ŠOLSKI CENTER VELENJE
ELEKTRO IN RAČUNALNIŠKA ŠOLA
Trg mladosti 3, 3320 Velenje

MLADI RAZISKOVALCI ZA RAZVOJ ŠALEŠKE DOLINE

RAZISKOVALNA NALOGA

APLIKACIJA Z DVOSMERNO SINHRONIZACIJO MED OPRAVILI IN OBLAČNIM SKLADIŠČEM

Tematsko področje: Računalništvo

Avtorja:
Anže Goršek, 4. letnik
Matija Osojnik, 4. letnik

Mentorja:
Islam Mušić, prof.
Samo Železnik, inž.

Velenje, 2021

Raziskovalna naloga je bila opravljena na ŠC Velenje, Elektro in računalniška šola, 2021.

Mentorja: Islam Mušić, prof., Samo Železnik, inž.

Datum predstavitve: april 2021

KLJUČNA DOKUMENTACIJSKA INFORMACIJA

ŠD ŠC Velenje, šolsko leto 2020/2021

KG spletna aplikacija za beleženje opravil / sinhronizacija s spletnim skladiščem

AV GORŠEK, Anže / OSOJNIK, Matija

SA MUŠIĆ, Islam / ŽELEZNIK, Samo

KZ 3320 Velenje, SLO, Trg mladosti 3

ZA ŠC Velenje, Elektro in računalniška šola, 2021

LI 2021

IN **APLIKACIJA Z DVOSMERNO SINHRONIZACIJO MED OPRAVILI IN OBLAČNIM SKLADIŠČEM**

TD Raziskovalna naloga

OP IX, 49 str., 1 pregl., 0 graf., 40 sl., 0 pril., 26 vir.

IJ SL

JI sl/en

AI Načrtovanje je eden od najpomembnejših delov vsakega dogodka, izdelka, storitve ..., saj lahko z dobrim oz. slabim načrtovanjem naredimo ogromno razliko v končnem produktu.

Kljub temu, da obstaja že veliko aplikacij za načrtovanje, sva se odločila, da raziščeva ravno to področje. Cilj naloge je za uporabnika enostavna spletna aplikacija za opravljanje in postavljanje opravil. Predvsem sva se želela osredotočiti na opravila, za katera je potrebno pripraviti veliko dokumentov (poročila za vaje, evidenca prisotnosti ...) ter v ta opravila vključiti veliko ljudi.

Ker aplikacija lahko hrani veliko število dokumentov, sva se zato odločila povezati aplikacijo z oblačno storitvijo, kot je Google Drive. Tako bi lahko uporabnik shranil vse svoje dokumente v oblak in ne bi imel skrbi, da jih izgubi, če spletna stran preneha delovati. Prav tako s tem uporabniku dajeva nadzor nad podatki, ki jih je zaupal najini aplikaciji, kar se nama zdi še posebej pomembno v današnjem svetu, kjer ima splet več podatkov o nas, kot pa se sami zavedamo.

KEY WORDS DOCUMENTATION

ND ŠC Velenje, school year 2020/2021

CX web application / web storage sinhronization / task management

AU GORŠEK, Anže / OSOJNIK, Matija

AA MUŠIĆ, Islam / ŽELEZNIK, Samo

PP 3320 Velenje, SLO, Trg mladosti 3

PB ŠC Velenje, Elektro in računalniška šola, 2021

PY 2021

TI **APLIKACIJA Z DVOSMERNO SINHRONIZACIJO MED OPRAVILI IN OBLAČNIM SKLADIŠČEM**

DT Research work

NO IX, 49 p., 1 tab., 0 graf., 40 fig., 0 ann., 26 ref.

LA SL

AL sl/en

AB Planning is one of the most important parts of every event, product, service ..., and it can make a huge difference with a good or poor design of the final product.

Even though there are already many existing planning applications, we decided to research exactly this area of research. The goal of the research is to create a simple web application, which allows the user to create and set tasks. Above all, we wanted to focus on tasks that require a lot of documents (exercise reports, attendance records ...) and involve a larger number of people in these tasks.

Because the application has the ability to store a large number of documents, we therefore decided to connect it to a cloud storage such as Google Drive. This way, the user could store all their documents in the cloud and would not have to worry about their data being lost in the case of a website failure. This way, we also give the user control over the data he has entrusted to our app, which we find especially important in today’s world where the web has more information about you than you realize.

VSEBINA

KAZALO SLIK

KAZALO TABEL

KRATICE

API - programski vmesnik (angl. *Application Programming Interface*)

AWS - Amazonove spletne storitve (angl. *Amazon Web Services*)

BSON - Binarna oznaka modelov v JavaScriptu (angl. *Binary JavaScript Object Notation*)

CRUD - Ustvari, beri, posodobi in izbriši (angl. *Create, read, update and delete*)

CSS – kaskadne stilske podloge (angl. *Cascading Style Sheets*)

HTML – jezik za označevanje nadbesedila (angl. *Hypertext Markup Language*)

JSON – Oznaka modelov v JavaScriptu (angl. *JavaScript Object Notation*)

JWT - spletni žeton JSON (angl. *JSON Web Token*)

MB – megabajt

NoSQL - Ne samo SQL (angl. *Not only SQL*)

SQL - Strukturirani povpraševalni jezik za delo s podatkovnimi bazami (angl. *Structured Query Language*)

1. UVOD

Načrtovanje je eden od najpomembnejših delov vsakega dogodka, izdelka, storitve ..., saj lahko z dobrim oz. slabim načrtovanjem naredimo ogromno razliko v končnem produktu. Kljub temu, da obstaja že veliko aplikacij za načrtovanje, sva se odločila, da raziščeva ravno to področje. Osredotočila sva se na opravila, za katera je potrebno pripraviti veliko dokumentov (poročila za vaje, evidenca prisotnosti, poročilo o dogodku ...) ter v ta opravila vključiti veliko ljudi. Za shranjevanje vseh teh podatkov oz. dokumentov bi uporabila oblačno storitev, kot je Google Drive in s tem uporabnikom omogočila popoln nadzor nad njihovimi podatki, kar se nama zdi še posebej pomembno v sodobnem svetu, kjer ima splet več podatkov o nas, kot pa se sami zavedamo. Poleg tega pa bi podatki in datoteke, shranjene na oblačnem skladišču, tvorili varnostno kopijo podatkov spletne aplikacije v primeru zrušitve aplikacije.

1.1 Hipoteze

1. S spletno aplikacijo se lahko povežemo preko programskega vmesnika (API) na poljubno oblačno skladišče.

2. Preko programskega vmesnika (API) lahko v oblačnih skladiščih ustvarjamo mape in nalagamo datoteke.

3. Preko programskih vmesnikov (API) lahko ustvarimo dvosmerno sinhronizacijo med opravili in mapami v oblačnem skladišču.

4. Skriptni programski jezik JavaScript ni dovolj zmogljiv za strežniški del aplikacije.

1.2 Namen in cilj

Namen raziskovalne naloge je za uporabnika ustvariti enostavno spletno aplikacijo za opravljanje in postavljanje opravil v raznih projektih in pri organizaciji dogodkov, s pomočjo katerih obdržimo svojo rdečo nit. Aplikacija mora omogočati dvosmerno sinhronizacijo med opravili in oblačnim skladiščem, kar pomeni, da ko v aplikaciji ustvarimo opravilo, se mora to shraniti tudi v oblačnem skladišču in obratno. Omogočati mora tudi dodajanje uporabnikov k opravilom in skladiščenje velikih datotek. Aplikacija mora biti varna za uporabo.

Aplikacijo bova predstavila potencialnim bodočim uporabnikom, kot so mali podjetniki z namenom pridobitve povratnih informacij. Zanimalo naju bo predvsem, če jim je bila aplikacija v pomoč pri načrtovanju izdelka, projekta ali storitve in kako lahko najino aplikacijo izboljšava in dopolniva, da bi bila zanimiva za trg.

2. PREGLED OBSTOJEČIH REŠITEV

Danes obstaja na svetu že mnogo in še več aplikacij, ki so bile ustvarjene z namenom lažjega skupinskega dela in povečane organizacije. Najbolj priznane aplikacije na današnjem trgu so Trello, Asana, Jira, ClickUp, Basecamp in še mnoge druge, ki nudijo podobne ali pa tudi enake možnosti, kot so:

- ustvarjanje projektov,
- ustvarjanje nalog in podnalog,
- dodelitev nalog posameznikom v projektu,
- končni datumi za naloge,
- nalaganje datotek iz računalnika ali iz že obstoječih oblačnih storitev, kot so Google Drive, OneDrive, DropBox …,
- komentarji,
- obvestila.

Lahko bi še govorila o vseh prednostih, ki jih te aplikacije nudijo, vendar se nama zdi bolj smiselno govoriti o tem, česa nimajo. Tukaj se vključi najina povezava aplikacije z možnostjo dvosmerne sinhronizacije med opravili in oblačnim skladiščem. Zveni precej podobno, kot le povezava z enim od ponudnikov oblačnih storitev, kot je Google Drive, ampak je ta del bolj kompleksen.

Podobne zmožnosti omogoča tudi aplikacija Slack, vendar temelj in cilj le-te nista podobna najinemu in je že sam način organizacije v tej aplikacije ustvarjen po drugačnih metodah, ki so bolj podobne Whatsappu ali Telegramu, kot tistim aplikacijam, ki so bile naštete na začetku tega poglavja.

