Organizacija i upravljanje dokumentima

Glavan, Hrvoje

Undergraduate thesis / Završni rad

2018

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Algebra University College / Visoko učilište Algebra**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:225:397618

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-19**

Repository / Repozitorij:

Algebra Univerity College - Repository of Algebra Univerity College

VISOKO UČILIŠTE ALGEBRA

ZAVRŠNI RAD

Organizacija i upravljanje dokumentima

Hrvoje Glavan

Zagreb, ožujak 2018.

Student vlastoručno potpisuje Završni rad na prvoj stranici ispred Predgovora s datumom i oznakom mjesta završetka rada te naznakom:

„Pod punom odgovornošću pismeno potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature, ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor, te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada“.

U Zagrebu, datum.

Ime Prezime

Predgovor

Poštovani,

ovo je moj pokušaj izrade i realizacije većeg projekta s primjenom modernih metodologija i alata unutar danih vremenskih ograničenja prema jasnom cilju same kvalitete.

Vrlo sam zadovoljan sa odabranom temom jer mi daje mogućnosti igranja i istraživanja raznih zanimljivih tehnologija i pristupa izrade softvera. Također je područje koje mene osobno zanima jer može se puno korisnih stvari naučiti i siguran sam da bi velik dio ove aplikacije mogao biti ponovno iskorišten i za druge svrhe.

Prilikom uvezivanja rada, Umjesto ove stranice ne zaboravite umetnuti original potvrde o prihvaćanju teme završnog rada kojeg ste preuzeli u studentskoj referadi

Sažetak

Ovo je moje osobno putovanje i iskustvo u kreiranju malo većeg projekta ispočetka. Glavni ciljevi uključuju učenje i istraživanje raznih dostupnih opcija u domeni postavljanja i upravljanja projektima, kreiranja ugodnog radnog okruženja te procesa gradnje. To uključuje isprobavanje i rad sa nekolicinom popularnih JavaScript *framework*-ova skupa sa Java Spring pozadinom sve dok smo adaptabilni i spremni na promjene.

This is my own journey and experience of creating a slightly larger project from scratch. My biggest goal is to learn and explore as many options as possible regarding project initialization and workflow, setting up a pleasant development environment and build process. That includes trying out and working with a few popular JavaScript frameworks with Java Spring backend while being able to adapt and respond to change.

Ključne riječi: istraživanje, postavljanje, inicijalizacija, projekt, Java, JavaScript.

Sadržaj

1. Uvod 1

2. Početni dizajn i određivanje funkcionalnosti 2

2.1. Razrađivanje prvih koraka 2

2.2. Upravljanje projektima 3

2.3. Sustavi za upravljanje dokumentima 3

2.4. Ispitivanje dostupnih tehnologija prije početka izrade 3

2.5. Početni dizajn 4

2.6. Postavljanje okruženja za izradu 4

2.7. Java i Spring konfiguracija 8

2.8. Odvojeni prednji i stražnji slojevi 8

2.9. Heroku App 10

2.10. EditorConfig 10

2.11. Lombok 11

3. Izrada 12

3.1. Definiranje modela 12

3.2. Definiranje kontrolera 13

3.3. Repozitoriji za podatke 15

3.4. Statička HTML stranica 15

3.5. Prvi prikaz 16

3.6. Kontejner klasa 17

3.7. Prezentacijska klasa 19

4. Pogled na aplikaciju 22

4.1. Početna stranica 22

5. Izdvojeni detalji 23
5.1. Omogućavanje REST poziva 23
5.2. Premošćivanje API poziva 23
5.3. Kompresija na poslužitelju 24
Zaključak 25
Popis kratica 26
Popis slika 27
Popis tablica 28
Popis kôdova 29
Literatura 30
Prilog 31

1. Uvod

Glavna ideja i vizija ovog rada je napraviti, isprobati i istražiti više različitih područja i problematika kod izrade aplikacija. Nije cilj samo doći do brzog rješenja zadanog problema nego proći kroz više modernih pristupa izrade te samim tim otvoriti put prema skalabilnosti i lakšem održavanju aplikacije kroz životni tijek.

Uvijek se izvorni kod mora čuvati pod *source version control* sistemu, pisan što je bolje mogućim praksama pisanja dobrog izvornog koda te uvijek spreman za refaktoriranje i izmjene.

Kao izazov sam postavio sebi da ću ići dublje nego što je potrebno za izradu zadataka koji u suštini nije pretjerano kompliciran, ali tehnologija ispod je ono što ga čini zanimljivim. Pojavljivat će se razne metodologije, okviri, *build* procesi zaokruženi terminologijom i vrlo aktualnim pristupima izrade softvera i stalnog dostavljanja vrijednosti kroz automatske *deploy*-eve na *development* ili produkcijske poslužitelje.

Neki od istaknutijih korištenih jezika, metodologija i alata za izradu su Spring, React.js, Redux, Vue.js, REST klijenti, Intellij Ultimate, Visual Studio Code, …

Također cilj je nastojati što bolje pratiti i dokumentirati sve važnije izmjene tijekom izrade bilo vezane sa konfiguraciju, procese građenja (engl. *build process*) ili konkretna rješenja za određeni problem.

