, add this field to NotesEditorComponent:

@ViewChild(NotesComponent) notesComponent:NotesComponent;

It will be give access from parent component (NotesEditorComponent) to child component (NotesComponent).

6) Now update canDeactivate method in CanDeactivateNote: we will get the note text and show confirmation if it was entered. If user will cancel the route change, section will not be changed.

const note = notesEditorComponent.notesComponent.text;  
if (note && note.length>0) {  
 return *window*.confirm(  
 `You have entered the note.  
 Do you really want to change section?`);  
} else return true;

Now you can restart page and see it in action. If you will enter note text and try to change the section, you will see the confirmation. If you press cancel, change of the section will not occur.

## Additional tasks

1) Move/copy note to another section.

2) Create web site on the base of notes. It should have the sections as menu items, and notes as the page contents. Show it on the separate route.

3) Implement subsections.

# Task 7. Form with validation

## Task description

In this task you will create a registration form and validation for the form.

## Time

3 hours

## Detailed description

### 7.1 Create the form

1) Add the button Register to top right of the page:

open app.component.html, and after application title

<div class="navbar-header">  
 <div class="navbar-brand">Notes tutorial app</div>  
</div>

add Register div and button:

<div class="navbar-form navbar-right">  
 <a class="btn btn-primary" routerLink="/register">Register</a>  
</div>

2) Define new route in app.module.ts:

{ path: 'register', component: UserFormComponent },

It should be before ':name' path!

3) Create userForm.component.ts having UserFormComponent. Define templateUrl as



templateUrl: 'app/userForm.component.html'

Don’t forget to register component in declarations of AppModule.

4) Create model/User.ts having User domail model object:

export class User {  
 name:string;  
 password: string;  
 password2: string;  
 subscribe: boolean;  
 email: string;  
 dateOfBirth: string;  
}

5) Add field **user:User** to **UserFormComponent**, initialize it with

new User()

6) Create file userForm.component.html having this code:

<div class="col-md-8 col-md-offset-2">  
 <div class="panel panel-primary">  
 <div class="panel-heading">User registration form</div>  
 <div class="panel-body">  
  
 <form class="css-form" name="userForm"  
 #userForm="ngForm" (ngSubmit)="onSubmit()" >

</form>  
  
 </div>  
 </div>  
</div>

Here we define the form, use **#userForm="ngForm"** to define template variable having access to form metadata. Also we define callback on form submission.

7) Add username field to the form:

<div class="form-group">  
 <label for="userName">User name</label>  
 <input type="text" class="form-control" id="userName"  
 name="userName" placeholder="Username"  
 #name="ngModel" [(ngModel)]="user.name">  
</div>

Here you can see #name which defines template variable referencing ngModel object with metadata (we can access validation results, etc.). Also we define two-way data binding with [(ngModel)]="user.name".

Required is the attribute to validate if this field value is not empty.

8) The same way add fields for **password**, **password2**, **dateOfBirth**, **email**.

9) We will add checkbox "subscribe for newsletters". If this checkbox will be checked, we will show e-mail input box. To implement this, add checkbox this way:

<div class="form-group">  
 <div class="checkbox">  
 <label><input type="checkbox" name="subscribe"  
 [(ngModel)]="user.subscribe">  
 Subscribe for newsletters  
 </label>  
 </div>  
</div>

10) In div containing e-mail we will check if subscribe checkbox is checked:

\*ngIf="user.subscribe"

E-mail will be shown only in this case.

11) In the end of the form define Submit button this way:

<button type="submit" class="btn btn-primary"> Submit</button>

### 7.2 Add validation to the form

1) We want to allow to press Submit only if the whole form is valid. To check this, add this code to Submit button:

[disabled]="!*userForm*.valid"

It will check userForm validity and disable button if any field in the form is validated to invalid value.

2) We will show the results of validation using CSS. To do that, add this code to UserFormComponent @Component decorator:

styles: [`  
 input.ng-touched.ng-invalid {  
 background-color: #ffe8f1;  
 }  
`]

It will show input fields which are touched but invalid with reddish background.

3) Add **required** attribute to <input> of name, password, password2 and dateOfBirth. Now you can reload page and see the form validation by visiting required fields but not entering any data. It will get reddish.

