Setup Angular Project with Express server

Notizbuch:	Arbeit		
Erstellt:	16.10.2017 13:30	**Geändert:**	18.10.2017 13:25
Autor:	farassinger@gmx.at		

VS-Code, NodeJs required

---

1. Create a folder on any location you want. I name mine "Todo" and go there with the command line of your choice.
2. In the folder call **`npm init`** and follow the instructions to create a package.json for the project.
3. Next we go back to the console and type **`npm install --save express`**, which will install us the express framework and save it to the package.json.
4. Then we go in VSCode and create a new folder called "server".
5. Within this folder "server" we create a new file called, "app.js", which contains the server.

app.js - basic:

```
// Load the express module.
let express = require('express');
const port = 3001;
let app = express();

// This function is called when the root or home page is opened.
app.get('/', function(req, res) {
res.send('Hello World');
});

// Start the server on the port you specify here.
app.listen(port);
```

6. In order to start the "app.js" change back to the terminal and change into the server folder and type **`node app.js`**. This will start the server.
7. If you go on "localhost:3001" you will see the result of the server printed.
8. Next we add another route to the server which is called "/todos" which will return the todos later.

app.js - basic2:

```
// Load the express module.
let express = require('express');
const port = 8001;
let app = express();

// This function is called when the root or home page is opened.
app.get('/', function(req, res) {
res.send('Hello World');
});
```

```
// This function is called upon visiting /todos and returns the todos.
app.get('/todos', function(req, res) {
res.send('Todos');
});

// Start the server on the port you specify here.
app.listen(port);
```

9. Next we fill the todos function with some todos that are sent back from our server. Normally we would make here a database call with mongodb and get the data from there.

app.js - basic 3:

```
// Load the express module.
let express = require('express');
const port = 8001;
let app = express();

// This function is called when the root or home page is opened.
app.get('/', function(req, res) {
res.send('Hello World');
});

// This function is called upon visiting /todos and returns the todos.
app.get('/api/1.0.0/todos', function(req, res) {
res.send([
{
title: 'First Todo',
content: 'I have a lot of work to do.'
},
{
title: 'Second Todo',
content: 'I have to write a lot of stuff.'
}
]);
});

// Start the server on the port you specify here.
app.listen(port);
```

10. The next part is the angular application that should be delivered by the server. So we create in the "server" folder a new subfolder called "views"
11. Then we add a basic index.html to the folder in order to test if the server delivers it.
12. Next we send the file if the user visits the home page "/". Therefore we make small changes to our app.js. We include the "path" module in order to provide relative paths for any windows or mac user.

app.js - basic 4:

```
// Load the express module.
let express = require('express');
// The path module in order to produce realtive paths for all OS.
let path = require('path');
```

```
const port = 8001;
let app = express();

// This function is called when the root or home page is opened.
app.get('/', function(req, res) {
res.sendFile(path.join(__dirname,'views', 'index.html'));
});

// This function is called upon visiting /todos and returns the todos.
app.get('/api/1.0.0/todos', function(req, res) {
res.send([
{
title: 'First Todo',
content: 'I have a lot of work to do.'
},
{
title: 'Second Todo',
content: 'I have to write a lot of stuff.'
}
]);
});

// Start the server on the port you specify here.
app.listen(port);
```

13. Next we want to place our angular application. Therefore we create a new folder in the root of our project, which is called "client".
14. Then we have to add the following dependencies to our package.json, if we dont generate it with the Angular CLI

package.json:

```
{
"name": "todo",
"version": "1.0.0",
"description": "",
"main": "index.js",
"scripts": {
"test": "echo \"Error: no test specified\" && exit 1"
},
"author": "",
"license": "ISC",
"dependencies": {
"express": "^4.16.2",
"@angular/common": "~2.4.0",
"@angular/compiler": "~2.4.0",
"@angular/core": "~2.4.0",
"@angular/forms": "~2.4.0",
"@angular/http": "~2.4.0",
"@angular/platform-browser": "~2.4.0",
"@angular/platform-browser-dynamic": "~2.4.0",
"@angular/router": "~3.4.0",

"angular-in-memory-web-api": "~0.2.4",
"systemjs": "0.19.40",
"core-js": "^2.4.1",
"rxjs": "5.0.1",
"zone.js": "0.7.4"
},
```

```
"devDependencies": {
"concurrently": "^3.1.0",
"lite-server": "^2.2.2",
"typescript": "~2.1.6",
"typings": "^1.4.0",
"canonical-path": "0.0.2",
"http-server": "^0.9.0",
"tslint": "^3.15.1",
"lodash": "^4.16.4",
"jasmine-core": "~2.4.1",
"karma": "^1.3.0",
"karma-chrome-launcher": "^2.0.0",
"karma-cli": "^1.0.1",
"karma-jasmine": "^1.0.2",
"karma-jasmine-html-reporter": "^0.2.2",
"protractor": "~4.0.14",
"rimraf": "^2.5.4",

"@types/node": "^6.0.46",
"@types/jasmine": "^2.5.36"
}
}
```