2. Početni dizajn i određivanje funkcionalnosti

Krećemo od vrlo jednostavne ideje: treba nam aplikacija za organiziranje vremena, dokumenata, rizika i potencijalno drugih resursa. Također ideja je primijeniti metodologije iz projektnog menadžmenta i vidjeti kako bi mogli ovakvu aplikaciju primijeniti u timskom okruženju. Jedna od takvih primjena je dijeljenje dokumenata unutar tima - potrebno je poslužiti se metodama projektnog menadžmenta u svrhu što kvalitetnijeg upravljanja projektnom dokumentacijom.

U biti ideja nije tako jednostavna jer što više zahtjeva postoji stvari postaju sve kompliciranije. Da bi ovakav pothvat bio uspješan trebalo bi razlomiti projekt u manje cjeline.

Za početak zacrtao sam sljedeće zahtjeve:

- Složiti funkcionalnu listu gdje se mogu dodavati novi zadaci
- Zadaci se mogu uređivati
- Zadatak može biti označen kao dovršen

To je vrlo osnovna i čvrsta baza od koje se može krenuti međutim između se još nalaze stotine manjih koraka. Nakon uspješnih prvih koraka će se raditi na dodatnoj funkcionalnosti.

2.1. Razrađivanje prvih koraka

Na konceptualnoj razini trenutačno imamo neku ideju što treba napraviti, ali ono što mene još više zanima je kako postaviti takav projekt, koji su tehnički zahtjevi i najbolji pristup ovakvom zadatku.

Svakako bi htio pratiti moderne trendove i istražiti više opcija i pristupa pa makar takvi pristupi nisu najbolji ili su previše komplicirani za ovakav zadatak. Samo ove osnovne korake se može riješiti na mali milijun načina i zasigurno postoji daleko jednostavnijih rješenja koji su efektivni ali nisu npr. skalabilni.

Idealno bi bilo imati aplikaciju koja nije teška za održavanje i koja se neće pretvoriti u neodrživi nered jednom kada se počnu dodavati nove funkcionalnosti.

2.2. Upravljanje projektima

Prilikom izrade projekta koristiti će se agilni pristup razvoju. Za ovaj projekt sam odabrao Gitlab kao platformu gdje ću pohranjivati izvorni kôd i simulirati agilno okruženje tako što će se pisati dokumentacija, otvarati novi zadaci s praćenjem problema uz organiziranu i ažuriranu ploču gdje će se moći dobiti osjećaj dodjeljivanja zadataka unutar tima kao i pisanje komentara, primjera programskog kôda i sličnih oblika komunikacije.

2.3. Sustavi za upravljanje dokumentima

Sustavi za upravljanje dokumentima su sustavi koji se koriste za praćenje, pohranu i raspolaganje dokumentima. Znatno smanjuju uporabu papira i postaju neophodni u industriji i projektnom menadžmentu.

Na tržištu već postoji velik broj kvalitetnih alata i sustava koji će zasigurno olakšati upravljanje dokumentima. Neki od istaknutijih su: Jira, Microsoft Teams, Asana, Basecamp, itd.

Dobar primjer kojeg sam pratio za izradu praktičnog dijela je Asana koja ima odlične to-do liste i neke dodatne značajke kao integracija s 100 različitih umetaka (engl. *plugin*), pregledavanje posla i zadataka na kalendaru te postavljanje prioriteta i ciljeva.

2.4. Ispitivanje dostupnih tehnologija prije početka izrade

Jedna od tipičnih opcija je izabrati Bootstrap i Jquery te krenuti sa razvojem iako se to skoro pa i ne može nazvati izbor jer trend je postao automatski uključivati te pakete u svaki projekt i dok bi bili savršeno validni i jednostavniji za koristiti za ovaj projekt odlučio sam izabrati nešto kompliciranije i ne toliko uobičajeno.

Diskutabilno je samo koliko je ne tipičan izbor s obzirom da React dolazi od Facebook-a i vrlo je popularan među modernim okvira (engl. *framework*) za izgradnju korisničkih sučelja (engl. *user interfaces*), ali se zato pokazuje kao jako dobar izbor jer podrška od strane zajednice (engl. *community*) je čvrsta i moguće je naći rješenja za probleme, ali i gotove komponente kao Semantic-UI-React koje sam koristio u projektu.

Druga pak opcija je bila Vue.js koji isto dobiva na popularnosti u zadnje vrijeme i dijeli dosta sličnosti sa prvom inačicom Angular-a u načinu korištenja. Treća opcija na

raspolaganju je isprobati kontroverzne alate i povući nekakve paralele u načinu korištenje jer oni ipak dijele sličnu funkciju.

Pored toga tu su industrijski standardi kao Spring i Java, Webpack bundler, Typescript i Semantic-UI.

2.5. Početni dizajn

Kao korisnu polaznu točku pronašao sam artikl među uputama za korištenje React-a dostupno na [1] https://reactjs.org/docs/thinking-in-react.html.

Jedna od najvažnijih točaka je upravo rastavljanje većih cjelina na komponente koje su u svojoj naravi jednostavnije za korištenje, implementaciju, održavanje ili možda još značajnije – ponovno su iskoristive i smanjuju ponovno pojavljivanje programskog koda (engl. *code duplication*).

Ono što je malo više specifično za React je definiranje gdje će „stanje“ aplikacije „živiti“ ili postojati jer kasnije kao što ćemo i sami saznati stvari se zakompliciraju jednom kada imamo više stanja za pratiti i više svojstva za prenositi među komponentama.

Da ne ulazim previše u React u ovom trenu za početak sve što bi htio je imati postavljen sustav spreman za izradu zadatka i iako postoje već neka rješenja kao **create-react-app** koji kreiraju *boilerplate code* krenuo sam na svoju ruku istraživati Webpack i moguće načine postavljanja projekta.