2.1 Metode organizacije dela

Metode organiziranja dela na področju informacijske tehnologije pomenijo okvir, ki se ga uporablja za strukturo, načrt in kontrolo procesa razvoja programske rešitve (Software Development Methodology, 2013). Razvoj programskih rešitev je del industrije, ki je podvržena zelo hitrim spremembam. Je ena novejših industrij in ima tendenco, da se v branžo

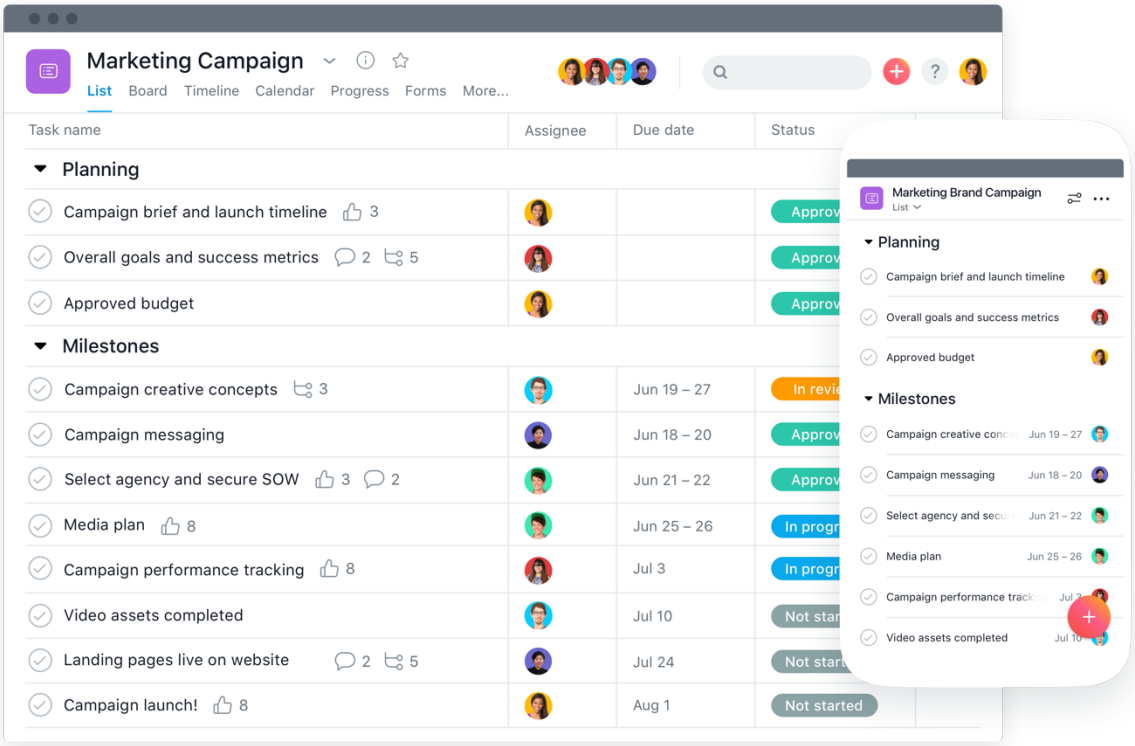
vključujejo mladi, kreativni in inovativni ljudje. Če združimo vse značilnosti, ugotovimo, da je programiranje razvijajoča se disciplina. Potreba po metodah je prišla s strani vodstva in svetovalcev in ne iz potrebe razvijalcev, ki metode navadno vidijo kot oviranje njihove svobode, kreativnosti in produktivnosti (Kittlaus, 2012).

Ključen del uspeha vsake organizacije v sodobnem svetu, poleg neprestanega inoviranja, je agilna organiziranost, kar pomeni, da se lahko hitro in lahko prilagaja spremembam, ter da je pripravljena na neprestano serijo le-teh.

2.2 Obstoječe alternative

2.2.1 Asana

Asana je orodje za nadzor nad opravili, ki temelji na spletnem oblaku. Podjetjem in ekipam omogoča upravljanje, sodelovanje, komunikacijo in organizacijo nalog in projektov. Njena prednost je možnost nadzora več projektov naenkrat in je primerna za vsa podjetja.



Slika 1: Osnovna oblika Asane, vir [1]

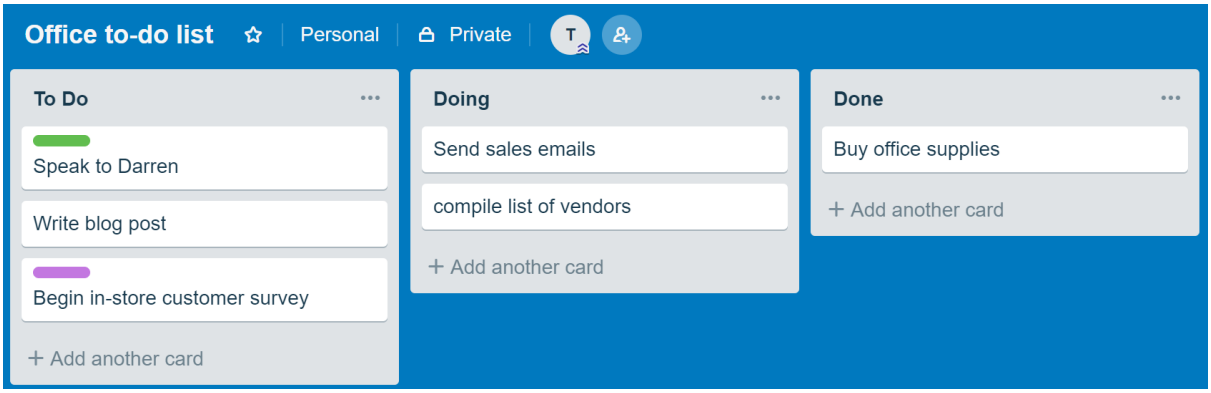
Samo delovanje Asane je razdeljeno na več delov:

- Projekti – Aplikacija omogoča organizacijo in deljenje vseh projektov kot sezname ali table za ohranjanje vseh zapisov in posnetkov s sestankov in programov.
- Naloge – Ena izmed večjih prednosti je zmožnost razdelitve opravil na manjša opravila, tako imenovana podopravila (angl. *subtasks*) in zmožnost dodelitve drugih članov ekipe direktno na opravila. Poleg tega omogoča še veliko več funkcij, kot so pravila, obrazci, cilji, zaključni datumi, nalaganje datotek.
- Komunikacija – Pomembno vlogo pripisujejo tudi dobri komunikaciji, zato je v aplikaciji ustvarjeno orodje za sporočanje, kjer lahko ekipa v živo piše in odgovarja.
- Integracija drugih orodij – Poleg storitve Google Drive, Dropbox in OneDrive nudi tudi možnost povezave z ostalimi aplikacijami, kot so Microsoft Teams, Office 365, Outlook, Jira cloud in še več z bolj popularnimi aplikacijami.

Aplikacija je brezplačna, vendar ta naročnina omogoča omejeno delovanje in izključitev določenih funkcij, pa tudi integracija drugih orodij ne deluje tako, kot mora. Največja negativna lastnost je zapletenost samega delovanja, saj nudi veliko funkcij tudi, če jih sami ne želimo uporabljati.

2.2.2 Trello

Trello je osnovno orodje za nazdor nad opravili, ki delujejo na principu Kanban organizacije dela. Vsaka naloga je shranjena v svoji kartici, ki je sama po sebi tudi seznam. Mogoče je imeti neomejeno kartic, vendar osnovna postavitev je sestavljena iz treh seznamov: kaj je še potrebno narediti, na čem se trenutno dela in končano. Zaradi lažje organizacije in razporeditve je mogoče sezname dodati in izbrisati ter povleči in spustiti naloge iz enega seznama v drugega.



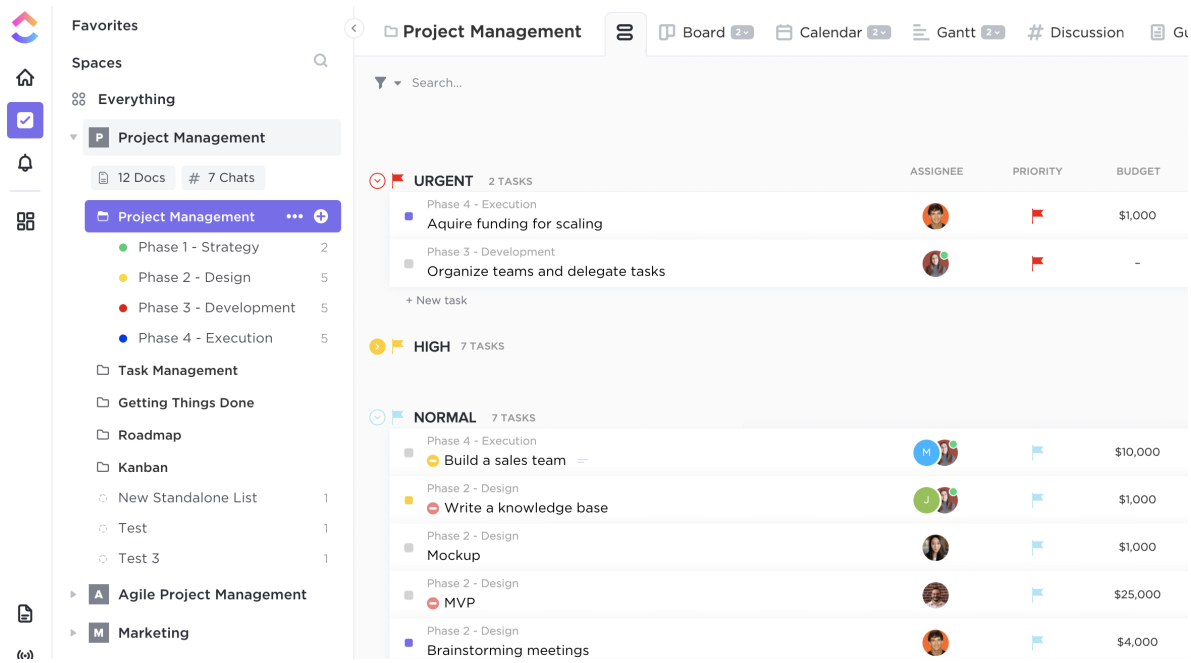
Slika 2: Primer oblike Trello Kanban sistema, vir [2]

V posamezne kartice je mogoče tudi dodati priponke in jim pripisati že obstoječe ali samostojno ustvarjene oznake. Težava nastane, ko več članov ekipe dela na več nalogah ob istem času in lahko hitro zgrešimo obvestila in vprašanja na le-teh. Ob primerjavah med Trellom in Asano je največja razlika ta, da je oblika Trella kot nek preprost seznam opravil, medtem ko je bila Asana že od začetka ustvarjena kot orodje za nadzor nad projekti za ekipe.

Kljub temu je zelo enostavna platforma za uporabo in zelo priporočljiva za manjše ekipe, ki želijo delati na enem projektu naenkrat. Verjetno je ta dostopnost in enostavnost tudi največja vrednost aplikacije, saj s tem omogoča uporabo vsakemu, ki išče nov dostopen seznam opravil.

2.2.3 ClickUp

ClickUp je delovna platforma v oblaku za vse vrste in velikosti skupin in podjetij. Združuje pomembne poslovne aplikacije in centralizira podatke o podjetju v eno samo spletno rešitev. Članom ekipe lahko dodeljujemo naloge, upravljamo projekte za stranke in sodelujemo z drugimi člani ekipe pri dokumentih.



Slika 3: Izgled aplikacije ClickUp, vir [3]

Poleg tega aplikacija omogoča ogled delovnih elementov in podatkov v več pogledih za boljše razumevanje in hitrejše sledenje. Možen je pogled v obliki seznama za opravila, v obliki plošče za potek dela, v obliki polja za nadzorne plošče ali v obliki Ganttovega pogled za urnike projektov. Med drugim ponuja tudi koledarski pogled, pogled dejavnosti, mentalne mape, pogled delovne obremenitve, pogled tabele in zemljevidov.

