RAZVOJ PODESIVOG MOCK POSLUŽITELJA

Rezić, Lea

Undergraduate thesis / Završni rad

2020

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:225:153265

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-18**

Repository / Repozitorij:

Algebra Univerity College - Repository of Algebra Univerity College

VISOKO UČILIŠTE ALGEBRA

ZAVRŠNI RAD

RAZVOJ PODESIVOG MOCK POSLUŽITELJA

Lea Rezić

Zagreb, veljača 2020.

„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.

Predgovor

Ovaj rad posvećujem svim ljudima kojih se i sjećam i ne sjećam, a koji su mi pomogli prilikom brojnih putovanja autostopom.

Htjela bih zahvaliti svojem mentoru, Bojanu Fulanoviću, što je prihvatio moj rad i pružio savjete. Zahvalila bih svim učiteljima, profesorima i poslovnim kolegama na prenesenom znanju.

Za kraj, zahvalila bih obitelji, prijateljima i partneru na podršci prilikom studiranja i života općenito.

(mjesto za potvrdu)

Sažetak

Rad se bavi problematikom specifičnom za proces razvoja i održavanja opsežnih JavaScript klijentskih aplikacija. Rad nastoji riješiti problem samostalnog rada na takvim aplikacijama razvojem podesivog alata za kreiranje umjetnih podataka i pokretanje *mock* poslužitelja koji će prilikom razvoja služiti kao zamjena za API servis s kojim aplikacija inače razmjenjuje podatke.

Ključne riječi: jednostranična aplikacija, razvoj softvera, održavanja softvera, mock poslužitelj, javascript, statička tipizacija, typescript.

Abstract

The aim of this paper is to produce a simple, yet higly configurable tool for generating statically typed mock data and running a mock server. The tool speeds up the development process by providing web developers with a flexible way to work on a frontend application without requiring direct access to the corresponding API service.

Keywords: single page application, software development, software maintenance, mock server, javascript, static typing, typescript.

Sadržaj

1. Uvod 1

2. Proces razvoja klijentskih aplikacija 2

2.1. Jednostranične aplikacije 2

2.1.1. Prednosti jednostraničnih aplikacija 4

2.1.2. Nedostatci jednostraničnih aplikacija 5

2.2. Modularizacija klijentskog kôda 7

2.3. Statička tipizacija klijentskog kôda 9

2.3.1. TypeScript 10

2.4. Radno okruženje 14

3. Razvoj podesivog mock poslužitelja 16

3.1. Motivacija za razvoj rješenja 16

3.1.1. Potrebe korisnika 16

3.1.2. Analiza potreba korisnika 17

3.1.3. Predloženo rješenje 18

3.2. Implementacija rješenja 18

3.2.1. Korištene tehnologije 21

3.2.2. Mock poslužitelj biblioteka 24

3.2.3. Konzolna aplikacija 28

3.2.4. GUI aplikacija 30

3.3. Upotreba za automatizirane testove 34

3.4. Nedostatci i moguća proširenja 35

Zaključak 37

Popis kratica 38

Popis slika..... 39

Popis kôdova 40

Literatura 41

1. Uvod

Od razvoja interneta do danas, internetski preglednici napredovali su od pretraživača i prikazivača tekstualnog sadržaja do okruženja u kojem se izvršavaju kompleksne aplikacije koje rješavaju širok spektar problema. Samim time, razvoj internetskih stranica i aplikacija uvelike se razlikuje danas u odnosu na svoje početke.

Jedna od posljedica tog razvoja jest pojava jednostraničnih internetskih aplikacija (eng: *single page application*), odnosno samostalnih klijentskih aplikacija koje se izvršavaju u JavaScript pokretačima unutar internetskih preglednika. Postoje razni pristupi i alati za razvoj takvih aplikacija, ali osnovna ideja svodi se na prenošenje prezentacijske logike iz poslužitelja na klijenta.

Rad se bavi problematikom specifičnom za proces razvoja i održavanja velikih klijentskih aplikacija. Ukazuje na prednosti modularizacije kôda, prednosti korištenja statičke tipizacije, te prednosti korištenja alata za automatizaciju čestih radnji prilikom programiranja.

Naglasak rada je na povećanju produktivnosti programera prilikom razvoja i održavanja klijentskih aplikacija. U nastojanju facilitacije tog procesa, razvijen je alat za brzo kreiranje, održavanje i pokretanje *mock* poslužitelja, koristeći statički tipizirane podatke. *Mock* poslužitelj odnosi se na računalni program kojem je zadaća posluživati unaprijed definirane umjetne podatke, te je kao takav gotovo nezaobilazan za efikasno razvijanje i održavanje samostalnih klijentskih aplikacija.

Struktura rada podijeljena je u 2 dijela. U prvom dijelu govorit će se o procesu razvoja i održavanja klijentskih aplikacija, bit će riječi o jednostaničnim aplikacijama, o modularizaciji i statičkoj tipizaciji klijentskog kôda te utjecaju radnog okruženja na produktivnost programera. U drugom dijelu bit će prezentirana motivacija za razvoj podesivog *mock* poslužitelja te opis implementacije rješenja. U sklopu drugog dijela bit će prezentirane i tehnologije korištene prilikom razvoja rješenja, kao i nedostatci te moguća proširenja rješenja.

2. Proces razvoja klijentskih aplikacija

U ovom poglavlju prikazat će se problematika specifična za razvoj i održavanje velikih klijentskih aplikacija. Započet će jednostraničnim aplikacijama, prednostima i nedostatcima takve arhitekture. Nakon toga osvrnut će se na poteškoće održavanja velikog repozitorija kôda pisanog u JavaScriptu, prikazati korisnosti modulariziranja kôda te korištenja statičke tipizacije. Budući da ovaj rad za postizanje statičke tipizacije koristi TypeScript, dio poglavlja posvetit će se TypeScriptu. Na kraju poglavlja, rad će se baviti problematikom optimizacije programerske produktivnosti u vidu radnog okruženja.

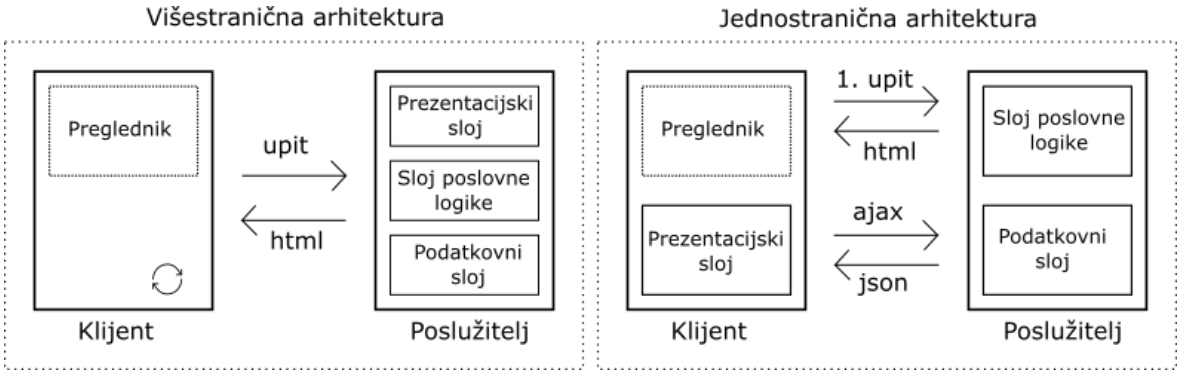
2.1. Jednostranične aplikacije

U klasičnoj višestraničnoj (engl. *multi page*) klijent-poslužitelj arhitekturi internetskih aplikacija, klijent je odgovoran isključivo za prikazivanje dokumenata pripremljenih od strane poslužitelja. Koristeći tehnologije HTML i CSS, klijent (internetski preglednik) prikazuje sadržaj i primjenjuje definirane stilove, a u stanju je vršiti i dinamičku manipulaciju nad DOM elementima izvršavanjem JavaScript skripti. Većina akcija korisnika rezultira slanjem upita poslužitelju, koji zatim obavlja sve potrebne radnje za specifičan upit te klijentu dostavlja novi dokument, nakon čega se korisniku osvježava preglednik i prikazuje nova stranica.

Pojednostavljeno, možemo reći da je u takvoj arhitekturi poslužitelj odgovoran za pristup podatcima, za pripremu i obradu podataka, za obavljanje poslovne logike nad podatcima te za pripremu i posluživanje prezentacijskih dokumenata koje će iscrtati klijent. Opisani pristup često se koristi prilikom razvoja internetskih aplikacija, a razvijanje istih olakšano je postojanjem brojnih alata i radnih okvira, poput .NET MVC, Java Spring ili Laravel radnih okvira.

Ukoliko se krajnjem korisniku želi ponuditi vrlo bogat i brz sustav interakcije s aplikacijom, razvijanje i održavanje prezentacijskog sloja aplikacije s velikim brojem različitih HTML stranica i JavaScript skripti može postati vrlo zahtjevan zadatak. Različiti autori navode interaktivnost aplikacije kao glavni razlog za razvijanje jednostraničnih aplikacija (Klauzinski, 2016; Scott, 2016).

Jednostranične aplikacije (engl. *single page application*) su arhitekturalni stil razvoja prezentacijskog sloja aplikacije kao samostalnog programskog rješenja koje se izvodi na klijentu. Razmotrimo usporedbu jednostavnog primjera višestranične arhitekture i jednostavnog primjera jednostranične arhitekture na sljedećoj slici (Slika 2.1):



Slika 2.1 Pojednostavljena usporedba primjera višestranične i primjera jednostranične klijent-poslužitelj arhitekture internetske aplikacije

Za razliku od višestraničnih aplikacija, gdje svaki upit klijenta rezultira pripremanjem i prikazivanjem nove stranice, u jednostraničnim aplikacijama klijent od poslužitelja zahtijeva potrebne prezentacijske dokumente samo u prvom upitu. Kod tog inicijalnog upita, klijent dobiva HTML dokument, te sav potreban CSS i JavaScript za dinamično prikazivanje sadržaja.

Daljnja komunikacija između klijenta i poslužitelja odvija se međusobnom razmjenom podataka. Ta razmjena implementirana je korištenjem XHR API-ja, programskog sučelja kojim okruženje unutar internetskog preglednika JavaScript programu omogućava asinkrono slanje zahtjeva preko HTTP protokola. Takvo korištenje XHR API-ja naziva se AJAX, čiji akronim dolazi od riječi *asynchronous JavaScript and XML*. Unatoč nazivu, podatci ne moraju biti u XML formatu, te je najčešći oblik u kojem se podatci razmjenjuju JSON.

Bitno je napomenuti kako prikazana slika (Slika 2.1) predstavlja primjer implementacije obiju arhitektura. Poslužiteljski programi su u pravilu puno kompleksniji, a i klijentske aplikacije moguće je implementirati na razne načine. Moguće je čitav ili samo djelomičan dio prezentacijskog sloja prebaciti na klijenta. Poslužitelj klijentu može slati samo podatke ili kombinaciju HTML-a i podataka, pri čemu podatci ne moraju biti u JSON obliku. Ostatak rada bavi se isključivo pristupom u kojem klijent jednom zahtijeva sve potrebne

prezentacijske podatke, a kasnije s poslužiteljem komunicira razmjenom podataka koristeći tehnologije AJAX i JSON, pri čemu se pretpostavlja kako je poslužiteljski API implementiran poštujući REST principe.

Jednostavnosti radi, kada se u ostatku rada spominju pojmovi *jednostranična aplikacija* ili *klijentska aplikacija*, misli se na klijentsku aplikaciju izvedenu kao samostalno programsko rješenje s prethodno opisanom arhitekturom. Riječi *poslužiteljska aplikacija* ili *API servis* odnose se na program s kojim klijentska aplikacija razmjenjuje podatke.

2.1.1. Prednosti jednostraničnih aplikacija

U nastavku teksta opisane su neke prednosti korištenja jednostranične arhitekture:

- Brzina izvršavanja
- Interaktivnost
- Razdvajanje kôda
- Razdvajanje domenskog znanja

Brzina izvršavanja

Kao što je spomenuto u prethodnom odlomku, jedan od glavnih motivatora za razvoj jednostraničnih aplikacija jest brzina izvršavanja. Budući da klijent samostalno reagira na korisničke akcije koje se tiču prezentacije, manje upita se šalje poslužitelju. Poslužitelj je manje opterećen jer se ne bavi kontinuiranim pripremanjem i posluživanjem prezentacijskih dokumenata. Drugim riječima, nakon inicijalnog posluživanja prezentacijskih dokumenata i skripti, između klijenta i poslužitelja izmjenjuju se samo podatci, što smanjuje količinu internetskog prometa. Prvi upit će trajati nešto dulje, ali nakon toga korisnicima će aplikacija raditi brže, što može biti vrlo važan faktor, pogotovo u uvjetima gdje korisnici aplikacije imaju lošu internetsku vezu.

Interaktivnost

Usko povezana uz brzinu izvršavanja jest i količina interaktivnosti koju korisnik dobiva od aplikacije. Budući da ne mora čekati osvježavanje stranice, korisnik brže dobiva reakcije od strane aplikacije, te prividno ima osjećaj kao da koristi aplikaciju na računalu, a na u internetskom pregledniku. Brzina izvršavanja i interaktivnost zajedno uvelike doprinose rastu korisničkog zadovoljstva aplikacijom (Scott, 2016).

Razdvajanje kôda

Prednosti jednostraničnih aplikacija ne odnose se samo na korisničko iskustvo. Jedan od bitnih doprinosa jednostranične arhitekture jest činjenica kako su klijentska i poslužiteljska aplikacija međusobno agnostične što se tiče tehnologija u kojima su izvedene. Drugim riječima, klijentska aplikacija ne ovisi o tehnologijama u kojima je izveden poslužitelj, i obratno, sve dok su obje strane u mogućnosti razmjenjivati HTTP poruke.

Takva neovisnost u vezi tehnologija omogućava razdvajanje poslužiteljskog i klijentskog kôda. Prilikom razvijanja i održavanja velike aplikacije, razdvajanje je vrlo korisno jer omogućava paralelan rad na zasebnim repozitorijima kôda. Nakon dogovora oko strukture poruka koje će se razmjenjivati, programer poslužiteljske aplikacije i programer klijentske aplikacije mogu istovremeno razvijati istu funkcionalnost.