2.6. Postavljanje okruženja za izradu

Vrlo aktualan u ovom trenu je Webpack, nevjerojatno moćan alat koji osim što omogućava vruće izmjene (engl. *hot reloading*) i razne druge programerske pogodnosti za vrijeme izgradnje također služi za kreiranje produkcijske verzije aplikacije uz razne optimizacije kao smanjivanje programskog koda (engl. *minification*).

On kao takav ima samo funkciju skupljanja (engl. bundle) različitih modula, slika i korištenih resursa u jedan ili više izlaza i razumije samo čisti JavaScript međutim njegova moć leži u tome što pomoću umetka (engl. *plugin*) može obavljati više korisnih funkcija i razumjeti tj. prevoditi skripte pisane u Typescript-u kao jedan primjer.

Slijedi primjer Webpack konfiguracije korištene u projektu:

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const webpack = require('webpack');
const BundleAnalyzerPlugin = require('webpack-bundle-
analyzer').BundleAnalyzerPlugin;

module.exports = {
  entry: './src/main/js/index',
  plugins: [
    new webpack.IgnorePlugin(/^\.\/locale$/, /moment$/),
    new BundleAnalyzerPlugin({openAnalyzer: false, analyzerMode:
'static'})
  ],
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname,
'src/main/resources/static/assets'),
    publicPath: '/assets/'
  },
  resolve: {
    modules: [
      'node_modules', 'src/main/js'
    ],
    extensions: [
      '.tsx', '.ts', '.js', '.vue'
    ],
    alias: {
      'vue$': 'vue/dist/vue.esm.js'
    }
  },
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        include: path.resolve(__dirname, "src/main/js"),
        use: [
          {
            loader: 'babel-loader',
            options: {
              babelrc: true,
              cacheDirectory: true
```

```
            }
          },
          'awesome-typescript-loader'
        ],
        // exclude: '/node_modules/'
      }, {
        test: /\.vue$/,
        loader: 'vue-loader'
      }, {
        test: /\.s?css$/,
        use: ['style-loader', 'css-loader', 'sass-loader']
      }, {
        test: /\.(png|svg|jpg|gif)$/,
        use: [
          {
            loader: 'url-loader',
            options: {
              mimetype: 'image/jpg',
              limit: 8192
            }
          }
        ]
      }, {
        test: /\.(woff|woff2|eot|ttf|otf)$/,
        use: ['file-loader']
      }
    ]
  }
};
```

Kôd 2.1 Zajednička Webpack konfiguracija

Obratiti pažnju kako Webpack ima čvrsto definirate ulaze i izlaze te definirane *plugine* koji omogućavaju skupljanje raznih resursa korištenih u projektu i kreiranje konačnog izlaza.

Vrijedno je i napomenuti kako je ovo zajednička (engl. *common*) konfiguracija i pravila koja će se uvijek koristiti, a za konfiguraciju okruženja za vrijeme izrade i za produkciju koriste se dvije odvojene datoteke koje se ovisno o trenutno postavljenoj varijabli okruženja (engl. *environment variable*) spajaju u traženu konfiguraciju.

Ovako izgleda konfiguracija za okruženje za izradu (engl. *development environment*):

```
const merge = require('webpack-merge');
```

```
const common = require('./webpack.common.js');
const webpack = require('webpack');
const path = require('path');

module.exports = merge(common, {
  devtool: 'inline-source-map',
  devServer: {
    contentBase: './src/main/resources/static',
    hot: true,
    port: 9000,
    compress: true,
    historyApiFallback: true,
    proxy: {
      '/api': 'http://localhost:8080'
    }
  },
  plugins: [
    new webpack.HotModuleReplacementPlugin(),
    new webpack.NamedModulesPlugin()
  ]
});
```

Kôd 2.2 DevServer konfiguracija

Ono što je tu specifično je što u biti pokrećemo *development* server koji podržava vruće izmjene što znači da svaki put kad se nešto promijeni unutar dokumenata u projektu server će osvježiti (engl. *refresh*) izgradnju i automatski prikazati izmjene bez da mi manualno trebamo ponovno osvježavati internet pretraživač.

Za produkciju je priča malo drugačija:

```
const merge = require('webpack-merge');
const UglifyJsPlugin = require('uglifyjs-webpack-plugin');
const common = require('./webpack.common.js');
const webpack = require('webpack');
const CleanWebpackPlugin = require('clean-webpack-plugin');

module.exports = merge(common, {
  devtool: 'source-map',
  plugins: [
    new UglifyJsPlugin({
      sourceMap: true
    }),
```

