Razvoj poslužiteljskih komponenti aplikacije za upravljanje rasporedom

Šturlan, Adriana

Undergraduate thesis / Završni rad

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Pula / Sveučilište Jurja Dobrile u Puli**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:137:022312

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-25**

Repository / Repozitorij:

Digital Repository Juraj Dobrila University of Pula

Sveučilište Jurja Dobrile u Puli

Fakultet informatike

Adriana Šturlan

RAZVOJ POSLUŽITELJSKIH KOMPONENTI APLIKACIJE ZA UPRAVLJANJE RASPOREDOM

Završni rad

Pula, rujan 2019. godine

Sveučilište Jurja Dobrile u Puli

Fakultet informatike

Adriana Šturlan

RAZVOJ POSLUŽITELJSKIH KOMPONENTI APLIKACIJE ZA UPRAVLJANJE RASPOREDOM

Završni rad

JMBAG: 0303061260, redoviti student

Studijski smjer: Sveučilišni preddiplomski studij informatike

Kolegij: Programsko inženjerstvo

Znanstveno područje: Društvene znanosti

Znanstveno polje: Informacijske i komunikacijske znanosti

Znanstvena grana: Informacijski sustavi i informatologija

Mentor: doc. dr. sc. Tihomir Orehovački

Pula, rujan 2019. godine

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisana, Adriana Šturlan, kandidat za prvostupnika informatike ovime izjavljujem da je ovaj završni rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio završnog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, rujan 2019. godine

IZJAVA

o korištenju autorskog djela

Ja, Adriana Šturlan, dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj završni rad pod nazivom „Razvoj poslužiteljskih komponenti aplikacije za upravljanje rasporedom“ koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, rujan 2019. godine

Potpis

_____

SAŽETAK

Ovaj rad dokumentira proces izgradnje poslužiteljskog dijela web aplikacije za upravljanje rasporedima na fakultetu, počevši od arhitekturnog dizajna koji opisuje postavljene zahtjeve na sustav te izgled korisničkog sučelja aplikacije. Nakon toga slijedi opis interne strukture po uzoru na model klijent-poslužitelj, uvid u funkcionalne aspekte i uloge klijenta i poslužitelja, definicija komunikacijskih kanala RESTful servisa te logički dizajn baze podataka. Predstavljanjem korištenih tehnologija, čitatelj se uvodi u proces fizičke implementacije poslužitelja te projektnu strukturu dok se u narednim poglavljima opisuje cjelokupna izrada projekta kroz faze počevši od konfiguracije poslužitelja, preko implementacije baze podataka i definiranja modela, izrade same poslužiteljske aplikacije i definiranja servisa, zaključno sa uvidom u dodatno korištene posredničke tehnologije.

Ključne riječi: *raspored sati, web aplikacija, Node.js, Express.js, RESTful, debeli klijent, PostgreSQL, Sequelize, Docker, JWT, Express-validator, Passport*

ABSTRACT

The contents of this paper cover the process of building server-side components of a web-based application for managing schedules in universities, starting with an architectural design which describes the requirements set on the system and the layout of the application's user interface.This is followed by an overview of the internal structure based on the client-server model, insights into the functional aspects and roles of the client and server respectively, the definition of restful based communication channels, and the logical design of the database. By introducing the utilized technologies, the reader is familiarized with the process of physical server implementation which provides insights into the structure and overall design of the project through phases, starting with the server configuration, then going through the implementation of the physical database, the definition of models and server application endpoints, concluding with insight into the used middleware technologies.

Keywords: *timetable, web application, Node.js, Express.js, RESTful, fat client, PostgreSQL, Sequelize, Docker, JWT, Express-validator, Passport*

SADRŽAJ

1. UVOD 1

2. ARHITEKTURA APLIKACIJE 2

2.1. ZAHTJEVI NA SUSTAV 2

2.2. FUNKCIONALNOSTI APLIKACIJE I KORISNIČKO SUČELJE 3

3. LOGIČKI DIZAJN APLIKACIJE 8

3.1. MODEL KLIJENT-POSLUŽITELJ 9

3.1.1. FUNKCIJE KLIJENTA 9

3.1.2. FUNKCIJE POSLUŽITELJA 9

3.1.3. REST SERVISI 10

3.2. DIZAJN BAZE PODATAKA 11

4. KORIŠTENE TEHNOLOGIJE 13

5. FIZIČKI DIZAJN APLIKACIJE 15

5.1. POSTAVLJANJE PROJEKTA I PROJEKTNA STRUKTURA 15

5.2. KONFIGURACIJSKE DATOTEKE I SERVER.JS 17

5.3. BAZA PODATAKA POSTGRESQL 19

5.3.1. VIRTUALNA OKOLINA DOCKER 19

5.3.2. POSTGRESQL KONEKCIJA I SEQUELIZE 20

5.3.3. SEQUELIZE MODELI 21

5.4. EXPRESS.JS APLIKACIJA 22

5.4.1. API 24

5.4.1.1. ZAHTJEV TIPA GET 24

5.4.1.2. ZAHTJEV TIPA DELETE 25

5.4.1.3. ZAHTJEV TIPA POST 26

5.4.2. POSREDNIK EXPRESS-VALIDATOR 27

5.4.3. POSREDNIK PASSPORT 28

ZAKLJUČAK 31

POPIS LITERATURE 32

1. UVOD

Raspored sati jedan od klasičnih modela upravljanja vremenom korišten u velikom broju poslovanja te obrazovnih ustanova, kod kojeg se općenito javlja optimizacijski problem alociranja resursa u vremenske jedinice uz uvjet zadovoljavanja postavljenih ograničenja. Problem optimizacije proizlazi iz velike količine nezavisnih varijabli te specifičnosti područja uporabe, tako primjerice u slučaju rasporeda u obrazovnoj domeni, problemi se javljaju kada uključimo varijable dostupnosti dvorana ili profesora u određeno vrijeme odnosno kapacitet dvorana i broj upisanih studenata. Zbog velikog broja kombinacija i mogućih rješenja, problem rasporeda generalno slovi kao NP-težak problem jer se optimalno rješenje ne može pronaći u polinomijalnom vremenu.

Kako je u današnje vrijeme neizbježno imati takav efikasan sustav, *Timetable Maker* (u daljnjem tekstu *TTM*) nudi svojevrsno rješenje za potporu izradi semestralnog rasporeda sati na primjeru fakulteta, a povodi se metodom lakoće pristupa, jednostavnosti korištenja te brzog procesa izrade u suprotnosti velikom broju već gotovih rješenja čiji su glavni nedostaci komplicirano korisničko sučelje, odužen period prilagodbe na aplikaciju te zamoran proces izrade rasporeda. TTM aplikacija sadrži potrebne funkcionalnosti za asistenciju u izradi rasporeda sati, počevši sa vizualnom reprezentacijom korisnički kreiranih rasporeda, metodom popunjavanja rasporeda popraćenom vizualnim asistencijama, pogledima na raspored po semestrima i po pojedinim profesorima te druge funkcionalnosti koje će biti podobnije opisane u poglavlju 5. Aplikacija je kreirana je s ciljem djelomične automatizacije cjelokupnog procesa izrade rasporeda te optimizacije navedene problematike kreiranja rasporeda na fakultetima gdje se nastava izvodi semestralno.

Ovaj rad dokumentira poslužiteljsku stranu aplikacije TTM, opisujući pritom korištene poslužiteljske tehnologije, dizajn, implementaciju i razvoj aplikacije, konstrukciju relacijske baze podataka uz odabranu ORM tehnologiju te dodatne tehnologije potrebne za adekvatnu realizaciju svih zahtjeva na sustav.

2. ARHITEKTURA APLIKACIJE

TTM aplikacija dizajnirana je sa ciljem lakog upravljanja sustavom rasporeda i kreiranja istih, uzimajući u obzir mane drugih sustava ponuđenih na tržištu. Glavna namjena aplikacije je vizualnom asistencijom pružiti potporu u izradi rasporeda te jednostavnim korisničkim sučeljem pružiti ugodno iskustvo korištenja.

2.1. ZAHTJEVI NA SUSTAV

Mogućnosti aplikacije proizlaze iz postavljanih zahtjeva na sustav koje uključuju sljedeće funkcionalnosti:

- Prijava u sustav
- Pregled kreiranih godišnjih rasporeda
- Dodavanje novog godišnjeg rasporeda
- Brisanje godišnjih rasporeda
- Aktiviranje godišnjeg rasporeda u svrhu uređivanja
- Uređivanje godišnjeg rasporeda

Uređivanje godišnjih rasporeda kao glavni aspekt aplikacije pruža mogućnost uređivanja pojedinih rasporeda unutar aktivnog godišnjeg rasporeda odabirom pristupa studiranju (redovni/izvanredni), odabirom razine studija (preddiplomski/diplomski) te odabirom semestra, čije dostupne vrijednosti ovise o tome pripada li godišnji raspored zimskom ili ljetnom semestru. Odabirom željene kombinacije pristupamo listi predmeta, odvojenih po kategorijama predavanja, vježbi i seminara, koje je potrebno, uz odabir željene dvorane, smještati u raspored sve dok se zadana semestralna satnica sve tri kategorije ne svede na nulu odnosno dok nije zadovoljen kriterij integriteta.