Razdvajanje domenskog znanja

Osim što se resursi tvrtke bolje iskorištavaju razdvajanjem klijentskog i poslužiteljskog kôda, dodatna prednost jest i razdvajanje domenskog znanja programera. Razumije se da je korisno i poželjno imati širok spektar znanja o svim tehnologijama za razvoj internetskih aplikacija, ali budući da je tu riječ o sve većem i kompliciranijem sustavu, teško je biti visoko stručan u svim tehnologijama.

Razdvajanjem domenskog znanja smanjuje se pritisak na programerima da moraju savršeno znati implementirati čitavu aplikaciju. Programer poslužiteljske aplikacije ne mora znati sve detalje oko implementacije klijentske aplikacije i obratno. Od obje strane se očekuje da zajedno rade na implementaciji neke funkcionalnosti, ali su visoko stručni u ograničenijem području.

2.1.2. Nedostatci jednostraničnih aplikacija

Neki nedostatci kod razvoja jednostraničnih aplikacija:

- SEO problematika
- Količina domenskog znanja
- Promjenjivost vanjskih biblioteka

SEO problematika

Unatoč tome što moderni SEO alati postaju sve uspješniji u indeksiranju jednostraničnih aplikacija, većina alata još uvijek puno uspješnije radi s višestraničnim aplikacijama u

kojima svaki HTML dokument ima jedinstven URL. Moguće je kreirati zasebne stranice za *SEO botove* ali ako je optimizacija internetske aplikacije za pretraživače vrlo bitna stavka za neki proizvod, jednostranična arhitektura možda nije zadovoljavajuće rješenje.

Količina domenskog znanja

Ovisno o programskom proizvodu i veličini tima koji taj proizvod razvija i održava, razdvajanje kôda i domenskog znanja opisanih u prethodnom odlomku mogu biti ili pozitivni ili negativni. Ukoliko je riječ o malenom timu koji razvija jednostavno programsko rješenje, te ako u timu ne postoji raspodjela programera na programere klijentske i poslužiteljske aplikacije, jednostranična arhitektura vjerojatno nije najbolji pristup.

Za razvoj kvalitetne jednostranične aplikacije potrebna je stručnost i razumijevanje JavaScripta, radnog okvira ili biblioteke koja se koristi za razvoj, te korištenje određenih pomoćnih alata. Ako je za neku internetsku aplikaciju dovoljna minimalna interakcija s DOM elementima, nema potrebe za uvođenjem novih tehnologija, odnosno klasična višestranična arhitektura s manjim brojem klijentskih skripti je sasvim dostatna. U tom slučaju se od programera ne očekuje da su stručni i u klijentskim i poslužiteljskim alatima, već sve slojeve aplikacije mogu implementirati u istom alatu.

Promjenjivost vanjskih biblioteka

U prethodnom odlomku spomenute su JavaScript biblioteke i radni okviri koji se koriste za razvoj klijentskih aplikacija. Moderni JavaScript odlikuje raznim karakteristikama koje ga čine prikladnim za razvoj stabilnih i pouzdanih aplikacija. Iako je moguće čitavu klijentsku aplikaciju napisati u čistom JavaScriptu, u praksi se za kreiranje prezentacijskih elemenata, za upravljanje tim elementima i za čuvanje informacija o stanju pojedinog elementa koriste već isprogramirane biblioteke ili radni okviri. U trenutku pisanja rada, najčešće korišteni alati su biblioteka React, te radni okviri Angular i Vue.

Klauzinski navodi kako je JavaScript danas među najpopularnijim jezicima, ali je prethodno bio smatran igračkom za pisanje malih skripti. U dijelu svoje knjige čak opisuje JavaScript kao nekoć naporan jezik za podizanje prozora za upozoravanje (Klauzinski, 2016). Iako su ti navodi možda malo ekstremni, ima istine u tome da dio programerske zajednice JavaScript ne smatra dovoljno ozbiljnim jezikom. Sukladno tome, alati za razvoj jednostraničnih aplikacija ponekad se smatraju pretjerano promjenjivima i nedovoljno stabilnima za razvoj i dugoročno održavanje aplikacija, što neke tvrtke i timove programera može odvratiti od razvoja jednostraničnih aplikacija.

Zaključno, neovisno o objektivnim prednostima i nedostatcima jednostranične arhitekture, odabir arhitekture i tehnologija za razvoj programskog rješenja ovisi o samom programskom rješenju te o ljudima koji će to rješenje razviti i održavati. Za neke tvrtke i proizvode jednostranična arhitektura je dobar izbor, dok za neke druge proizvode ili tvrtke takva arhitektura nije prihvatljiva ili je, uvjetno rečeno, nepotrebno kompleksna.

2.2. Modularizacija klijentskog kôda

Na službenom blogu tvrtke Mozilla, Jason Orendorff, jedan od programera JavaScripta, navodi kako se opsežnost JavaScript programa značajno povećala, te je za uspješan razvoj i održavanje JavaScript programa neophodan sustav modularizacije. Modularizacija se odnosi na raspodjeljivanje smislenih cjelina kôda u različite mape i datoteke, uz mogućnost da se pojedine cjeline mogu međusobno pozivati te da se sav taj raspodijeljeni kôd u konačnici smisleno objedini i uspješno dostavi klijentu ili izvrši na računalu.[1]

Bez modularizacije, moguće je sav JavaScript napisati u jednoj datoteci ili napisati proizvoljan broj različitih skripti, koje se pomoću zasebnih *script* označnih polja unose u istu ili u različite HTML stranice. Takav pristup otvara određene probleme prilikom održavanja. Naime, unutar JavaScript skripti, sve varijable definirane izvan tijela funkcije su globalne varijable, te je redoslijed referenciranja skripti veoma bitan. Drugim riječima, programeri moraju paziti da ne bi greškom modificirali neku vrijednost o kojoj ovisi druga skripta, te prilikom korištenja funkcija i varijabli iz drugih skripti moraju paziti na to kojim redoslijedom je koja skripta učitana u HTML dokument te kako se točno pojedina varijabla ili funkcija zove.

Možemo zaključiti kako održavanje velikog broja skripti za jednu aplikaciju lako može postati veoma težak zadatak. Ukoliko se prezentacijski sloj aplikacije implementira kao zasebna klijentska aplikacija, ta klijentska aplikacija u pravilu će biti veliki repozitorij JavaScript kôda, za čiji je razvoj i održavanje prijeko potrebno kvalitetno modularizirati kôd.

Od 2015. godine, JavaScript podržava module na razini jezika, te je sve veća podržanost JavaScript modula od strane internetskih preglednika.[2] Jednostavnim jezikom, modul je zasebna JavaScript datoteka, dio kôda čije su varijable i funkcije ograničene na područje tog modula. Unutar modula mogu se izvoziti elementi, bilo da se žele izvesti svi ili samo pojedini

[1] Mrežni izvor, https://hacks.mozilla.org/2015/08/es6-in-depth-modules/
[2] Mrežni izvor, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

elementi. Ako je neki element izvezen ključnom riječju *export*, taj se element može koristiti u drugim modulima ključnom riječju *import*. Izvoženje može biti imenovano (engl. *named export*) ili jedan modul može izvoziti jednu vrijednost ključnom riječju *export default.*

Razmotrimo sljedeći prikaz korištenja JavaScript modula s imenovanim izvozom (Kôd 2.1)

```
// DATOTEKA stringUtil.js
const defaultFallback = '-';

export const safeToString =
 (value, fallback = defaultFallback) =>
   value != null ? String(value) : fallback;

export const safeIsNullOrEmpty = (value?) =>
  safeToString(value, '').trim() === '';

// DATOTEKA validate.js
import { safeIsNullOrEmpty } from '../../utils/stringUtil';

const defaultFallback = 2;

export const validateCustomProxyFile = (proxyFile) => {
  if (safeIsNullOrEmpty(proxyFile)) {
    throw new Error('Custom proxy file cannot be empty.');
  }
// ...
}
```

Kôd 2.1 Prikaz korištenja JavaScript modula imenovanim izvozom (engl. *named export*)

Ovakav pristup rješava probleme koji mogu nastati korištenjem istih globalnih varijabli u različitim skriptama, te uvelike pojednostavljuje korištenje varijabli ili funkcija u različitim dijelovima kôda. U danom primjeru modul validate.js iz modula stringUtil.js može koristiti samo one vrijednosti koje je modul stringUtil.js prefiksirao ključnom riječju *export*. Nadalje, to što oba modula koriste varijablu `defaultFallback` ne može uzrokovati neočekivane posljedice, budući da su obje varijable ograničene na opseg (engl. *scope*) modula unutar kojeg su deklarirane.

Modularizacija kôda dodatno je omogućila brojnim alatima koje programeri koriste u svakodnevnom radnom okruženju da povećaju svoju produktivnost funkcionalnostima poput

dovršavanja rečenica koda, podcrtavanja neispravno referenciranih modula, optimizacije i kreiranja objedinjene skripte i slično. Kratki pregled nekih alata bit će dan u odlomku 2.4.

2.3. Statička tipizacija klijentskog kôda

JavaScript je dinamički tipiziran skriptni jezik, što znači da se tip varijable određuje tek prilikom njenog izvršavanja. U pojedinu memorijsku lokaciju, odnosno varijablu deklariranu ključnom riječju *var*, moguće je staviti bilo koji dozvoljeni tip podatka, te je kasnije dozvoljeno na tu istu lokaciju pospremiti neki drugi tip podatka.

Od specifikacije ECMAScript 2015, u JavaScriptu je moguće razlikovati *const* i *let* varijable koje donekle rješavaju problematiku. *const* je tip varijable koji se prilikom deklaracije mora i inicijalizirati te se nakon toga vrijednost ne smije mijenjati. U slučaju vrijednosnih varijabli, *const* je vrijednosna konstanta, a u slučaju referentnih vrijednosti, *const* je referentna konstanta, odnosno vrijednost *const* varijable mora uvijek pokazivati na istu lokaciju u memoriji, iako se objekt na koji ona pokazuje smije modificirati. *let* je sličan *var* tipu varijable po tome što se vrijednost smije mijenjati, ali je razlika u tome što se unutar opsega u kojem je deklarirana smije deklarirati samo jednom. Također, ako je izvan i unutar nekog opsega deklarirana varijabla istog imena, koristeći *var* varijablu to će biti ista varijabla, dok će kod korištenja *let* varijable to biti dvije zasebne varijable.

Unatoč tome što moderni JavaScript pruža sigurnije korištenje varijabli, održavanje velikog repozitorija kôda teško je bez potpunog razumijevanja tipova podataka koje pojedina varijabla sadrži ili funkcija prima i vraća. U klasičnom JavaScriptu moguće je deklarirati funkciju koja prima tri parametra, a zatim ju pozvati s niti jednim ili deset parametara. Kao što je već spomenuto, takve karakteristike čistog JavaScripta ne predstavljaju problem za male programe, ali za velike programe otežavaju proces razvoja, razumijevanja i održavanja.

Problem je moguće riješiti vrlo pedantnom dokumentacijom pojedinih varijabli i funkcija, ali to može rezultirati pretjerano velikom količinom komentara. Kako bi se omogućila statička tipizacija klijentskih aplikacija, programeri su razvili razne alate i čak zasebne jezike za pisanje klijentskih programa. U trenutku pisanja rada, najčešće korištene tehnologije za postizanje statičke tipizacije su Flow, sustav anotacija tipova, te TypeScript, nadskup JavaScripta koji dozvoljava opcionalno određivanje tipova.

Od ostalih tehnologija valja spomenuti CoffeeScript i Dart, zasebne jezike za pisanje klijentskih aplikacija. Rad neće u detalje opisivati pojedinu tehnologiju, već će ukratko prikazati TypeScript, budući da je isti korišten u razvoju podesivog *mock* poslužitelja.

2.3.1. TypeScript

Na službenoj stranici, TypeScript je definiran kao JavaScript za ozbiljne aplikacije. Razvijen je 2012. godine u tvrtki Microsoft, a glavni arhitekt mu je Anders Hejlsberg, koji je i glavni arhitekt jezika C#. Objavljen je pod Apache 2.0. licencom otvorenog kôda, te je u trenutku pisanja rada najnovija inačica 3.7.[3]

U TypeScript Programming Language knjizi, navodi se kako opsežan JavaScript program lako postaje neuredan, kako je teško takav program održavati i ponovno iskorištavati. Nedostatak objektno orijentiranih principa, nedostatak statičke tipizacije i nedostatak otkrivanja grešaka prilikom prevođenja (engl. *compile time*) čine JavaScript nedovoljno snažnim jezikom za razvijanje ozbiljnih aplikacija. To je ujedno i glavna motivacija zašto je TypeScript razvijen (TypeScript Publishing, 2019).

TypeScript je objektno orijentirani statički tipizirani jezik koji se mora prevoditi, a ujedno i set alata. Sastoji se od jezika kao nadskupa JavaScripta, sastoji se od TypeScript prevoditelja te jezičnih servisa (engl. *TypeScript language service*) koji omogućuju programima za uređivanje teksta da vrše funkcionalnosti poput dovršavanja rečenica, praćenje poveznica, refaktoriranja i slično.

Za razliku od klijentskih jezika CoffeeScript i Dart, koji za izvršavanje zahtijevaju zasebno razvojno okruženje, TypeScript je superset ili nadskup JavaScripta što znači da ne treba zasebno razvojno okruženje te je svaki JavaScript program ispravan TypeScript program (TypeScript Publishing, 2019). Nadalje, TypeScript je unutar jezika ugradio napredne funkcionalnosti koje su planirane za JavaScript ali još nisu dio aktualne ECMAScript specifikacije. Na taj način, programeri klijentskih aplikacija mogu koristiti napredne mogućnosti jezika, ali će se program kasnije prevesti u obični i standardizirani JavaScript. Prilikom prevođenja TypeScripta moguće je podesiti željenu verziju JavaScripta u koji će program biti preveden, pri čemu je zadana vrijednost ES3 (ECMAScript 3).