V primerjavi s Trellom in Asano omogoča ClickUp več funkcij brezplačno, vendar po ocenah uporabnikov mogoče tudi preveč. Ima bolj kompleksen in težje razumljiv uporabniški vmesnik za začetnike, vendar je, kot že prej omenjeni aplikaciji, dobro dodelan in ponuja vse funkcije brezplačno.

2.2.4 Primerjava že obstoječih rešitev z najino aplikacijo

Vse zgoraj opisane aplikacije rešujejo enak problem, ki ga imenujemo organizacija dela in nadzor dela pri projektih. Vse uporabljajo kot temelj strukture agilne metode. Trello je bolj osredotočen na enostavnost in praktičnost reševanja seznamov opravil, medtem ko sta Asana in ClickUp bolj kompleksni aplikaciji in sta že od začetka bili osredotočeni na večje ekipe in kolaboracijo v le-teh. Seveda pa so pomembne funkcije, ki jih te aplikacije nudijo. Trello omogoča le en pogled nad opravili, ki se imenuje Kanban, omogoča razdelitev opravil med uporabniki, lahek sistem prestavljanja nalog med seznami in pa seveda možnost obvestil in

komentarjev. Asana in ClickUp omogočata vse to in še več, z več možnostmi pogledov nad opravili, sporočanje in komunikacijo znotraj aplikacije, Asana omogoča tudi integracijo številnih oblačnih storitev, kot so Google Drive, OneDrive, DropBox in drugih aplikacij.

Zgoraj navedene aplikacije omogočajo najrazličnejše funkcije, zato sva se odločila, da se bova osredotočila na tiste funkcije, ki sva jih pri že obstoječih pogrešala, in to je predvsem zasebnost podatkov in dvosmerna sinhronizacija med opravili in oblačnim skladiščem. Ker sva se odločila aplikacijo povezati z oblačno storitvijo, sva sklenila, da bova uporabnikom omogočila tudi nalaganje velikih datotek na najino aplikacijo oz. na njihovo oblačno skladišče. Zasebnost njihovih podatkov bova zagotovila tako, da nimava na aplikaciji shranjene nobene datoteke, ampak vse shraniva na njihovo oblačno skladišče.

3. OGRODJA ZA IZDELAVO APLIKACIJE

3.1 Izbira programskega okolja

Izbira programskega okolja je bil postopek, ki sva mu namenila več časa, saj sva želela izdelati aplikacijo v programskem okolju, ki bi bilo najprimernejšo za najino aplikacijo, in to je okolje, ki omogoča mnoga operiranja z mnogo sočasnimi zahtevki, horizontalno skalabilnost in operiranje z velikimi datotekami in ravno to nama omogoča uporaba izvajalnega okolja Node.js, nadgrajena s knjižnico Express.js, ki skrbi za zaledni del aplikacije. Za izdelavo uporabniškega vmesnika sva izbrala spletno ogrodje Vue, ker omogoča enostavno povezavo z okoljem Node.js in prilagodljivost zahtevam aplikacije. Poleg vsega tega ima možnost uporabe knjižnice Vuetify, ki je eno od boljših ogrodij za izdelavo uporabniških vmesnikov, zgrajeno na podlagi ogrodja Vue in s tem sva lahko v čim krajšem času ustvarila uporabniku razumljiv in enostaven vmesnik. Ena od prednosti izbire Node.js in Express.js je, da lahko uporabiš skriptni programski jezik JavaScript tudi za zaledni del aplikacije in s tem prihraniš čas, ki bi ga porabil za učenje dveh programskih jezikov.

3.1.1 Node.js

Node.js je odprtokodno izvajalno okolje, ki omogoča izvajanje kode, napisane v programskem jeziku JavaScript izven spletnega brskalnika, za katerega je bil prvotno namenjen programski jezik JavaScript. Za razvijalca je to ogromen plus, saj lahko uporablja enak programski jezik v celotni aplikaciji. To mu uspe z Googlovim mehanizmom V8 JavaScript, saj le-ta JavaScript kodo predela direktno v računalniku razumljivo kodo in jo ob tem še optimizira.

3.1.1.1 Prednosti Node.js

1. Node.js je asinhron sistem, kar pomeni, da lahko upravlja naenkrat z mnogo opravili/zahtevami za razliko od sinhronih sistemov, ki lahko naenkrat operirajo le z enim opravilom oz. zahtevo.
2. Uporablja jezik JavaScript, kar omogoča razvijalcu uporabo enakega jezika na uporabniški strani aplikacije in na strežniški strani.
3. Ogromno število knjižnic, ki razvijalcu prihranijo veliko časa pri kodi aplikacije, ki je skoraj enaka za vsako aplikacijo (povezava s podatkovno bazo ali avtentikacija). Prav tako

se knjižnice redno posodabljajo, kar je še dodaten plus, saj je skrb za stalno posodabljanje in sledenje novim smernicam zelo časovno potratno za razvijalca.

4. Aktivna in velika skupnost ter že napisane kode, s katero si lahko pomagamo.
5. Podpora za velike datoteke.

3.1.1.2 Slabosti Node.js

1. Glavna slabost Node.js je, da ne podpira vodoravne skalabilnosti, kar pomeni, da slabo izkorišča vse prednosti več jedrnih procesorjev, en procesor pa v današnjih strežniški strojni opremi pogosto ne zadostuje.
2. Zapletena programska koda, kadar imamo opravka z relacijsko podatkovno bazo.
3. Potrebno je poglobljeno poznavanje JavaScript programskega jezika.

3.1.1.3 Zakaj torej Node.js?

Kot vsak sistem ima tudi Node.js svoje prednosti in slabosti, zato je najpomembnejše, da razumemo naloge naše aplikacije in šele potem se lahko pravilno odločimo za pravo okolje. Za Node.js se bomo odločili, če bo naša aplikacija operirala z mnogo zahtevami istočasno, ne bo pa intenzivno obremenjevala procesorja z računskimi operacijami. Po možnosti izberemo tudi nerelacijsko podatkovno bazo, da si olajšamo delo ter seveda moramo znati pisati v programskem jeziku JavaScript.

Midva sva izbrala Node.js predvsem zaradi njegove asinhrone arhitekture, ki omogoča operiranje z mnogo zahtevami naenkrat, ker meniva, da aplikacija ne bo preveč obremenjevala procesorja. Prav tako sva se že prej odločila za podatkovno bazo MongoDB, zato nisva imela tu nikakršnih pomislekov. Z Node.js pridobiva tudi možnost operiranja z velikim datotekami, kar je še dodaten plus. Tu je tudi programski jezik JavaScript, s katerim sva oba že delala in nama je to predstavljalo še eno prednost.

3.1.2 Express

Express.js oz. Express je v osnovi JavaScript knjižnica, s pomočjo katere lahko ustvarimo zaledno ogrodje spletne aplikacije in programskih vmesnikov.

Express ima veliko alternativ, kot so Ruby on Rails, Django, Laravel, Fastify ... Ampak zanj sva se odločila, ker nama ponuja 3 ključne stvari, in sicer lahko operirava z veliko zahtevki sočasno, ima močno skupnost, ki nama lahko pomaga, ko naletiva na težave in podpira mnogo knjižnici za lažji razvoj.

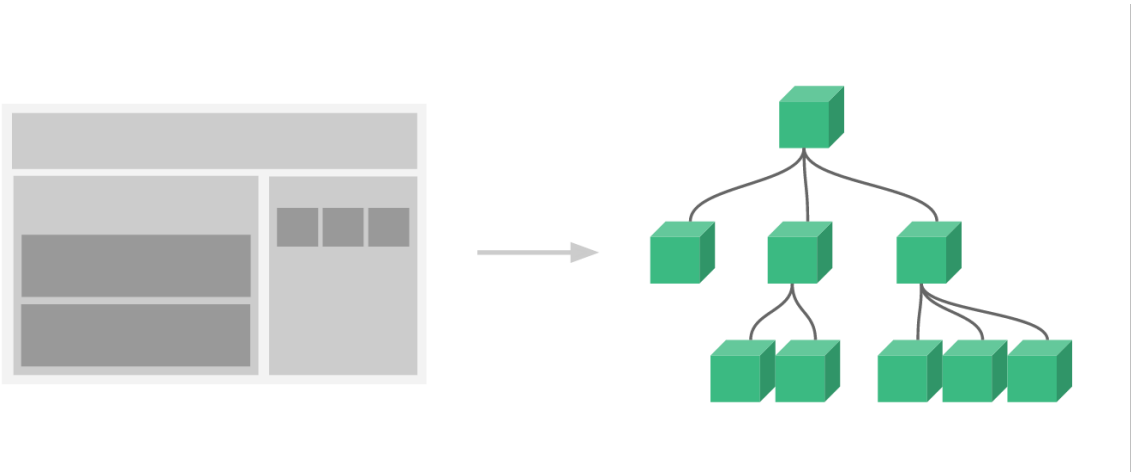
Express omogoča integracijo z izvajalnim okoljem v dveh korakih. Prvi korak je, da naložimo express.js preko ukazne vrstice. Da ga lahko uporabljamo v sami aplikaciji, ga moramo le še poklicati (glej sliko 4). Deluje pa na principu zahtevkov in odzivov. Iz zahtevka express sprejme podatke, ki jih potem obdela, nazaj pa vrne odziv. Odziv je lahko le status 200, ki pomeni, da vse deluje pravilno. V odzivu lahko tudi pošlje podatke, ki so najpogosteje zapisani v formatu JSON. V primeru, da pride do napake, nam to tudi javi z ustreznim statusom (npr. status 500 pomeni, da je prišlo do napake na strežniku).

```
const express = require("express");
const router = express.Router();
```

Slika 4: Klic knjižnice Express v aplikaciji, lasten vir

3.1.3 Vue.js

Vue (prebran /vju/, kot angl. *view*) je odprtokodno progresivno JavaScript ogrodje, namenjeno izdelavi uporabniških vmesnikov in enostranskih aplikacij, ki je zgrajeno na podlagi komponent (glej sliko 5). Za razliko od že obstoječih monolitnih ogrodij je bil Vue izdelan za prilagodljivost uporabniku. Kot je že razvidno iz imena, želijo graditelji ogrodja podati velik poudarek na izgledu, kar pa dosegajo z uporabo le vidnega sloja ter možnosti povezave in integracije z že mnogimi knjižnicami in obstoječimi projekti, kar je eden od mnogih razlogov za izbiro tega ogrodja kot čelni (angl. *front-end*) del najine aplikacije.



