# Lab: Using the CRUD app with a Web Services

## Introduction

REST stands for Representational State Transfer. (It is sometimes spelled "ReST".) It relies on a stateless, client-server, cacheable communications protocol -- and in virtually all cases, the HTTP protocol is used. REST is an architecture style for designing networked applications.

REST-compliant web services allow requesting systems to access and manipulate textual representations of [web resources](https://en.wikipedia.org/wiki/Web_resource) using a uniform and predefined set of [stateless](https://en.wikipedia.org/wiki/Stateless_protocol) operations.

In this lab you will use a RESTful web service in the Todo app to provide data persistence(Picture editing programs or word processors, for example, achieve [state](https://en.wikipedia.org/wiki/State_(computer_science)) persistence by saving their documents to files.). This lab is based on this excellent article about Angular 2 Observable Data Services –read before starting:

An Observable is a collection that arrives over time. Observables can be used to model events, asynchronous requests, and animations.

The Promise object is used for asynchronous computations. ... The executor function is executed immediately by the Promise implementation, passing resolve and reject functions (the executor is called before the Promise constructor even returns the created object).

RxJs stands for Reactive Extensions for Javascript

implementation of Observables for Javascript

**polyfill** is code that implements a feature on [web browsers](https://en.wikipedia.org/wiki/Web_browser) that do *not* support the feature. Most often, it refers to a [JavaScript](https://en.wikipedia.org/wiki/JavaScript) [library](https://en.wikipedia.org/wiki/Library_(computing)) that implements an [HTML5](https://en.wikipedia.org/wiki/HTML5) [web standard](https://en.wikipedia.org/wiki/Web_standard), either an established standard (supported by some browsers) on older browsers, or a proposed standard (not supported by any browsers) on existing browsers. Formally, "a polyfill is a [shim](https://en.wikipedia.org/wiki/Shim_(computing)) for a browser [API](https://en.wikipedia.org/wiki/Application_programming_interface)".

var obs = Rx.Observable.interval(1000).take(5);

<https://coryrylan.com/blog/angular-2-observable-data-services>

Also <http://blog.angular-university.io/functional-reactive-programming-for-angular-2-developers-rxjs-and-observables/>

Observables are used in to manage asynchronous data in the application. The application will use a local copy of the Todo list which will be loaded from and saved to an external web service in the background. The user does not need to know that it may take time to save the data remotely and certainly won’t want to wait. Observables enable this behaviour.

While this lab is based on the article above, there are some important changes (e.g. the article demo is incomplete), so please follow this carefully.

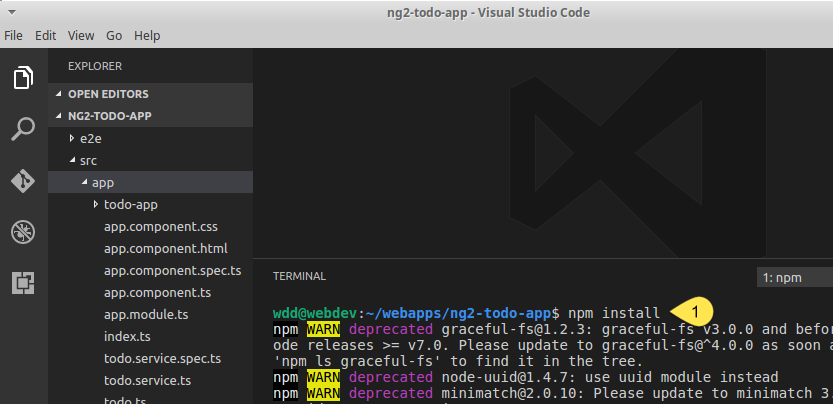
## Getting started

Start this lab by importing the following repository into your own Bitbucket or GitHub account.

https://elee-ittdublin@bitbucket.org/elee-ittdublin/ng2-todo-ws-start.git

Clone your new repository and open in visual studio code

Run npm install in the VS Code integrated terminal (enable it from the view menu)