Tipovi su u TypeScriptu opcionalni, te se označavaju tako što se nakon imena varijable napiše dvotočka pa ime tipa koji će varijabla sadržavati. Ako se tip ne dodijeli eksplicitno,

[3] Mrežni izvor: https://www.typescriptlang.org/index.html

TypeScript će sam pretpostaviti o kojem je tipu riječ (engl. *type inference*). TypeScript nudi mogućnost tipiziranja varijabli, parametara i povratnih vrijednosti funkcija. Moguće je kreirati generičke funkcije, kao i preopterećenja funkcija, odnosno više puta deklarirati istoimenu funkciju s različitim potpisom. Moguće je definirati vlastite tipove, enume i sučelja. Bitno je napomenuti kako u TypeScriptu sučelje nije ograničeno na opisivanje ponašanja (kao što je slučaj u nekim jezicima), već se koristi za opis izgleda objekta, slično klasi. Razlika je u tome što će se nakon prevođenja od klasa generirati JavaScript funkcije, dok je sučelje samo indikator tipa koji razumiju TypeScript jezični servisi. Drugim riječima, korištenjem sučelja omogućava se statička tipizacija i provjera ispravnosti kôda tijekom njegova pisanja ili prevođenja, ali bez prevođenja u ekvivalentni JavaScript kôd.

Sustav tipova je vrlo bogat u TypeScriptu. Uz osnovne tipove kakve nalazimo u JavaScriptu, podržava određene kombinacije tipova kao operacije nad skupovima, podržava uvjetne tipove, a u jezik su ugrađeni i pomoćni tipovi koji omogućavaju transformacije tipova. U sljedećem primjeru prikazano je korištenje nekih tipova (Kôd 2.2):

```
type CustomType = 'a' | 'b' | 'c';

interface ICustomObject {
    propertyA?: number;
    propertyB: string;
    propertyC: CustomType;
    propertyD: (prop: number | null) => string;
}

const customObj: ICustomObject = {
    propertyB: 'B',
    propertyC: 'c',
    propertyD: ((prop: number | null) => 'test')
}
const functionA = <T, K extends keyof T>(prop1: T, prop2: K)
      => console.log(`${prop2}: ${prop1[prop2]}`);

functionA<ICustomObject, 'propertyB'>(customObj,'propertyB');
// propertyB: B

const objOmitD: Omit<ICustomObject, 'propertyD'> = {
    propertyA: 1,
```

```
    propertyB: 'B',
    propertyC: 'c',
}
```

Kôd 2.2 Primjer korištenja nekih TypeScript tipova

Iz primjera je vidljivo kako proizvoljni tip možemo definirati kao uniju određenih vrijednosti korištenjem znaka „|“. Isti simbol možemo koristiti i za dodjeljivanje unije tipova određenoj varijabli. U navedenom primjeru, parametar `prop` može biti tipa „number“ ili tipa „null“. Nadalje, prikazan je primjer korištenja sučelja za opis izgleda objekta, pri čemu pojedini parametar može biti bilo koji dozvoljeni tip, uključujući druga sučelja, klase ili potpise funkcije. Opcionalne parametre moguće je definirati korištenjem znaka „?“ nakon imena varijable, a prije dvotočke koja prethodi oznaci tipa.

Korištenjem generičkih i pomoćnih tipova, moguće je definirati kompleksne vrste tipova. U prethodnom primjeru, funkcija s nazivom `functionA` prima dva generička tipa. Vrijednost drugog generičkog tipa mora biti ime bilo kojeg parametra definiranog u prvom generičkom tipu. Varijabla nazvana `objOmitD` mora poprimiti tip jednak sučelju `ICustomObject`, ali bez parametra koji se zove `propertyD`. Ukoliko bismo pokušali u varijabli `objOmitD` definirati parametar s nazivom `propertyD`, dobili bismo iznimku.

Kao objektno orijentirani jezik, TypeScript omogućava pisanje programa po objektno orijentiranoj paradigmi naprednije od JavaScripta. Razmotrimo sljedeći primjer (Kôd 2.3):

```
class Student {
    protected isProcrastinator: boolean;
    constructor(procrastinator: boolean) {
        this. isProcrastinator = procrastinator;
    }
    procrastinate() {
      this.isProcrastinator ? console.log('Yay!')
            : console.log('Work time!');
    }
}
enum Deadline {
    FAR_AWAY = 'FAR_AWAY',
    NEAR = 'NEAR',
    TOMORROW = 'TOMORROW',
}
```

```
interface IBatchelorThesisWriter {
    writeThesis: (deadline: Deadline) => void;
}
class LazyStudent extends Student implements
IBatchelorThesisWriter {
    constructor() { super(true); }
    writeThesis(deadline: Deadline) {
        deadline === Deadline.TOMORROW
            ? console.log('Dear batchelor thesis...')
            : console.log('Enough time left...');
    }
}
class DiligentStudent extends Student implements
IBatchelorThesisWriter {
    constructor() { super(false); }
    writeThesis(deadline: Deadline) {
        deadline !== Deadline.FAR_AWAY
            ? console.log('Dear batchelor thesis...')
            : console.log('Really enough time left...`);
    }
}
const lazyStudent = new LazyStudent();
const diligentStudent = new DiligentStudent();
lazyStudent.procrastinate(); // Yay!
diligentStudent.procrastinate(); // Work time!
lazyStudent.writeThesis(Deadline.NEAR);
// Enough time left...
diligentStudent.writeThesis(Deadline.NEAR);
// Dear batchelor thesis...
```

Kôd 2.3 Primjer korištenja nekih od objektno orijentiranih koncepta u jeziku TypeScript

Primjer je vrlo jednostavan, ali prikazuje deklariranje i kreiranje instanci klase, nasljeđivanje klase i implementacije sučelja. Za razliku od klasičnog JavaScripta i prototipnog nasljeđivanja, u TypeScriptu su objektno orijentirani koncepti bogatiji i realizirani puno sličnije popularnim jezicima za pisanje objektno orijentiranih programa, poput jezika Java ili C#.

Zaključno, možemo reći da TypeScript olakšava razvoj i održavanje velikih klijentskih aplikacija. Poznavajući tipove varijabli te poznavajući tipove parametara funkcija i povratnih vrijednosti funkcija, programer lakše razumije i koristi tuđi kôd, te isto tako piše kôd

razumljiviji drugima. Korištenjem dodatnih alata za uređivače teksta, poput TypeScript priključka za program Visual Studio Code, Visual Studio, Sublime Text i druge, programer ima vrlo napredne mogućnosti dovršavanja linija koda, upozoravanja na pogreške prije i za vrijeme prevođenja, te mu je uvelike olakšan proces refaktoriranja kôda.

Također, budući da donosi pristup objektno orijentiranim principima drugačije od JavaScripta, a puno sličnije onome u jezicima C# i Java, TypeScript je potencijalno ugodniji za korištenje programerima koji dolaze iz takve jezične pozadine.

2.4. Radno okruženje

U svojoj knjizi Pragmatic Programmer, autori Hunt i Hurst posvetili su jedno poglavlje radnom okruženju. Ističu važnost radnog okruženja te daju savjete kako maksimizirati učinkovitost upotrebom ispravnih alata te automatizacijom čestih radnji. Navode kako programeri pripravnici često nauče raditi u isključivo jednom alatu, poput snažnog IDE[4] programa te smatraju kako je to loše za opću produktivnost. Autori tvrde kako je za povećanje produktivnosti potrebno poznavati naredbeni redak, koristiti postojeće alate i kreirati vlastite alate za automatizaciju (Hunt i Hurst, 2019).

Kroz ovo poglavlje rada dosta puta su spomenuti alati i to s razlogom. Razvijanje modernih JavaScript programa, bilo za internetske preglednike ili za računala u okruženju Node.js, podrazumijeva korištenje širokog spektra različitih alata koji zajedno teže sličnom cilju, optimizaciji programerske učinkovitosti automatizacijom posla.

Modularno posloženi kôd zahtijeva čitače modula (engl. *module loaders*) kako bi se pojedini moduli međusobno ispravno referencirali. Ukoliko programer želi od velikog broja modula u konačnici kreirati jedan skupni modul, može to učiniti alatima za spajanje modula (engl. *module bundlers*), a koji nude i razne oblike optimizacije konačnog kôda ovisno o definiranim postavkama. Jedan od najpoznatijih alata za spajanje modula jest webpack, veoma podesiv alat za koji postoje brojni nadodatci (engl. *plugins*) za optimizaciju kôda.

Nadalje, modularnost je rezultirala olakšanjem procesa korištenja tuđeg koda. Umjesto da se u HTML stranice referenciraju vanjske skripte na CDN poslužiteljima ili da se takve skripte drže na poslužitelju, programerima je omogućen sustav preuzimanja i instaliranja

[4] IDE – engl. *integrated development environment*, vrsta računalnog programa za pisanje kôda, sastoji se od uređivača teksta te dodatnih funkcionalnosti poput prevoditelja, alata za kreiranje izvršne verzije programa i slično

vanjskih modula odnosno biblioteka uz pomoć programa za upravljanje paketima (engl. *packet manager*). Najpoznatiji upravljači paketima su npm i yarn, a velika prednost kod takvog korištenja biblioteka jest što sami kôd vanjskih modula nije statički dio kôda aplikacije. Drugim riječima, dovoljno je samo čuvati podatak o tome koji su paketi potrebni i u kojoj verziji, što repozitorij čini manji i preglednijim. Prilikom puštanja programa u produkciju ili pokretanja programa na lokalnom računalu, potrebno je pokrenuti naredbu koja će ovisno o navedenim ovisnostima o vanjskim modulima preuzeti sve potrebne module.

Iako su opisane funkcionalnosti možda ustaljena praksa i norma za neke jezike i razvojna okruženja, za pisanje JavaScript programa riječ je o relativno novim funkcionalnostima i alatima, što od programera naučenih na tradicionalno korištenje JavaScripta kao jednostavnog jezika za pisanje malih skripti zahtijeva dodatno učenje i razumijevanje.

Nadalje, moderne tehnologije omogućile su programerima klijentskih aplikacija da dio brige oko kompatibilnosti s internetskim preglednicima ostave na alatima. Omogućeno im je da pišu uredan i moderan kôd koristeći napredne mogućnosti JavaScripta, a brigu o tome hoće li se on moći izvršiti na starim verzijama internetskih preglednika ostave na posebnim prevoditeljima koji prevode JavaScript u JavaScript, ali iz novije verzije ECMAScript specifikacije u stariju specifikaciju jezika (engl. *transpiler*). Među popularnijim JavaScript *transpilerima* jest Babel, a istu funkcionalnost sadrži i prethodno spomenuti TypeScript prevoditelj. Programerima je olakšano i pisanje CSS-a postojanjem alata koji automatski prefiksiraju određena CSS pravila kako bi ista ispravno radila u različitim preglednicima. Primjer takvog alata jest Autoprefixer, koji se može koristiti kao nadodatak za prije spomenuti webpack alat.

Webpack također nudi i webpack-dev-server, alat koji omogućuje pokretanje jednostranične aplikacije na lokalnom poslužitelju, a zanimljivo je da je unutar konfiguracije moguće zadati i adresu na koju programer želi preusmjeravati promet. Tako je za jednostavne potrebe samostalnog razvoja klijentske aplikacije omogućeno da programer ne mora prevoditi i pokretati poslužiteljski kôd, već može sve upite preusmjeravati na željeni lokalni poslužitelj.

U želji za povećanjem produktivnosti automatizacijom česte radnje prilikom samostalnog razvoja klijentskih aplikacija, u sklopu ovog rada razvijen je podesivi alat za kreiranje i pokretanje *mock* i preusmjeriteljskog poslužitelja. Motivacija za razvoj rješenja, kao i implementacija opisani su u idućem poglavlju.

3. Razvoj podesivog mock poslužitelja

U ovom poglavlju predstavit će se motivacija za razvoj podesivog *mock* poslužitelja. Navest će se potrebe korisniki te će se prikazati kako je rješenje povezano s prethodnim poglavljem, odnosno kako olakšava razvoj klijentskih aplikacija automatizacijom radnog okruženja, koristeći pri tome prednosti modularizacije kôda i statičke tipizacije. Detaljnije će se prikazati sama implementacija rješenja, a za kraj će se poglavlje osvrnuti na korištenje statički tipiziranih podataka u automatiziranim testovima, na nedostatke rješenja, kao i mogućnosti proširivanja.

3.1. Motivacija za razvoj rješenja

Kao što je spomenuto u prethodnom poglavlju, razvoj klijentskih aplikacija može se činiti dosta kompleksnim po pitanju potrebnih alata i tehnologija za postavljanje radnog okruženja. No, takvi alati postoje s razlogom i uvelike olakšavaju proces razvoja. Među tim alatima nalaze se i programi za imitiranje poslužitelja, kako programeri klijentskih aplikacija ne bi morali konstantno pokretati poslužiteljski kôd. Na tržištu postoje alati za kreiranje *mock* poslužitelja, poput JSON Server aplikacije kojom je vrlo brzo moguće postaviti čisti JSON poslužitelj. Aplikacija Postman također ima mogućnost kreiranja *mock* servera, iako je pretežito namijenjena testiranju REST servisa bez klijenta. Korištenjem webpack-dev-servera prilikom razvoja klijentskih aplikacija moguće je automatski preusmjeravati zahtjeve na željeni poslužitelj u razvojnom okruženju.

Navedeni alati rješavaju dio problema, ali ne i sve probleme imitacije poslužitelja. U nastavku teksta predstavljene su potrebe korisnika u obliku korisničkih priča, analiza potreba korisnika te predloženo rješenje za kreiranje i pokretanje *mock* poslužitelja.

3.1.1. Potrebe korisnika

U nastakvu teksta prikazane su potrebe korisnika u obliku korisnčkih priča. Korisničke priče predstavljaju stvarne potrebe programera tvrtke Oradian d.o.o., ali se mogu primijeniti općenito na programere koji rade na razvoju i održavanju velikih klijentskih aplikacija. Budući da je u svim korisničkim pričama korisnik programer klijentske aplikacije, skup riječi „Kao programer klijentske aplikacije“ bit će skraćen na „Kpka“.