Slika 5: Uporaba komponent v ogrodju Vue, vir [4]

Zaradi uporabe deklaracijskega upodabljanja in sestave komponent je viden tudi velik napredek v hitrosti nalaganja in prikazovanja spletne strani uporabnikom. Kot je bila že omenjena možnost integracije z mnogimi knjižnicami, je pomembno poudariti tudi usmerjanje poti strani s pomočjo knjižnice Vue Router, ki omogoča ugnezdeno preslikavo poti oz. pogleda, določanje parametrov poti, shranjevanje zgodovine in vsesplošno urejen pregled stanja naslovov spletnih strani, s čimer pridobimo na hitrosti izvajanja.

Pomembno je tudi poudariti, da je aplikacija enostranska (angl. single-page), kar pomeni, da deluje znotraj brskalnika in ne potrebuje ponovnega nalaganja strani med svojim delovanjem. Je le ena sama stran, ki jo obiščemo in na kateri se nato naloži vso ostalo vsebino s pomočjo JavaScripta.

3.1.3.1 Zgradba ogrodja Vue

Kot je že bilo omenjeno na začetku tega poglavja, je ogrodje Vue zgrajeno na podlagi komponent. Komponente, ki jih lahko ponovno uporabimo, so lahko procesirane kot le manjši delujoč del kode in vdelane v različne dele aplikacije. Osnovna zgradba Vue komponente je prikazana spodaj.

```
<template>
  <!-- HTML del kode -->
  <h1 class="naslov">Pozdravljen svet!</h1>
</template>

<script>
//Tukaj uvozimo vse željene knjižnice in komponente
export default {
  components: {
    //Uporaba komponent
  },
  data: () => ({
    //Podatki aplikacije
  }),
  mounted() {
    //Kaj želimo izvesti, ko se stran naloži
  },
  props: {
    //Določitev tipa podatkov, ki bodo uvoženi iz drugih komponent
  },
  computed: {
    //Izvedba funkcij ob spremembah
  },
  methods: {
    //Vse metode in funckije
  },
};
</script>

<style scoped> /* CSS del kode */
.naslov {
    font-size: 20px;
    color: rgb(45, 45, 45)
}
</style>
```

Slika 6: Osnovna zgradba Vue komponente, lasten vir

Kot je razvidno s slike 6, je vsaka Vue komponenta zgrajena iz treh ločenih segmentov: <template> (HTML), <script> (JavaScript) in <style> (CSS). Vue uporablja sintakso predloge na podlagi HTML-ja, ki omogoča razvijalcu združitev upodobljenega objektnega modela dokumenta s podatki, ki so deklarirani v Vue komponenti. Te predloge vsebujejo preprost HTML, ki ga lahko razumejo in obdelajo vsi brskalniki, operacijski sistemi in HTML-razčlenjevalniki. Vse zaledne funkcije, kot so API-klici, izvedba ukazov, povezave z bazo, zaledne metode …, so obravnavane v <script> sekciji. V zadnji sekciji, imenovani <style>, je oblika komponente ustvarjena s CSS ali poljubnim predprocesorjem, ki omogoča razvijalcu uporabo lastnosti, ki niso del splošnega CSS-standarda.

3.1.4 Vuex

Vuex je vzorec in knjižnica za upravljanje stanj v Vue aplikaciji. Deluje kot centralizirana shramba za vse komponente v aplikaciji s pravili, ki zagotavljajo, da je lahko stanje mutirano le na predvidljiv način.

3.1.4.1 Vzorec za upravljanje stanj

Najlažje se nama je zdelo prikazati sam koncept vzorca za upravljanje stanj s kratko Vue aplikacijo, ki omogoča prištevanje števil (glej sliko 7). Ta aplikacija je sestavljena iz treh delov.

- stanje (angl. state) je vir resnice, ki ga uporablja aplikacija,
- pogled (angl. view) omogoča deklarativno preslikavanja stanja,
- akcije oz. dejanja (angl. actions) so možni načini spreminjanja stanja, ki se zgodijo ob interakciji uporabnika s stranjo.

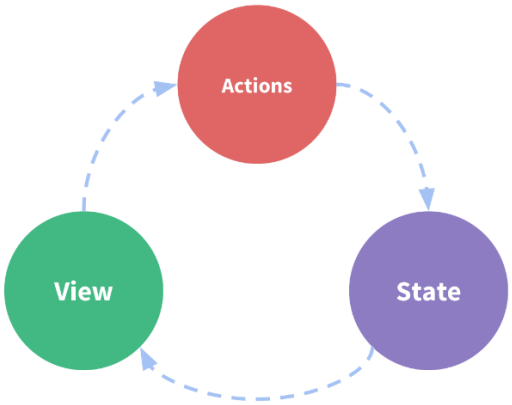
```
<template>
  <!-- Pogled -->
  <div>{{ stetje }}</div>
</template>

<script>

export default {
  //Stanje
  data: () => ({
    stetje: 0,
  }),
  //Dejanja oz. akcije
  methods: {
    pristej() {
      this.stetje++;
    },
  },
};
</script>
```

Slika 7: Primer vzorca za beleženje stanj, lasten vir

3.1.4.2 Težave enosmernega toka podatkov

Vue uporablja za privzeti način upravljanja stanj tako imenovani koncept enosmernega toka podatkov (glej sliko 8). Ta lahko postane hitro preveč kompleksen ali pa nezanesljiv, ko si več komponent deli skupno stanje. Primeri, v katerih lahko postane to kompleksno:

- Več pogledov se nanaša na isti del stanja.
- Akcije iz različnih pogledov morajo spreminjati isti del stanja.

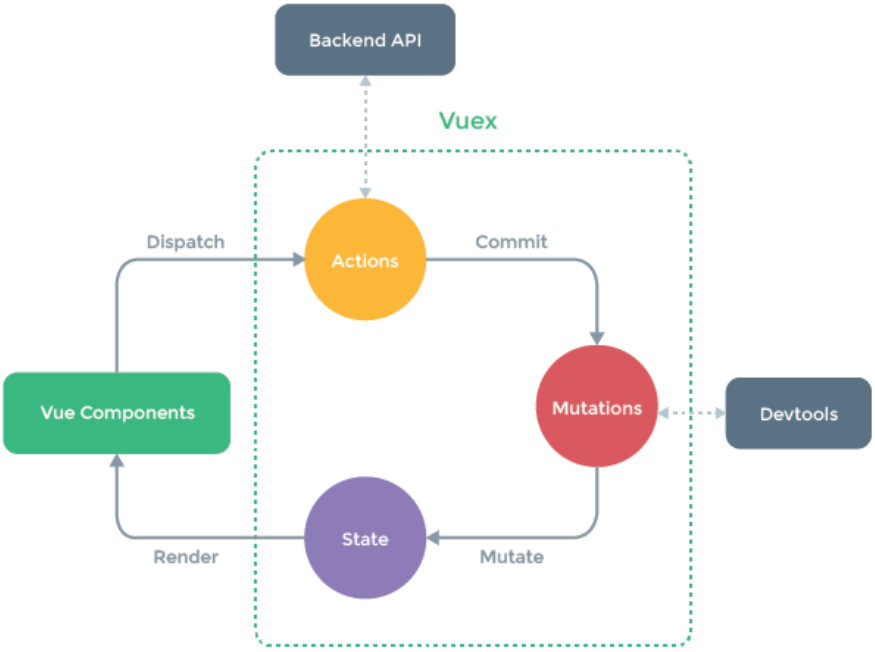


Slika 8: Koncept enosmernega toka podatkov, vir [9]

Rešitev prvega problema je posredovanje lastnosti globoko gnezdenim komponentam, kar pa je zamudno in ne deluje za povezane komponente. Pri drugem problemu pa poseganje direktno v starševske komponente ali poskušanje posodabljanja stanj skozi dogodke vodi v kodo, ki jo je težko vzdrževati.

3.1.4.3 Rešitev problemov z uporabo knjižnice Vuex

Delovanje knjižnice Vuex je zelo podobno že obstoječim knjižnicam, kot so Flux, Redux ali arhitektura Elm. Ideja je, da deljeno stanje izločimo iz komponent in ga upravljano v enem globalnem središču. Tako komponentno drevo postane eno veliko drevo, ki omogoča vsem komponentam, da dostopajo do podatkov in na njih kličejo akcije oz. metode, s čimer pa se znebimo odvisnost pogledov od stanja.



Slika 9: Koncept upravljanja deljenega stanja, vir [9]

Delovanje Vuex knjižnice temelji na več glavnih konceptih, kot so:

- Stanje (angl. *state*) - Vuex uporablja eno drevo stanja, ki je edini objekt, ki vsebuje celotno stanje aplikativnih plasti. To pomeni, da imamo eno shrambo za celotno Vue aplikacijo. Za večjo preglednost pa lahko stanje razdelimo tudi v več različnih modulov.
- Pridobivalci (angl. *getters*) – So kot izračunane lastnosti stanja. Torej lahko filtriramo ali izvajamo poljubne operacije in jih tako izpostavimo kot pridobivalce.
- Mutacije (angl. *mutations*) – So edini način spreminjanja stanj v Vuex stanju. Podobne so dogodkom, vsaka mutacija pa ima tekstovni tip in funkcijo, ki izvede spremembo stanja.
- Akcije (angl. actions) – Akcije so podobne mutacijam, razlikujejo se po tem, da namesto spreminjanja stanja sprožijo mutacije in lahko vsebujejo asinhrone operacije
- Moduli (angl. *modules*) – Zaradi uporabe samo enega drevesa stanja je celotno stanje aplikacije v enem velikem objektu, kar v večjih aplikacijah postane nakopičeno in težko

vzdržljivo. Zato lahko stanja delimo v module, ta pa lahko vsebujejo svoja stanja, mutacije, akcije, pridobivalce in pa tudi gnezdene module.

3.2 Izbira podatkovne baze

Za podatkovno bazo, kjer se shranjujejo vsi podatki aplikacije, sva izbrala dokumentno podatkovno bazo, imenovano MongoDB, saj je najbolj priljubljena in uveljavljena med NoSQL-podatkovnimi bazami (glej sliko 10). Razlog za izbiro NoSQL-baze in ne SQL-baze je sama struktura in večja hitrost, ki nam jo le-ta omogoča. Glavna razlika je, da MongoDB ne pozna relacij med dokumenti, kot jih pozna relacijska podatkovna baza. Odvisnosti rešuje drugače, in sicer z združitvenimi operacijami (angl. *join operations*). Najbolj pogosto se podatke odvisnega dokumenta doda kot poddokument (glej sliko 11). Primer tega pri najini aplikaciji je povezava projektov in njegovih elementov z uporabniki.