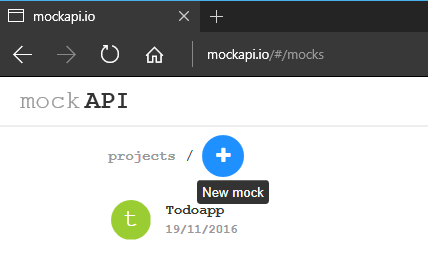
When complete start the application by entering ng serve

## 1. Create a Web Service for testing

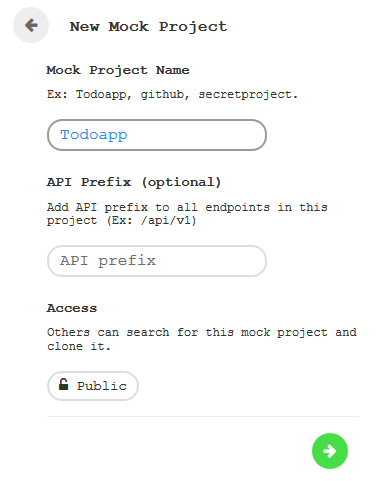
In a future lab we will make use of a real web service but for now use the free web API testing service at <http://www.mockapi.io/>

After creating an account (or signing in with your GitHub account..

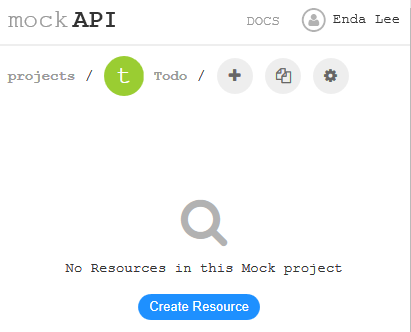
1. Click the **+** to add a new mock



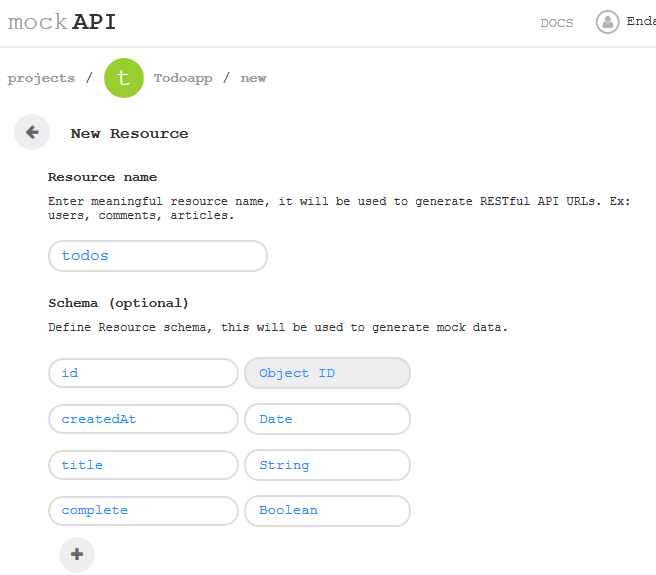
1. Name the project and continue



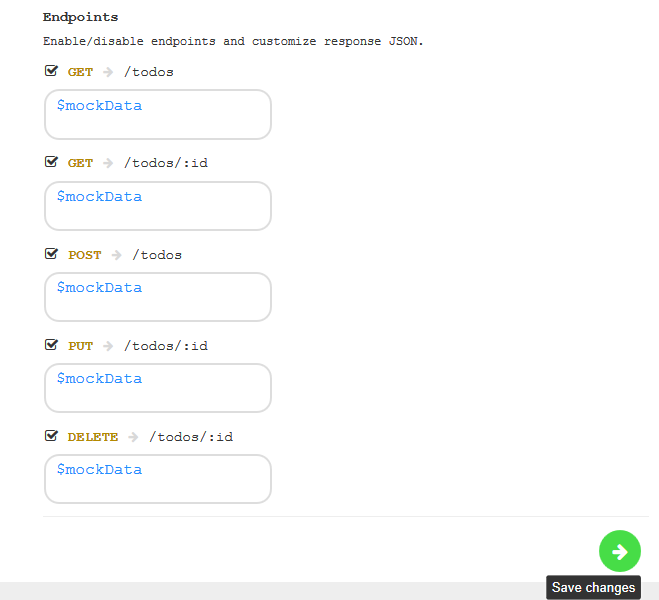