1. Kpka, želim mogućnost rada na klijentskoj aplikaciji bez pokretanja poslužiteljske aplikacije, kako bi proces razvoja bio brži.
2. Kpka, želim program koji će posluživati *mock* podatke, kako bih mogao raditi na klijentskoj aplikaciji bez pokretanja poslužiteljske aplikacije.
3. Kpka, želim podesiv program koji može programski posluživati *mock* podatke, kako bih imao veću kontrolu nad podatcima koje dobivam.
    a. Kpka, želim program za *mock* poslužitelja koji ovisno o zahtjevu može donijeti odluku o tome koje *mock* podatke će poslužiti.
    b. Kpka, želim program za *mock* poslužitelja koji uz minimalno mijenjanje postavki može za iste zahtjeve posluživati različite setove *mock* podataka.
4. Kpka, želim program koji će preusmjeravati promet klijentske aplikacije na lokalni ili udaljeni poslužitelj, kako bih mogao raditi na klijentskoj aplikaciji bez pokretanja poslužiteljske aplikacije nakon promjena u kôdu.
5. Kpka, želim podesiv alat kojim ću lako kreirati, održavati i pokretati *mock* poslužitelja, kako ne bih morao ručno raditi program za *mock* poslužitelja za svaku novu situaciju.
    a. Kpka, želim alat koji će od stvarnog HTTP prometa generirati *mock* podatke koje ću kasnije koristiti za *mock* poslužitelja, kako ne bih morao ručno kreirati puno dokumenata.
    b. Kpka, želim alat koji će kombinirati preusmjeravanje i posluživanje *mock* podataka, kako bih mogao raditi na klijentskoj aplikaciji preusmjeravanjem prometa, ali posluživanjem *mock* podataka ukoliko pojedina funkcionalnost još ne postoji na poslužitelju.
6. Kpka, želim program za *mock* poslužitelja koji je svjestan tipova *mock* podataka, kako bih bio siguran da su *mock* podatci u ispravnom obliku te da nisam napravio grešku prilikom pisanja *mock* podataka.

3.1.2. Analiza potreba korisnika

Svim korisničkim pričama zajednički je korisnik, odnosno programer klijentske aplikacije koji želi mogućnost rada na klijentskoj aplikaciji bez pokretanja poslužiteljske aplikacije, kako bi brže mogao vidjeti stanje programa na kojem radi nakon učinjenih promjena. Za tu potrebu želi program koji će posluživati *mock* podatke, želi program koji će preusmjeravati

promet na poslužiteljsku aplikaciju (lokalno ili udaljeno), te želi program koji je u stanju raditi i preusmjeravanje i posluživanje *mock* podataka.

Korisnik želi mogućnost provjere strukture *mock* API odgovora i podataka, te želi mogućnost brze promjene postavki za pokretanje *mock* poslužitelja. Ponajviše, korisnik želi alat koji objedinjuje navedene korisničke priče, te programeru omogućava da na automatiziran način stvori, modificira i uz promjenjive postavke pokreće program koji će donekle zamijeniti poslužiteljsku aplikaciju.

Te potrebe proizlaze iz problematike kontinuiranog rada na velikoj poslovnoj klijentskoj aplikaciji. Za omanju aplikaciju kao i aplikaciju gdje se poslužiteljska aplikacija brzo prevodi i izvršava, nema tolike potrebe za ovakvim alatom. No, za održavanje programskog proizvoda čija se poslužiteljska aplikacija dugo prevodi, kojoj treba puno vremena da se pokrene nakon promjena u kôdu i/ili nije lako automatski bazu podataka popuniti podatcima, vrlo je bitno da programer klijentske aplikacije može raditi bez poslužiteljske aplikacije.

3.1.3. Predloženo rješenje

Predloženo rješenje jest razvoj biblioteke koja programeru pruža sučelje (API) za generiranje statički tipiziranih podataka snimanjem prometa za vrijeme preusmjeravanja, te sučelje za pokretanje *mock* ili preusmjeriteljskog poslužitelja, uz opciju da se kod preusmjeravanja posluži *mock* podatak ukoliko preusmjeritelj dobije odgovor s HTTP statusom 404 (resurs nije pronađen). Predloženo je i razvijanje pomoćne aplikacije koja koristi takvu biblioteku te pruža korisničko sučelje za pokretanje *mock* poslužitelja uz unos proizvoljnih parametara.

3.2. Implementacija rješenja

Sukladno predloženom, razvijen je TypeScript program u obliku biblioteke koja prema korisniku nudi sve potrebne funkcije i sučelja (engl. *interface*) za stvaranje i pokretanje *mock* poslužitelja uz poznavanje tipova podataka. U trenutku pisanja rada, prototip biblioteke je objavljen u GitHub repozitoriju te se može preuzeti i koristi. Valja spomenuti kako će nakon određenog perioda korištenja od prototipa biti razvijena biblioteka spremna za objavu u npm repozitoriju otvorenog kôda. Korisnici će tada rješenje moći koristiti izravnim dodavanjem biblioteke u svoj program kao poveznice (engl. *dependency*) na odgovarajući npm repozitorij.

Budući da rješenje nije specifično za pojedinačnu klijentsku ili poslužiteljsku aplikaciju, implementirano je poštujući modulariziranje kôda, te je u stanju raditi s bilo kojom klijentskom aplikacijom koja s poslužiteljem razmjenjuje podatke koristeći AJAX i JSON tehnologije, pri čemu se očekuje da je poslužitelj razvijen poštujući REST principe.

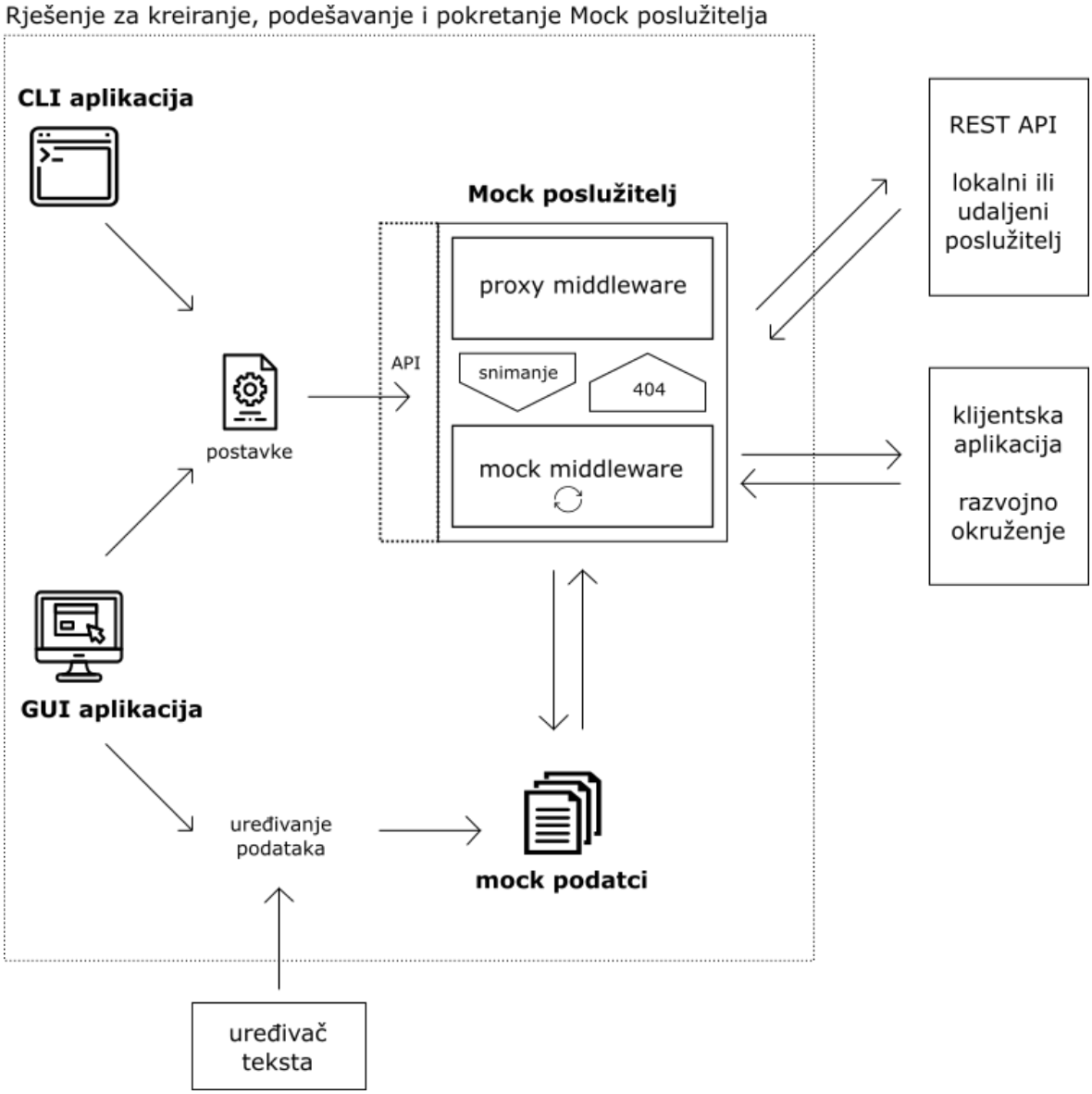
Kako bi se postigla statička tipizacija podataka, *mock* podatci pohranjuju se u obliku TypeScript datoteka. Unatoč tome što je možda neintuitivno podatke spremati u tekstualne datoteke, u ovom slučaju to je opravdano. Hunt i Hurst osvrću se na tekstualni format kao samodokumentirajući oblik podataka čije je semantičko značenje i razumijevanje neovisno o programu koji ih je generirao. Na taj način podatke je moguće razumjeti bez poznavanja konteksta u kojem su nastali (Hunt i Hurst, 2019).

Ovakav pristup pohrani podataka omogućava i ručnu manipulaciju nad podatcima kroz program za uređivanje teksta, odnosno podatci se mogu uređivati bez zasebne aplikacije s korisničkim sučeljem. Dodatna prednost jest i mogućnost verzioniranja *mock* podataka. Ukoliko korisnik želi *mock* podatke držati unutar verzioniranog repozitorija projekta u kojem koristi rješenje, tekstualni format je vrlo prikladan za praćenje promjena.

Zbog modularnosti, rješenje je također lako moguće prilagoditi da podržava i druge tipove podataka, te prilagoditi da umjesto TypeScript datoteka podatke sprema u drugačijem formatu, bilo u datoteke ili bazu podataka, ako za to ima potrebe.

Dodatno su razvijene i dvije pomoćne aplikacije kao sloj korisničkog sučelja za korištenje glavne biblioteke. Razvijena je jednostavna konzolna aplikacija za pokretanje *mock* poslužitelja iz naredbenog retka, te pomoćna aplikacija s grafičkim korisničkim sučeljem za upravljanje *mock* poslužiteljem i aktualnim podatcima. Valja naglasiti da pomoćne aplikacije nisu dio biblioteke, već je na krajnjim korisnicima odluka žele li koristiti biblioteku i korisničko sučelje, ili žele preuzeti samo biblioteku i sami kreirati pomoćno sučelje ili isključivo programski pokretati *mock* poslužitelj iz JavaScript ili TypeScript kôda.

Za predočavanje pojednostavljene arhitekture rješenja, razmotrimo sljedeću sliku (Slika 3.1):



Slika 3.1 Pojednostavljen prikaz razvijenog rješenja, Mock poslužitelj biblioteke te pomoćnih aplikacija s korisničkim sučeljem

Cjelokupno rješenje sastoji se od četiri glavna dijela. Centralni dio rješenja jest biblioteka za kreiranje i pokretanje Mock poslužitelja. Biblioteka pruža API, odnosno sučelje pomoću kojeg je moguće upravljati *mock* poslužiteljem. Biblioteka za rad zahtijeva postavke u definiranom obliku te je neovisna o tome kako će se te postavke generirati. Sukladno postavkama, kreira se poslužitelj na željenom portu i koriste se *mock* podatci sa željene lokacije. Biblioteka očekuje određenu strukturu datoteka, ali će po potrebi istu generirati na lokaciji definiranoj postavkama. Na taj način, *mock* podatci su odvojeni od same biblioteke čime se dobiva na fleksibilnosti korištenja, te možemo reći kako su *mock* podatci donekle neovisan dio rješenja.

Ovisno o postavkama, *mock* poslužitelj može raditi u nekoliko načina (engl. *mode*) što će detaljnije biti opisano u poglavlju (3.2.2). Izvan opsega rješenja, a u komunikaciji s *mock* poslužiteljem, nalaze se lokalni ili udaljeni poslužitelj prema kojem *mock* poslužitelj može preusmjeravati promet te klijentska aplikacija u razvojnom okruženju koja će koristiti *mock* poslužitelj.

Pomoćne aplikacije, CLI (engl. *command line interface*) ili konzolna aplikacija i GUI (engl. *graphic user interface*) ili aplikacija s grafičkim korisničkim sučeljem, koriste opisanu biblioteku i otvaraju dodatno sučelje prema krajnjem korisniku. Njihova primarna zadaća je od korisničkog unosa kreirati postavke te pokrenuti *mock* poslužitelj.

Aplikacija s grafičkim sučeljem dodatno pruža i sučelje za uređivanje podataka koji se koriste tijekom rada *mock* poslužitelja. Na taj način, korisnik podatke može uređivati ili preko pomoćne aplikacije, ili preko bilo kojeg programa za uređivanje teksta, poput alata kojeg koristi za pisanje kôda. Pomoćne aplikacije bit će detaljnije opisane u nadolazaćeim poglavljima (poglavlje 3.2.3 i poglavlje 3.2.4).

3.2.1. Korištene tehnologije

Cjelokupno rješenje razvijeno je u okruženju Node.js i jeziku TypeScript. Korištene su i dodatne biblioteke i radni okviri, od kojih je najbitnije spomenuti Express i React. Typescript je opisan u prethodnom dijelu rada (poglavlje 2.3.1), a u nastavku odlomka opisane su ostale ključne tehnologije.