Zahtjevi na sustav modula za uređivanje godišnjih rasporeda primjenjuju se na pojedinačne vremenske jedinice (polja) unutar rasporeda, a oni su sljedeći:

- Vizualni prikaz popunjenih polja i pripadajućih detalja
- Vizualno označavanje polja nedostupnih za popunjavanje trenutnom kombinacijom odabranog predavanja i dvorane
- Vizualno označavanje polja dostupnih za popunjavanje trenutnom kombinacijom odabranog predavanja i dvorane

- Oslobađanje popunjenih polja od zauzeća
- Premještanje dodanih predavanja na druga dostupna polja

Uz navedene zahtjeve na sustav, u obzir treba uzeti i potrebu prikazivanja presjeka dostupnosti polja kroz sve rasporede unutar godišnjeg rasporeda kako bi se spriječio zauzimanje dvorane već zauzete na nekom drugom rasporedu u to vrijeme ili zauzimanje polja u čijoj je vremenskoj oznaci označeni profesor već zauzet.

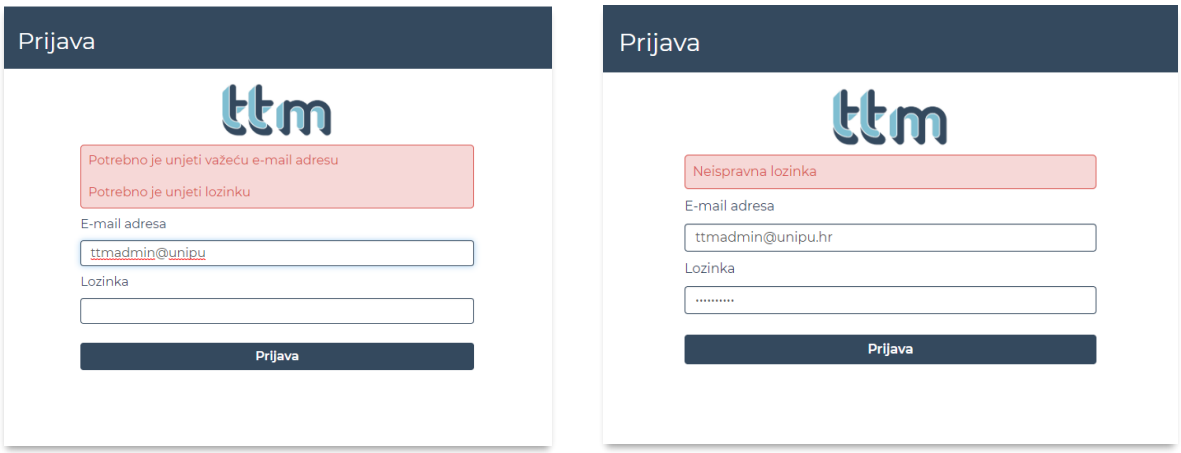
2.2. FUNKCIONALNOSTI APLIKACIJE I KORISNIČKO SUČELJE

Rad pod nazivom „Razvoj klijentskih komponenti aplikacije za upravljanje rasporedom sati 'TTM' " (Vučković, 2019) opisuje implementaciju klijentskog dijela aplikacije TTM u vidu razvoja korisničkog sučelja i funkcionalnosti aplikacije. Ovo poglavlje u sažeto opisuje dio tematike navedenog komplementarnog rada čime nastoji pružiti cjelokupan uvid u izgradnju aplikacije.

Pri ulasku u aplikaciju korisnik se prvo susreće sa zaslonom za prijavu u sustav. Prijava u aplikaciju vrši se putem administratorskog e-maila i lozinke dobivene prije početka rada u sustavu. S obzirom da je aplikacija zaštićena od neovlaštenog pristupa, nije moguće nastaviti s radom ukoliko se korisnik uspješno ne prijavi u sustav. Slika 1. prikazuje zaslon gdje se unose dobiveni korisnički podaci.

Slika 1. Zaslon za prijavu u aplikaciju

U slučaju unosa pogrešnih podataka za prijavu, sustav sukladno tome obavještava korisnika (slika 2.).

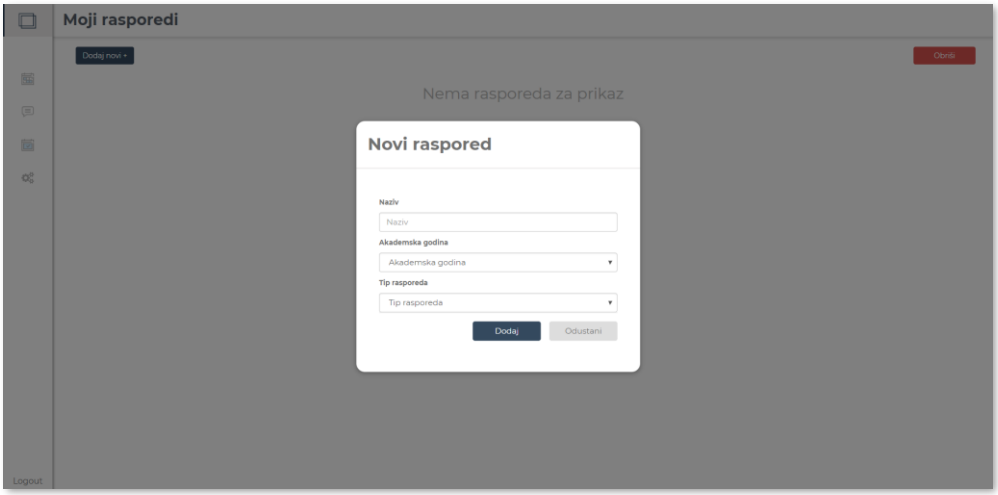


Slika 2. Obavještajne poruke na zaslonu za prijavu

Nakon uspješne prijave u sustav aplikacija se generalno može podijeliti na dva modula: Upravljanje godišnjim rasporedima i uređivanje godišnjih rasporeda. Sljedeća slika prikazuje korisničko sučelje modula za upravljanje godišnjim rasporedima.

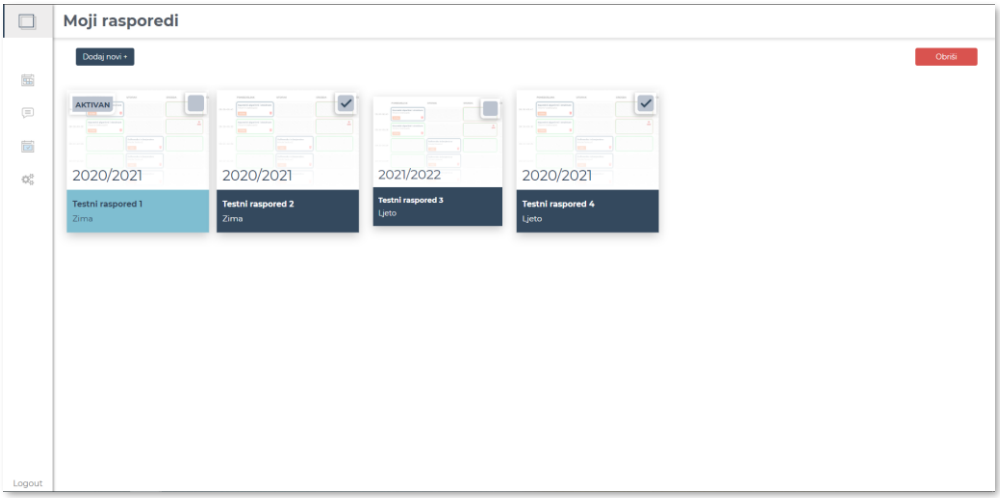
Slika 3. Modul za upravljanje godišnjim rasporedima

Modul za upravljanje godišnjim rasporedima direktno je vezan za uređivanje rasporeda jer se zbog logike aplikacije ne može pristupiti modulu za uređivanje prije nego li se aktivira jedan od godišnjih rasporeda. Novi godišnji rasporedi mogu se dodavati klikom na gumb *Dodaj novi* u gornjem lijevom kutu, pri čemu se otvara prozor za dodavanje novog rasporeda, kao što je prikazano na slici 4.



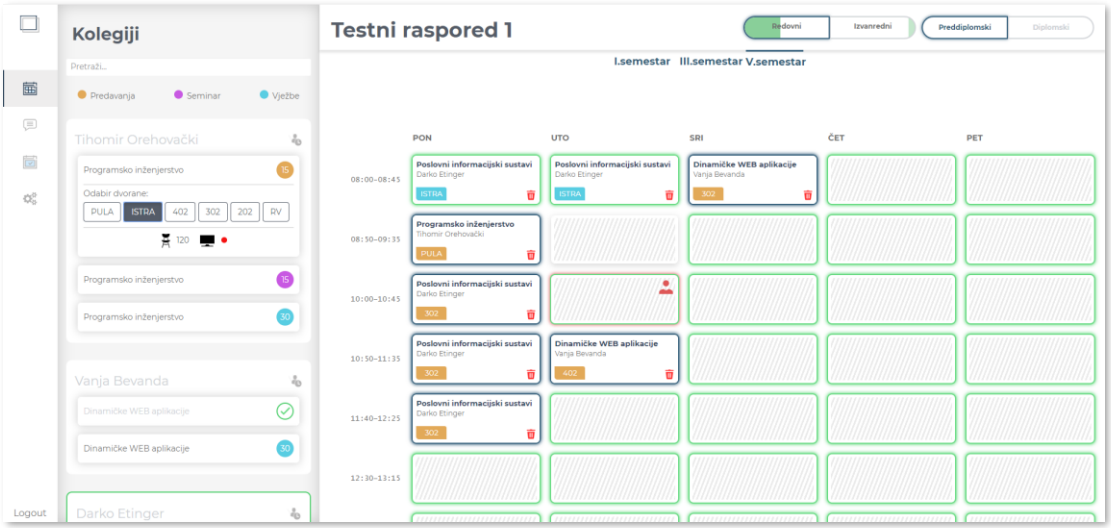
Slika 4. Dodavanje novog godišnjeg rasporeda

Dodavanjem novih godišnjih rasporeda otvara se mogućnost aktiviranja rasporeda te odabir pojedinih rasporeda za potrebe brisanja (slika 5.).