1. Create a Resource



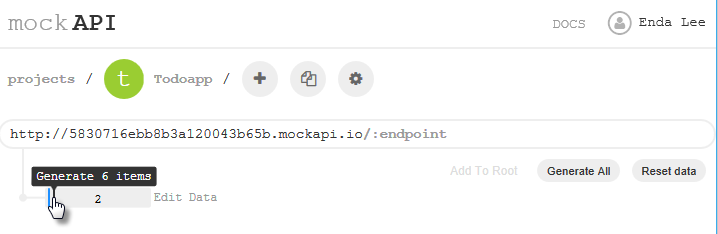
1. Setup the resource - name and a schema to match our Todo class (later)



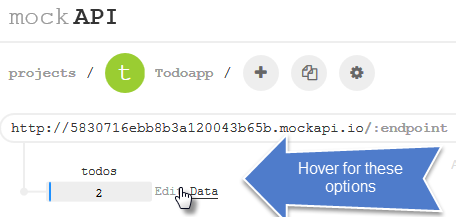
1. Note the endpoints and save changes



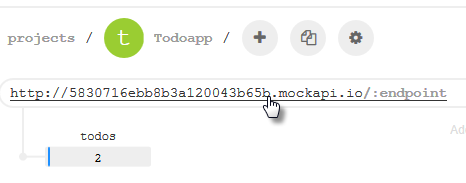
1. Generate some data (hover mouse over grey todos box, then click for number of items)

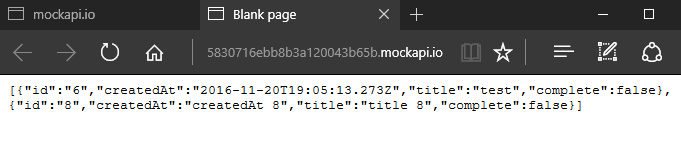


1. To Edit endpoints or data



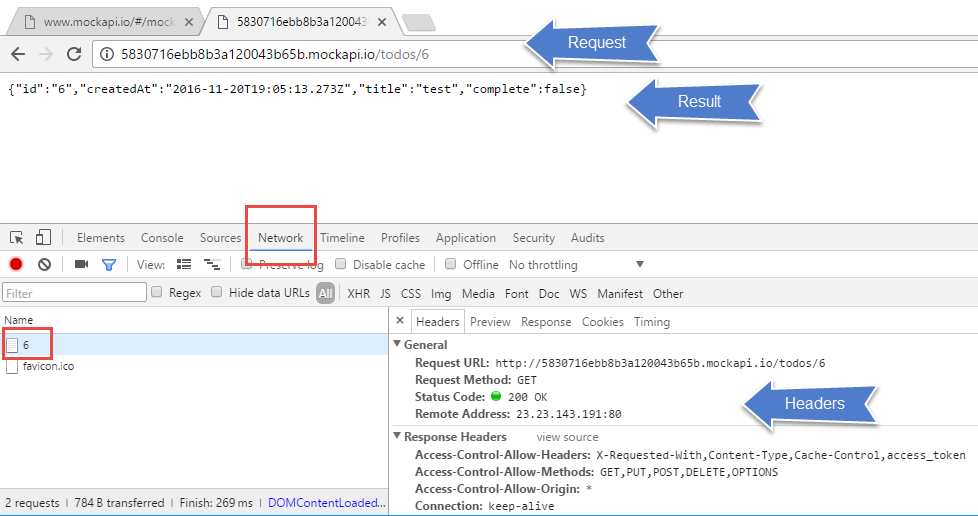
1. To see the raw data, click the URL

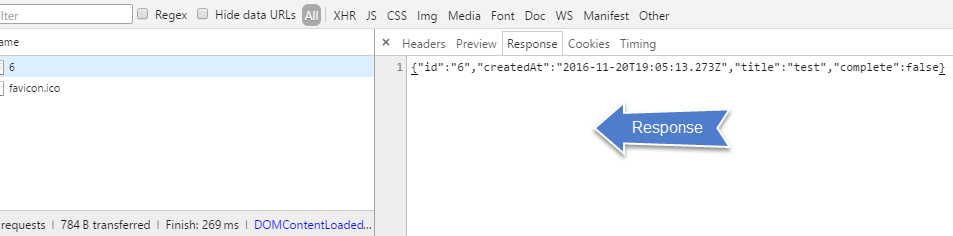


2. 