15. Next we go back one layer in the terminal and execute the command **`npm install`**. This will install us all further dependencies.
16. Now we go back to the "index.html" in order to start the application appropriately we have to add a few stuff to the index.html.

index.html:

```
<!DOCTYPE html>
<html>
<head>
<title>Todos</title>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
integrity="sha384-
BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
crossorigin="anonymous">

<link rel="stylesheet" href="styles.css">

<!-- Polyfill(s) for older browsers -->
<script src="node_modules/core-js/client/shim.min.js"></script>

<script src="node_modules/zone.js/dist/zone.js"></script>
<script src="node_modules/systemjs/dist/system.src.js"></script>

<!-- <script src="systemjs.config.js"></script>
<script>
System.import('app').catch(function(err){ console.error(err); });
</script>
</head>

<body>
<my-app>Loading...</my-app>
</body> -->
```

```
</html>
```

17. Furthermore we need to adept the app.js file in order to deliver the "node_modules" folder also. As currently the webserver only would server the index.html file.

app.js - basic 5:

```
// Load the express module.
let express = require('express');
// The path module in order to produce realtive paths for all OS.
let path = require('path');
let app = express();

const port = 8001;
const nodeModulesPath = path.join(__dirname, '..', 'node_modules');
// This tells express to server also the folders in the path. The first
parameter
// is important to let the path start with node_modules. Otherwise it would
server the content.
app.use('/node_modules', express.static(nodeModulesPath));

// This function is called when the root or home page is opened.
app.get('/', function(req, res) {
res.sendFile(path.join(__dirname, 'views', 'index.html'));
});

// This function is called upon visiting /todos and returns the todos.
app.get('/api/1.0.0/todos', function(req, res) {
res.send([
{
title: 'First Todo',
content: 'I have a lot of work to do.'
},
{
title: 'Second Todo',
content: 'I have to write a lot of stuff.'
}
]);
});

// Start the server on the port you specify here.
app.listen(port);
```

18. Now we need to add also the "systemjs.config.js" file which tells Angular how to load modules and build our configuration. The idea behind the file is that the browser exactly knows where to load the code from. Therefore we create a new file called "systemjs.config.js" in the "client" folder and add the following code to it.

systemjs.config.js:

```
/**
 * System configuration for Angular samples
 * Adjust as necessary for your application needs.
 */
(function (global) {
    System.config({
```

```
        paths: {
            // paths serve as alias
            'npm:': 'node_modules/'
        },
        // map tells the System loader where to look for things
        map: {
            // our app is within the app folder
            app: 'app',

            // angular bundles
            '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
            '@angular/common': 'npm:@angular/common/bundles/common.umd.js',
            '@angular/compiler':
'npm:@angular/compiler/bundles/compiler.umd.js',
            '@angular/platform-browser': 'npm:@angular/platform-
browser/bundles/platform-browser.umd.js',
            '@angular/platform-browser-dynamic': 'npm:@angular/platform-
browser-dynamic/bundles/platform-browser-dynamic.umd.js',
            '@angular/http': 'npm:@angular/http/bundles/http.umd.js',
            '@angular/router': 'npm:@angular/router/bundles/router.umd.js',
            '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',

            // other libraries
            'rxjs':                      'npm:rxjs',
            'angular-in-memory-web-api': 'npm:angular-in-memory-web-
api/bundles/in-memory-web-api.umd.js'
        },
        // packages tells the System loader how to load when no filename
and/or no extension
        packages: {
            app: {
                main: './main.js',
                defaultExtension: 'js'
            },
            rxjs: {
                defaultExtension: 'js'
            }
        }
    });
})(this);
```