Slika 5. Mogućnosti modula za upravljanje godišnjim rasporedima

Odabirom ikone rasporeda na glavnom izborniku pristupamo modulu za uređivanje godišnjih rasporeda. Ovaj modul obilježava lak pristup svim rasporedima unutar godišnjeg rasporeda pomoću navigacije u gornjem desnom kutu aplikacije, čijom se kombinacijom prikazuje željeni raspored u modu za uređivanje. Ovisno o odabranoj kombinaciji, na lijevoj traci aplikacije prikazani su svi predmeti koje je potrebno unijeti u raspored. Svaki se predmet može podijeliti na dvije do tri kategorije izvođenja nastave – predavanja, seminari, vježbe - od kojih svaka kategorija ima svoje semestralno opterećenje te specifične zahtjeve na opremljenost dvorane izvođenja. Svaki predmet i kategorija predmeta, kao preduvjet izvođenja, može zahtijevati da dvorana izvođenja bude opremljena računalima, što se uzima u obzir kod ponude dvorana. Ukoliko dvorana nije opremljena računalima, a predmet zahtjeva da bude, dvorana neće biti prikazana u listi dostupnih. Isto tako svaki predmet ima određeni broj upisanih studenata. Ukoliko je kapacitet dvorane manji od broja upisanih studenata, dvorana također neće biti prikazana na listi dostupnih dvorana. Podno odabrane dvorane vizualno se prikazuje se njezin kapacitet i opremljenost. Bitno je napomenuti da ukoliko odabrani predmet ne zahtjeva dodatnu opremljenost, korisnik i dalje kao izbor ima odabir dvorane opremljene računalima.



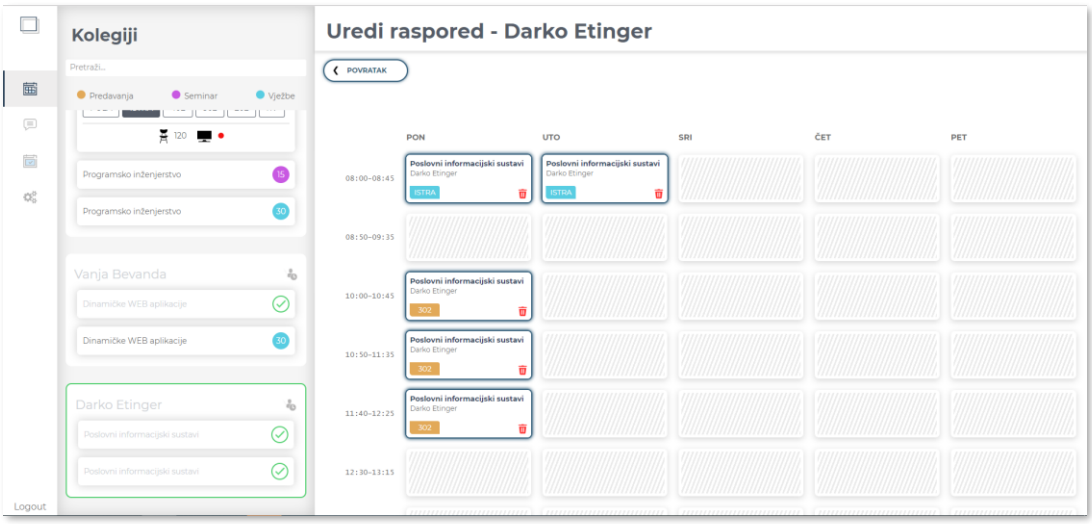
Slika 6. Modul za uređivanje rasporeda

Nakon odabira kombinacije, klikom na odabrano zeleno polje, ono postaje popunjeno detaljima predavanja te se sa ukupne satnice oduzima broj jednak zbroju tjedana u odabranom semestru. Na slici 6. možemo vidjeti stanja u kojima se polja

mogu nalaziti, pa tako zelena boja označava dostupnost dvorane i profesora a crvena, sa dodatkom ikone čovječuljka, nedostupnost odabranog profesora u to vrijeme. Ukoliko je polje uokvireno i crvenom i zelenom bojom, to znači da je odabrana dvorana u to vrijeme dostupna, no odabrani profesor nije. Kada je polje popunjeno predavanjem uokviruje se plavom bojom. Prazno neuokvireno polje označava da je u to vrijeme odabrana dvorana nedostupna te da se ne održava niti jedno predavanje u trenutnom rasporedu. Također, bitno je naglasiti da se u bilo kojem trenutku popunjeno polje može osloboditi od predavanja klikom na crvenu kanticu.

Uređivanje jednog rasporeda unutar godišnjeg rasporeda reflektira izradu ostalih pojedinačnih rasporeda. To rezultira različitim kombinacijama okvira pojedinih polja te predstavlja glavnu funkcionalnost aplikacije TTM.

Jednom kada se unesu sva potrebna predavanja određenog profesora, taj se profesor, zajedno sa predavanjima, stavlja na dno liste te se uokviruje zelenom bojom. Posljednjoj funkcionalnosti odnosno prikazu rasporeda za pojedinačne profesore može se pristupiti na dva načina: klikom na sivog čovječuljka pokraj imena profesora u lijevoj traci aplikacije ili klikom na crvenog čovječuljka unutar polja. Taj nam prikaz služi kao pomoć u izradi rasporeda optimalnog i za studente i za profesore te ga nije moguće uređivati (slika 7.) (Vučković, 2019).



Slika 7. Prikaz rasporeda pojedinih profesora

3. LOGIČKI DIZAJN APLIKACIJE

Logički dizajn i pravila jamče ispravan rad aplikacije i sigurnost počevši od autentikacije na samom ulazu u aplikaciju te autorizacije tokom korištenja aplikacije. Prilikom prijave u sustav nad unesenim korisničkim podacima vrši se validacija na strani poslužitelja koja će, u slučaju neispravnih autentikacijskih podataka, vratiti greške koje se potom prikazuju korisniku, ili u slučaju uspješne prijave, vratiti token koji će biti spremljen na klijentu i korišten za autorizaciju prilikom slanja zahtjeva na poslužitelj u toku rada aplikacije. Token koji korisnički agent sprema u svojem lokalnom spremištu koristi se za autorizaciju ruta te ima svoje vrijeme isteka, pa stoga ukoliko je token istekao, korisnik se mora ponovno prijaviti u aplikaciju. Metoda validacije koja se koristi na zaslonu za prijavu koristi se i kod unosa novog godišnjeg rasporeda. Validacija se vrši na poslužitelju štiteći tako istinitost podataka koji se unose u bazu. U narednim poglavljima biti će opisan proces izrade tokena nad objektom korisnika na poslužitelju te proces validacije podataka.

U svom radu, TTM aplikacija osim korisnički kreiranih podataka, koristi unaprijed unesene podatke. Oni, ovisno o korisničkom izboru, selektivno uključuju: dostupne predmete i profesore, dvorane, broj upisanih studenata, broj semestara, razine studija i pristup studiranju. Podaci koje korisnik generira svojim radom u aplikaciji su godišnji rasporedi te predavanja unesena u polja rasporeda tijekom kreiranja i uređivanja rasporeda. Takav pristup aplikaciji daje slobodu integracije s drugim sustavima u budućem razvoju te rasterećuje korisnika nepotrebnih koraka samostalnog unosa podataka potrebnih za rad.

S obzirom da je TTM aplikacija kreirana kao web aplikacija, moguće joj je pristupiti s bilo kojeg uređaja putem pretraživača s internetskom vezom, a podaci se spremaju u stvarnom vremenu i nakon svake izmjene što rezultira rasterećenjem korisnika samostalnog spremanja promjena te sprječava gubitak podataka.

3.1. MODEL KLIJENT-POSLUŽITELJ

TTM aplikacija implementirana je kao distribuirani sustav koristeći model klijent-poslužitelj. Ovaj se model bazira na nezavisnoj komunikaciji klijenta i poslužitelja pri čemu klijent od poslužitelja očekuje odgovor dok poslužitelj odgovara na zahtjeve klijenta. Ni jedna strana nije direktno svjesna postojanja druge, no obje strane moraju imati uniformni jezik komunikacije (format) kako bi postigli traženu inter-procesnu komunikaciju.

3.1.1 FUNKCIJE KLIJENTA

Unutarnja logika izrade rasporeda s uključenom vizualnom asistencijom, navigacijom i ostalim funkcionalnostima odvija se na strani klijenta, čineći tako TTM aplikaciju raspodijeljenim sustavom klijent-poslužitelj gdje su funkcije klijenta sljedeće:

- Isporuka korisničkog sučelja
- Provjera sintakse kod unosa podataka
- Slanje formatiranog upita poslužitelju
- Primanje i obrada odgovora
- Prikaz obrađenih podataka korisniku

Kako se na strani klijenta izvršava logička obrada podataka a poslužitelju se pristupa samo u slučaju dohvata podataka, autentikacije i verifikacije, u TTM aplikaciji klijentska strana implementirana je po principu debelog klijenta (Vučković, 2019).

3.1.2 FUNKCIJE POSLUŽITELJA

Poslužiteljska strana aplikacije ima sljedeće funkcionalnosti:

- Prihvaćanje zahtjeva klijenta
- Upravljanje pristupom
- Obrada zahtjeva
- Dohvaćanje podataka iz baze
- Obrada podataka
- Slanje formatiranih podataka klijentu

Poslužiteljski dio aplikacije odgovara na zahtjeve klijenta te vraća potrebne podatke natrag klijentu u JSON formatu putem standardnog HTTP protokola. Zahtjevi se šalju na definirane REST servise te se pristigli podaci zatim obrađuju ili se prema

zahtjevu klijenta dohvaćaju iz baze podataka. Nakon obrade odgovor se šalje natrag klijentu (codecademy.com, n.d.).