## 2. Testing and Debugging

Requests sent to a web service need to be well formed (i.e. be syntactically correct in both physically and logically). Browser developer tools can be very useful for diagnosing problems – particularly the network tools which allow the request and response to be inspected.

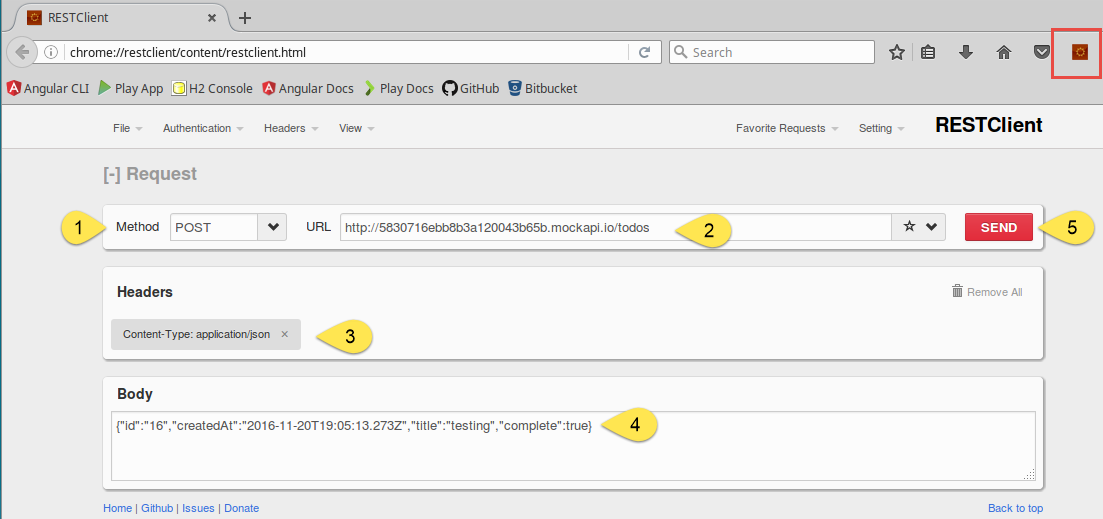




**2.1 Testing with Browser Plugins**

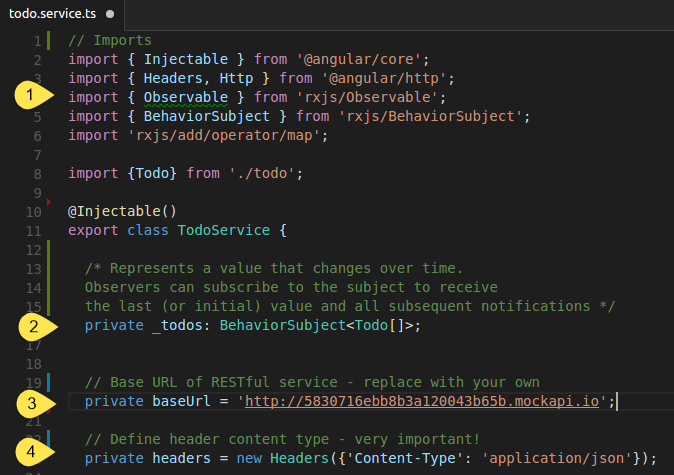
There are many browser plugins and extensions which can help test and diagnose issues, for example RESTClient for Firefox. This tool makes it easy to generate HTTP requests using any method (particularly useful for POST). Headers can easily be modified.