4) We will also apply regular expression validation. For **dateOfBirth** it will be

pattern="[0-9][0-9]\.[0-1][0-9]\.[1-2][0-9][0-9][0-9]"

5) To show the hint if user is using the wrong date format, add this code after dateOfBirth <input> tag:

<span \*ngIf="*dateOfBirth*.errors && *dateOfBirth*.errors.pattern && *dateOfBirth*.touched">  
 Date of birth should be in format dd.mm.yyyy  
</span>

It will check if there’re some errors in date format, and show hint.

6) Do the same for the e-mail field.

### 7.3 Add check that password and repeat password are the same

We want to check if password and password2 match. To do this, we will need to create custom validator validateEqual. We will be using it this way:

<input type="password" name="password2" placeholder="Retype password" [(ngModel)]="user.password2" validateEqual="password">

You see validateEqual attribute – that’s our custom directive. It will take the name of second field as the parameter and will check if these fields are equal. Lets create this directive.

1) Create file directives/EqualToValidator.ts.

2) Define class EqualToValidator implements Validator with this decorator:

@Directive({  
 selector: '[validateEqual][ngModel]',  
 providers: [{provide:NG\_VALIDATORS,

useExisting: EqualToValidator, multi: true}]  
})

Also add EqualToValidator to declarations in AppModule.

3) Define constructor where we will take attribute value of “validateEqual” field:

constructor( @Attribute("validateEqual") public validateEqual: string) {}

This will be used to pass the name of the field which we will be comparing to.

4) Define validate method:

validate(c: AbstractControl): {[key: string]: any} {  
 let v = c.value;  
 let e = c.root.get(this.validateEqual);

if (e && v !== e.value) return { validateEqual: false };  
 return null;  
}

Here we take the value of the field and find another field with which we will compare to.

5) Add the results of validation to the form: after password2 <input> add this:

<span \*ngIf="*password2*.errors && !*password2*.errors.validateEqual && *password2*.touched">  
 Passwords should match  
</span>

It will show error if password2 was touched and not equal to password.

6) However, there’s still one problem: in case if we change password after we have set password2, validation is not re-executed. This happens because validation is working only after changing of the validated field. To fix this, do the following:

In EqualToValidator we will subscribe to all changes in password field:

*// subscribe to future changes in password*  
this.subscription = e.valueChanges.subscribe((val:string)=> {  
 if (val != v) c.setErrors({validateEqual: false});  
 else c.setErrors(null);  
 }  
);

Also add property **subscription: Subscription** to class **EqualToValidator**. We need to keep subscription reference to allow Angular to unsubscribe from valueChanges before subscribing to another callback, on next validate() call.

Now if you will change password, validation will be working correctly.

### 7.4 Add check that userName is unique

That’s all good, but some validations can’t be done on the client. For example, we cannot check the uniqueness of user name on client side, because we need to do database query. For such cases we can use asynchronous validation. Lets define UserUniqueValidator which will check user.

1) Create file directives/UserUniqueValidator.ts

2) Define class

UserUniqueValidator implements Validator

With constructor

constructor( private http: Http) {}

We need http service to access server.

3) Define this @Directive decorator:

@Directive({  
 selector: '[userUniqueValid][formControlName],[userUniqueValid][ngModel]',  
 providers: [{provide:NG\_ASYNC\_VALIDATORS,

useExisting: UserUniqueValidator, multi: true}]  
})

4) Define validate method:

validate(c: AbstractControl): Promise<{[key: string]: any}> {  
 const user = c.value;  
 return new *Promise*(resolve =>  
 this.http.get("checkUserUnique", {params:{user}})  
 .map(response => response.json())  
 .subscribe(res =>  
 res?resolve(null):resolve({userUniqueValid:false})));  
}

Here validator returns not the value, but the promise.

5) Add "/**checkUserUnique**" route to server.js:

Since we don't have users table and can't check userName for uniqness, let create some mock implementation for a while (we’ll implement it later):

app.get("/checkUserUnique", function(req,res) {

res.send(req.query.user.length>2);

});

6) Add **userUniqueValid** attribute to username <input> in the form

7) Add this error message after username <input>:

<span \*ngIf="*name*.errors && *name*.value?.length>0  
 && !*name*.errors.userUniqueValid && *name*.dirty">  
 User name is not unique. Please select another one.  
</span>

### 7.5 Add user to mongodb

1) In server.js add new collection users:

db.collection('users', function(error, users) {

db.users = users;