3.1.3 REST SERVISI

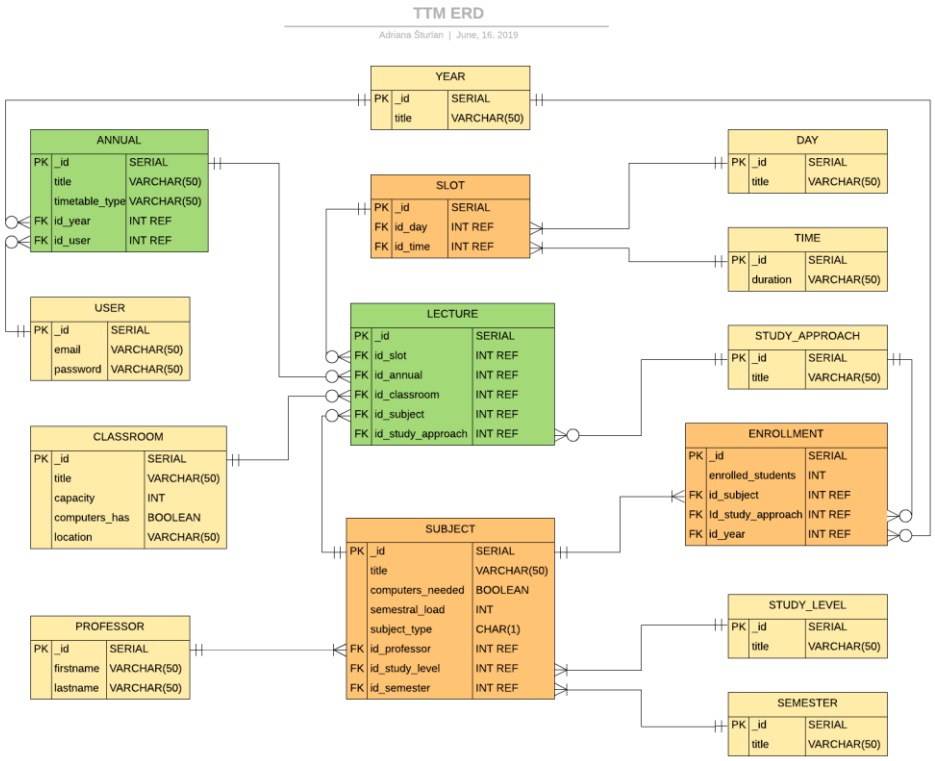
REST (*representational state transfer*), kao standard stiliziranja web servisa, omogućuje nezavisnu komunikaciju klijenta i poslužitelja na bazi HTTP protokola. Uniformnim setom pre definiranih operacija, REST servisi manipuliraju tekstualnom reprezentacijom web resursa i to čine na način da niti jedan zahtjev ne ovisi o drugom niti je njega svjestan. Kada šaljemo REST zahtjev, šaljemo ga na specifičan URI, te očekujemo odgovor u obliku jednog od standardnih formata poput HTML-a, XML-a ili, u slučaju TTM aplikacije, JSON formata. Osim odgovora u obliku podataka, klijent može dobiti potvrdu da je zahtijevana operacija izvršena. Neke od najčešće korištenih REST operacija su GET, POST, PUT i DELETE. Te operacije bazirane su na nezavisnoj klijentsko-poslužiteljskoj komunikaciji tako što su svi nadolazeći zahtjevi modularni i neovisni jedan o drugome te se informacije o prethodnim zahtjevima ne pohranjuju (osim u slučaju keširanja). Uz takvu vrstu transfera podataka, omogućeno je da više klijenata zahtjeve šalje na isti servis te dobiva iste odnosno individualizirane odgovore, što naravno ovisi o prirodi samog zahtjeva. Također je omogućeno i nesmetano korištenje posrednika (eng. *proxy*) te ekvilibrista opterećenja (eng. *load balancer*), kao posrednika u komunikaciji.

Svaki se zahtjev šalje na određenu putanju definiranu ciljanim poslužiteljskim servisom. Kod kreiranja putanja pozornost se pridaje jasnoći i čitljivosti te se nastoji putanju održati samo opisnom radi lakog razumijevanja njezine funkcije. U zaglavlju zahtjeva klijent, između ostalog, šalje i tip odgovora koji očekuje od poslužitelja (takozvani MIME tip) poput tipa *txt/html*, *text/css* ili *image* te se ista informacija šalje kao odgovor od poslužitelja na taj zahtjev. Tip odgovora se mora podudarati sa tipom navedenim u zahtjevu kako bi se odgovor mogao obraditi sukladno definicijama protokola. Na odgovoru se također nalazi i statusni kod koji informira klijenta o uspješnosti provođenja tražene operacije. Tako se primjerice kod slanja GET zahtjeva očekuje odgovor sa statusom 200 (ok), kod POST zahtjeva status 201 (stvoreno) te DELETE zahtjeva status 204 (bez sadržaja) (codecademy.com, n.d.).

U aplikaciji TTM klijent sa poslužiteljem komunicira kroz REST servise kreirane putem aplikacijskog okvira Node.js programskog okruženja, a podaci se u odgovorima šalju u formatu JSON.

3.2. DIZAJN BAZE PODATAKA

Nakon autorizacije klijentskih zahtjeva, u slučaju potrebe za podacima, poslužitelj se povezuje sa relacijskom PostgreSQL bazom podataka putem ORM alata za transakcijsku podršku. Logički dizajn korištene baze podataka predstavljen je strukturalnim ER dijagramom koji prikazuje sve entitete i veze unutar baze uz pripadajuće kardinalnosti te odgovarajuće tipove podataka.



Slika 8. ER dijagram baze podataka

ER dijagram (prikazan na slici 8.) sastoji se od 14 međusobno povezanih relacija od kojih svaka kao primarni ključ sadrži jedinstveni id te ostale atribute koji odgovaraju fizičkoj implementaciji u bazi. Primarni ključ svake relacije u bazi se generira

sekvencijalno pa je stoga označen sa PostgreSQL tipom podatka SERIAL. Također su i ostali tipovi podataka usklađeni sa Postgres specifikacijom. Ovaj model prikazuje i strane ključeve relacija te veze i kardinalnosti između relacija Martinovom notacijom. Iz dijagrama se mogu izdvojiti relacije označene zelenom bojom (*Lecture* i *Annual*) jer se CRUD operacije na bazu izvršavaju isključivo nad njima. Te relacije, građene po uzoru na model pahuljice, sadrže strane ključeve okolnih relacija (dimenzija), pa se tako primjerice relacija Lecture sastoji od stranih ključeva svih relacija koje sadrže detalje o predavanjima, a s obzirom da je korisnik taj koji kreira polja s predavanjima uređivanjem rasporeda, redaka u toj tablici može biti relativno mnogo (ovisno o broju godišnjih rasporeda i svih rasporeda unutar pojedinog godišnjeg rasporeda). Iz tog razloga, relacija Lecture kreirana je u normaliziranom obliku sa minimalnom redundancijom podataka. S druge strane, žuta boja relacije označava relacije koje ne sadrže strane ključeve već isključivo podatke koji se koriste u ostalim relacija putem stranih ključeva poput narančasto označenih relacija (na kojima se, u suprotnosti zeleno obojanim relacijama, ne izvršava operacija INSERT).

Svaka relacija na ovom dijagramu na strani poslužitelja predstavljena je Sequelize modelom. Naredna poglavlja opisuju tehnologije korištene na poslužitelju te na stvarnim primjerima detaljno prikazuju povezanost baze podataka i poslužitelja.

4. KORIŠTENE TEHNOLOGIJE

U ovom poglavlju biti će opisane tehnologije korištene u izgradnji poslužiteljskog dijela TTM aplikacije, no jednako je bitno spomenuti implementaciju na strani klijenta kako bi se dobio uvid u širinu cjelokupnog projekta. TTM aplikacija građena je kao web aplikacija modernog i fluidnog dizajna ostvarenog korištenjem CSS3 i Bootstrap tehnologija, a funkcionalnosti na klijentu implementirane su putem Vue.js Javascript okvira uz potporu Vuex i Vue-Router tehnologija te ostalih biblioteka potrebnih za slanje zahtjeva, izgradnju modula, kompiliranje i slično. Kako se poslužiteljska strana aplikacije izvršava nezavisno od klijenta, bilo je moguće u potpunosti odvojiti njihove funkcionalnosti i građu, a komunikaciju ostvarivati prethodno spomenutim RESTful pristupom. Poslužiteljska strana aplikacije građena je u Node.js *run-time* programskom okruženju koje omogućuje korištenje Javascript jezika za skriptiranje na strani poslužitelja, no iako se ne smatra programskim okvirom, Node.js uključuje bogatu biblioteku Javascript modula korištenih za kreiranje poslužitelja.

Do pojave Node.js okruženja JavaScript je bio jezik korišten isključivo za skriptiranje na strani klijenta, no s početkom u 2009. godini kreiranje dinamičnog sadržaja postalo je moguće i na strani poslužitelja. Iako je Node.js okruženje jednodretveni proces, smatra se učinkovitom i skalabilnom platformom sa malim zahtjevima na sustav zbog korištenja I/O modela temeljenog na događajima i asinkronom izvođenju operacija, dok petlja događaja (eng. *event-loop*) omogućuje neblokirajuće izvođenje unutarnjih procesa. Zbog navedenih svojstava, klijent nikada neće čekati odgovor na zahtjev, već će nastaviti izvođenje ostalih operacija sve dok ne zaprimi odgovor koji se, nakon što dođe na red u stogu događaja (eng. *event stack*), prikladno obrađuje (tutorialspoint.com, n.d.).