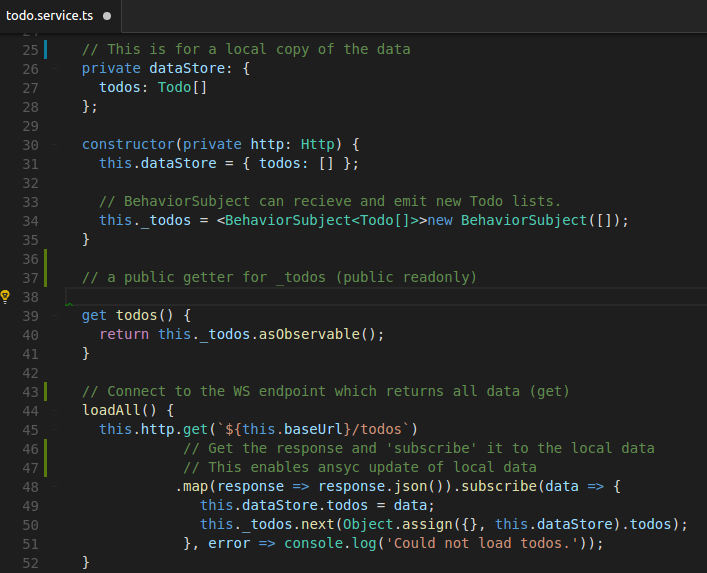
The following is an example of a POST request (1) for creating a new Todo item using a mockapi endpoint (2). This was very useful for diagnosing a problem in the application where POSTs were sent successfully but the server returned an error stating that the request body(4) was empty. It turned out that the wrong headers were being sent. The default content type is text/html but application/json was required – this needs to be explicitly set (3).