Node.js

Node.js je okruženje (engl. *runtime environment*) za izvršavanje JavaScript programa na računalu zasnovano na V8 JavaScript pokretaču (engl. *engine*). Tijekom rada, V8 od JavaScript kôda generira apstraktno sintaktičko stablo, pretvara to stablo u *bytecode* koji zatim prevodi u strojni jezik. Pojednostavljeno rečeno, V8 direktno prevodi JavaScript u strojni jezik koristeći *just-in-time* prevođenje, pri čemu provodi i određene optimizacije prije izvršavanja. V8 napisan je u jeziku C++ i otvorenog je kôda. Razvijen je za izvršavanje JavaScript programa u internetskom pregledniku Google Chrome i ostalim preglednicima zasnovanima na Chromim-u, ali se može koristiti i izvan preglednika, odnosno može se integrirati u druge tehnologije, poput okruženja Node.js.[5]

[5] Mrežni izvor, https://v8.dev/docs

Node.js je otvorenog kôda i može se koristiti na svim najzastupljenijim operacijskim sustavima. Najčešće se upotrebljava za razvoj poslužiteljskih aplikacija, makar se može koristiti i u druge svrhe, primjerice za programiranje integriranih uređaja. Node.js se u službenoj dokumentaciji opisuje kao okruženje vođeno događajima (engl. *event-driven*) koje omogućava pisanje asinkronog kôda zahvaljujući specifičnom modeliranju I/O operacija kao operacija koje ne blokiraju daljnje izvršavanje programa.[6]

U knjizi Node.js 8 the Right Way, Wilson jednostavnim jezikom objašnjava tu arhitekturu: glavni proces može vezati određenu funkciju uz rezultat neke sistemske operacije, ali ne mora čekati rezultat, već može nastaviti s izvršavanjem programa. Okruženje će se pobrinuti za dostavljanje rezultata nakon što isti bude dostupan i aktivirati funkciju koja je taj rezultat očekivala. Takva funkcija s dobivenim rezultatom (engl. *callback function*) dolazi na kraj reda za izvršavanje funkcija. Svaka funkcija izvršava se od početka do kraja, a proces završava onda kada nema niti jedne aktivne funkcije niti funkcije koja iščekuje neki događaj. Sustav koji omogućava opisano ponašanje naziva se *event loop* (Wilson, 2018).

Od verzije 8, Node omogućava korištenje `async/await` sintakse koja pauzira izvršavanje funkcije za vrijeme čekanja rezultata neke operacije. Umjesto da se funkcija izvrši od početka do kraja, a da se uz sistemsku operaciju veže određena *callback* funkcija, moguće je odblokirati glavni proces usred tijela funkcije. Nakon što funkcija dobije rezultat i ponovno dođe na red za izvršavanje, nastavit će se izvršavati od onog dijela kôda koji je inicijalno odblokirao glavni proces.

Budući da je JavaScript jednodretveni jezik, problem dostupnosti poslužiteljske aplikacije uslijed velikog broja upita nije moguće riješiti stvaranjem paralelizma pomoću većeg broja dretvi, što je česta praksa u višedretvenim jezicima. Iz tog razloga implementiran je opisani sustav pristupa sistemskim operacijama i asinkronog obrađivanja događaja. Valja napomenuti kako je sustav opisan veoma pojednostavljeno i da sami Node.js nije jednodretveni proces, ali to je izvan opsega rada.

Express

Node.js okruženje sadrži module za mrežno programiranje. Korištenjem net modula moguće je kreirati TCP ili IPC poslužiteljske i klijentske programe, a korištenjem http modula

[6] Mrežni izvor, https://nodejs.org/en/about/

olakšava se korištenje HTTP protokola i omogućava baratanje podatcima HTTP poruka u obliku toka podataka (engl. *data stream*).

Express je radni okvir za razvoj poslužiteljskih aplikacija. Predstavlja sloj apstrakcije nad izvornim modulima Node.js okruženja za mrežno programiranje i olakšava proces razvoja poslužiteljskih aplikacija pružanjem unaprijed programiranih funkcija za česte potrebe poslužiteljskih programa. Primjeri funkcionalnosti koje nudi Express su uspoređivanje URL putanji u nadolazećim HTTP zahtjevima prema definiranim uzorcima (engl. *pattern matching*), odbijanje loše formuliranih zahtjeva, jednostavno korištenje proizvoljnih *middleware* funkcija, centralno procesiranje iznimki prilikom izvršavanja programa i slično.

React

React je JavaScript biblioteka za razvoj korisničkih sučelja. Koristi se za razvoj jednostraničnih aplikacija i mobilnih aplikacija. Za razliku od nekih radnih okvira, primjerice Angulara, React biblioteka koristi se isključivo za manipulaciju i prikaz podataka u DOM-u, što znači da je za dodatne funkcionalnosti, poput čuvanja globalnog stanja aplikacije, komunikacije s API servisom ili navigaciju, potrebno koristiti dodatne biblioteke. [7]

Korištenjem React biblioteke, pojedine stranice i elementi prezentacijskog sloja predstavljaju se zasebnim komponentama. Programer definira komponente, komponente postavlja u hijerarhijsku strukturu, te za različita stanja aplikacije prikazuje različitu kombinaciju komponenti. React uvodi JSX, dodatak JavaScriptu kao kombinaciju JavaScripta i označnog jezika XML čime se olakšava pisanje komponenti. Za JSX možemo reći da je sintaktički šećer (engl. *syntax sugar*) jer su JSX označni elementi samo praktična zamjena za pisanje standardnih JavaScript funkcija koje dinamički kreiraju HTML elemente. Programer na taj način može od proizvoljnog broja drugih JSX i izvornih HTML elemenata kreirati komponentu i zatim tu komponentu koristiti sintaksom vrlo sličnom HTML sintaksi.

Komponenta može biti klasa ili funkcija, a zadaća joj je iscrtati određeni dio prezentacijskog sloja aplikacije, ovisno o stanju aplikacije. Komponenta može ovisiti o internom stanju, kao i o stanju roditeljske komponente što se postiže prosljeđivanjem parametara iz roditeljske u komponentu dijete.

[7] Mrežni izvor, https://en.wikipedia.org/wiki/React_(web_framework)

React implementira virtualni DOM odnosno memorijsku reprezentaciju DOM-a. React kod promjene stanja neće ponovno iscrtati sve elemente, već će usporediti trenutno i nadolazeće stanje u memorijskoj reprezentaciji DOM-a te ponovno iscrtati samo one elemente koji sadrže vrijednosti koje su u novom stanju drugačije.

Čitavo stanje aplikacije moguće je držati unutar klasnih komponenti, ali se kod React aplikacija najčešće koriste dodatne biblioteke koje programeru olakšavaju održavanje globalnog stanja aplikacije kreiranjem centralnog sustava za manipulaciju stanjem i čuvanje stanja u zasebnoj strukturi u memoriji. Jedna od najpopularnijih biblioteka za održavanje globalnog stanja aplikacije jest Redux, te je ista korištena u implementaciji dijela rješenja ovog rada.

Od verzije 16.8. u Reactu je moguće koristiti i *React hooks*, odnosno specifične funkcije koje omogućavaju korištenje stanja unutar funkcijskih umjesto klasnih komponenti, ali to je izvan opsega rada budući da ta specifična tehnologija nije korištena u razvoju rješenja.

3.2.2. Mock poslužitelj biblioteka

Centralni dio rješenja implementiran je kao biblioteka za generiranje *mock* podataka, te pokretanje i zaustavljanje *mock* poslužitelja koji, ovisno o postavkama, može raditi na nekoliko načina. *Mock* podatci su opisani pomoću dva glavna modela, modela kojim se opisuje određeni API zahtjev te modela koji predstavlja odgovor na zahtjev. Ta dva modela nazvana su ruta i odgovor. Sukladno REST principima imenovanja, ruta se smatra jedinstvenom kao kombinacija URL-a i HTTP metode. Uz ta dva parametra, svaka ruta opisana je i listom odgovora te opcionalno željenim indeksom odgovora. Pojedini element u listi odgovora može sadržavati jedan od dva različita tipa podatka. Odgovor može biti funkcija koja prima objekte zaglavlja i tijela HTTP zahtjeva i vraća objekt odgovora, a može biti i putanja prema pospremljenom *mock* odgovoru. Bilo da je odgovor dobiven učitavanjem datoteke ili izvršavanjem funkcije, mora biti točno određene strukture te sadržavati HTTP statusni kôd i objekt koji predstavlja tijelo odnosno podatke odgovora. Na taj način sustav je modeliran poprilično jednostavno, ali ostavlja dovoljno fleksibilnosti oko definiranja željenog odgovora. Pojedini *mock* podatak može opcionalno biti opisan sučeljem, kako bi se omogućila statička tipizacija i provjera ispravnosti strukture nakon mijenjanja podataka.

U vezi jedinstvenosti ruta, valja napomenuti kako je omogućen sustav definiranja parametara unutar URL putanje. Ukoliko programer definira opis rute s putanjom *trgovina/proizvod/:id*,

nadolazeće putanje koje odgovaraju tom uzorku bit će smatrane identičnima. Primjerice, rute *trgovina/proizvod/1* ili *trgovina/proizvod/365*. Naravno, zahtjevi u primjeru moraju imati i istu HTTP metodu. Navedeno ponašanje omogućeno je pretvaranjem pospremljenih URL putanji u regularne izraze prilikom usporedbe s nadolazećom URL putanjom, na isti način na koji je to implementirano u Express radnom okviru.

Za pokretanje *mock* poslužitelja potrebno je opisati postavke u točno određenom obliku, makar je velik broj stavki opcionalan, budući da nisu svi podatci potrebni za svaki način rada. Osnovni podatci koje biblioteka svakako očekuje jesu port na kojem će poslužitelj biti pokrenut te apsolutna putanja do korijenske datoteke koju će po potrebi koristiti prilikom rada. Budući da rješenje nudi funkcionalnost automatskog generiranja podataka, ovisi o točno definiranoj hijerarhijskoj strukturi direktorija te načinu imenovanja određenih datoteka. Radi lakšeg razumijevanja, prikazan je oblik strukture direktorija.

```
mock-data
  interfaces
    <uniquename>.ts
  data-sets
    <setname>
      <uniquename>.ts
    <othersetname>
  logs
    log_<timepstamp>.txt
  routes.ts
```

Kôd 3.1 Prikaz strukture direktorija potrebnih za rad *mock* poslužitelja na određenom projektu

Unutar glavnog direktorija mora se nalaziti jedna datoteka s opisom svih ruta, te zasebni poddirektoriji za spremanje sučelja, za spremanje *mock* podataka i za spremanje logova. Unutar direktorija za spremanje *mock* podataka moguće je definirati proizvoljan broj poddirektorija koji sadrže podatke. Rješenje je implementirano na taj način kako bi za iste opise ruta ponudilo jednostavno mijenjanje čitavog seta podataka koje će poslužitelj koristiti u radu. Ukoliko se rješenje želi koristiti za različite projekte, potrebno je samo u postavkama definirati drugačiju korijensku lokaciju podataka. Važno je napomenuti kako je rješenje fleksibilno po pitanju imenovanja svih datoteka izuzev datoteke `routes.ts`, ali je za ispravno korištenje funkcionalnosti generiranja podataka poželjno da se određena *mock* datoteka i pripadajuća datoteka sa sučeljem jednako zovu, osim što *mock* datoteka dodatno u nazivu može imati znak „-„ popraćen proizvoljnim brojem znamenki. Takav sustav imenovanja omogućava automatsko pohranjivanje različitih odgovora s istim sučeljem.

Ovisno o postavkama, *mock* poslužitelj može se pokrenuti na nekoliko načina:

- Preusmjeriteljski poslužitelj
  - Čisti preusmjeritelj
  - Preusmjeritelj koji poslužuje *mock* podatke u slučaju odgovora 404
  - Preusmjeritelj koji generira *mock* podatke
- Poslužitelj unaprijed definiranih *mock* podataka

Preusmjeriteljski poslužitelj

Za pokretanje preusmjeriteljskog poslužitelja programer mora odrediti lokaciju na koju želi preusmjeravati promet te odrediti želi li dobivati *mock* podatke ako preusmjeritelj dobije odgovor sa statusnim kôdom 404. Omogućeno mu je definiranje putanje do certifikata, ukoliko je isti potreban za povezivanje na udaljeni poslužitelj, kao i mogućnost definiranja proizvoljnih HTTP zaglavlja, te pozivanja *callback* funkcija za određene statusne kôdove odgovora. Primjerice, programer može podesiti *mock* poslužitelj tako da prilikom dobivanja statusnog odgovora 401 (*unauthorised*) pokrene *callback* funkciju, pa zatim obaviti u toj funkciji prijavu na udaljeni poslužitelj te osvježiti podatak koji prosljeđuje u obliku proizvoljnog zaglavlja preusmjeritelja, a koji se odnosi na neki obliku identifikatora korisnika i sjednice na udaljenom poslužitelju.

Za lakši razvoj funkcionalnosti preusmjeravanja korištena je npm biblioteka http-proxy-middleware[8]. Navedena biblioteka uz parametre kreira *middleware* funkciju koja se može koristiti unutar Express aplikacija, pri čemu je dio parametara proslijeđen od korisničkog unosa, a dio je implementiran u sklopu biblioteke. Rješenje prati nadolazeće tokove podataka i sprema kopiju tijela zahtjeva i odgovora kako bi se mogla implementirati funkcionalnost posluživanja *mock* podataka ili snimanja odgovora.

Funkcionalnost posluživanja *mock* podataka implementirana je vrlo jednostavno. Ukoliko preusmjeritelj dobije odgovor sa statusnim kôdom 404, potražit će se pohranjeni opisi ruta te poslužiti željeni *mock* odgovor, ako isti postoji, bilo da je riječ o funkciji ili putanji prema datoteci. Ova jednostavna funkcionalnost omogućava vrlo korisnu stvar prilikom razvoja aplikacija. Programer može razvijati klijentsku aplikaciju i preusmjeravati promet na željeni poslužitelj, kako bi vidio ponašanje slično produkcijskom stanju aplikacije, a istovremeno može za neki API zahtjev koji još nije implementiran na poslužitelju dobiti unaprijed definirani *mock* odgovor. Na taj način skraćuje se vrijeme potrebno za testiranje kako će se

[8] https://www.npmjs.com/package/http-proxy-middleware

neka nova funkcionalnost ponašati nakon što se implementira i odgovarajući API na poslužiteljskoj strani.