19. To load the file we have to further adapt our server in order to server again our file. Furthermore we need to uncomment the previous lines in the index.html file in order to load the configuration.

app.js - basic 6:

```
...
const clientPath = path.join(__dirname, '..', 'client');
app.use('/client', express.static(clientPath));
...
```

20. To load the module or the entry point of Angular we need to create the "main.js" and the "app" folder where our angular application lives. Therefore we create a new Subfolder in the "client" folder called "app" and within there a file "main.js"
21. Next we need to adapt the systemjs.config.js and add the "client/app" path before the

"app" path. Also the same goes for the index.html which looks like this then

index.html:

```
<!DOCTYPE html>
<html>
<head>
<title>Todos</title>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
integrity="sha384-
BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
crossorigin="anonymous">

<link rel="stylesheet" href="styles.css">

<!-- Polyfill(s) for older browsers -->
<script src="node_modules/core-js/client/shim.min.js"></script>

<script src="node_modules/zone.js/dist/zone.js"></script>
<script src="node_modules/systemjs/dist/system.src.js"></script>

<script src="client/systemjs.config.js"></script>
<script>
System.import('app').catch(function(err){ console.error(err); });
</script>
</head>

<body>
<my-app>Loading...</my-app>
</body>
</html>
```

22. Now we need to add typescript support for our application in order to program all in typescript. So first of all we rename our "main.js" to "main.ts" and then add the appropriate scripts to the package.json in order to automatically compile our typescript code to javascript.

package.json:

```
...
"scripts": {
"tsc": "tsc",
"test": "echo \"Error: no test specified\" && exit 1"
},
...
```

23. Before we need to create a "tsconfig.json" file in the root of our project and add the following configurations to it.

tsconfig.json:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": true,
    "lib": [ "es2015", "dom" ],
    "suppressImplicitAnyIndexErrors": true
  }
}
```

24. Now we can compile our .ts files to .js and .map files by running the command **`npm run tsc`**.
25. Next we want to add the further code to the main.ts that loads our angular application. The problem though is that we cant load the libraries without "typings". These allow us to include javscript modules in our typescript file. Therefore we need to create a file called "typings.json" that contains information about the types and how to load them and add them. The file needs to be in our root folder again.

typings.json:

```
{
"globalDependencies": {
"angular-protractor": "registry:dt/angular-
protractor#1.5.0+20160425143459",
"core-js": "registry:dt/core-js#0.0.0+20160725163759",
"jasmine": "registry:dt/jasmine#2.2.0+20160621224255",
"node": "registry:dt/node#6.0.0+20160831021119",
"selenium-webdriver": "registry:dt/selenium-
webdriver#2.44.0+20160317120654"
}
}
```

26. Now we need to tell our project how to load the type definitions. Therefore we add another script to our package.json.

package.json:

```
...
"scripts": {
"start": "tsc && concurrently \"tsc -w\" \"node server/app.js\"",
"run-server": "node server/app.js",
"tsc": "tsc",
"tsc:w": "tsc -w",
"test": "echo \"Error: no test specified\" && exit 1"
},
...
```

27. Afterwards we go back to our console and type in the project root **`npm run typings install`**, which will intsall all typings we specified in the .json.
28. Forget the typings part!!!!
29. Start from the final project "FINAL_START"
30. The typings part is obsolete and depricated with typescript 2.0.
31. Then we only need to create our "app.module.ts" file with the following content, in our "app" folder of the "client".

app.module.ts:

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

@Component({
    selector: 'todo-app',
    template: `<p>Angular is loaded!</p>`
})
export class AppComponent {

}
```

32. Our "main.ts" looks like this.

main.ts:

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

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

platformBrowserDynamic().bootstrapModule(AppModule);
```

33. One more time the full package.json.

package.json:

```
{
"name": "todo",
"version": "1.0.0",
"description": "",
"main": "index.js",
"scripts": {
"start": "tsc && concurrently \"tsc -w\" \"node server/app.js\"",
"run-server": "node server/app.js",
"tsc": "tsc",
"tsc:w": "tsc -w",
"test": "echo \"Error: no test specified\" && exit 1"
},
"author": "",
"license": "ISC",
"dependencies": {
"@angular/common": "4.1.3",
"@angular/compiler": "4.1.3",
"@angular/core": "4.1.3",
"@angular/forms": "4.1.3",
"@angular/http": "4.1.3",
```