```
    new webpack.DefinePlugin({
      'process.env.NODE_ENV': JSON.stringify('production')
    }),
    new CleanWebpackPlugin(['src/main/resources/static/assets'])
  ]
});
```

Kôd 2.3 Produkcijska Webpack konfiguracija

Zanimljivo kod ovoga je što zapravo uopće ne pokrećemo server nego `CleanWebpackPlugin` potpuno očisti direktorij u koji skupljamo izvorni kod te stvara novi produkcijski svežanj (engl. *production bundle*). `DefinePlugin` postavlja naše okruženje tj. varijablu na 'production' i `UglifyJsPlugin` radi velik broj optimizacija po zadanom (engl. *default*).

2.7. Java i Spring konfiguracija

Za Java konfiguraciju sam koristio Spring Boot koji odmah izvan kutije (engl. *out-of-the-box*) postavlja ogroman broj konfiguracijskih postavka automatski ovisno o tome što se nalazi na ***classpath***-u.

Kao primjer koristim zadani **application.properties** dokument za definiranje konekcijskih varijabli za bazu te zajedničkih postavki kao npr. Hibernate način generiranja tablica u bazi: `spring.jpa.hibernate.ddl-auto=update` koji će zadržati postojeće podatke i samo ažurirati tablice tj. pokušati ažurirati tablice jer za sve veće promjene čini se preporučljivo koristiti `create` ili `create-drop` koji generiraju čiste tablice sa pravilnom strukturom, ali brišu podatke.

Osim zajedničkih postavka koriste se na zadanoj putanji lokalne konfiguracije kao npr. application-local.properties s tim da moramo kod pokretanja aplikacije reći koji lokalni profil koristimo. Tu spremamo postavke specifične za lokalni razvojni stroj kao npr. konekcijska veza na različitu bazu podataka.

2.8. Odvojeni prednji i stražnji slojevi

Jedan bitan element kod ovog projekta mi je bio imati odvojene slojeve i izgradnju (engl. *build*) za *frontend* i *backend*. Na taj način oni ne ovisi jedan o drugom tj. moguće je raditi samo na izgledu aplikacije bez da ikad pokrenemo logiku i server za podatke u pozadini.

Lokalno sam to postigao tako da sam jednostavno pokretao svoj Webpack-dev-server za *frontend* na portu 9000, a Spring aplikaciju na portu 8080. Kasnije su bile potrebne dodatne konfiguracije radi ove specifičnosti.

Za produkcijski *build* integrirao sam *frontend* dio unutar Maven procesa izgradnje. Za to je bio potreban "Frontend Maven Plugin".

Ovo je primjer postavki koje sam koristio unutar build procesa definiranog unutar pom.xml:

```
<plugin>
        <groupId>com.github.eirslett</groupId>
        <artifactId>frontend-maven-plugin</artifactId>
        <version>1.6</version>

        <executions>

          <execution>
            <id>install node and npm</id>
            <goals>
              <goal>install-node-and-npm</goal>
            </goals>
            <configuration>
              <nodeVersion>v9.4.0</nodeVersion>
            </configuration>
          </execution>

          <execution>
            <id>npm install</id>
            <goals>
              <goal>npm</goal>
            </goals>
            <configuration>
              <arguments>install</arguments>
            </configuration>
          </execution>

          <execution>
            <id>npm run build</id>
            <goals>
              <goal>npm</goal>
            </goals>
            <configuration>
```

```
            <arguments>run build</arguments>
          </configuration>
        </execution>

      </executions>
    </plugin>
```

Kôd 2.4 Koraci za izgradnju unutar pom.xml

Ovaj umetak u biti pokreće Node komande koje bi ja inače ručno pisao kako bi dobio produkcijski *build* na lokalnoj mašini, ali ovo će biti izrazito bitno jednom kada budem htio postaviti tj. *deploy* aplikaciju na server.

2.9. Heroku App

Heroku je davatelj usluga (engl. *host*) koji sam odabrao za svrhe ovog projekta jer to nije tipični statički server već on sam pokreće čitavi *build* proces koji sam ja definirao te pokreće Java aplikaciju isto kao što bi ju ja pokretao lokalno na svom stroju.

Inače bi bilo potrebno konfigurirati korake koje će Heroku koristiti za izradnju aplikacije unutar «PROCFILE» dokumenta, ali u ovom slučaju čak nije bilo potrebno jer Heroku je automatski prepoznao Java aplikaciju i zna koje korake mora odraditi da pokrene i posluži aplikaciju.

2.10. EditorConfig

EditorConfig je manji *plugin* kojeg podržavaju svi poznatiji alati za pisanje koda (engl. *code editors*) te promovira konzistentnost kroz dokumente unutar projekta.

Pravila koja sam koristio za projekt su sljedeća:

```
root = true

[*]
charset = utf-8
indent_style = space
indent_size = 2
end_of_line = lf
insert_final_newline = true
trim_trailing_whitespace = true
```

Kôd 2.5 EditorConfig postavke

Vrlo jednostavno, učinkovito i svakako preporučljivo kod postavljanja okruženja za izradu bilo kakvog projekta.

2.11. Lombok

Lombok je vrlo koristan *plugin* za Javu koji smanjuje pisanje generičkog koda (engl. *boilerplate code*), ali i automatski se odrazi na sve promjene. Npr. ako imamo (a imamo ih u malo većem broju slučajeva) *getter*-e i *setter*-e ne moramo pisati metode koje obavljaju tu funkciju i s time je sama klasa preglednija. Možda još bitnije je što ako se nešto promijeni u modelu ne moramo mijenjati *getter* ili *setter* nego Lombok to odradi sam.

U projektu često koristim anotaciju `@Data` koja postavlja getter-e i setter-e za svako polje (engl. *field*) ili anotaciju `@NoArgsConstructor` koja jednostavno definira konstruktor za klasu bez ikakvih parametara.

3. Izrada

U ovom poglavlju velik dio pažnje se obraća na saznanja koja su došla tijekom same izrade konkretnog zadatka, mišljenja i usporedbe iskustava.

Glavni fokus stavljam na model i logiku iza REST poziva te React aplikaciju za korisničko sučelje. Kasnije povlačim paralelu sa nekim drugim iskustvima kao npr. korištenje Vue.js za postizanje praktički istog cilja.

3.1. Definiranje modela

Za prvi korak želim imati mali broj entiteta koje ću kasnije nadopuniti i povezati. Jedan takav je `TodoItem` model:

```
package io.praktikum.jupiter.model;

import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.*;
import java.util.Date;

@Entity
@NoArgsConstructor
@Data
public class TodoItem {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private long id;
  private String text;
  private String description;
  private boolean completed;

  @Temporal(TemporalType.TIMESTAMP)
  private Date created;

  public TodoItem(String text, String description, Date created) {
    this.text = text;
```

```
    this.description = description;
    this.completed = false;
    this.created = created;
  }
}
```

Kôd 3.1 TodoItem model

Koristimo anotaciju `@Entity` da označimo ovo kao model prema kojem će Hibernate izgraditi tablice. Također potrebno je za Hibernate definirate primarni ključ i identitet tablice sa `@Id` te `@GeneratedValue` koji automatski generira vrijednosti za svaki novi unos.

Osim tekstualnih polja koristi se i `boolean` polje koje će se kasnije koristiti u kutiji za označavanje (engl. *checkbox*) je li zadatak dovršen ili nije.

Za datum i vrijeme kada je ovaj zadatak kreiran potrebno je definirati polje pomoću `@Temporal` anotacije.

Ovaj entitet će biti dovoljan za prikazati jednostavnu listu elemenata na stranici.

3.2. Definiranje kontrolera

Ovako izgleda kontroler (engl. *controller*) koji koristimo u primjeru:

```
package io.praktikum.jupiter.controller;

import io.praktikum.jupiter.model.TodoItem;
import io.praktikum.jupiter.repository.TodoItemRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping(value = "/api/todos")
public class JupiterController {

  private TodoItemRepository jupiterRepository;

  @Autowired
  JupiterController(TodoItemRepository jupiterRepository) {
    this.jupiterRepository = jupiterRepository;
```

```
  }

  @GetMapping
  public Iterable<TodoItem> todos() {
    return jupiterRepository.findAllByOrderByCreated();
  }

  @RequestMapping(method = RequestMethod.POST)
  public ResponseEntity<TodoItem> add(@RequestBody TodoItem
todoItem) {
    TodoItem newItem = new TodoItem(todoItem.getText(),
todoItem.getDescription(), todoItem.getCreated());
    TodoItem saved = jupiterRepository.save(newItem);
    return new ResponseEntity<>(saved, HttpStatus.CREATED);
  }

  @RequestMapping(value = "/{id}", method = RequestMethod.DELETE)
  public Iterable<TodoItem> remove(@PathVariable Long id) {
    jupiterRepository.deleteById(id);
    return todos();
  }

  @RequestMapping(method = RequestMethod.PUT)
  public ResponseEntity<TodoItem> edit(@RequestBody TodoItem
todoItem) {
    return new ResponseEntity<>(jupiterRepository.save(todoItem),
HttpStatus.ACCEPTED);
  }
}
```

Kôd 3.2 Kontroler za TodoItem entitete

Prvo što treba primijetiti je da koristimo `@RestController` anotaciju koja u biti ne radi mnogo osim što automatski postavlja `@ResponseBody` na metode unutar kontrolera (što se apsolutno može i ručno dodati na metode ako se **ne** koristi `@RestController`).

Zanimljivo u ovom projektu je što uopće nisam trebao definirati klasični kontroler koji u biti **mora** vratiti statičku stranicu koju koristim da učitam JavaScript skripte nego je Spring Boot automatski pronašao na putanji dokument pod nazivom index.html te ga servira pozivom na korijen (engl. *root*) url putanju.

Same metode unutar kontrolera koriste REST servise i obavljaju standardne CRUD (engl. Create Retrieve Update Delete) operacije nad `TodoItem` modelom.

3.3. Repozitoriji za podatke

Sljedeće je primjer repozitorija za podatke:

```
package io.praktikum.jupiter.repository;

import io.praktikum.jupiter.model.TodoItem;
import org.springframework.data.repository.CrudRepository;
import
org.springframework.data.repository.PagingAndSortingRepository;

public interface TodoItemRepository extends
PagingAndSortingRepository<TodoItem, Long> {

  Iterable<TodoItem> findAllByOrderById();
  Iterable<TodoItem> findAllByOrderByCreated();
}
```

Kôd 3.3 Repozitorij za rad s bazom podataka

Vrlo malo kodiranja u biti jer `PagingAndSortingRepository` u ovom slučaju već sadrži većinu potrebnih operacija koje nam trebaju nad bazom.

Metode koje sam na vlastitu ruku definirao u biti sortiraju vraćene podatke jer sam htio prikazati elemente u određenom redu (što je uostalom moguće riješiti na više načina i pristupa).

3.4. Statička HTML stranica

Jedina .html datoteka koja nam je potrebna u ovom slučaju je:

```
<!DOCTYPE html>
<html lang="en-US">
<head>
  <meta charset="utf-8"/>
  <meta content="ie=edge" http-equiv="x-ua-compatible"/>
  <title>Jupiter</title>
  <meta content="width=device-width, initial-scale=1"
name="viewport"/>
```

```
  <link rel="stylesheet"
href="//cdnjs.cloudflare.com/ajax/libs/semantic-
ui/2.2.12/semantic.min.css"/>
</head>
<body>