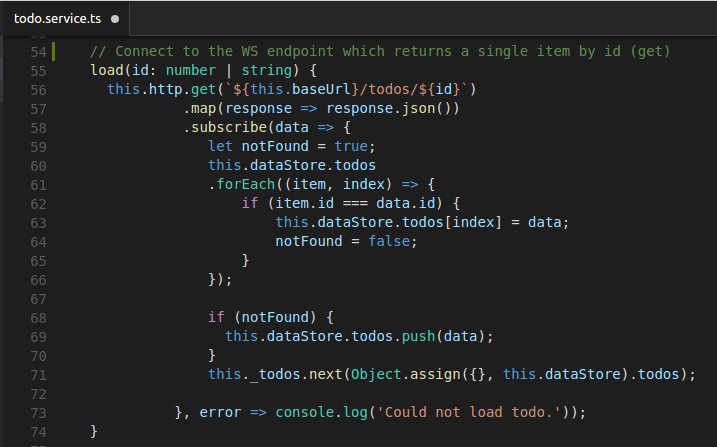


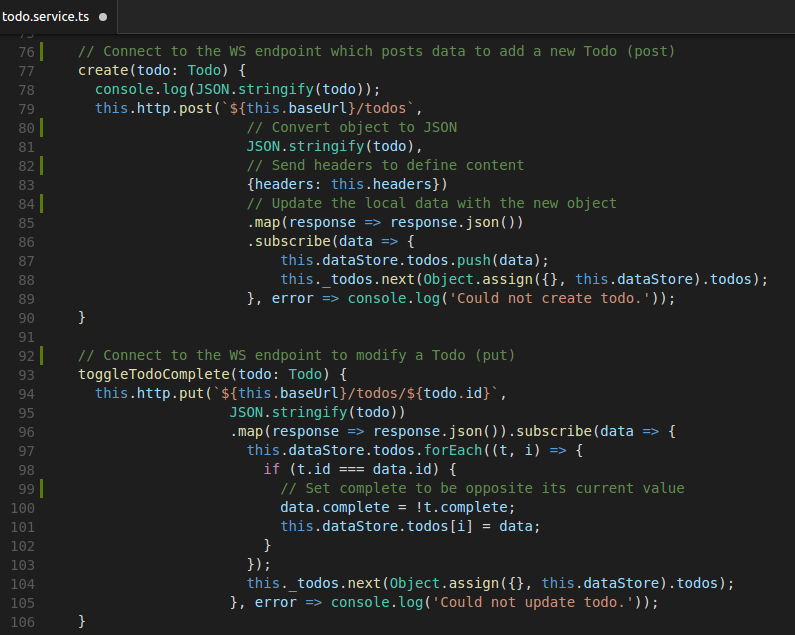
## 3. The ToDoService

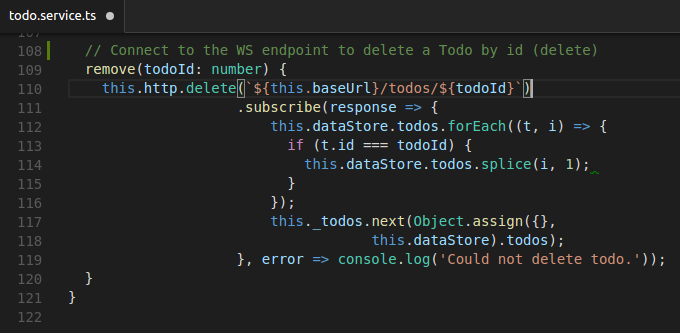
Read the comments. (3) Change the RESTful URL to your own -created using mock api











**src/app/todo.service.ts:**

// Imports

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

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

import { Observable } from 'rxjs/Observable';

import { BehaviorSubject } from 'rxjs/BehaviorSubject';

import 'rxjs/add/operator/map';

import {Todo} from './todo';

@Injectable()

export class TodoService {

/\* Represents a value that changes over time.

Observers can subscribe to the subject to receive

the last (or initial) value and all subsequent notifications \*/

private \_todos: BehaviorSubject<Todo[]>;

// Base URL of RESTful service - replace with your own

private baseUrl = '';

// Define header content type - very important!

private headers = new Headers({'Content-Type': 'application/json'});

// This is for a local copy of the data

private dataStore: {

todos: Todo[]

};

constructor(private http: Http) {

this.dataStore = { todos: [] };

// BehaviorSubject can recieve and emit new Todo lists.

this.\_todos = <BehaviorSubject<Todo[]>>new BehaviorSubject([]);

}

// a public getter for \_todos (public readonly)

get todos() {

return this.\_todos.asObservable();

}

// Connect to the WS endpoint which returns all data (get)

loadAll() {

this.http.get(`${this.baseUrl}/todos`)

// Get the response and 'subscribe' it to the local data

// This enables ansyc update of local data

.map(response => response.json()).subscribe(data => {

this.dataStore.todos = data;

this.\_todos.next(Object.assign({}, this.dataStore).todos);

}, error => console.log('Could not load todos.'));

}

// Connect to the WS endpoint which returns a single item by id (get)

load(id: number | string) {

this.http.get(`${this.baseUrl}/todos/${id}`)

.map(response => response.json())

.subscribe(data => {

let notFound = true;

this.dataStore.todos

.forEach((item, index) => {

if (item.id === data.id) {

this.dataStore.todos[index] = data;

notFound = false;

}

});

if (notFound) {

this.dataStore.todos.push(data);

}

this.\_todos.next(Object.assign({}, this.dataStore).todos);

}, error => console.log('Could not load todo.'));

}

// Connect to the WS endpoint which posts data to add a new Todo (post)

create(todo: Todo) {

console.log(JSON.stringify(todo));

this.http.post(`${this.baseUrl}/todos`,

// Convert object to JSON

JSON.stringify(todo),

// Send headers to define content

{headers: this.headers})

// Update the local data with the new object

.map(response => response.json())

.subscribe(data => {

this.dataStore.todos.push(data);

this.\_todos.next(Object.assign({}, this.dataStore).todos);

}, error => console.log('Could not create todo.'));

}

// Connect to the WS endpoint to modify a Todo (put)

toggleTodoComplete(todo: Todo) {

this.http.put(`${this.baseUrl}/todos/${todo.id}`,

JSON.stringify(todo))

.map(response => response.json()).subscribe(data => {

this.dataStore.todos.forEach((t, i) => {

if (t.id === data.id) {

// Set complete to be opposite its current value

data.complete = !t.complete;

this.dataStore.todos[i] = data;

}

});

this.\_todos.next(Object.assign({}, this.dataStore).todos);

}, error => console.log('Could not update todo.'));

}

// Connect to the WS endpoint to delete a Todo by id (delete)

remove(todoId: number) {

this.http.delete(`${this.baseUrl}/todos/${todoId}`)