```
"@angular/platform-browser": "4.1.3",
"@angular/platform-browser-dynamic": "4.1.3",
"@angular/router": "4.1.3",
"@angular/upgrade": "4.1.3",
"angular-in-memory-web-api": "~0.3.2",
"body-parser": "^1.15.2",
"bootstrap": "^3.3.6",
"core-js": "^2.4.1",
"express": "^4.14.0",
"reflect-metadata": "^0.1.10",
"rxjs": "5.4.2",
"systemjs": "0.20.12",
"zone.js": "0.8.11",

"@types/node": "^6.0.46",
"@types/jasmine": "^2.5.36"
},
"devDependencies": {
"concurrently": "^2.2.0",
"lite-server": "^2.2.2",
"typescript": "^2.0.3",
"typings": "^1.4.0",
"canonical-path": "0.0.2",
"http-server": "^0.9.0",
"tslint": "^3.15.1",
"lodash": "^4.16.1",
"jasmine-core": "~2.5.2",
"karma": "^1.3.0",
"karma-chrome-launcher": "^2.0.0",
"karma-cli": "^1.0.1",
"karma-htmlfile-reporter": "^0.3.4",
"karma-jasmine": "^1.0.2",
"karma-jasmine-html-reporter": "^0.2.2",
"protractor": "^3.3.0",
"rimraf": "^2.5.2"
},
"repository": {}
}
```

34. Then we create the "app.module.ts" file that is the entry point for our application.

app.module.ts:

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

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

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

35. Furthermore the index.html gets updated as well to match our new angular component.

index.html:

```
<!DOCTYPE html>
<html>
<head>
<title>Todos</title>

<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1">

<!-- Polyfill(s) for older browsers -->
<script src="node_modules/core-js/client/shim.min.js"></script>

<script src="node_modules/zone.js/dist/zone.js"></script>
<script src="node_modules/reflect-metadata/Reflect.js"></script>
<script src="node_modules/systemjs/dist/system.src.js"></script>

<script src="client/systemjs.config.js"></script>
<script>
System.import('app').catch(function(err){ console.error(err); });
</script>
</head>
<body>
<todo-app></todo-app>
</body>
</html>
```

36. Dann ist es noch wichtig die Skripte der package.json zu erklären. Wir fügen hier einen Watcher hinzu der automatisch die Datei neu lädt wenn sich etwas ändert. Auch werden alle wieder kompiliert.
37. Der **`npm start`** Befehl führt ein paar Dinge gleichzeitig aus. Dadurch kann einfach der Server gestartet werden und auch das Programm.
38. Now wer recrate the Todo app and add Todos from a database. So first of all we start with the Project "FINAL_START".

FINAL_START:

---

1. We create a new file called "rxjs-operators.ts" in our "app" folder, which will be used for all functions that we need with our Observables.

rxjs-operators.ts:

```
// import 'rxjs/Rx'; // Would add all rxjs statics and operators for
Observables

// Statics
import 'rxjs/add/observable/from';

// Operators
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/mergeMap';
```

2. Next we need to import that in our app.module.ts, by adding the "import './rxjs-operators'" to our module file.
3. Then we add the HTTP Module also to our app.module.ts in order to simulate a database or server.

app.module.ts:

```
import './rxjs-operators';

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

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

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

4. Now we fetch the todos from our server not directly in our app component but we create a service in the "app" folder of our project. Therefore we create a file called "todo.service.ts".
5. We also need to add the TodoService of course to our "app.module.ts" as import again and the respective declaration in the "providers" array of the NgModule().
6. We can than give our Service to our constructor in the app.component.ts and initialize it.
7. Within our Service we save the todos as an observable array. Which will contain all new todos.
8. First we need to create a new Class for a todo. Thats why we create a new file in the "app" root folder called "todo.ts".
9. In order to fetch data from the express server, we have to create an ajax request within our service. We do this in the constructor.
10. We create an extra function that makes an http get request on the server and the server returns the list of todos.
11. Then we save the response of the request in the observable.
12. In the app.component.ts we can then get the Observable in the ngOnInit() lifecycle event and save it in a component specific variable.
13. We initialize an observable also in our app.component.ts and save the observable from the server there.
14. In the "li" element where we print out the todos we use the "| async" pipe in order to display the todos and update the "li" elements if new todos would arrive.
15. So far the files look like this.