Instalacijom Node.js-a, u paketu dobivamo najpoznatiji upravitelj zavisnih modula, takozvani *node package manager* (skr. *npm*), pomoću kojeg pristupamo najvećem repozitoriju otvorenog koda koji jednostavno možemo preuzeti te koristiti u aplikaciji. Zbog velike popularnosti, *npm* se ne koristi isključivo za poslužiteljski Node, već i za klijentske aplikacije.

Umjesto uporabe suštinskog Node.js-a, aplikaciju gradi aplikacijski okvir Express.js čija je svrha apstrahiranje procesa izgradnje aplikacije te ubrzavanje razvoja servisa. Među ostalim tehnologijama nalaze se i *ekspress-validator* kao

posrednik u validaciji podataka na strani poslužitelja, zatim *passport* i *JWT* za ovjeru autentičnosti pristupa, *babel* i *webpack* za es6 podršku i objedinjenje modula te *nodemon* za praćenje promjena i ponovno pokretanje poslužitelja u toku razvoja aplikacije. Za interakciju s bazom podataka korišteni su *postgres* moduli sa svojom kolekcijom funkcija i pre-definiranih SQL upita, te Sequelize ORM (eng. *object relational mapping*) moduli koji, bazirani na obećanjima, pružaju podršku izvođenju CRUD operacija kroz odabrani dijalekt. Nabrojane tehnologije biti će podobnije opisane na primjerima programskog rješenja.

5. FIZIČKI DIZAJN APLIKACIJE

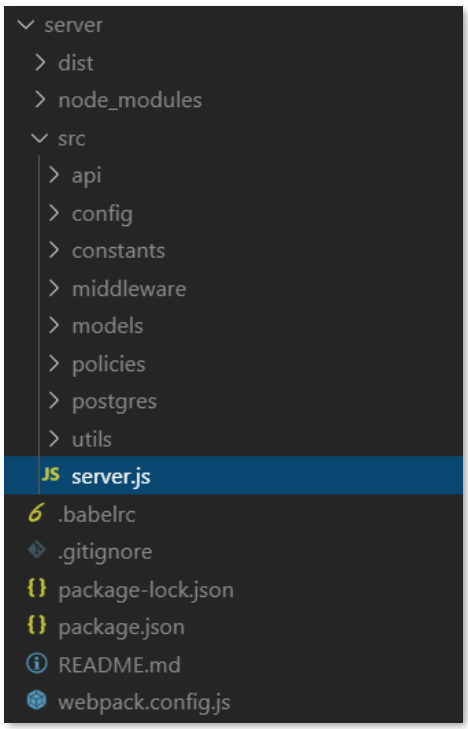
5.1. POSTAVLJANJE PROJEKTA I PROJEKTNA STRUKTURA

Korištenjem Node.js tehnologije na poslužitelju, postavljanje projekta relativno je jednostavno te se odvija u svega nekoliko koraka. Prvi korak je odabir direktorija u kojemu će se projekt nalaziti unutar njega pokrenuti naredbu *npm init.* Ovom naredbom npm za nas kreira datoteku *package.json* koja, između ostalog, sadrži meta podatke o projektu poput naziva, verzije, npm naredbi, dediciranog repozitorija, početne točke aplikacije te liste zavisnih modula koje koristimo u izgradnji aplikacije. Osim modula potrebnih za rad aplikacije, postoje i moduli korišteni isključivo u razvojnom okruženju koje nazivamo razvojne zavisnosti. Te zavisnosti nisu potrebne u radu aplikacije nakon završetka razvojne faze.

Sljedeći korak je instalacija modula koje namjeravamo koristiti u razvoju aplikacije. Instalacija modula odvija se kroz npm upravitelj naredbom *npm install <naziv paketa>* te se oni, nakon instalacije, automatski dodaju na listu zavisnosti package.json datoteke, što je svakako korisno jer u budućnosti, ukoliko želimo pokrenuti isti projekt na drugom računalu, nismo primorani ručno instalirati svaki paket već jednostavno pokrenemo naredbu *npm install* koja prolazi kroz zavisnosti definirane unutar package.json datoteke, dohvaća ih te smješta u mapu node_modules od kuda se moduli mogu koristiti. Također, naredbom *npm update* ažuriramo korištene pakete na zadnju semantičku verziju. Slika 9. prikazuje dio package.json datoteke sa aplikacijskim i razvojnim zavisnostima TTM aplikacije.

Slika 9. Projektne i razvojne zavisnosti aplikacije TTM

Datoteke TTM aplikacije strukturirane su po standardima prakse na način prikazan na slici 10.



Slika 10. Datotečna struktura poslužitelja

Svi postojeći servisi definirani su unutar mape *api* te strukturirani na način da svaka datoteka nosi naziv ciljanog segmenta nadolazećeg zahtjeva. Ako se primjerice od poslužitelja očekuje da kao odgovor vrati sve predmete u JSON formatu, onda bi URL trebao biti strukturiran na sljedeći način: *host:port/api/v1/subject .* Nadalje, mape *config*, *middleware* i *postgres* sadržavaju konfiguracijske datoteke, *constants* definicije ruta, *models* definirane Sequelize modele, *policies* politike za autentikaciju dok *utils* sadržava često korištene funkcionalnosti. O datotekama unutar navedenih mapa biti će riječi u narednim poglavljima.

5.2. KONFIGURACIJSKE DATOTEKE I SERVER.JS

Paralelno s mapom *src* definirane su konfiguracijske datoteke *webpack* i *babel* modula koji će se koristiti kod pripremanja aplikacije za produkcijsku fazu a uključuju pretvorbu sintakse u es5 standard te umotavanje mreže zavisnosti u jednu jedinstvenu datoteku. Ulazna skripta poslužitelja nazvana je *server.js* i pokreće se naredbom *npm run start* (slika 11.). Ta naredba najprije postavlja globalnu varijablu okruženja NODE_ENV na vrijednost „local“ kroz *cross-env* modul, čijim korištenjem izbjegavamo modificiranje naredbe za potrebe standarda različitih platformi. Također se pokreću *nodemon* te *babel-node* koji radi isto što i naredba *node*, uz prednost prevođenja u es5 sintaksu te pokretanja neobaveznih dodataka koji pružaju potporu za specifična svojstva korištenog jezika.

```
"scripts": {
  "start": "cross-env NODE_ENV=local && nodemon --exec babel-node src/server.js",
  "build": "rm -rf dist && webpack",
  "build-start": "node ./dist/app.bundle.js"
},
```

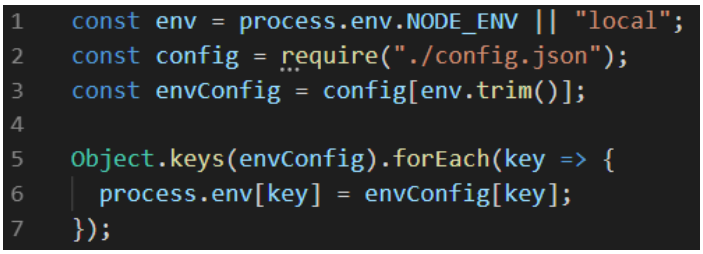
Slika 11. Datoteka *package.json* – npm naredbe

Ulazna datoteka programa koristi ostale datoteke potrebne za izvođenje aplikacije, uključujući postavke za konekciju sa PostgreSQL bazom podataka, konfiguraciju globalnih varijabli te postavke Express aplikacije. Slika 12. prikazuje strukturu datoteke *server.js*.

```
import express from "express";

require("./config/config");
require("./postgres/postgres");

const server = express();

server.use("/", require("./api/v1"));

server.listen(process.env.PORT, () => {
  console.log(`Server running on port ${process.env.PORT}`);
});
```

Slika 12. Datoteka *server.js*

Datoteka *config.js* (slika 13.) postavlja globalne varijable na vrijednosti definirane u unutar *config.json* datoteke ovisno o vrijednosti NODE_ENV varijable postavljenje putem cross-env modula pokretanjem naredbe *npm start.*



```
const env = process.env.NODE_ENV || "local";
const config = require("./config.json");
const envConfig = config[env.trim()];

Object.keys(envConfig).forEach(key => {
  process.env[key] = envConfig[key];
});
```

Slika 13. Datoteka config.js

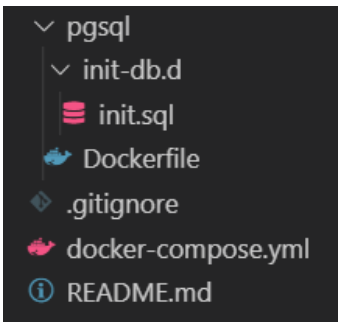
Ukoliko je primjerice vrijednost NODE_ENV varijable jednaka „development“, sve će globalne varijable procesa biti postavljene na vrijednosti definirane ključem „development“ unutar config.json datoteke. Općenito, globalnim varijablama može pristupiti bilo koja skripta te ih koristiti za donošenje odluka ili konfiguraciju. U slučaju TTM aplikacije, globalne varijable sadrže vrijednosti koje se koriste u procesu konekcije na bazu podataka poput pristupnog porta, naziva korisnika, lozinke te odabranog dijalekta. Ovisno je li okruženje razvojno ili produkcijsko, vrijednosti tih varijabli mogu se mijenjati.