  <div id="app"></div>

  <script src="assets/bundle.js"></script>
</body>
</html>
```

Kôd 3.4 Statička početna html stranica

Automatski se servira pozivom '/' putanje i ono bitno što radi je učitavanje skupljenih (engl. *bundled*) skripti koje je izbacio Webpack i na taj način učitava *frontend* aplikaciju.

Osim toga učitava *CSS* stilove potrebne za komponente koje React koristi iz Semantic UI.

3.5. Prvi prikaz

React je u drugu ruku jedna potpuno drukčija zvijer, skoro pa odvojeni svijet u usporedbi sa svim što sam opisao do sad. Koristi svoju vlastitu sintaksu kako bi opisao **izgled** i **logiku** aplikacije koja inače ima posebnu ekstenziju `.jsx` tj. u ovom slučaju `.tsx` jer koristim Typescript.

Ovako izgleda korijen React aplikacije:

```
import * as React from 'react';
import * as ReactDOM from 'react-dom';

import App from './containers/App';
import './index.css';

ReactDOM.render(
  <App />,
  document.getElementById('app') as HTMLElement
);
```

Kôd 3.5 Izvorna React skripta

Tu povlačimo (engl. *import*) uz pomoć Webpack-a module koji nam trebaju unutar skripte i ovaj izvorni služi funkciji da povuče App ovisnosti (engl. *dependencies*) i stilove za

aplikaciju te ih prikaže (engl. *render*) i zakači na element naziva 'app' na stranici. (na statičkoj stranici u prošlom poglavlju smo imali `div` *tag* sa identifikatorom 'app').

3.6. Kontejner klasa

```
import * as React from 'react';
import { Divider } from 'semantic-ui-react';

import TodoForm from 'components/TodoForm';
import TodoItemList from 'components/TodoItemList';
import InfoMessage from 'components/InfoMessage';
import IconHeader from 'components/IconHeader';

import axios, { AxiosInstance, AxiosResponse, AxiosRequestConfig }
from 'axios';
axios.defaults.baseURL = "api/todos";

interface State {
  todos: TodoItem[];
  showMessage: boolean;
  status: string;
  loader: boolean;
}

class Todos extends React.Component<{}, State> {

  constructor(props) {
    super(props);
    this.state = { todos: [], status: "", showMessage: false,
loader: true };

    this.removeHandler = this.removeHandler.bind(this);
    this.addHandler = this.addHandler.bind(this);
    this.editItemHandler = this.editItemHandler.bind(this);
  }

  removeHandler(id) {
    axios.delete("/" + id)
      .then((response: AxiosResponse<TodoItem[]>) => {
        this.setState({ todos: response.data });
```

```
      });
  }

  addHandler(e) {
    axios.post("/" , e)
      .then((response: AxiosResponse<TodoItem>) => {
        let todos = this.state.todos;
        todos.push(response.data);
        this.setState({ status: response.data.text, showMessage:
true });
        this.setState({ todos: todos });
    });
  }

  editItemHandler(e) {
    let todos = this.state.todos;
    let exists: TodoItem = todos.find(item => item.id === e.id);
    let idx = todos.indexOf(exists);
    todos[idx] = e;

    this.setState({ todos: todos })

    axios.put("/", e)
      .then((response: AxiosResponse<TodoItem>) => {
      })
  }

  componentDidMount() {
    axios.get('/')
      .then((response: AxiosResponse<TodoItem[]>) => {
        this.setState({ todos: response.data, loader: false });
      });
  }

  render() {
    return(
      <span>
        <IconHeader iconName="calendar outline"
cornerIconName="trophy" />
        <TodoForm action={this.addHandler} />
        <TodoItemList
```

```
            todos={this.state.todos}
            removeAction={this.removeHandler}
            updateAction={this.editItemHandler}
            loader={this.state.loader}
          />

          <InfoMessage
            message={this.state.status}
            hidden={!this.state.showMessage}
            action={() => this.setState({ showMessage: false })}
          />

          <Divider />
        </span>
      );
    }
}
export default Todos;
```

Kôd 3.6 Primjer kontejner klase za React

Tu stvari postaju malo kompliciranije iako ova React klasa ima prilično jednostavnu funkciju: Obaviti neku logiku (npr. dohvatiti podatke) i proslijediti ih svojem djetetu (engl. *child component*) kao svojstvo (engl. *property*).

Jako bitno napomenuti je da je ovo kontejner klasa, drugim riječima ova klasa nije zadužena za prikazivanje podataka nego za obavljanje logike i držanje podataka unutar svojeg stanja (engl. *state*). Za prikaz tog stanja su zadužena djeca tj. obične komponente koje idealno ne bi trebale sadržavati vlastito stanje.

Za više o ovoj praksi pročitati: [2] https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0.

3.7. Prezentacijska klasa

```
import * as React from 'react';
import { List, Icon, Button, Checkbox, CheckboxProps, Dimmer,
Segment, Loader } from "semantic-ui-react";
import moment from 'moment';
```

```
import { TodoItemRemoveButton } from
'components/TodoItemRemoveButton';
import ItemEditModal from 'containers/ItemEditModal';

interface Props {
  todos: TodoItem[];
  removeAction?: (id, e) => void;
  updateAction?: ((todoItem: TodoItem, e) => void);
  loader: boolean;
}

class TodoItemList extends React.Component<Props, {}> {

  checkboxHandler(e: React.FormEvent<HTMLInputElement>, data:
CheckboxProps, todo: TodoItem) {
    todo.completed = data.checked;

    return this.props.updateAction(todo, e);
  }

  render() {
    return(
      <List>
        <Loader active={this.props.loader} inline='centered' />
        {
          this.props.todos.map((todo: TodoItem) =>