app.component.ts:

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs/Observable';

import { TodoService } from './todo.service';
import { Todo } from './todo';

@Component({
    selector: 'todo-app',
    template: `
    <h1>Todo List</h1>
    <ul>
        <li *ngFor='let todo of todos$|async'>
            <strong>{{ todo.title }}</strong>
            <span>{{ todo.content }}</span>
        </li>
    </ul>
    `
})
export class AppComponent {
    todos$: Observable<Todo[]>;

    constructor(private _todoService: TodoService)  {
    }

    // On initialization of the component we load the todo list first and
save it locally.
    ngOnInit() {
        this.todos$ = this._todoService.todos$
    }
}
```

app.module.ts:

```
import './rxjs-operators';

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

import { AppComponent } from './app.component';
import { TodoService } from './todo.service';

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

todo.service.ts:

```
import { Injectable } from '@angular/core';
```

```
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';

import { Todo } from './todo';

// Tells other modules and components it's a service
@Injectable()
export class TodoService {

// We get an array of todos back from the server. So we get always a new
array of todos and
// replace the observable with the new list.
public todos$: Observable<Todo[]> = null;

// Initialize the http service.
constructor(private _http: Http) {
this.loadTodos();
}

/**
* This function loads our todos from the server.
*/
public loadTodos() {
this.todos$ =
this._http.get('/api/1.0.0/todos')
.map((response: Response) => { // First we return a json from the server
response.
return response.json();
})
.map((list: any) => { // Second we confert the Object type from the server
to Todo
let todoList: Todo[] = []; // Create an empty list of Todos
for (let elm of list) {
todoList.push(new Todo(elm['title'], elm['content']));
}
return todoList;
});
}
}
```

todos.ts:

```
export class Todo {
constructor(public title: string, public content: string) { }
}
```

16. Now we adapt the app.js in order to let our application store new todos and not only retrive those 2 we already defined.
17. In order to add the todos to a database we could use mongoDb or MySQL but this would be far too much now. Therefore we save them in a local array on the server.
18. To add new todos we create a form in our app.component.ts. We double bind our variables with ngModel again and add the new todo to the server within a button click function.
19. So in order to use ngModel we need to import the "FormsModule" in our "app.module.ts".
20. In order to save our todos we also need to add a method in our todo.service.ts file to save them to the server.
21. There we make a put request to the server.

22. On the server we have to save the new todo in our array or our database if we have one.
23. So first of all we need to install a new module with **`npm install --save body-parser`**. This will help us with the save of the todo.
24. Then we can access the body part of the request and read out the properties.
25. The updated files look like this.

app.js:

```
// Load the express module.
let express = require('express');
// The path module in order to produce realtive paths for all OS.
let path = require('path');
// Load the body parser module to deconstruct an put request.
let bodyParser = require('body-parser');
let app = express();

const port = 8001;
const nodeModulesPath = path.join(__dirname, '..', 'node_modules');
// This tells express to server also the folders in the path. The first
parameter
// is important to let the path start with node_modules. Otherwise it would
server the content.
app.use('/node_modules', express.static(nodeModulesPath));

const clientPath = path.join(__dirname, '..', 'client');
app.use('/client', express.static(clientPath));

// Tell our app that we want to use the body parser module with json
functionality.
app.use(bodyParser.json());

// This function is called when the root or home page is opened.
app.get('/', function(req, res) {
res.sendFile(path.join(__dirname, 'views', 'index.html'));
});

let todos = [
{
title: 'First Todo',
content: 'I have a lot of work to do.'
},
{
title: 'Second Todo',
content: 'I have to write a lot of stuff.'
}
];

// This function is called upon visiting /todos and returns the todos.
app.get('/api/1.0.0/todos', function(req, res) {
res.send(todos);
});

// This function is called upon visiting /todos/create and adds a new todo
to our local array.
app.put('/api/1.0.0/todos/create', function(req, res) {
todos.push({
title: req.body['title'],
content: req.body['content']
});
```

```
res.send({success: true});
});

// Start the server on the port you specify here.
app.listen(port);
```

app.component.ts:

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs/Observable';

import { TodoService } from './todo.service';
import { Todo } from './todo';

@Component({
    selector: 'todo-app',
    template: `
    <h1>Todo List</h1>
    <ul>
        <li *ngFor='let todo of todos$|async'>
            <strong>{{ todo.title }}</strong>
            <span>{{ todo.content }}</span>
        </li>
    </ul>
    <h3>Add new Todo:</h3>
    <input type='text' [(ngModel)]='newTodo.title'/>
    <input type='text' [(ngModel)]='newTodo.content'/>
    <button (click)='onAddTodo()'>Save Todo</button>
    `
})
export class AppComponent {
    todos$: Observable<Todo[]>;
    newTodo = new Todo('', '');

    constructor(private _todoService: TodoService)  {
    }

    // On initialization of the component we load the todo list first and
save it locally.
    ngOnInit() {
        this.todos$ = this._todoService.todos$
    }

    /**
    * This method creates a new todo for the service and initializes the
call to the server.
    */
    onAddTodo() {
        this._todoService.create(this.newTodo);
    }
}
```

todo.service.ts:

```
import { Injectable } from '@angular/core';
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';

import { Todo } from './todo';

// Tells other modules and components it's a service
```

```
@Injectable()
export class TodoService {

// We get an array of todos back from the server. So we get always a new
array of todos and
// replace the observable with the new list.
public todos$: Observable<Todo[]> = null;

// Initialize the http service.
constructor(private _http: Http) {
this.loadTodos();
}

/**
* This function loads our todos from the server.
*/
public loadTodos() {
this.todos$ =
this._http.get('/api/1.0.0/todos')
.map((response: Response) => { // First we return a json from the server
response.
return response.json();
})
.map((list: any) => { // Second we confert the Object type from the server
to Todo
let todoList: Todo[] = []; // Create an empty list of Todos
for (let elm of list) {
todoList.push(new Todo(elm['title'], elm['content']));
}
return todoList;
});
}

/**
* This method create a new todo by making a get request to the server.
* @param todo we want to create
*/
public create(todo: Todo) {
// We need to use the .forEach in order to make the request. It will not be
executed till
// the function is called as the observable is 'lazy'. It waits till it
really is needed.
this._http.put('api/1.0.0/todos/create', todo).forEach((response: Response)
=> {
console.log(response);
});
}
}
```

26. To update the angular app automatically when we add a new todo we need to add a new Class called "Subject" in our todo.service.ts. This will allow us to notify our main component when we have created a new observable in our loadTodos() method.
27. We change our todos$ observable to a Subject which has more functionality and can add functionlity from outside.
28. Now the instance of Subject always stays the same and we can update the component.
29. So afterwards we can call loadTodos() again and refresh the page.
30. The final file looks like this.

todo.service.ts:

```
import { Injectable } from '@angular/core';
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';
import { Subject } from 'rxjs/Subject';

import { Todo } from './todo';

// Tells other modules and components it's a service
@Injectable()
export class TodoService {

// We get an array of todos back from the server. So we get always a new
array of todos and
// replace the observable with the new list. We change it to a Subject
which has more functionality.
public todos$: Subject<Todo[]> = new Subject();

// Initialize the http service.
constructor(private _http: Http) {
this.loadTodos();
}

/**
* This function loads our todos from the server.
*/
public loadTodos() {
this._http.get('/api/1.0.0/todos')
.map((response: Response) => { // First we return a json from the server
response.
return response.json();
})
.map((list: any) => { // Second we confert the Object type from the server
to Todo
let todoList: Todo[] = []; // Create an empty list of Todos
for (let elm of list) {
todoList.push(new Todo(elm['title'], elm['content']));
}
return todoList;
})
.forEach((list: Todo[]) => {
this.todos$.next(list);
});
}

/**
* This method create a new todo by making a get request to the server.
* @param todo we want to create
*/
public create(todo: Todo) {
// We need to use the .forEach in order to make the request. It will not be
executed till
// the function is called as the observable is 'lazy'. It waits till it
really is needed.
this._http.put('api/1.0.0/todos/create', todo).forEach((response: Response)
=> {
console.log(response);
this.loadTodos();
});
}
}
```