Zahvaljujući prethodno opisanoj strukturi direktorija, funkcionalnost generiranja *mock* podataka također je implementirana prilično jednostavno. Nakon što preusmjeritelj završi s dobivanjem podataka, program provjerava postoji li već ruta s tim URL-om i metodom. Ako ne postoji, dodat će novi opis rute i pripadajuća *mock* datoteka. Ukoliko ruta postoji, provjerit će se postoji li već identični odgovor unutar trenutnog podatkovnog seta. Ako postoji, neće se generirati duplikat, a ukoliko ne postoji takav odgovor, generirat će se nova *mock* datoteka. Prilikom generiranja datoteke s *mock* podatkom, po potrebi se generira i prikladno sučelje. Ovisno o nazivu, program će *mock* datoteku generirati tako da koristi postojeće sučelje s odgovarajućim imenom, odnosno generirati novo sučelje. Generiranje sučelja implementirano je korištenjem biblioteke typescript-interface-generator[9] koja za zadani objekt i naziv generira tekstualnu reprezentaciju sučelja, ovisno o tipovima podataka koje na zadanom objektu pronađe.

Funkcionalnost snimanja zamišljena je primarno kao brz način generiranja inicijalnog seta podataka za rad *mock* poslužitelja ili za brzo generiranje novog, odvojenog seta podataka prilikom rada na istom projektu. Za dugoročno održavanje statički tipiziranih *mock* podataka, na programeru je izbor hoće li nastaviti koristiti generirana sučelja, definirati svoja sučelja ili koristiti sučelja koja je već definirao u nekom drugom dijelu projekta. Budući da su ciljni korisnici prvenstveno TypeScript programeri, vrlo vjerojatno već imaju definirana brojna sučelja koja opisuju strukturu odgovora za određene API zahtjeve.

Poslužitelj unaprijed definiranih *mock* podataka

Do sada su opisani načini rada *mock* poslužitelja koji uključuju preusmjeravanje zahtjeva na lokalni ili udaljeni API servis, ali rješenje se može koristiti i za pokretanje čistog *mock* poslužitelja. Programer u postavkama može definirati željeni podatkovni set, pri čemu je zadana vrijednost prvi podatkovni set unutar direktorija u kojem se nalaze *mock* podatci. Nakon pokretanja, *mock* poslužitelj za nadolazeće rute poslužuje definirane podatke. Taj sloj namjerno je implementiran u obliku *middleware* funkcije, a ne klasičnim dodavanjem ruta i *callback* funkcija na objekt express aplikacije, kako bi se podatci mogli promijeniti tijekom rada poslužitelja, a sukladne promjene odraziti bez potrebe da se poslužitelj ponovno pokreće.

[9] https://www.npmjs.com/package/typescript-interface-generator

Funkcionalnost pokretanja rješenja kao čistog *mock* poslužitelja korisna je za nekoliko situacija. Korisna je u slučaju kada programer nema pristup udaljenom poslužitelju ili kada iz bilo kojeg razloga ne može na lokalnom računalu pokrenuti poslužitelsku aplikaciju. Naravno, programer može koristiti čisti *mock* poslužitelj i iz jednostavnog razloga što se on brže pokreće, pa će programer brže vidjeti promjene tijekom rada na aplikaciji, odnosno proces razvoja bit će ubrzan.

Zaključno, možemo reći kako je centralni dio rješenja implementiran kao jednostavna ali fleksibilna biblioteka koja na objedinjen način rješava probleme samostalnog rada na klijentskoj aplikaciji. Omogućava brzo generiranje statički tipiziranih *mock* podataka i opisa ruta, omogućava preusmjeravanje prometa, posluživanje *mock* podataka ili kombinaciju zadnje dvije funkcionalnosti. Na taj način programeru je olakšan proces razvoja klijentske aplikacije tako što ne mora ovisiti o poslužiteljskoj aplikaciji.

3.2.3. Konzolna aplikacija

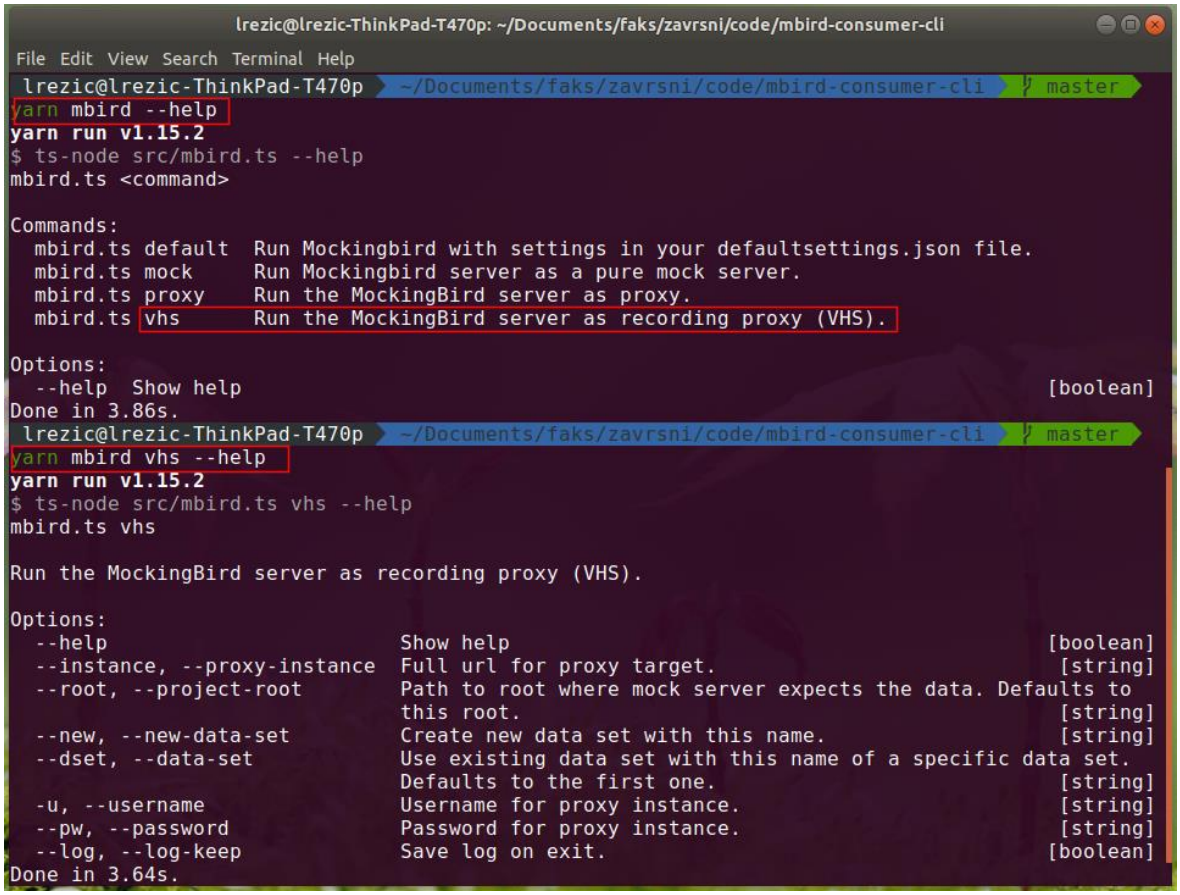
Pomoćna konzolna aplikacija kreirana je za brzo i jednostavno pokretanje *mock* poslužitelja uz promjenjive postavke te sadrži vrlo malen broj funkcionalnosti. Implementirana je korištenjem biblioteke yargs[10] koja pruža sučelje za razvoj jednostavnih i preglednih konzolnih aplikacija uz dobre mogućnosti parsiranja korisničkog unosa. Biblioteka yargs namijenjena je za Node.js aplikacije, licencirana je MIT licencom te se može preuzeti preko npm repozitorija. Konzolna aplikacija napisana je u TypeScriptu te se može pokrenuti prevođenjem u JavaScript i korištenjem Node.js okruženja, ili izravnim pozivanjem skripte korištenjem ts-node alata. Za veću jednostavnost korištenja, aplikaciju je moguće pokrenuti i korištenjem alata yarn ili npm za pokretanje prethodno definirane imenovane skripte koja će zatim u pozadini koristiti potrebne tehnologije za izvršavanje.

Aplikacija se pokreće kao kombinacija imena programa, naziva željene naredbe te zatim konfiguracijskih parametara. Za svaku od ponuđenih naredbi kreiran je pomoćni prikaz koji izlistava sve moguće parametre i njihove opise. Nakon pokretanja *mock* poslužitelja, od korisnika se ne očekuje daljnja interakcija s konzolnom aplikacijom, osim izlaska korištenjem tipki „CTRL + C“, odnosno davanja standardnog signala za prekidanje aktualnog procesa. Budući da je nad podatcima moguća manipulacija programima za uređivanje teksta, dodatne funkcionalnosti bile bi suvišne. Opisani način pokretanja

[10] https://www.npmjs.com/package/yargs

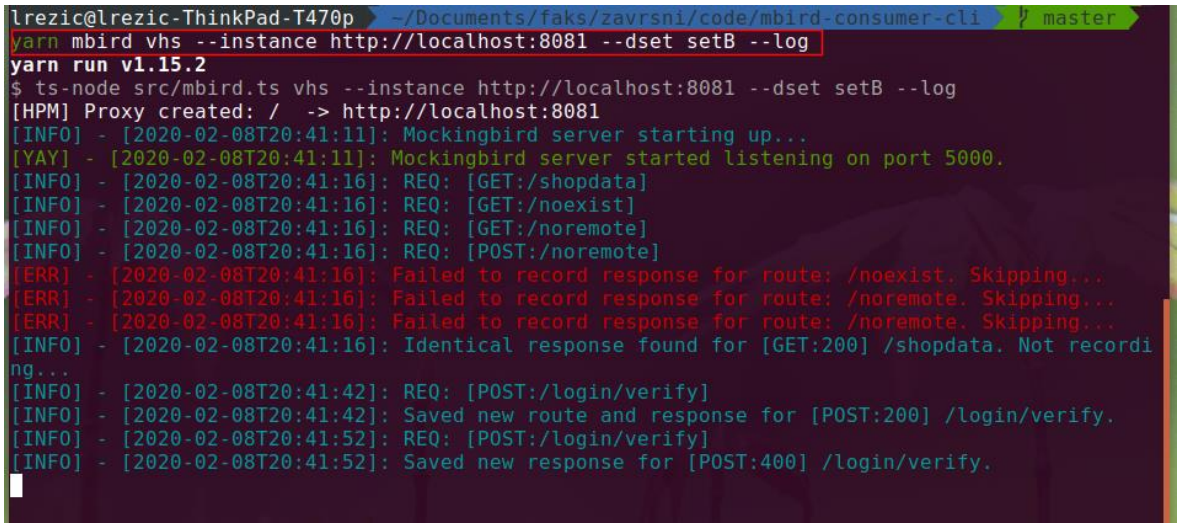
aplikacije prosljeđivanjem svih potrebnih parametra prilikom pokretanja, umjesto postepenog interaktivnog unosa pojedinačnog parametra, ubrzava proces pokretanja te dodatno omogućava i pisanje funkcija ili alijasa za naredbeni redak. Drugim riječima, programer ima slobodu sam napisati pomoćnu funkciju koja će pokrenuti ovu konzolnu aplikaciju s predefiniranim parametrima.

Razmotrimo primjer korištenja na idućoj slici (Slika 3.2):



Slika 3.2 Primjer korištenja *help* naredbe u konzolnoj aplikaciji

Pokretanjem naredbe „yarn mbird –help“ dobiven je prikaz svih mogućih naredbi i njihovih opisa, dok je naredba „yarn mbird <imeNaredbe> --help“ prikazala detalje željene naredbe. Nakon odabira naredbe i parametara, aplikacija se pokreće, pri čemu je *mock* poslužitelj konfiguriran da događaje zapisuje u konzolu kako bi korisnik imao pregled nad radom poslužitelja. Slijedi primjer rada aplikacije nakon pokretanja poslužitelja (Slika 3.3):



Slika 3.3 Izgled konzolne aplikacije nakon pokretanja *mock* poslužitelja

3.2.4. GUI aplikacija

Aplikacija s grafičkim korisničkim sučeljem kreirana je kao internetska aplikacija korištenjem prethodno opisanih tehnologija Express i React. Nameće se pitanje odabira tehnologije, odnosno opravdanosti korištenja internetske aplikacije koja će se upotrebljavati samo na lokalnom računalu. Razlog tome bio je vrlo jednostavan. Budući da su izvorni krajnji korisnici programeri klijentske aplikacije, želja je bila izbjeći uvođenje dodatnih alata i tehnologija potrebnih za pokretanje pomoćne aplikacije. Aplikacija s grafičkim sučeljem nazvana je Mocking-Bird Manager te će se s tim nazivom oslovljavati u ostatku teksta.

Za razliku od prethodno opisane konzolne aplikacije, koja je namijenjena isključivo za lakše pokretanje *mock* poslužitelja, Mocking-Bird Manager aplikacija dodatno nudi i mogućnost zaustavljanja poslužitelja te uređivanja podataka koji se trenutno koriste za rad poslužitelja, ukoliko je isti pokrenut u nekom od načina koji koriste *mock* podatke. Nakon odabira željenog načina rada poslužitelja, korisnik putem forme unosi potrebne parametre i pokreće *mock* poslužitelj. Ukoliko se *mock* poslužitelj uspješno pokrene, korisnik je preusmjeren na stranicu za pregled trenutnih postavki. Na toj stranici korisnik vidi s kojim je parametrima pokrenuo poslužitelj, vidi informacije koje poslužitelj odašilje te ima opciju zaustaviti rad poslužitelja. Pregled zapisanih događaja omogućen je pretplaćivanjem na događaje koje odašilje *mock* poslužitelj te asinkronim slanjem poruka klijentu korištenjem lokalne *web socket* konekcije. Na taj način korisniku se iscrtavaju aktualni događaji bez da klijent mora eksplicitno tražiti podatke.