5.3. BAZA PODATAKA POSTGRESQL

Kao DBMS aplikacije TTM odabrana je poznata relacijska baza podataka PostgreSQL. S obzirom da se shema baze podataka sastoji od relacija koje su inicijalno popunjene podacima spremnim za korištenje, bilo je potrebno kreirati skriptu koja će se pri pokretanju učitati u bazu podataka te kreirati potrebne tablice, pripremajući tako bazu za korištenje. U svrhu inicijalizacije baze podataka korištena je virtualna okolina Docker koja, osim automatizacije procesa postavljanja okruženja, omogućava replikaciju razvojnog okruženja nezavisno od korištenog sustava.

5.3.1. VIRTUALNA OKOLINA DOCKER

Kao virtualizacijski softver i PAAS, Docker nam u suštini služi za izoliranje dijelova softvera u takozvane spremnike te njihovo nesmetano pokretanje na bilo kojoj platformi i okruženju. U slučaju TTM aplikacije, virtualna okolina Docker kreira spremnik sa slikom postgreSQL baze podataka na standardnom Postgres portu te, u toku inicijalizacije, učitava i pokreće pre definiranu *init.sql* skriptu. Datoteke korištene za postavljanje baze podataka možemo vidjeti na slici 14.



Slika 14. Struktura datoteka potrebnih za pokretanje PostgreSQL baze podataka

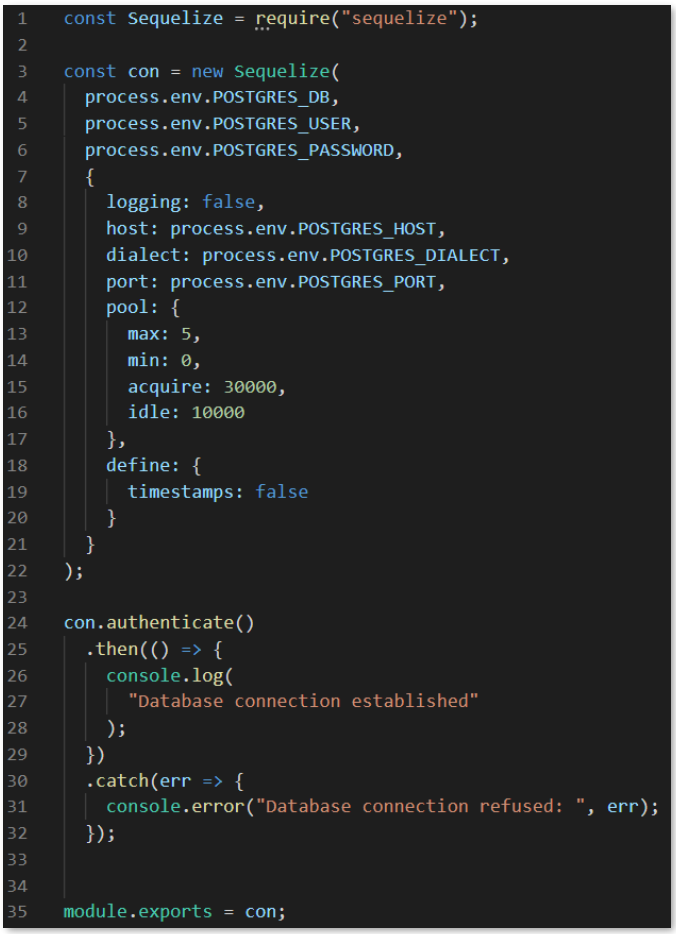
Unutar datoteke *docker-compose.yml* nalazi se definicija servisa koje pokrećemo naredbom *docker-compose up.* Docker najprije kreira spremnike za navedene servise te ih zatim instalira. Specifično u slučaju TTM aplikacije, jedini servis koji je potrebno pokrenuti je Postgres. Sukladno definicijama, Docker pristupa kreiranoj bazi podataka akreditivima definiranim u Dockerfile datoteci te pokreće

skriptu init.sql koja sadržava naredbe za populaciju baze podataka. Baza je zatim, nakon konekcije s poslužiteljem, spremna za korištenje.

5.3.2. POSTGRESQL KONEKCIJA I SEQUELIZE

Prije kreiranja aplikacijskog dijela poslužitelja, potrebno je ostvariti konekciju sa kreiranom bazom podataka kako bi funkcije poslužitelja mogle pristupati podacima i izvršavati svoje zadatke. Proces konekcije sa bazu podataka odvija se putem Sequelize modula neposredno nakon postavljanja globalnih varijabli. Sequelize je poznata Javascript biblioteka koje se koristi kao ORM alat na poslužiteljskoj strani web aplikacija sa namjerom izvođenja CRUD operacija (*Create, Read, Update, Delete*) dijalektom odabrane baze podataka. Korištenje ORM alata je velika prednost u odnosu na pisanje čistih SQL naredbi jer programeri mogu nesmetano raditi neovisno o bazi podataka koju koriste. Glavna uloga ORM alata je mapiranje objektne sintakse na shemu baze podataka, dok specifično Sequelize koristi Javascript objektnu sintaksu pa se stoga može koristiti u kombinaciji sa drugim Node.js bibliotekama za izgradnju poslužitelja.

Unutar mape *postgres* nalazi se datoteka korištena za ostvarivanje konekcije s bazom podataka te je upravo ta datoteka pozvana u toku izvođenja ulazne skripte aplikacije. U toj se datoteci, koju možemo vidjeti na slici 15., kreira nova instanca klase Sequelize čijem se konstruktoru predaju parametri poput prethodno spomenutih globalnih varijabli koje sadrže detalje potrebne za ostvarivanje uspješne konekcije s bazom podataka. Također, kreiranom objektu možemo predati i neobavezne parametre koji vrše internu konfiguraciju i postavljaju ograničenja. Od neobaveznih parametara definirani su *host*, *dialect* i *port* te objekt *pool* koji služi za konfiguraciju ponovnog iskorištavanja ostvarenih konekcija na bazu. Neke od tih konfiguracija su minimalan broj konekcija potreban za kreiranje bazena, maksimalan broj konekcija koje je moguće zadržati u bazenu te vremensko ograničenje u milisekundama koje govori koliko vremena konekcija smije biti besposlena prije nego li se otpusti iz bazena. Nakon kreiranja i konfiguriranja Sequelize objekta, možemo pokušati povezati poslužitelj s bazom podataka putem metode *authenticate.* Ukoliko je povezivanje uspješno, konzola će ispisati prikladnu poruku.

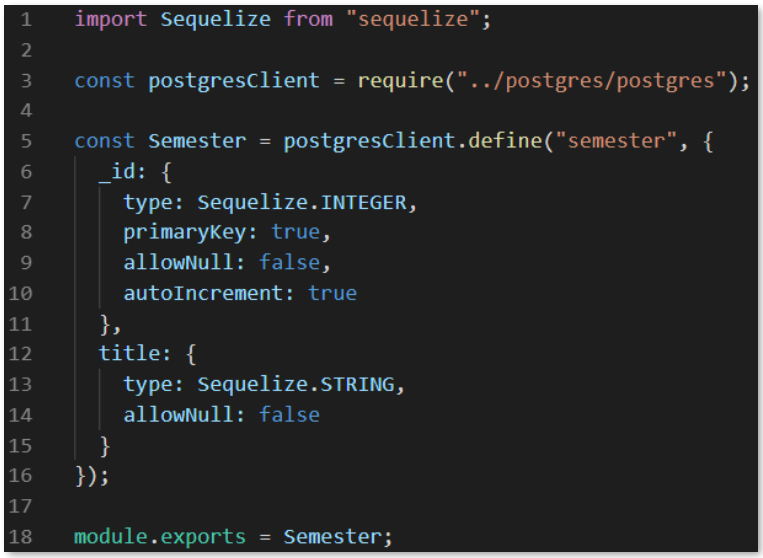


```
const Sequelize = require("sequelize");

const con = new Sequelize(
  process.env.POSTGRES_DB,
  process.env.POSTGRES_USER,
  process.env.POSTGRES_PASSWORD,
  {
    logging: false,
    host: process.env.POSTGRES_HOST,
    dialect: process.env.POSTGRES_DIALECT,
    port: process.env.POSTGRES_PORT,
    pool: {
      max: 5,
      min: 0,
      acquire: 30000,
      idle: 10000
    },
    define: {
      timestamps: false
    }
  }
);

con.authenticate()
  .then(() => {
    console.log(
      "Database connection established"
    );
  })
  .catch(err => {
    console.error("Database connection refused: ", err);
  });


module.exports = con;
```

Slika 15. Datoteka postgres.js

5.3.3. SEQUELIZE MODELI

Svaka relacija sheme baze podataka odražena je na pojedinačni Sequelize model unutar aplikacije. Sequelize funkcije korištene nad definiranim modelom generiraju naredbe CRUD operacija te ih prosljeđuju pg biblioteci koja se zatim spaja sa bazom podataka i šalje generirane upite. Nakon što se zadana operacija izvrši na bazi, zahtijevani podaci vraćaju se pg biblioteci u originalnom obliku. Zatim se ti podaci prosljeđuju Sequelize modulima koji ih raščlanjuju u Javascript objekte te tako pripremaju za daljnju uporabu na poslužitelju, mapirajući pritom svaku stavku vraćenog objekta na jedan redak u tablici. Modeli su definirani kao struktura podataka koju očekujemo primiti kao odgovor na SQL upit pa tako, uz očekivani naziv tablice iz baze, definiramo i očekivanu strukturu podataka. Slika 16. prikazuje jedan od definiranih modela aplikacije TTM.



```
import Sequelize from "sequelize";

const postgresClient = require("../postgres/postgres");

const Semester = postgresClient.define("semester", {
  _id: {
    type: Sequelize.INTEGER,
    primaryKey: true,
    allowNull: false,
    autoIncrement: true
  },
  title: {
    type: Sequelize.STRING,
    allowNull: false
  }
});

module.exports = Semester;
```

Slika 16. Sequelize model „*Semester*“

Osim definicije tipova podataka, unutar datoteke modela mogu se definirati i prilagođene funkcije koje kao zadatak imaju izvršavanje CRUD operacija nad bazom podataka a također je moguće definirati validacijske parametre kod unosa novih podataka u bazu ili postaviti zadane vrijednosti stupaca. Svaki model definiran u aplikaciji TTM ima svoju mapu i datoteku index.js koja se zatim uključuje u one datoteke gdje su specifični modeli potrebni.