Rank			DBMS	Database Model	Score		
Mar 2021	Feb 2021	Mar 2020			Mar 2021	Feb 2021	Mar 2020
1.	1.	1.	Oracle	Relational, Multi-model	1321.73	+5.06	-18.91
2.	2.	2.	MySQL	Relational, Multi-model	1254.83	+11.46	-4.90
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1015.30	-7.63	-82.55
4.	4.	4.	PostgreSQL	Relational, Multi-model	549.29	-1.67	+35.37
5.	5.	5.	MongoDB	Document, Multi-model	462.39	+3.44	+24.78
6.	6.	6.	IBM Db2	Relational, Multi-model	156.01	-1.60	-6.55
7.	7.	↑ 8.	Redis	Key-value, Multi-model	154.15	+1.58	+6.57
8.	8.	↓ 7.	Elasticsearch	Search engine, Multi-model	152.34	+1.34	+3.17
9.	9.	↑ 10.	SQLite	Relational	122.64	-0.53	+0.69
10.	↑ 11.	↓ 9.	Microsoft Access	Relational	118.14	+3.97	-7.00
11.	↓ 10.	11.	Cassandra	Wide column	113.63	-0.99	-7.32
12.	12.	↑ 13.	MariaDB	Relational, Multi-model	94.45	+0.56	+6.10
13.	13.	↓ 12.	Splunk	Search engine	86.93	-1.61	-1.59
14.	14.	14.	Hive	Relational	76.04	+3.72	-9.34
15.	↑ 16.	15.	Teradata	Relational, Multi-model	71.43	+0.53	-6.41
16.	↓ 15.	↑ 23.	Microsoft Azure SQL Database	Relational, Multi-model	70.88	-0.41	+35.44
17.	17.	↓ 16.	Amazon DynamoDB	Multi-model	68.89	-0.25	+6.38

Slika 10: Priljubljenost podatkovnih baz, vir [5]

Ena izmed značilnosti MongoDB-ja, ki ga razlikuje od drugih podatkovnih baz in jo je vredno omeniti, je BSON (Binary JSON – JavaScript Object Notaition). Ta služi za komunikacijo med aplikacijskim in podatkovnim nivojem. Na primer dokument, ki se ga želi shraniti, se na aplikacijskem nivoju pretvori v BSON-dokument in tega pošlje podatkovni bazi. Njegove

prednosti so, da se lahko z njim prenašajo poljubni podatki, in ker je binarne oblike, je prenos zelo hiter.

Ostale funkcionalnosti MongoDB-podatkovne baze sovpadajo z osnovnimi značilnostmi NoSQL-podatkovnih baz, kot je porazdeljen sistem ali horizontalna skalabilnost, drobljenje podatkov in replikacija podatkov.

Tabela 1: Primerjava relacijske podatkovne baze s podatkovno bazo MongoDB, lasten vir

Relacijska podatkovna baza	**MongoDB**
Tabela	Kolekcija
Vrstica	Dokument
Stolpec	Polje dokumenta
Spoj	Poddokument
Tuji ključ	Referenca na dokument

Elementi, ki jih prikazuje tabela, so si med seboj podobni, vendar precej različni v sintaksi in načinu delovanja. Prva razlika je v tem, da relacijska podatkovna baza definira podatke na nivoju tabele. To pomeni, da so podatki v vrsticah tabel med seboj strukturno enaki, medtem ko MongoDB definira podatke na nivoju dokumenta, kar pomeni, da se lahko dokumenti v isti kolekciji med seboj strukturno razlikujejo. Kolekcije pa služijo zgolj kot mape, v katere odlagamo dokumente, podobno kot na datotečnem sistemu.

```
const ItemSchema = new Schema({
    title: {
        type: String,
        required: true
    },
    description: {
        type: String,
        required: false
    },
    tags: {
        type: String,
        required: false
    },
    deadline: {
        type: Date,
        required: false
    },
    dateAdd: {
        type: Date,
        required: true
    },
    dateModify: {
        type: Date,
        required: true
    },
    parentItem: {
        type: mongoose.Types.ObjectId,
        ref: 'Item'
    },
    owner: {
        type: mongoose.Types.ObjectId,
        ref: 'User',
        required: true
    },
```

Slika 11: Model elementa poddokumenta parentItem in owner, lasten vir

3.3 GitHub

GitHub je ena izmed največjih in najnaprednejših razvijalnih platform za oblačno shranjevanje programske kode. Ena od glavnih prednostih platforme GitHub je, da beleži vso zgodovino kode. To lahko uporabimo, kadar gre kaj narobe in s preprostim klikom obnovimo aplikacijo na zadnjo delujočo verzijo. Zelo priročen je tudi, ko aplikacijo razvija skupina razvijalcev oziroma v najinem primeru midva, saj je vsa koda shranjena na oblaku in imava do nje dostop ves čas, kar nama omogoča razvoj dostop do zadnje verzije aplikacije, kar prihrani veliko časa. Priročna funkcija platforme so tudi združevanje dokumetov oz. datotek, kar nam omogoča

sočasno razvijanje programske kode, ki se potem združi s funkcijo združevanja (angl. *merging*).

Slika 12: Logotip platforme GitHub, vir [8]

3.4 Heroku

Heroku je eden najpopularnejših platform za gostovanje spletnih aplikacij med razvijalci. Razloga za to sta predvsem dva in ta sta, da je Herkou brezplačen za aplikacije, ki so velike do 500 MB, po tem pa omogoča preprosto nadgraditev. Drugi razlog je, da je namestitev aplikacije na gostovanje skoraj avtomatizirana, kar nam prihrani veliko časa, in če uporabljamo Heroku, nam ni treba skrbeti za vzdrževanje strežnikov, strojne opreme in infrastrukture, kar nam omogoča, da se osredotočimo na varnost in sam razvoj aplikacije.

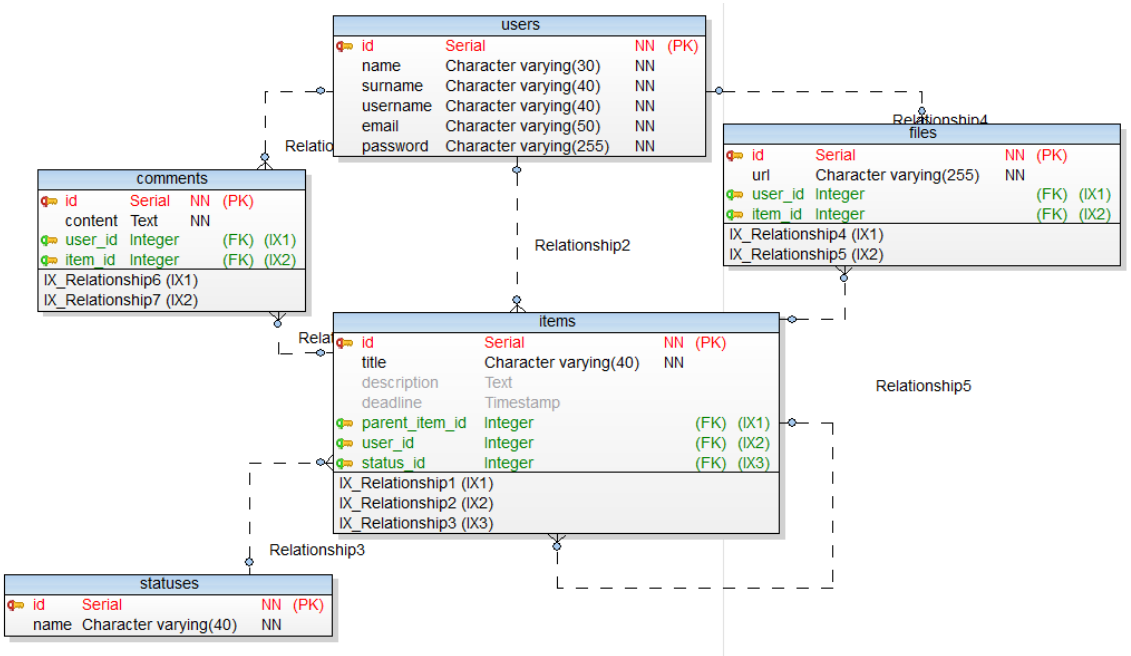
Slika 13: Logotip podjetja Heroku, vir [10]

4. IZDELAVA APLIKACIJE

Izdelava spletne aplikacije je kompleksna, zato sva jo razdelila na čelni del aplikacije, zaledni del aplikacije in podatkovno bazo. Za gostovanje podatkovne baze MongoDB sva izbrala kar globalnega ponudnika oblačnega gostovanja podatkovnih baz MongoDB, to je MongoDB Atlas. Zaledni del aplikacije predstavlja API, ki je narejen v Node.js in Expressu. S tem sva ločila zaledni in čelni del aplikacije, kar nama omogoča večjo preglednost. Čelni del aplikacije predstavlja Vue, ki komunicira z API s pomočjo zahtevkov. Oba dela aplikacije (čelni in zaledni) gostujeta na platformi Heroku.

4.1 Podatkovna baza

Podatkovno bazo sva najprej grafično skicirala kot relacijsko podatkovno bazo, da sva si jo lažje predstavljala. Za to sva uporabila program Toad Data Modeler in se odločila, da bo najin glavni dokument Item, katerega posebnost je, da ima sklic (angl. *reference*) na samega sebe, kar nama omogoča neskončno mnogo ugnezdenih kolekcij v kolekciji, kar pomeni, da imava lahko neskončno mnogo opravil v opravilih. Skico sva potem ustrezno prilagodila, da sva jo lahko uporabila pri izdelavi nerelacijske podatkovne baze MongoDB.



Slika 14: Grafična skica podatkovne baze v Toad Data Modelerju, lasten vir

Podatkovno bazo sva ustvarila na ponudniku oblačnega gostovanja MongoDB Atlas, kjer najina podatkovna baza sedaj gostuje. MongoDB Atlas sva izbrala, ker omogoča brezplačno shrambo podatkov do 512 MB, prav tako pa omogoča enostavno nadgradnjo plana, če bi potrebovala več prostora. Nudi nama tudi brezplačno podporo v primeru težav in dokumentacijo za povezavo na podatkovno bazo.

4.2 Zaledni del aplikacije

Zaledni del aplikacije je v najinem primeru programski vmesnik oz. API, narejen v izvajalnem okolju Node.js in s knjižnico Express. Deluje tako, da iz čelnega dela aplikacije dobi zahtevek s podatki v formatu JSON, in glede na dobljene podatke, vrne ustrezen rezultat v formatu JSON. V primeru napake ali neustreznih podatkov v zahtevi nam to javi v obliki odziva s statusom in opisom napake.