});

2) Add possibility to post user to users collection:

app.post("/users", function(req,res) {

db.users.insert(req.body, function(resp) {

req.session.userName = req.body.name;

res.end();

});

});

3) In UserFormComponent add **onSubmit()** method:

onSubmit() {  
 this.http.post("users", this.user).subscribe(res=>{  
 this.router.navigateByUrl("");  
 });  
}

4) Also modify constructor of UserFormComponent:

constructor(private http:Http, private router: Router) {}

Now you can register the user. To check that user was added to database, execute mongo and look at list of the users:

db.users.find();

### 7.6 Implement check userName for uniqueness

Implement it yourself.

#### Additional tasks

1) Check that user's age is >12 (create special directive)

2) Implement drop-down to select country and city (list of countries and cities should be loaded from database), so that if user selects country, the list of cities would be loaded

# Task 8. Multiuser support

## Task description

In this task you will submit the form and create users on server side.

## Time

2 hours

## Detailed description

### 8.1 Show sections and notes for the current user

1) Execute mongo

2) Run

use tutor

db.notes.update({},{$set:{userName:"demo"}}, {multi:true})

db.users.insert({userName:"demo",password:"demo", sections:[{title:"Work"},{title:"Vacations"},{title:"Children"}]})

3) Check updates:

show collections

db.users.find()

db.notes.find()

4) In **server.js**

1. Add function **setUser** to add **userName** to query objects:

function setUserQuery(req) {

req.query.userName = req.session.userName || "demo";

}

1. Update all queries to include user:

app.get("/notes", function(req,res) {

setUserQuery(req);

db.notes.find(req.query)

.toArray(function(err, items) {

res.send(items);

});

});

Also update post notes by adding this:

req.body.userName = req.session.userName || "demo";

1. Define sections as the array in users table.

New structure of user in users collection will be like this:

{ userName: "John",

password:"jpass",

sections: [{title:"Work"}, {title:"Vacation"}, {title:"Hobby"}]

}

For this:

4.1 Update app.get("/sections"):

app.get("/sections", function(req,res) {

var userName = req.session.userName || "demo";

db.users.find({userName:userName})

.toArray(function(err, items) {

var user = items[0];

res.send(user.sections||[]);

});

});

4.2 Rewrite app.post("/sections/replace"):

app.post("/sections/replace", function(req,res) {

var userName = req.session.userName || "demo";

db.users.update({userName:userName},

{$set:{sections:req.body}},

function() {

res.end();

});

});

### 8 .2 Implement login form

Now lets implement the login functionality.

1) Create **LoginService** in **app/service**

@Injectable()  
export class LoginService {

private loginUrl = 'login'; *// URL to web api*  
private logoutUrl = 'logout'; *// URL to web api*  
loggedIn: boolean = false;

constructor(private http: Http) { }

}

2) Define class **LoginUser** in **LoginService**:

export class LoginUser {  
 userName: string;  
 password: string;  
}

3) Add **login()** and **logout()** methods to **LoginService**:

login(user: LoginUser): Observable<boolean> {  
 return this.http.post(this.loginUrl, user)  
 .map(response => response.json() as boolean)  
 .do(res => { if (res) this.userLogin(user) });  
}  
  
logout() {  
 return this.http.get(this.logoutUrl)  
 .do(res => this.userLogout());  
}

Also define userLogin() and userLogout() methods:

userLogin(user: LoginUser) {  
 this.loggedIn = true;  
}  
  
userLogout() {  
 this.loggedIn = false;  
}

4) Create login form component app/loginForm.component.ts

@Component({  
 selector: 'login-form',  
 templateUrl: 'app/loginForm.component.html'  
})  
export class LoginFormComponent {  
 userForm: LoginUser = new LoginUser();  
 failedLogin: boolean;  
  
 constructor(private loginService: LoginService, private router: Router) {}  
  
 login() {  
 this.loginService.login(this.userForm)  
 .subscribe(res=>res?this.onSuccessLogin():this.onFailLogin());  
 }  
  
 logout() {  
 this.loginService.logout().subscribe(res=>this.onLogout());  
 }  
  
  
 onSuccessLogin() {  
 this.router.navigateByUrl("/");  
 }  
  
 onFailLogin() {  
 this.failedLogin = true;  
 }  
  
 onLogout() {  
 this.router.navigateByUrl("/");  
 }  
}

Note that we are redirecting to start page on login and logout.