.subscribe(response => {

this.dataStore.todos.forEach((t, i) => {

if (t.id === todoId) {

this.dataStore.todos.splice(i, 1);

}

});

this.\_todos.next(Object.assign({},

this.dataStore).todos);

}, error => console.log('Could not delete todo.'));

}

}

// Imports

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

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

import { Observable } from 'rxjs/Observable';

import { BehaviorSubject } from 'rxjs/BehaviorSubject';

import 'rxjs/add/operator/map';

import { Todo } from './todo';

@Injectable()

export class TodoService {

private \_todos: BehaviorSubject<Todo[]>;

private baseUrl = 'http://583dd1c195d29812004e4440.mockapi.io';

private headers = new Headers({ 'Content-Type': 'application/json' });

private dataStore: {

todos: Todo[]

};

constructor(private http: Http) {

this.\_todos = <BehaviorSubject<Todo[]>>new BehaviorSubject([]);

}

get todos() {

return this.\_todos.asObservable();

}

loadAll() {

this.http.get('${this.baseUrl}/todos')

.map(response => response.json()).subscribe(data => {

this.dataStore.todos = data;

this.\_todos.next(Object.assign({}, this.dataStore).todos);

}, error => console.log('Could not load todos.'));

}//end loadAll

load(id: number | string) {

this.http.get('${this.baseUrl}/todos/${id}')

.map(response => response.json())

.subscribe(data => {

let notFound = true;

this.dataStore.todos

.forEach((item, index) => {

if (item.id === data.id) {

this.dataStore.todos[index] = data;

notFound = false;

}

});

if (notFound) {

this.dataStore.todos.push(data);

}

this.\_todos.next(Object.assign({}, this.dataStore).todos);

}, error => console.log('Could not load todo.'));

}//end load

create(todo: Todo) {

console.log(JSON.stringify(todo));

this.http.post('${this.baseUrl}/todos',

JSON.stringify(todo),

{ headers: this.headers })

.map(response => response.json()).subscribe(data => {

this.dataStore.todos.forEach((t, i) => {

if (t.id === data.id) {

data.complete = !t.complete;

}

});

this.\_todos.next(Object.assign({}, this.dataStore).todos);

}, error => console.log('Could not update todo.'));

}//end create

remove(todoId: number) {

this.http.delete('${this.baseUrl}/todos/${todoId}')

.subscribe(response => {

this.dataStore.todos.forEach((t, i) => {

if (t.id === todoId) {

this.dataStore.todos.splice(i, 1);

}

});

this.\_todos.next(Object.assign({},

this.dataStore).todos);

}, error => console.log('Could not delete todo.'));

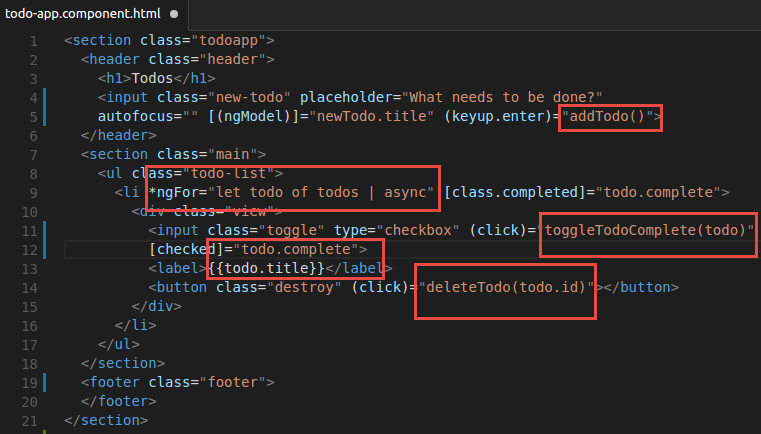
}//end remove

}//end TodoService

## 4. todo-app component

### 4.1 HTML template

This is mostly unchanged but note that the for loop is now working asynchronously (| async in the highlighted area). Also verify that methods and parameters are correct.



### 4.2 Component class

Similar to original but with added awareness of async behaviour