          <List.Item key={todo.id}>
              <List.Content>
                <List.Header>

                  <Checkbox id={todo.id} checked={todo.completed}
label={todo.text}
                    onChange={
                      (e, data) => this.checkboxHandler(e, data,
todo)
                    }
                  />

                    
```

```
                    <ItemEditModal
                      todoItem={todo}
                      updateAction={this.props.updateAction}
                      removeAction={this.props.removeAction}
                    />

                    <TodoItemRemoveButton id={todo.id}
action={this.props.removeAction} />

                  </List.Header>
                  <List.Description>
                    {moment(todo.created).format("dddd, MMMM Do YYYY,
h:mm:ss a")}
                  </List.Description>
                </List.Content>
              </List.Item>
            )
          }
        </List>
      );
    }
}

export default TodoItemList;
```

Kôd 3.7 Primjer prezentacijske klase za React

Ovo je *child* klasa kojoj naš kontejner prosljeđuje podatke. Kao što se vidi u primjeru klasa ne sadrži vlastito stanje te je zaduženja sa prikaz proslijeđenih svojstava.

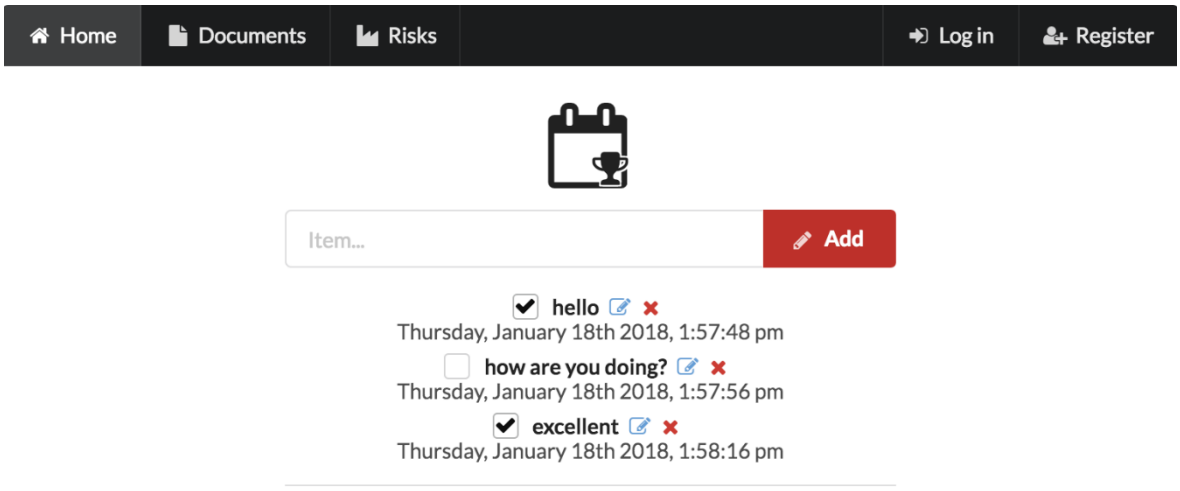
Koristi `map` funkciju nad poljem kako bi svaki `TodoItem` mapirala i prikazala kao zasebni element liste.

Neki od elemenata za prikaz dolaze iz Semantic-UI-React zavisnosti i za prikaz vremena u boljem obliku koristi se Moment.js.

4. Pogled na aplikaciju

4.1. Početna stranica

U dokument prilažem izgleda početne stranice (engl. *homepage*) gdje se nalazi `TodoItem` lista opisivana u radu.



Slika 4.1 Početna stranica

5. Izdvojeni detalji

5.1. Omogućavanje REST poziva

Da bi ovi primjeri funkcionirali potrebno je bilo obaviti veći broj međukoraka. Jedan od njih je bio omogućavanje poziva sa različitih odredišta (engl. *Cross-Origin Resource Sharing*).

To proizlazi iz činjenice da u *developer* okruženju vrtimo dva različita servera na dva različita utora (engl. *port*).

Primjer konfiguracije:

```
@Configuration
public class WebMvcConfig implements WebMvcConfigurer {
  @Override
  public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/api/**")
      .allowedOrigins("http://localhost:9000")
      .allowedMethods("GET", "POST", "DELETE", "PUT");
  }
```

Kôd 5.1 Omogućavanje CORS poziva

Osim omogućavanja poziva sa lokalne adrese različitog utora potrebno je i omogućiti obavljanje metoda koje inače nisu dozvoljene po zadanom npr. `DELETE` metoda.

5.2. Premošćivanje API poziva

Zanimljiv problem koji se javio kraj ovakvog pristupa je što u produkciji ne koristimo 2 zasebna server-a nego sve ide preko Tomcat poslužitelja. Znači da za REST pozive neće bit potrebno zvati različitu adresu kao kod okruženja za izradu.

Taj problem sam riješio prvo ovako:

```
declare var process : {
  env: {
    NODE_ENV: string
  }
}

axios.defaults.baseURL = "api/todos";
```

```
if (process.env.NODE_ENV !== "production") {
  axios.defaults.baseURL = "http://localhost:8080/api/todos";
}
```

Kôd 5.2 Primjer rješenja za pozive s različite adrese

Deklarirao sam provjeru za radno okruženje i u slučaju da nismo u produkciji zovemo različitu adresu za API pozive.

Nije bilo baš lijepo rješenje jer svaki kontejner koji koristi Axios logiku za podatke bi morao raditi sličnu provjeru. Umjesto toga moguće je bilo definirati premošćivanje (engl. *proxy*) na bazi Webpack *dev server*-a:

```
devServer: {
  contentBase: './src/main/resources/static',
  hot: true,
  port: 9000,
  compress: true,
  historyApiFallback: true,
  proxy: {
    '/api': 'http://localhost:8080'
  }
}
```

Kôd 5.3 Bolji način za riješiti pozive s različitih adresa

S ovom `proxy` postavkom svi pozivi prema `'/api'` putanji **isključivo** dok koristimo `devServer` će ići na postavljenu adresu.

5.3. Kompresija na poslužitelju

Da se ne troše ogromne količine podataka (engl. *bandwidth*) pod application.properties bilo je vrijedno definirati `server.compression.enabled=true` kako bi Tomcat kod posluživanja koristio metode kompresije sadržaja (engl. *Content-Encoding*).

Npr. ako bi izvorna veličina glavne JavaScript skripte bila oko 600kb, sa gzip kompresijom bi se efektivno spustila na ~130 kb.

Zaključak

Kao jedno od najkorisnijih i najzanimljivijih iskustava je bilo upravo postavljanje ovakvog projekta. Ne samo da su se stvari zakomplicirale povezivanjem više različitih tehnologija u cjelinu, ali u konačnici su stvorile jedno jako ugodno iskustvo za radno okruženje u kakvom bi htio idealno graditi projekt.

Ljepota kod takvog pristupa je što velik dio programskog kod-a ponovno iskoristiv i sadrži temelje za postavljanje novog projekta s možda potpuno različitim idejama.

Sama funkcija aplikacije na kraju možda nije bila pretjerano originalna koliko bi htio, ali omogućila je puno istraživanja i učenja npr. React okvira ili Redux metodologije uz Spring *backend*.

Osim toga vidio sam koliko vremenska ograničenja utječu na projekt i da nisam napisao testova koliko bi idealno bilo poželjno pri izradi projekta. Ali sa novo dobivenim znanjima već se osjećam sigurniji pri stvaranju ovakvog projekta, pogotovo što se tiče nekih stvari koje su bile iznimno zahtjevne u samom početku kao npr. rad sa nekolicinom kompliciranijih React klasa uz korištenje i učenje Typescript-a.

Također sam zadovoljan s iskustvom oko postavljanja ovakve aplikacije na živi poslužitelj koji automatski vrti proces izgradnje (engl. *build process*) prema repozitoriju izvornog koda gdje sam redovito čuvao sve promjene.

U konačnici imam jednu vrlo prilagodljivu aplikaciju koja možda nema ogroman broj mogućnosti kao npr. Asana, ali ima čvrste temelje i esencijalne funkcije za lakšu i bolju organizaciju i upravljanje dokumentima.

Popis kratica

API	*Application Programming Interface*	aplikacijsko programibilno sučelje
REST	*REpresentational State Transfer*	reprezentacijski prijenos stanja
CORS	*Cross-Origin Resource Sharing*	dijeljenje resursa s različitih odredišta

Popis slika

Slika 1 Početna stranica 22

Popis tablica

Popis kôdova

Kôd 2.1 Zajednička Webpack konfiguracija 6

Kôd 2.2 DevServer konfiguracija 7

Kôd 2.3 Produkcijska Webpack konfiguracija 8

Kôd 2.4 Koraci za izgradnju unutar pom.xml 10

Kôd 2.5 EditorConfig postavke 11

Kôd 3.1 TodoItem model 13

Kôd 3.2 Kontroler za TodoItem entitete 14

Kôd 3.3 Repozitorij za rad s bazom podataka 15

Kôd 3.4 Statička početna html stranica 16

Kôd 3.5 Izvorna React skripta 16

Kôd 3.6 Primjer kontejner klase za React 19

Kôd 3.7 Primjer prezentacijske klase za React 21

Kôd 5.1 Omogućavanje CORS poziva 23

Kôd 5.2 Primjer rješenja za pozive s različite adrese 24

Kôd 5.3 Bolji način za riješiti pozive s različitih adresa 24

Literatura

[1] THINKING IN REACT. https://reactjs.org/docs/thinking-in-react.html, 19.1.2018.

[2] PRESENTATIONAL AND CONTAINER COMPONENTS, https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0 , 26.1.2018.

Prilog

Završni rad može imati priloge, ali se oni ne prilažu uz pisanu verziju završnog rada već se mogu priložiti na završnom ispitu ukoliko povjerenstvo na završnom ispitu tako odluči. Važno je čuvati svu poratnu dokumentaciju koja je nastala pri izradi završnog rada.

S unutarnje strane na zadnjim koricama originala, kao i svake kopije završnog rada, pričvršćuje se CD s kompletnim završnim radom u izvornom formatu (npr. .doc) i .pdf formatu sa svom popratnom dokumentacijom i programima. Pri čemu je obvezno da na tom CD- u postoji i dokument koji opisuje kako se rezultat njegova diplomskog rada (softver ili hardver) koristi (ili kako se npr. izvode mjerenja koja je opisao u radu). Ako se radi o softveru nužno je opisati i kako se programska podrška instalira.

Algebra
visoka škola za
primijenjeno računarstvo

Organizacija i upravljanje dokumentima

Pristupnik: Hrvoje Glavan, 0246027570

Mentor: Prof. Marko Šimac