5) Define template for login form component in app/loginForm.component.html:

<form \*ngIf="!loggedIn" style="margin-bottom: auto">  
 <div class="form-group">  
 <input type="text" placeholder="Username" name="username"  
 class="form-control" [(ngModel)]="userForm.userName"  
 (keyup.enter)="login()">  
 </div>  
 <div class="form-group">  
 <input type="password" placeholder="Password" name="password" class="form-control" [(ngModel)]="userForm.password">  
 </div>  
 <button type="submit" class="btn btn-success" (click)="login()">

Sign in

</button>  
  
 <ng-content></ng-content>  
 <div \*ngIf="failedLogin" style="color:white">  
 Wrong username or password  
 </div>  
</form>  
<div \*ngIf="loggedIn" style="color:white">  
 User: <b>{{userForm?.username}}</b>  
 <button type="submit" class="btn btn-success" (click)="logout()">

Logout

</button>  
</div>

Here you see the login form with Sign in button. If user is already logged in, he will see user name and Logout button.

We will use **loggedIn** flag from **LoginService** to see if user logged in.

To retrieve it from component, add this method to **LoginFormComponent**:

get loggedIn() {  
 return this.loginService.loggedIn;  
 }

Also we will show message in case of wrong login/password. We’ll show it during 1 second, then it disappears. To implement this, modify **onFailLogin()** method by adding this line:

*setTimeout*(() => this.failedLogin = false, 1000);

6) Now add login component to **AppComponent** template (in place of Register button – now Register button will be inside the form):

<div class="navbar-form navbar-right">  
 <login-form>  
 <a class="btn btn-primary" routerLink="/register">Register</a>  
 </login-form>  
</div>

Also modify "Notes tutorial app" div to make it a link:

<div class="navbar-brand" routerLink="/" style="cursor:hand">

Notes tutorial app

</div>

7) Implement login and logout on server side:

app.post("/login", function(req,res) {  
 db.users.*find*(  
 {userName:req.body.userName,  
 password:req.body.password})  
 .toArray(function(err, items) {  
 if (items.length>0) {  
 req.session.userName = req.body.userName;  
 }  
 res.send(items.length>0);  
 });  
});

app.get("/logout", function(req, res) {  
 req.session.userName = null;  
 res.end();  
});

Now you can reload page and login.

### 8.3 Reload sections and notes on login

However, you will see that on login sections and notes are not updated. To reload it, we have to notify **SectionsComponent** to reread list of sections and update notes. To do it, let’s add possibility to subscribe to Login event in **LoginService**:

1) Add properties

private userLoginSource = new Subject<LoginUser>();  
userLogin$ = this.userLoginSource.asObservable();

Notice that you have to import Subject this way:

import { Subject } from 'rxjs/Subject';

2) Also modify userLogin() and userLogout() methods to emit the event:

userLogin(user: LoginUser) {  
 this.loggedIn = true;  
 this.userLoginSource.next(user);  
}  
  
userLogout() {  
 this.loggedIn = false;  
 this.userLoginSource.next(null)  
}

Now we can subscribe to the login event in **SectionsComponent**:

3) Add **LoginService** to constructor injection in **SectionsComponent**

4) Add subscription to constructor:

this.loginService.userLogin$.subscribe(user => this.readSections())

This will fire readSections() on login of the new user – and we’ll see the updated list of sections and notes. You can try.

### 8.4 Login after registration

When we register a new user, we would like to login into system right after registering. To do this, modify **UserFormComponent**: add **LoginService** to constructor arguments to be injected. Also modify **onSubmit()** to do login after submitting the form:

onSubmit() {  
 this.http.post("users", this.user).subscribe(res=>{  
 this.loginService.login({username:this.user.userName,  
 password: this.user.password})  
 .subscribe(res=>{if (res) this.router.navigateByUrl("/")});  
 });  
}

Also we need to get information about registered user in LoginForm to correctly show user data in right upper corner. To do this, add this line to the constructor of LoginFormComponent:

**this**.loginService.**userLogin$**  
.subscribe(user=>**this**.**userForm** = user || **new** LoginUser());

Now after registration of the new user you will see user logged in.

## Additional tasks

1) After page reload user needs to login again. How to fix it?

2) Implement edit of the user data