5.4. EXPRESS.JS APLIKACIJA

Kako aplikacija sada ima definirane modele i uspješno ostvarenu konekciju na bazu podataka, slijedi pokretanje poslužitelja. Kao tehnologija za razvoj servisa na poslužitelju odabran je Express.js koji slovi kao standardni okvir Node.js-a zbog brojnosti korisnika te kvalitetne programerske zajednice. Express.js, kao biblioteka otvorenog koda, svoju je popularnost stekla jednostavnim pristupom izgradnji servisa, asinkronim izvođenjem operacija te podržavanjem MVC strukture. Kao i ostali paketi, Express.js instalira se putem *npm* upravitelja, te se uključuje u početnu točku aplikacije. Sljedeći korak je, putem funkcije *listen*, pokrenuti poslužitelj i omogućiti da aplikacija sluša nadolazeće zahtjeve na portu definiranom globalnom varijablom. Kako je Express.js samo omotač standardnim Node.js funkcionalnostima, tako je i metoda listen apstrakcija originalne *createServer* funkcije.

Prethodno opisane http metode RESTful pristupa lako se implementiraju pomoću Express.js biblioteke. Poslužitelj mora adekvatno odgovoriti na zahtjev tako što će, u ovisnosti o nadolazećem zahtjevu, pozvati metode mapirane na dani URL te tip zahtjeva. Te metode kao parametre primaju *request* i *response* koji se, ovisno o tipu zahtjeva, koriste u svrhu izvršavanja operacija ili slanja adekvatnog odgovora. Putanje u kombinaciji s tipom zahtjeva definiraju takozvanu krajnju točku (eng. *endpoint*).

Aplikacija TTM svoje krajnje točke definira u posebnoj datoteci kako bi se, u slučaju potrebe u budućnosti, one lako mogle izmijeniti. Definirane krajnje točke prikazane su na sljedećoj slici:

```
1   const router = require("express").Router();
2
3   const bodyParser = require("body-parser");
4   const cors = require("../../middleware/cors");
5   const passport = require("./middleware/passport");
6
7   router.use(bodyParser.json());
8   router.use(cors);
9   router.use(passport)
10
11  router.use("/", require("./annual"));
12  router.use("/", require("./professor"));
13  router.use("/", require("./approach"));
14  router.use("/", require("./level"));
15  router.use("/", require("./semester"));
16  router.use("/", require("./subject"));
17  router.use("/", require("./year"));
18  router.use("/", require("./classroom"));
19  router.use("/", require("./enrollment"));
20  router.use("/", require("./day"));
21  router.use("/", require("./user"));
22
23  module.exports = router;
```

Slika 17. Definicija statičnog dijela krajnjih točaka

Svaka skupina krajnjih točaka modularno odijeljena u zasebne datoteke ovisno o funkcionalnosti. U ovoj se datoteci također koristi posrednik na razini usmjerivača - objekt *router* – koji ima jedinstvenu ulogu obrade zahtjeva i slanja odgovora. Ovakav pristup poželjan je za održavanje modularnosti aplikacije čak i na još nižim razinama tako što definirani *router* kao svoje posrednike koristi druge instance objekta *router* izvezenih iz ostalih datoteka, te ostale posrednike kao što su *cors, passport* i

bodyParser. Na ovaj način lako je implementirati i kontrolu ciklusa ukoliko je ona potrebna.

5.4.1. API

Većina zahtjeva na poslužitelj manifestira se u obliku tipa GET od kojeg se očekuje odgovor u JSON formatu te obliku tipa POST za unos podataka u bazu. Definirane rute koriste funkcije Sequelize modela za manipulaciju podacima koje potom klijentu šalje kao odgovor na zahtjev uz adekvatan statusni kod.

5.4.1.1. ZAHTJEV TIPA GET

Na sljedećoj slici možemo vidjeti neke od definiranih krajnjih točaka za zahtjeve tipa GET.

```
const router=require('express').Router()
const api=require('../../../constants')
const sendErrorResponce=require("../../../utils")
import authenticate from '../../../policies/AuthenticationPolicy';

const YearModel=require("../../../models/year")

router.get(api.YEARS, authenticate, (req,res)=>{
    try{
        YearModel.findAll().then(years=>{
            const Years=years.map(year=>{
                return Object.assign({},{
                    id : year._id,
                    title : year.year,
                })
            })
            res.status(200).json(Years)
        })
    }catch{
        sendErrorResponce(res,ex)
    }
})
router.get(api.YEARS + "/:id_year", authenticate, (req,res)=>{
    try {
        YearModel.findOne({
            where : {
                _id : req.params.id_year
            }
        }).then((year)=>{
            res.status(200).json(year)
        })
    }catch(ex){
        sendErrorResponce(res, ex);
    }
})
module.exports=router;
```

Slika 18. Zahtjevi tipa GET

GET krajnja točka rute */year* iz baze dohvaća sve retke i stupce tablice *Year* te svaki dohvaćeni redak modificira i sprema u zaseban objekt koji se zatim šalje klijentu

kao odgovor uz status 200. Može se primijetiti da se isti princip kreiranja objekta nije koristio u donjoj GET krajnjoj točci. Razlog tomu je potreba za modifikacijom originalnog rezultata Sequelize funkcije *findAll*. Krajnja točka rute */year/:id_year* uz funkciju *findOne* dohvaća jedan redak iz baze čiji *id* odgovara *id*-u dostavljenom kao parametar zahtjeva.

Još jedan primjer zahtjeva tipa GET možemo vidjeti u datoteci *classroom.js* (slika 19.) podno definicije modela. Ondje definirana funkcija *findByCapacity* iz baze vraća sve dvorane čiji je kapacitet veći ili jednak ograničenju dostavljenom kao parametar zahtjeva. Navedena funkcija koristi se unutar *classroom* modula kod zaprimanja zahtjeva tipa GET na URL *classrooms/:capacity.*

```
19  Classroom.findByCapacity = async function(capacity) {
20    const Class = this;
21    const result = await Class.findAll({
22      where: {
23        capacity: {
24          [Op.gte]: capacity
25        }
26      }
27    });
28    return result;
29  };
```

Slika 19. Funkcija *findByCapacity*

5.4.1.2. ZAHTJEV TIPA DELETE

Zahtjev tipa DELETE možemo vidjeti na primjeru brisanja godišnjeg rasporeda (slika 20.) gdje se Sequelize funkcijom *destroy,* pozvanom nad modelom *AnnualModel*, brišu svi godišnji rasporedi koji su stigli u tijelu zahtjeva u obliku polja id-jeva. Iako ovaj način slanja podataka nije direktno RESTful, odabran je u svrhu rasterećenja klijenta zaobilaskom slanja pojedinih DELETE zahtjeva za svaki raspored posebno.

```
47  router.delete(api.ANNUAL, authenticate, (req,res) => {
48      req.body.forEach(id => {
49          try{
50              AnnualModel.destroy({
51                  where: {_id : id}
52              })
53          }catch(ex){
54              sendErrorResponce(res,ex);
55          }
56      })
57      res.status(204).send('Deleted')
58  })
```

Slika 20. Zahtjev tipa DELETE

5.4.1.3. ZAHTJEV TIPA POST

U slučaju potrebe unosa novih stavaka u bazu podataka, poslužitelj očekuje da adekvatno strukturirani podaci pristignu unutar zahtjeva tipa POST. Unos novih redaka u bazu podataka odvija se kroz Sequelize metodu *create.* S obzirom da se u tijelu zahtjeva nalazi objekt čiji ključevi odgovaraju definiciji podataka modela *Annual*, metodi *create* možemo direktno predati tijelo zahtjeva po čijem će se uzoru kreirati redak u bazi. Kako se u ovom slučaju radi o unosu korisnički definiranih podataka, prije poziva navedene funkcije potrebno je provesti validaciju korištenjem odabranog validacijskog posrednika (slika 21.).

```
34  router.post(api.ANNUAL, authenticate, validation('CreateAnnual'), async (req,res)=>{
35      try{
36          const errors=validationResult(req)
37          if(!errors.isEmpty()){
38              return res.status(422).json({ errors: errors.array() });
39          }else{
40              const ANNUAL=await AnnualModel.create(req.body)
41              res.status(200).json(ANNUAL)
42          }
43      }catch(ex){
44          sendErrorResponce(res,ex);
45      }
46  })
```

Slika 21. Zahtjev tipa POST i validacija podataka

Svaki se upit na bazu, neovisno o tipu zahtjeva, izvršava u *try-catch* bloku gdje se u slučaju neuspjelog povezivanja ili greške izvršava skripta definirana u mapi *utils.* Slika 22. prikazuje korištenu *sendErrorResponse* funkciju.

```
const sendErrorResponse = (res, ex) => {
  if (ex.message) {
    return res.status(400).send(ex.message);
  }
  res.status(400).send(ex);
};

module.exports = sendErrorResponse;
```