API je razdeljen na kontrolerje, modele in poti (angl. *routes*). V modelih so definirane sheme za vsak dokument v podatkovni bazi, medtem ko kontrolerji skrbijo za operiranje z zahtevki, poti pa skrbijo za usmerjanje zahtevkov do kontrolerjev, kar pomeni, da sprejmejo zahtevek in ga pošljejo do ustreznega kontrolerja.

4.2.1 Avtentikacija

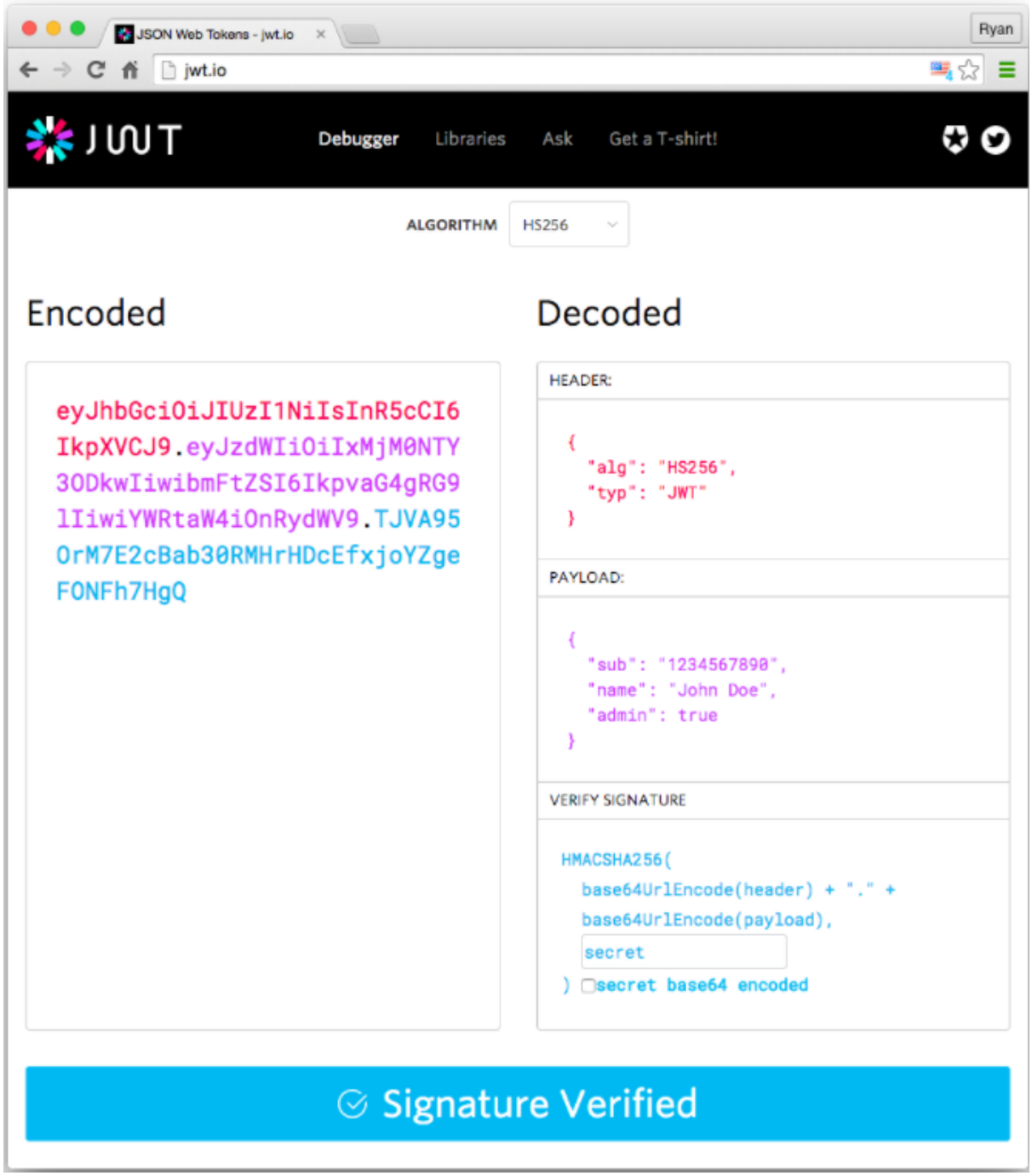
Za avtentikacijo sva uporabila knjižnico Passport, ki omogoča preko 500 strategij avtentikacije. Midva sva se odločila za JWT. JWT je odprtokoden standard, ki nam omogoča varen prenos informacij. Deluje tako, da shrani informacije v žeton, ki je sestavljen iz treh delov, ločenih s pikami. Prvi del se imenuje glava in v njem je shranjena vrsta žetona in vrsta zapisa, ki ga algoritem uporablja.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Slika 15: Primer glave JWT-žetona, vir [6]

Drugi del se imenuje tovor (angl. *payload*) in v njem so shranjene informacije, ki jih prenašamo, v najinem primeru so to podatki o uporabniku (npr. uporabniško ime). Informacije so v obeh deli zakodirane. Zadnji del žetona se imenuje podpisni del, zanj potrebujemo kodirano glavo, kodiran tovor, skrivnost (angl. *secret*) in algoritem, zapisan v glavi, dobimo pa ga s podpisom prej naštetih podatkov. Podpis uporabljamo za preverbo točnosti podatkov, da se ni kaj vmes izgubilo.

JWT deluje tako, da se žeton pošlje v glavi zahtevka po Bearer shemi (Bearer shema deluje na principu varnostnih žetonov, ki se pošiljajo v glavi zahtevkov. Drugače bi lahko razložili to shemo tudi kot preverjanje, če ima lastnik žetona dovoljen dostop do tistega dela spletne aplikacije, na katero pošilja zahtevek.) na strežniško stran aplikacije, v najinem primeru do programskega vmesnika, kjer se žeton dekodira in preveri, ali ima uporabnik dostop do klicanih virov.



Slika 16: Primer kodiranega in izvirnega žetona, vir [6]

4.2.1.1 Passport

Passport je knjižnica, ki nam olajša avtentikacijo na spletni aplikaciji, ki uporablja izvajalno okolje Node.js. Integracija v spletno aplikacijo zahteva le dva koraka, in sicer namestitev knjižnice preko ukazne vrstice in klic knjižnice v sami kodi aplikacije. Ko to naredimo, lahko uporabljamo vse funkcije knjižnice, za katere dokumentacijo najdemo na njihovi spletni strani. Passport je živa knjižnica, kar pomeni, da se razvija in sledi najnovejšim varnostnim

standardom, in to je velik plus za razvijalce, saj ni treba nam skrbeti za posodabljanje varnostnih standardov, ampak to naredi knjižnica namesto nas.

4.2.2 Povezava s podatkovno bazo

Za delo s podatkovno bazo sva uporabila knjižnico mongoose, ki je najbolj uporabljena in podprta knjižnica. Mongoose omogoča uporabo CRUD-metod za podatkovno bazo MongoDB in povezavo z le enim naslovom, v katerem sta skrita ime in geslo povezave, ki sta shranjena v lokalnih datotekah, kar poveča varnost povezave med aplikacijo in podatkovno bazo.

```
mongoose.connect(config.db, {
    useNewUrlParser: true,
    useUnifiedTopology: true
    // useCreateIndex: true,
    // useFindAndModify: false
});
```

Slika 17: Povezava aplikacije s podatkovno bazo, lasten vir

Poleg povezave s podatkovno bazo nam mongoose omogoča tudi, da naredimo shemo za vsak dokument posebej. V shemi definiramo vsa polja dokumentov in za vsako polje določimo podatkovni tip in obveznost. To nam pomaga pri preverjanju ustreznosti dobljenih podatkov s strani čelnega dela aplikacije oziroma uporabnika.

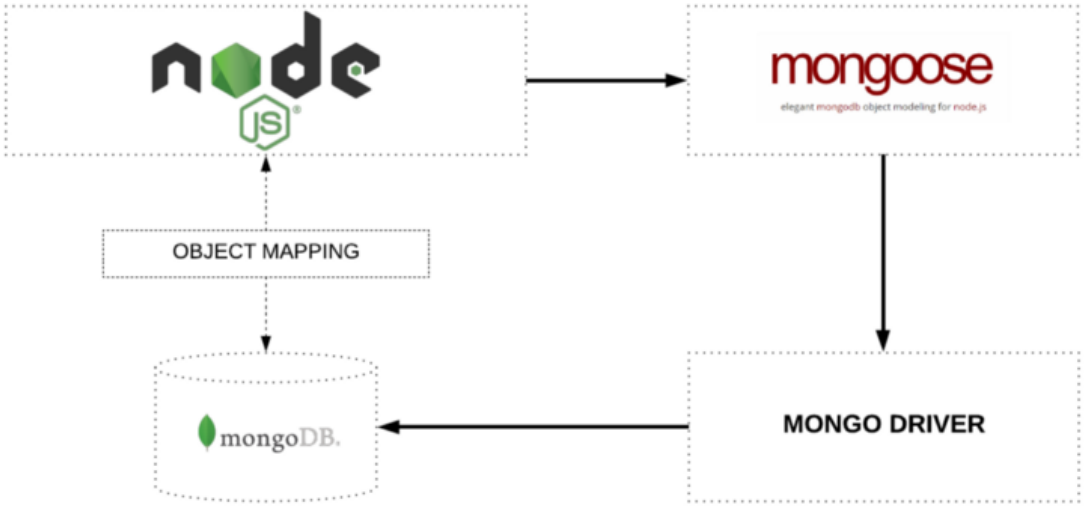
```
1  const mongoose = require('mongoose')
2  const Schema = mongoose.Schema;
3
4  const StatusSchema = new Schema({
5      name: {
6          type: String,
7          required: true
8      }
9  })
10
11 const Status = mongoose.model('Status', StatusSchema)
12 module.exports = Status
```

Slika 18: Primer sheme dokumenta, lasten vir

4.2.2.1 Mongoose

Mongoose je objektno relacijsko preslikovalno ogrodje (angl. *object-relational mapping framework*) za izvajalno okolje Node.js. Omogoča nam dostop do CRUD-ukazov za podatkovno bazo MongoDB.

Mongoose skrbi za relacije med podatki in poskrbi za prenos in prevod objektov v kodi programskega vmesnika v objekte, ki se shranjujejo v podatkovni bazi MongoDB. Poleg tega pa priskrbi še validacijsko shemo, ki preverja pravilno obliko podatkov, ki jih hočemo zapisati v podatkovno bazo.