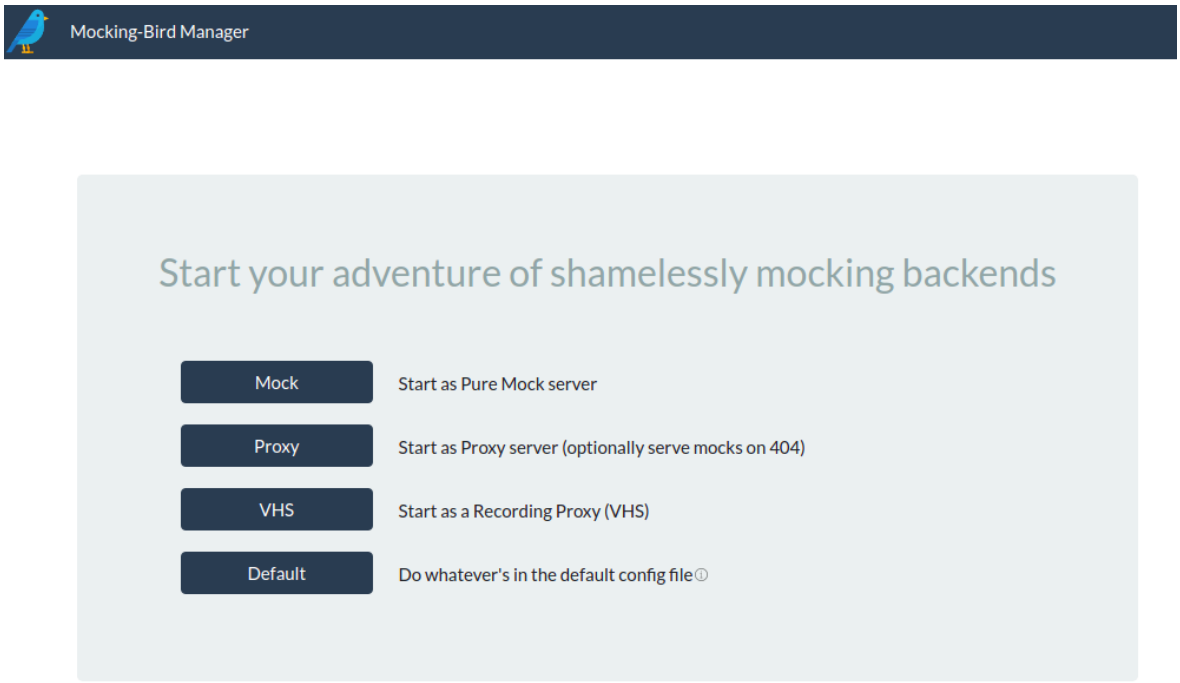
Ukoliko je *mock* poslužitelj pokrenut u načinu koji koristi podatke, korisnik će dodatno imati pristup funkcionalnostima za pregled svih opisa ruta, pregled svih odgovora unutar aktualnog podatkovnog seta, te za uređivanje izvornih datoteka, bilo da je riječ o datoteci koja sadrži opise ruta ili datoteci koja sadrži jedan *mock* podatak. Za uređivanje datoteka korištena je Monaco[11] biblioteka za uređivanje kôda unutar internetskog preglednika. Monaco biblioteka je postavljena da koristi TypeScript, te su joj kod prikazivanja pojedine datoteke proslijeđena sučelja za koja program pretpostavlja da su relevantna, čime je osigurana određena količina statičke tipizacije. Prilikom uređivanja kôda, korisniku će se prikazivati greške i nuditi mogućnosti dovršavanja kôda, identično kao u nekim računalnim programima za pisanje kôda. Naravno, navedena funkcionalnost nije zamišljena kao zamjena za korištenje stvarnog programa za uređivanje teksta, već olakšava jednostavne promjene za vrijeme rada *mock* poslužitelja, pružajući ograničenu statičku tipizaciju i upozoravanje na pogreške.

Uz mogućnost pregleda, pretraživanja i uređivanja izvornih datoteka, korisniku je omogućeno jednostavno mijenjanje željenog odgovora za pojedinu rutu. Budući da jedna ruta može sadržavati više odgovora, ova funkcionalnost je veoma korisna ako brzo želimo vidjeti promjenu u aplikaciji koja koristi *mock* poslužitelj ovisno o *mock* odgovoru. Primjerice, za neku API rutu na koju se šalje forma, korisnik može definirati *mock* odgovor s podatcima nakon uspješnog slanja podataka i *mock* odgovor s greškom nakon slanja forme. Koristeći Mocking-Bird Manager aplikaciju, vrlo brzo i lako može mijenjati željeni odgovor za provjeru je li ispravno implementirao pojedino stanje.

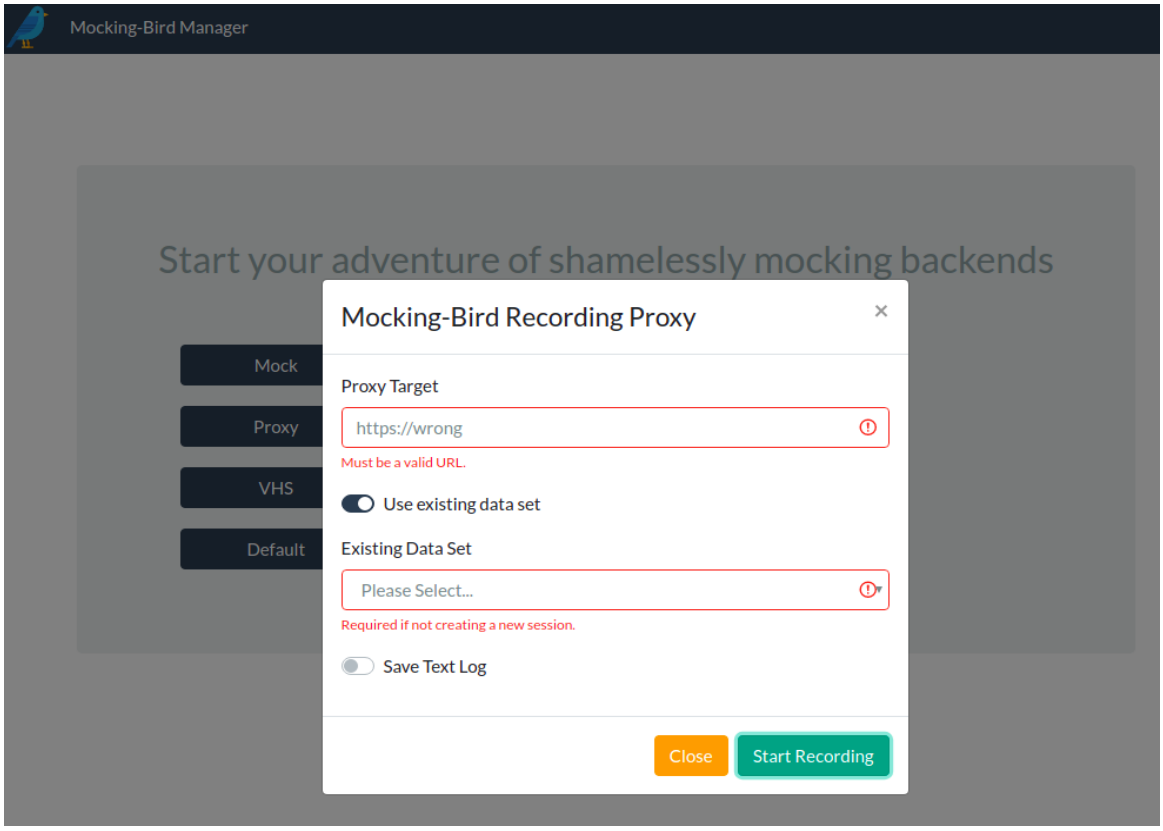
U nastavku slijedi nekoliko prikaza korištenja

- Početna stranica za pokretanje *mock* poslužitelja (Slika 3.4)
- Primjer forme za unos parametarra za pokretanje *mock* poslužitelja (Slika 3.5)
- Pregled trenutnog stanja nakon pokretanja *mock* poslužitelja (Slika 3.6)
- Pregledavanje ruta unutar nekog seta podataka (Slika 3.7)
- Primjer uređivanja odgovora (Slika 3.8)

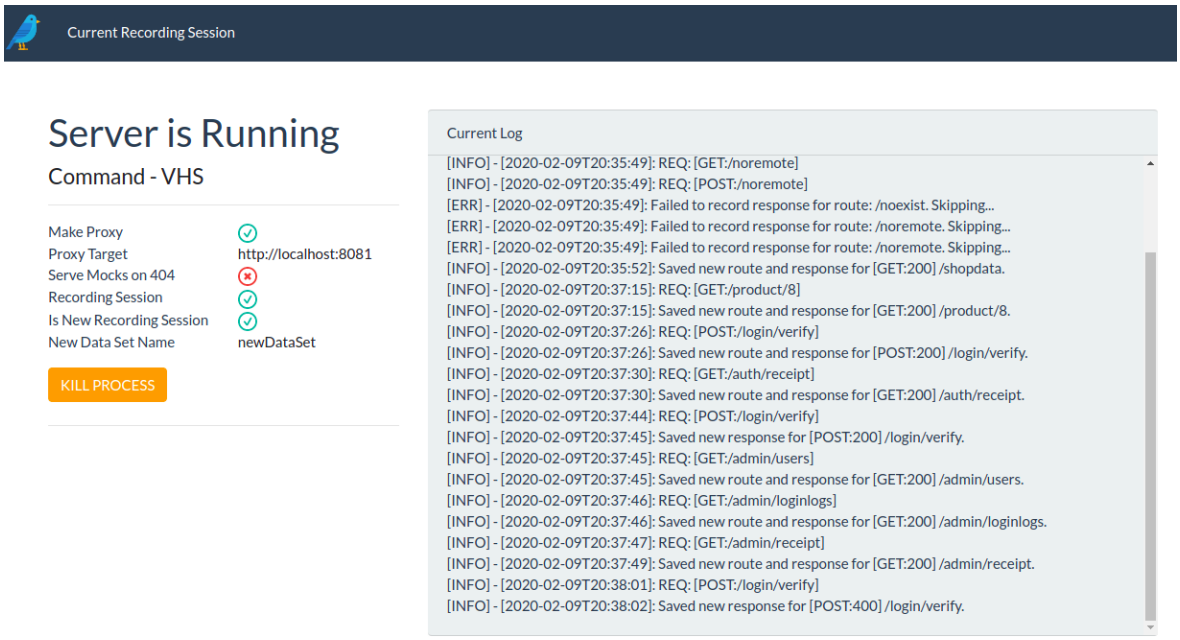
[11] https://microsoft.github.io/monaco-editor/



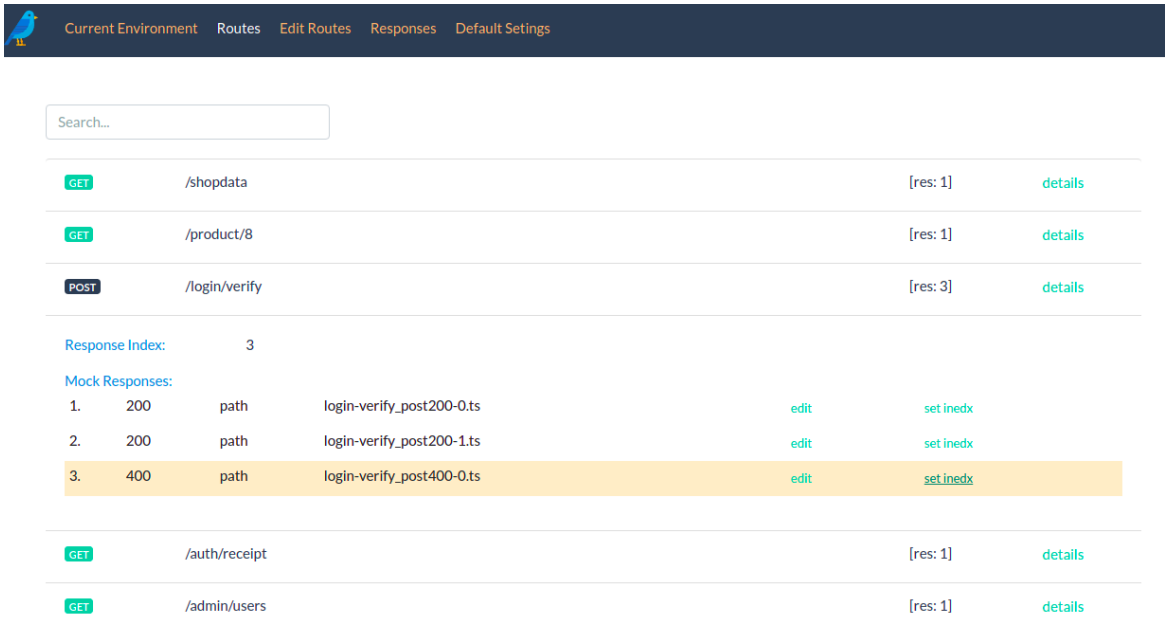
Slika 3.4 Grafičko sučelje za pokretanje poslužitelja