Slika 22. Odgovor poslužitelja u slučaju pogreške

5.4.2. POSREDNIK EXPRESS-VALIDATOR

Express-validator je skup posrednika biblioteke validator.js, namijenjenih korištenju unutar Node.js aplikacije u svrhu validacije i sanacije korisnički unesenih podataka. Na slici 21. vidimo da se validacijska funkcija koristi kao posrednik te da joj se kao parametar predaje prigodan naziv slučaja *switch* bloka koji se mora izvršiti u svrhu validacije podataka iz zahtjeva.

```
import {check} from 'express-validator';
function validate(method){
    switch(method){
        case 'CreateAnnual' : {
            return [
                check('title').not().isEmpty().withMessage("Polje 'Naziv' ne smije biti prazno"),
                check('id_year').isNumeric().withMessage("Polje 'Akademska godina' ne smije biti prazno"),
                check('timetable_type').isIn(['Ljeto','Zima']).withMessage("Polje 'Tip rasporeda' ne smije biti prazno")
            ]
        }
        case 'UserLogin' : {
            return [
                check('email').isEmail().withMessage("Potrebno je unjeti važeću e-mail adresu"),
                check('password').not().isEmpty().withMessage("Potrebno je unjeti lozinku")
            ]
        }
    }
};
export default validate;
```

Slika 23. Datoteka validation.js

Navedeni switch blok nalazi se unutar datoteke *validation.js* (slika 23.) u funkciji *validate* koja prolazi kroz stavke tijela zahtjeva te, u slučaju kreiranja novog godišnjeg

rasporeda, kroz funkciju *check* vrši validaciju korisničkog unosa. Na taj se način provjerava primjerice je li dani naziv rasporeda prazan te je li dani tip rasporeda naveden kao jedan od važećih izbora. Ukoliko je validacija neuspješna, *express-validator* puni objekt *ValidationResult* porukama predanim *withMessage* funkciji. Nakon prolaska kroz validacijski posrednik, ukoliko postoje greške u korisničkom unosu, klijentu se šalje odgovor u obliku polja grešaka sa statusnim kodom 422, a ukoliko je validacija uspješna i polje grešaka je prazno, kreira se novi redak u bazi podataka te se klijentu šalje novi kreirani godišnji raspored uz statusni kod 200.

5.4.3. POSREDNIK PASSPORT

Passport je autentikacijski posrednik namijenjen uporabi unutar Express.js aplikacija čije funkcije uključuju strategije za provođenje autentikacijskog procesa nad danim korisničkim podacima te, osim za bazične prijave, pružaju podršku za prijavljivanje putem društvenih mreža, tokena i e-mail opskrbljivača. Kada se korisnik nakon uspješne validacije prijavi u sustav, poslužitelj za njega generira token koji se šalje poslužitelju i sprema u lokalni spremnik korisničkog agenta.

```
6   const {validationResult} = require('express-validator')
7   import validation from '../../../middleware/validation'
8   import signUserjwt from '../../../middleware/authentication'
9
10  router.post(api.USER, validation('UserLogin'), async (req,res)=>{
11      try{
12          const errors=validationResult(req)
13          if(!errors.isEmpty()){
14              return res.status(422).json({ errors: errors.array() });
15          }else{
16              const User= await UserModel.findOne({
17                  where:{
18                      email:req.body.email,
19                  }
20              })
21              if(!User) {
22                 return res.status(403).json({ errors: [{msg:"Nepostojeća e-mail adresa"}]});
23              }
24              const isPasswordValid=req.body.password===User.password
25              if(!isPasswordValid){
26                 return res.status(403).json({ errors: [{msg:"Neispravna lozinka"}]});
27              }
28              const userJson=User.toJSON()
29              res.status(200).send({
30                  user: userJson,
31                  token: signUserjwt(userJson)
32              })
33          }
34      }catch(ex){
35          sendErrorResponce(res,ex);
36      }
37  })
38  module.exports=router;
```

Slika 24. Zahtjev tipa POST – prijava korisnika u sustav

Funkcija na slici 24. odgovara na zahtjev tipa POST na način da ponajprije izvršava validacija korisničkih podataka za prijavu putem prethodno opisanog posrednika *express-validator*, koristeći pritom funkciju *findOne* za usporedbu dane e-mail adrese sa postojećim adresama u bazi podataka. U svrhu boljeg korisničkog iskustva prikazom specifičnih validacijskih poruka, provjera ispravnosti lozinke vrši se odvojeno od provjere ispravnosti e-mail adrese. Nakon uspješne validacije objekt modela *UserModel* potpisuje se tokenom generiranim unutar funkcije *signUserjwt.* Ta je funkcija definirana u datoteci *authetication.js* (slika 25.) unutar mape *middleware* koja kroz SHA256 algoritam, koristeći predani objekt te globalnu varijablu JWT_SECRET kreira BASE64 tekstualnu reprezentaciju rezultata algoritma odnosno takozvani token. Sve dok poslužitelj radi, objekt *jwt* u sebi će sadržavati potrebne podatke za dekodiranje, a kreirani se token sam otpušta ukoliko istekne vrijeme njegovog trajanja definirano kao neobavezni parametar funkcije *sign.*

```
1   import jwt from 'jsonwebtoken';
2
3   function signUserjwt (user) {
4       try{
5           return jwt.sign(user, process.env.JWT_SECRET, {expiresIn: "24h"})
6       }catch(e){
7           console.log(e)
8       }
9   }
10
11  export default signUserjwt;
```

Slika 25. Funkcija *signUserjwt*

Token se potom šalje natrag klijentu i sprema u lokalni spremnik, a koristi se kao teret u zaglavlju kod slanja bilo kojeg zahtjeva na poslužitelj. Svaki zahtjev prolazi posredničku autorizacijsku funkciju za provjeru autentičnosti zahtjeva što se moglo primijetiti na prethodnim primjerima. Ova funkcija, prikazana na slici 26., *passport* metodom *authenticate* utvrđuje autorizacijski status zahtjeva te sukladno tome propušta ili odbija zahtjev.

```
const passport = require ('passport')

module.exports = function (req,res,next){
    passport.authenticate('jwt', function(err,user){
        if(err || !user){
            res.status(403).json({
                error:'Neautorizirani pristup'
            })
        }else{
            req.user=user
            next()
        }
    })(req,res,next)
}
```

Slika 26. Posrednička autorizacijska funkcija

Kako bi se autorizacijska funkcija uspješno izvršila, posredniku passport unutar datoteke *passport.js* definirana je strategija[1] koju koristi za dohvaćanje tokena iz zaglavlja zahtjeva i provjeru postojanja korisnika.

[1] Korištena strategija pod imenom passport-jwt može se pronaći na službenoj web stranici http://www.passportjs.org/packages/passport-jwt/

ZAKLJUČAK

TTM kao moderna web aplikacija svojom jednostavnošću i fluidnim dizajnom kao cilj ima upotpuniti korisničko iskustvo i olakšati izradu semestralnih rasporeda sati unutar fakultetske organizacije. Opisani proces izrade poslužiteljskih komponenti povodi se standardima struke te nastoji jasno prikazati primjere svih funkcionalnih aspekata aplikacije od prvobitnog postavljanja projekta pa sve do viših i složenijih procesa. Kroz cijeli rad, naglasak se stavlja na lepezu odabranih tehnoloških rješenja, počevši tako od samih temelja poput Node.js-a, preko korištenih biblioteka *Express.js* i *Seqelize,* baze podataka *PostgreSQL,* do posrednika *express-validator* i *passport* te pomoćnih tehnologija poput *Webpack*-a, *Babel*-a i *JWT*-a. Potkrijepljen stvarnim primjerima, rad nastoji pružiti uvid u cjelokupnu sliku procesa izrade poslužitelja.

Korištenjem RESTful servisa kao metode komunikacije između klijenta i poslužitelja relativno je jednostavno povući granicu između klijentskog i poslužiteljskog koda pa se stoga izrada cijele aplikacije dijeli na dva dijela, od kojih je klijentska strana, zajedno sa klijentskim tehnologijama, opisana u komplementarnom radu. Kako je aplikacija kreirana usporedno s dvije strane, bilo je potrebno koristiti metode verzioniranja, konkretno servis *GitHub,* za nesmetan proces kolaboracije između projekata. Također, za potrebe testiranja korišten je servis *Postman* koji olakšava ispitivanje ispravnosti definiranih API servisa.

Izradom aplikacije TTM postavljen je temelj za buduću nadogradnju i proširivanje, no i u ovom stadiju, aplikacija pruža sve osnovne funkcionalnosti potrebne za kreiranje rasporeda. Vizualna asistencija te ugrađena ograničenja čine izradu rasporeda polu automatiziranim procesom, idući tako korak dalje od standardne ručne izrade rasporeda sati.

Implementacija poslužitelja može se pogledati na linku:

https://github.com/asturlan/server

POPIS LITERATURE

Kvalifikacijski radovi:

[1] Vučković, Klara (2019.) »Razvoj klijentskih komponenti aplikacije za upravljanje rasporedom«, završni rad, Fakultet intormatike u Puli.

Web izvori:

[1] »Back-end Architecture« (n.d.), codecademy.com, https://www.codecademy.com/articles/back-end-architecture [20.08.2019]

[2] »What is rest (n.d.)«, codecademy.com, https://www.codecademy.com/articles/what-is-rest [01.09.2019]

[3] »Node.js – introduction (n.d.)«, tutorialspoint.com, https://www.tutorialspoint.com/nodejs/nodejs_introduction.html [01.09.2019]