Slika 19: Grafični prikaz povezave med MongoDB in node.js s knjižnico mongoose, vir [7]

4.2.3 Google Drive API

Google Drive API sva uporabila za povezavo spletne aplikacije z oblačnim skladiščem Google Drive. Z Google Drive API lahko iz Google Drive beremo datoteke in mape. Prav tako lahko z njim na oblačno skladišče nalagamo datoteke in ustvarjamo mape. Ponuja nam tudi funkcije za manipulacijo podatkov, kar pomeni, da lahko iščemo določene datoteke oz. mape na Google Drive.

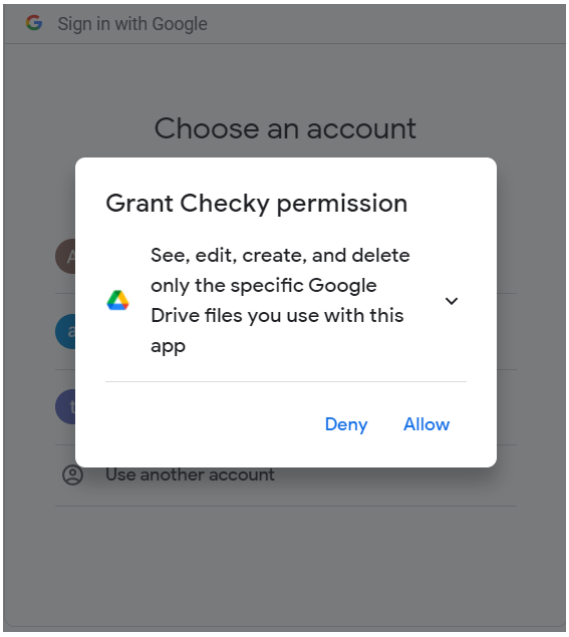
4.2.3.1 Integracija Google Drive API

Da lahko uporabljaš Google Drive API na svoji spletni aplikaciji, moraš na Google Cloud Platform ustvariti nov projekt. Od tam dobiš potrebne parametre (npr. id-uporabnika, id-projekta, skrivnost), da lahko API integriraš v svojo aplikacijo. Vse te parametre vsa shranila v lokalno datoteko.

Povezava na sam Google Drive deluje tako, da najprej prebereva vse parametre, ki sva jih dobilo od Google Cloud Platform iz lokalne datoteke. Vsi ti parametri se združijo v zahtevek in pošljejo na Google Drive API, ki naju, če so parametri pravilni, preusmeri na Google strežnik, kjer nama mora uporabnik dovoliti dostop do njegovega Google Drive računa. Dostop nama mora uporabnik dovoliti le, ko se odloči prvič povezati svoj uporabniški račun na najini aplikaciji z Google Drive računom, saj ko nama uporabnik omogoči dostop do Google Drive, nama Google Drive API pošlje aktivacijski žeton, s katerim imava trajen dostop do uporabnikovega Google Drive računa.

```
function getAccessToken(oAuth2Client, callback, code, res) {
  const authUrl = oAuth2Client.generateAuthUrl({
    access_type: 'offline',
    scope: SCOPES,
  });
  if(code != undefined && code != null){
    oAuth2Client.getToken(code, (err, token) => {
      if (err) return console.error('Error retrieving access token', err);
      oAuth2Client.setCredentials(token);
      // Store the token to disk for later program executions
      fs.writeFile(TOKEN_PATH, JSON.stringify(token), (err) => {
        if (err) return console.error(err);
        console.log('Token stored to', TOKEN_PATH);
      });       You, a week ago • working on google api
      callback(oAuth2Client);
      res.sendStatus(200);
    });
  }
  else{
    res.redirect(authUrl);
  }
}
```

Slika 20: Izvleček kode, ki poskrbi za shranjevanje aktivacijskega žetona, lasten vir



Slika 21: Dovoljenje aplikaciji za uporabo oblačnega skladišča Google Drive, lasten vir

4.2.3.2 Avtentikacija uporabnika

Pred vsakim klicem na Google Drive API morava uporabnika avtenticirati, to pa storiva tako, da najprej preveriva, če nama je uporabnik že dovolil dostop do njegovega Google Drive računa, oz. če imava njegov aktivacijski žeton. Če žetona nimava, pa ga preusmeriva na Google, da nama dovoli dostop in postopek ponoviva.

4.2.3.3 Sinhronizacija aplikacije z Google Drive

Ko se uporabnik odloči sinhronizirati svoje projekte z Google Drive oz. jih shraniti nanj, je potrebno uporabnika najprej avtenticirati. Po uspešni avtentikaciji se na Google Drive naredi za vsak projekt svoja mapa in za vsako opravilo v njem svoja mapa znotraj prejšnje mape. Poleg map se za vsako opravilo shranijo ostali podatki v JSON-datoteko. Na opravilo je lahko vezana tudi ena ali več datotek, ki se shranijo v enako mapo kot JSON-datoteka.

```
const drive = google.drive({
  version: 'v3',
  auth
});
var fileMetadata = {
  'name': 'Checky',
  'mimeType': 'application/vnd.google-apps.folder'
};
drive.files.create({
  resource: fileMetadata,
  fields: 'id'
}, function (err, file) {
  if (err) {
    // Handle error
    console.error(err);
  } else {
    console.log('Folder Id: ', file.data.id);
```

Slika 22: Izsek kode, ki ustvari mapo v Google Drive, lasten vir

Uporabnik lahko uvozi podatke tudi iz Google Drive. To pa poteka tako, da se za vsako glavno mapo ustvari projekt in za vse mape znotraj glavne mape ustvarijo opravila. Da se opravila pravilno ustvarijo, mora mapa vsebovati tudi JSON-datoteko, v kateri so zapisani še ostali potrebni podatki za uvoz opravila. Na njo se vežejo tudi vse datoteke, ki so v mapi opravila. Najpomembnejši del pri tem je, da preverimo prej vsa opravila oz. mape, ki že obstajajo, da ne pride do podvajanja podatkov.

```
for (let x = 0; x < Items.length; x++) {
    for (let y = 0; y < Items[x].items.length; y++) {
        let itemId = [];
        if (Items[x].items[y].parentItem) {
            if (ids.length > 0) {
                itemId = ids.map(id => {
                    if (id.item_id == Items[x].items[y]._id) {
                        return id.folder_id
                    } else {
                        return null
                    }
                });
            }
        } else {
            itemId.push(token.folder_id);
        }

        var fileMetadata = {
            'name': Items[x].items[y].title,
            'mimeType': 'application/vnd.google-apps.folder',
            parents: [itemId[0]]
        };
        itemId = []
        drive.files.create({
            resource: fileMetadata,
            fields: 'id'
        }, function (err, file) {
```

Slika 23: Izsek kode, ki ustvari ugnezdene mape na Google Drive, lasten vir

4.3 Čelni del aplikacije

Čelni del aplikacije predstavlja uporabniški vmesnik in v najinem primeru povezavo le-tega z zalednim delom aplikacije. To vključuje oblikovanje, slike, barve, gumbe, obrazce, tipografijo, animacije in vsebine. Na kratko je to vse, kar lahko uporabnik spletne strani vidi.

Najin čelni del je napisan v Vue.js ogrodju in uporablja naslednje knjižnice:

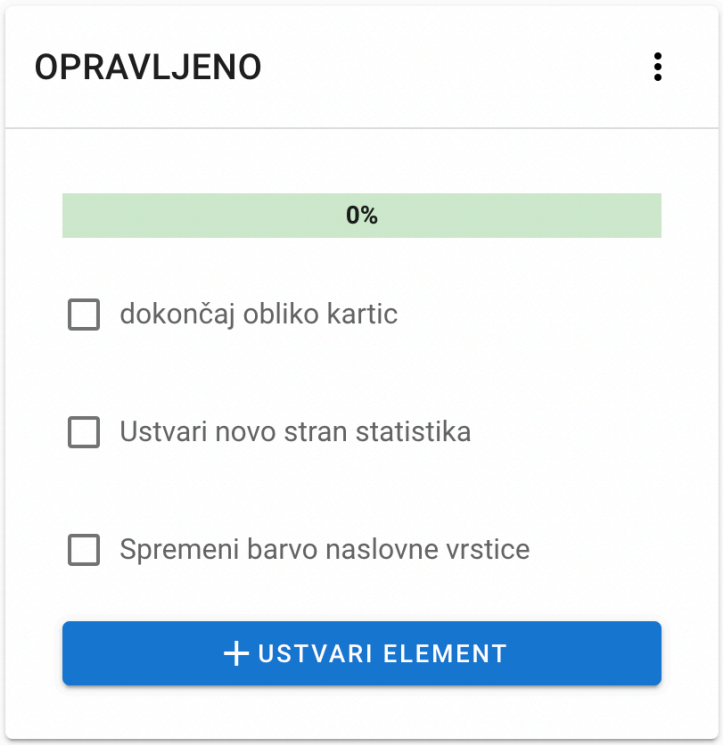
- vuetify – komponentno ogrodje,
- vuex – drevesna struktura stanja,
- vue-router – usmerjevalnik poti,
- vuex-persistedstate – shranjevanje stanja v LocalStorage,
- axios – knjižnica za Ajax zahtevke,
- jwt-decode – knjižnica za dekodiranje JSON-žetonov.

Za ustvarjanje projekta sva uporabila osnovno ogrodje za izdelavo in upravljenje Vue projektov, imenovano Vue CLI. Pri tem sva lahko izbrala, katere knjižnice želiva imeti nameščene in katere ne, ogrodje jih je nato samo namestilo v najino aplikacijo.

Slika 24: Ustvarjanje projekta v Terminalu z Vue CLI, lasten vir

4.3.1 Vuetify

Nato sva dodala še semantično kompaktno ogrodje Vuetify, ki nama je prihranilo čas pri oblikovanju same strani in nama tudi podalo možnosti že obstoječih komponent za obliko strani, kot so grafi, navigacijske vrstice, animacije, preddefinirane tipografije in barve ter seveda kartice, ki igrajo pomembno vlogo v izgledu samih projektov in seznamov (glej sliko 25).



Slika 25: Izgled kartice, kreirane v semantičnem ogrodju Vuetify, lasten vir

Tudi oblika ostalih delov strani je izdelana na enak način, saj sva želela čim bolj poenostaviti in izboljšati uporabniško izkušnjo pri izdelavi projektov.