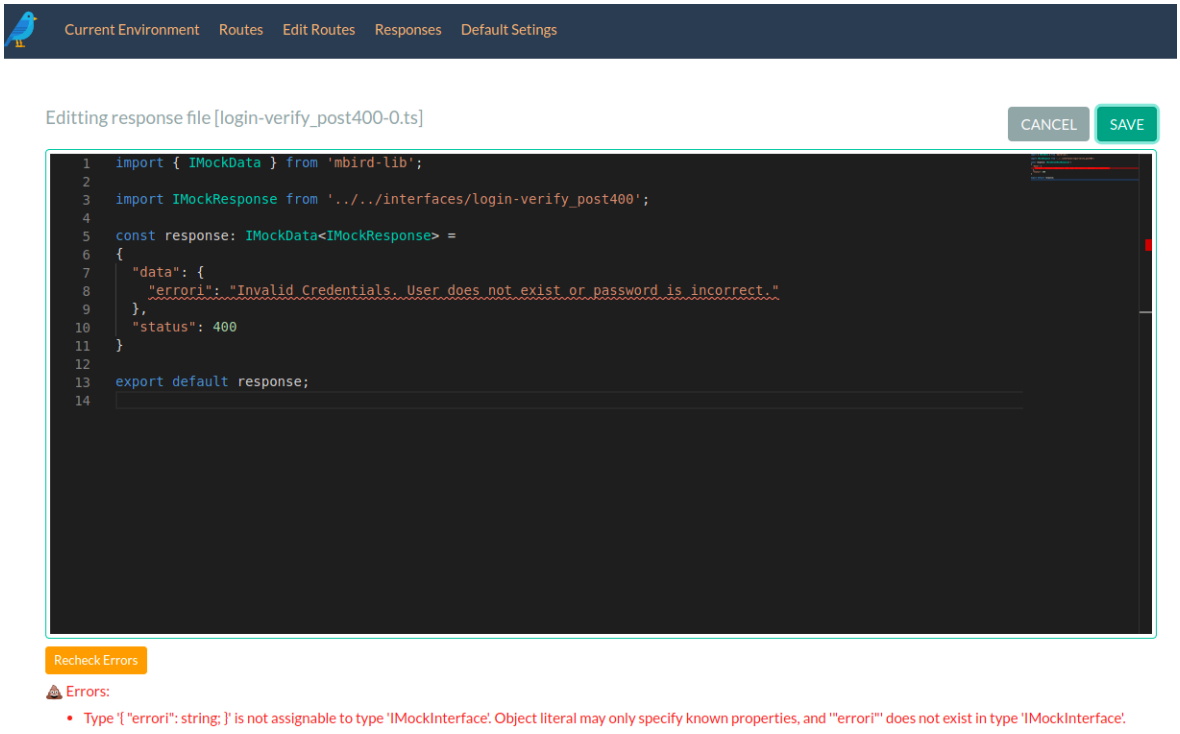
Slika 3.5 Prikaz forme za unos željenih parametara



Slika 3.6 Prikaz trenutnog stanja *mock* poslužitelja



Slika 3.7 Pregled ruti i detalja odgovora za pojedinu rutu



Slika 3.8 Stanje aplikacije nakon što se datoteka pokuša spremiti s greškama

Iz prikazanih primjera vidljivo je da aplikacija ne nudi velik broj funkcionalnosti niti vrlo bogato vizualno sučelje, ali bitno je imati na umu da su krajnji korisnici alata programeri koji će podatcima većinom baratati u programu u kojem pišu kôd, dok je Mocking-Bird Manager aplikacija samo pomoćni vizualni alat za preglednije pretraživanje postojećih podataka i vršenje malih i brzih promjena nad podatcima. Nadalje, budući da se aplikacija neće koristiti u komercijalne svrhe, niti mora „privlačiti“ krajnje korisnike, minimalistički dizajn s prikazivanjem samo onoliko informacija koliko je potrebno činio se kao ispravan pristup implementaciji rješenja.

3.3. Upotreba za automatizirane testove

Mogućnost generiranja i čuvanja statički tipiziranih umjetnih podataka u formatu neovisnom o programu koji ih je generirao, omogućuje iskorištavanje tih istih podataka u druge svrhe. Jedan od primjera jest korištenje tih podataka prilikom pisanja testova. Ako želimo testirati ponašanje aplikacije ovisno o određenom stanju podataka, možemo dio automatiziranih testova implementirati tako da ne rade s poslužiteljskom aplikacijom već s *mock* podatcima. Naravno da za testiranje cjelokupnog ponašanje aplikacije treba pisati testove koji uključuje

sve komponente sustava i oponašaju produkcijsko okruženje (engl. *end to end test*), no takvi testovi dugo traju i zahtijevaju veliku količinu resursa. Prebacivanjem dijela kompleksnih testova na testove koji koriste *mock* podatke, možemo u kraćem vremenskom periodu izvršiti veći broj testova.

Budući da rješenje potiče korištenje sučelja, te može generirati sučelja ovisno o dobivenim tipovima podataka, olakšava se mogućnost korištenja dodatnih alata koji će kreirati nasumične podatke ovisno o sučelju. Rješenje ne sadrži takvu funkcionalnost samo po sebi, ali korištenjem drugih biblioteka, primjerice intermock[12], može se generirati veći broj podataka za pokretanje testova s različitim vrijednostima parametara.

3.4. Nedostatci i moguća proširenja

Najveći nedostatak rješenja krije se u izjavi: „što je neki program više iskoristiv, to ga je teže koristiti“. Biblioteka je razvijena s ciljem da automatizira proces kreiranja *mock* i preusmjeriteljskog poslužitelja, uz dovoljno fleksibilnosti oko postavki. Neovisno o tome, različite aplikacije i poslužitelji mogu biti implementirani na vrlo različite načine, te se biblioteka, barem u trenutnom stanju prototipa, sigurno ne može koristiti u određenim situacijama, pogotovo u situacijama gdje je potrebno kompleksno postavljanje parametara za preusmjeravanje prometa. Bez dovoljno prakse je teško reći, ali rješenje bi zasigurno moglo biti prošireno dodatnom konfiguracijom. No, postavlja se pitanje do koje razine kompleksnosti ima smisla razvijati generičko i podesivo programsko rješenje. Ukoliko je sama konfiguracija vrlo kompleksna, možda je jednostavnije izravno napisati program za *mock* poslužitelja umjesto korištenja biblioteke razvijene u ovom radu.

Osim produbljivanjem mogućnosti pokretanja poslužitelja, rješenje bi se moglo proširiti tako što unutar postavki dozvoljava korištenje nekog drugog tipa dokumenta za spremanje *mock* podataka. Primjerice, konfiguracija bi mogla dozvoliti da se rješenja spremaju u JSON ili XML formatu, bez korištenja TypeScripta i generiranja i korištenja sučelja.

Dodatno moguće proširenje moglo bi biti korištenje određenih ključnih riječi za pojedine vrijednosti. Primjerice, ako želimo da *mock* podatak vraća informaciju s današnjim datumom, mogli bismo koristiti ključnu riječ @TODAY@ i kod čitanja podataka tu ključnu riječ zamijeniti današnjim datumom korištenjem regularnih izraza. To otvara problematiku

[12] https://www.npmjs.com/package/intermock

specifičnosti očekivanih odgovora za pojedinu aplikaciju, te potencijalnih neželjenih posljedica, što nas vraća na prvu rečenicu ovog odlomka. Problem je donekle riješen tako što ruta za odgovor može imati funkciju. U navedenom primjeru mogli bismo za rutu koja unutar podataka očekuje današnji datum definirati funkciju koja će kreirati i vraća taj podatak, ali prekomjerno korištenje funkcija unutar datoteke u kojoj su opisane rute rezultira manjom preglednošću. Funkcije kao odgovori su primarno zamišljeni za jednostavne funkcije koje će za neku rutu i POST metodu vratiti nazad dobiveno tijelo zahtjeva, budući da je to česta praksa kod kreiranja novih resursa. U svakom slučaju, rješenje bi moglo biti fleksibilnije po pitanju posluživanja dinamički generiranih podataka.

Što se tiče pomoćnih aplikacija, moguća proširenja ograničena su samo maštom, ali recimo da bi korisno bilo imati fleksibilnije sučelje za pretraživanje i sortiranje podataka, kao i sučelje s više funkcionalnosti za manipulaciju nad podatcima. Pomoćne aplikacije namjerno su izvedene donekle jednostavno jer je glavna zadaća pomoćnih alata da pokreću *mock* poslužitelj, dok će se primarna manipulacija nad podatcima vršiti preko programa za uređivanje teksta. Kompleksnija korisnička sučelja bila bi potrebna u slučaju da se podatci spremaju u bazu podataka ili u neki drugi nečitki format, ali onda bi postala upitna svrha takvog alata, odnosno alat za upravljanje *mock* poslužiteljem postao bi pretjerano kompleksan za to što rješava.

Zaključak

Rad je predstavio problematiku procesa razvoja i održavanja opsežnih klijentskih programskih rješenja. Prikazani su određeni nedostatci JavaScript jezika, te prezentirane prednosti korištenja pojedinih pristupa i tehnologija za rad na velikom repozitoriju JavaScript kôda. Rad je ukazao na problem potencijalne ovisnosti klijentske aplikacije o poslužiteljskoj aplikaciji ili servisu. Problem se očituje u usporavanju procesa razvoja ako programer prilikom promjena u prezentacijskom dijelu aplikacije mora ponovno pokretati čitavu aplikaciju.

Ukratko su prikazana neka od postojećih rješenja, a u želji da se proces samostalnog rada na klijentskoj aplikaciji dodatno ubrza, razvijen je podesivi *mock* poslužitelj. Rješenje je implementirano razvojem *Mock* poslužitelj biblioteke te razvojem dviju pomoćnih aplikacija koje predstavljaju dodatni sloj korisničkog sučelja za korištenje biblioteke. Iako je razvijeno za stvarne potrebe programera tvrtke Oradian d.o.o., rješenje je objavljeno u repozitoriju otvorenog kôda što ga čini dostupnim široj zajednici programera.

Razvijeno rješenje programerima klijentskih aplikacija pruža alat za generiranje statički tipiziranih umjetnih podataka te alat za pokretanje preusmjeriteljskog ili *mock* poslužitelja. Posljedično, alat olakšava proces razvoja klijentskih aplikacija bez ovisnosti o poslužiteljskoj aplikaciji ili servisu čime ubrzava cjelokupni proces razvoja i održavanja nekog programskog proizvoda. Uz rješavanje praktičnog problema, prikazana je i činjenica kako jednostavni alati za automatizaciju čestih radnji prilikom programiranja mogu imati bitnu ulogu u produktivnosti programera.

Popis kratica

AJAX	*Asynchronous JavaScript and XML*	asinkroni JavaScript i XML
API	*Application Program Interface*	aplikacijsko programsko sučelje
CDN	*Content Delivery Network*	mreža isporuke podataka
CLI	*Command Line Interface*	konzolno sučelje
CSS	*Cascading Style Sheets*	kaskadirajuće stilske stranice
DOM	*Document Object Model*	dokument objekt model
ECMA	*Ecma International*	(ime organizacije)
GUI	*Graphic User Interface*	grafičko korisničko sučelje
HTML	*Hyper Text Markup Language*	hipertekstualni označni jezik
HTTP	*Hyper Text Transfer Protocol*	hipertekstualni transportni protokol
I/O	*Input/Output*	ulaz / izlaz (ulazno izlazne operacije)
IDE	*Integrated Development Environment*	integrirano razvojno okruženje
IPC	*Inter Process Communication*	međuprocesna komunikacija
JSON	*JavaScript Object Notation*	JavaScript notacija objekata
JSX	*JavaScript and XML*	JavaScript i XML
MIT	*Massachusetts Institute of Technology*	(ime ustanove)
MVC	*Model View Controller*	model pogled upravljač
REST	*Representational State Transfer*	reprezentativno stanje prijenosa
SEO	*Search Engine Optimisation*	optimizacija za pretraživače
TCP	*Transmission Control Protocol*	protokol za kontrolu prijenosa
URL	*Uniform Resource Locator*	uniformni lokator resursa
XHR	*XML Http Request*	XML HTTP zahtjev
XML	*Extensible Markup Language*	proširivi označni jezik
npm	*Node Package Manager*	upravljač paketima okruženja Node.js

Popis slika

Slika 2.1 Pojednostavljena usporedba primjera višestranične i primjera jednostranične klijent-poslužitelj arhitekture internetske aplikacije 3

Slika 3.1 Pojednostavljen prikaz razvijenog rješenja, Mock poslužitelj biblioteke te pomoćnih aplikacija s korisničkim sučeljem 20

Slika 3.2 Primjer korištenja *help* naredbe u konzolnoj aplikaciji 29

Slika 3.3 Izgled konzolne aplikacije nakon pokretanja *mock* poslužitelja 30

Slika 3.4 Grafičko sučelje za pokretanje poslužitelja 32

Slika 3.5 Prikaz forme za unos željenih parametara 32

Slika 3.6 Prikaz trenutnog stanja *mock* poslužitelja 33

Slika 3.7 Pregled ruti i detalja odgovora za pojedinu rutu 33

Slika 3.8 Stanje aplikacije nakon što se datoteka pokuša spremiti s greškama 34

Popis kôdova

Kôd 2.1 Prikaz korištenja JavaScript modula imenovanim izvozom (engl. *named export*) . 8

Kôd 2.2 Primjer korištenja nekih TypeScript tipova 12

Kôd 2.3 Primjer korištenja nekih od objektno orijentiranih koncepta u jeziku TypeScript 13

Kôd 3.1 Prikaz strukture direktorija potrebnih za rad *mock* poslužitelja na određenom projektu 25

Literatura

Svaki autor piše popis literature na kraju rada. Popis literature se piše stilom literatura.

[1] KLAUZINSKI, P. *Mastering JavaScript Single Page Application Development*. Birmingham: Packt Publishing, 2016.

[2] SCOTT, E. A. JR. *SPA Design and Architecture.* Shelter Island: Manning Publications Co., 2016.

[3] *Mozilla Hacks*, URL: https://hacks.mozilla.org/2015/08/es6-in-depth-modules (Pristupljeno: 2020-02-02)

[4] *MDN Web Docs,* URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules (Pristupljeno: 2020-02-02)

[5] *TypeScript,* URL: https://www.typescriptlang.org/index.html (Pristupljeno: 2020-02-02)

[6] *TypeScript Programming Language,* TypeScript Publishing, 2019. (e-knjiga bez autora)

[7] THOMAS, D., HUNT, A. *The Pragmatic Programmer: your journey to mastery, 20th Anniversary Edition.* London: Pearson Education, 2019.

[8] *V8,* URL: https://v8.dev/docs (Pristupljeno: 2020-03-02)

[9] *Node.js Docs*, URL: https://nodejs.org/en/about (Pristupljeno: 2020-03-02)

[10] WILSON, J. R. *Node.js 8 the Right Way, Practical Server Side JavaScript That Scales.* Raleigh: The Pragmatic Bookshelf, 2018.

[11] *Wikipedia*, URL: https://en.wikipedia.org/wiki/React_(web_framework) (Pristupljeno: 2020-03-02)