4.3.2 Axios

Poleg oblike pa je pomembna tudi povezava z zalednim delom aplikacije, ki pa nam jo omogoča knjižnica, imenovana Axios. Ta skrbi za pošiljanje zahtevkov na naslov Node.js aplikacije, ki v tem primeru deluje kot API. V najini aplikaciji sva ustvarila več datotek za posamezne zahtevke, vendar glavna povezava se izvede v datoteki axios.js (glej sliko 26), kjer

s pomočjo naslova API-ja ustvariva spremenljivko Axios, ki počaka 10 sekund na odziv zalednega dela in če v tem času ni odziva, pošlje napako na uporabniški vmesnik. Če zazna JWT-žeton, tega tudi shrani v lokalno shrambo, ki pa omogoča avtentikacijo uporabnika. Na koncu datoteke se ta spremenljivka izvozi in je lahko uporabljena še v ostalih posameznik zahtevkih.

```
import axios from 'axios'
let baseurl = 'http://localhost:5000/api'


const axiosInstance = axios.create({
    baseURL: baseurl,
    timeout: 10000
})

axiosInstance.interceptors.request.use((config) => {
    const token = localStorage.getItem('checklist-jwt') || ''
    if (token) {
        config.headers.Authorization = `JWT ${token}`
    }
    return config
}, (err) => {
    return Promise.reject(err)
})

export default axiosInstance
```

Slika 26: Ustvarjanje in priprava knjižnice Axios, lasten vir

Kot sva že omenila, so datoteke, ki omogočajo pošiljanje zahtevkov na posamezne dele API-ja, ustvarjene ločeno. Razlog za to je le večja preglednost in urejenost same drevesne strukture aplikacije. Za začetek uvoziva v te datoteke spremenljivko Axios, ki sva jo opisala v prejšnjem odstavku in nato ustvariva metode za posamezno CRUD-operacijo. Kot primer, funkcija index kliče vse že ustvarjene projekte, v samo metodo pa moramo vstaviti tudi JWT-žeton (angl. *token*), saj je za sam zahtevek obvezna avtentikacija uporabnika, ki pa posledično v zalednem delu aplikacije pomaga najti vse uporabnikove projekte. V naslednjem delu kličemo Axios metodo GET na naslov API, ki sva ga določila v axios.js datoteki in zraven dodava še del 'project', tako da je celoten naslov http://localhost:5000/api/project. Te funkcije se pa potem izvedejo na uporabniškem vmesniku, ob nalaganju strani, ob spremembah elementov ali ob samih klikih na gumbe.

```
import axios from '@/services/axios'
export default {
    index(token) {
        return axios.get('project', {
            headers: {
                Authorization: 'Bearer ' + token,
            }
        })
    },
    single(id, token) {
        return axios.get(`project/${id}`, {
            headers: {
                Authorization: 'Bearer ' + token,
            }
        })
    },
    post(data, token) {
        return axios.post('project', data, {
            headers: {
                Authorization: 'Bearer ' + token,
            }
        })
    },
```

Slika 27: Ustvarjanje metod CRUD s pomočjo knjižnice Axios, lasten vir

Izvajanje omenjenih funkcij je izvedeno na strani Vue datotek v sekciji methods. V spodaj navedenem primeru je definirana asinhrona funkcija loadProjects() (slika 26), ki je izvedena ob nalaganju strani in kliče funckijo, opisano v prejšnjem odstavku, imenovano index (angl. *token*), ki pa kot parameter sprejme JWT-žeton, ki je pa shranjen kot stanje v knjižnici Vuex. Če dobiva odziv iz najinega API-ja, tega shraniva v spremenljivko response, in te datoteke shraniva v Vue spremenljivko projects, ki pa potem naloži in prikaže vse uporabnikove projekte. Če pride do napake, se nama ta zapiše v konzolo, da lahko preveriva, kaj je šlo narobe.

```
async loadProjects() {
  try {
    const response = await ProjectService.index(this.$store.state.token);
    if (response) {
      this.projects = response.data.items;
    }
  } catch (err) {
    console.log(err);
  }
},
```

Slika 28: Funkcija loadProjects(), ki naloži vse uporabnikove projekte, lasten vir

4.3.3 Vuex

Sama sva uporabila knjižnico Vuex za shranjevanje podatkov o uporabniku ob njegovi avtentikaciji ter za shranjevanje JSON-spletnih žetonov ter za shranjevanje podatkov o projektih in njihovih elementih, kar nama je omogočilo uporabo teh stanj po vsej aplikaciji brez težavnosti pošiljanja podatkov preko povezanih komponent (glej sliko 29).

```
state: {
  user: null,
  token: null,
  statuses: null,
  isUserLoggedIn: false
},
mutations: {
  setToken(state, token) {
    state.token = token
    state.isUserLoggedIn = !!(token)
  },
  setUser(state, user) {
    state.user = user
  },
  setStatuses(state, statuses) {
    state.statuses = statuses
  }

},
actions: {
  setUser({
    commit
  }, user) {
    commit('setUser', user)
  },
  setToken({
    commit
  }, token) {
    commit('setToken', token)
  },
  setStatuses({commit}, statuses) {
    commit('setStatuses', statuses)
  }
},

})
```

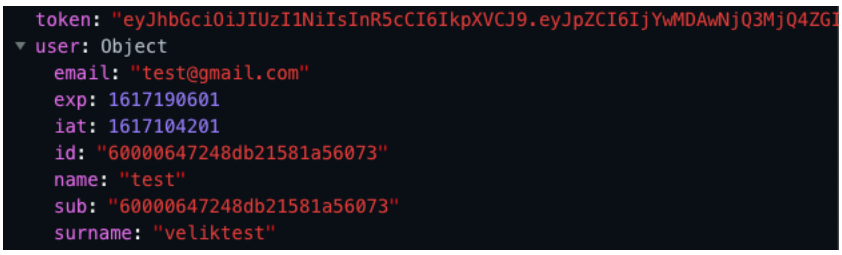
Slika 29: Prikaz glavnega Vuex modula, lasten vir

Za začetek sva se osredotočila na samo shranjevanje stanj ob avtentikaciji uporabnika, saj sva morala podatke uporabnika prikazovati na več različnih sorodnih komponentah in ob vsakih zahtevkih na zaledni del pošiljati tudi JSON-spletni žeton, ki nama je v zalednem delu po dekodiranju pokazal vse podatke uporabnika in ga tudi avtenticiral za potrditev vrnitve podatkov na čelni del aplikacije.

```
const response = await AuthenticationService.login({
  email: this.email,
  password: this.password,
});

if (response.data.token) {
  this.showPanel = true;

  setTimeout(() => {
    this.loginSuccess = true;
  }, 1500);

  const decoded = jwtDecode(response.data.token);

  setTimeout(() => {
    this.$store.dispatch("setToken", response.data.token);
    this.$store.dispatch("setUser", decoded);

    this.loginSuccess = false;
    this.showPanel = false;
    this.$router.push({ name: "home" });
  }, 2500);
} else {
  console.log("no response");
}
```

Slika 30: Prikaz uporabe stanj ob avtentikaciji uporabnika, lasten vir

Na sliki 30 je prikazana funkcija za prijavo uporabnika, ki na začetku pošlje zahtevek na zaledni del aplikacije s podatkoma o e-pošti in geslu ter ob uspešnem preverjanju vrne JSON- spletni žeton, ki se shrani v spremenljivko **response**. Zatem ta žeton dekodirava s pomočjo knjižnice **jwtDecode** in dekodiran žeton shraniva v spremenljivko **decoded**. Tako imava vse podatke o uporabniku ter sam žeton in lahko te podatke shraniva v Vuex shrambo s pomočjo akcij (angl. actions), ki smo jih prikazali v prejšnji sliki, **setToken** in **setUser**. Tako se ta dva podatka uspešno shranita kot stanja v najino aplikacijo (glej sliko 31) in ostaneta tam, dokler se uporabnik ne odjavi ali izbriše piškotke na brskalniku.



Slika 31: Primer shranjenih stanj uporabnika test, lasten vir

Zdaj, ko imava JSON-spletni žeton in uporabnika varno shranjena kot stanja v najini aplikaciji, pa lahko ta dva podatka kličemo, kadarkoli jih potrebujeva. Eden izmed takšnih primerov je metoda za ustvarjanje novega projekta (glej sliko 32).

```
async createProject() {
  this.waitBeforeClick = true;
  try {
    const response = await ProjectService.post(this.project, this.$store.state.token);
    if (response) {
      setTimeout(() => {
        this.waitBeforeClick = false;
        this.$router.push({
          path: "/",
        });
      }, 3000);
    }
  } catch (err) {
    console.log(err);
    setTimeout(() => (this.waitBeforeClick = false), 3000);
    setTimeout(() => (this.errors = []), 5000);
  }
},
```

Slika 32: Funkcija za ustvarjanje novega projekta, lasten vir

Metoda **post** v storitvenem modulu **ProjectService** zahteva dva parametra. Projekt v obliki objekta in pa JSON-spletni žeton, ki smo ga shranili kot stanje s pomočjo knjižnice Vuex. Le s pomočjo ene kratke vrstice kode **this.$store.state.token** dobimo iz naše Vuex shrambe podatek o žetonu, ki je bil shranjen ob prijavi uporabnika. Na enak način deluje tudi pridobivanje podatkov o uporabniku.

Samo ustvarjanje vseh metod in mutacij je lahko malce časovno zahtevno, vendar je nama pri aplikaciji prihranilo veliko časa na dolgi rok in tudi izboljšala se nama je organiziranost in urejenost kode, kar pa je pri takem velikem projektu pomemben del izdelave.

4.4 Od uporabnika do podatkovne baze

Postopek pridobivanja informacij, ustvarjanja novih elementov in projektov, posodobaljanje in brisanje le-teh si ves čas sledi po istem kopitu. Iz najinega uporabniškega vmesnika pošljeva zahtevo s pomočjo knjižnice Axios na zaledni del aplikacije, kjer se ta zahteva procesira in glede na zahtevan spletni naslov izvede že v naprej definirana funkcija, v kateri se izvede zahteva na podatkovno bazo in ta v primeru nobenih napak vrne željene podatke. Ti se shranijo v novo spremenljivko, ki je nato v obliki JSON poslana nazaj na čelni del aplikacije. Tam se vrnjeni podatki shranijo v novo spremenljivko, in če želimo, lahko te podatke tudi prikažemo na uporabniškemu vmesniku, ali, tako kot to imenuje Vue, na pogledu. Vsa povezava med temelji aplikacije je prikazana spodaj (glej sliko 33).

Slika 33: Primer povezave med čelnim delom, zalednim delom in podatkovno bazo, vir [10]

4.4.1 Primer povezave

Kot primer bova opisala delovanje ustvarjanja seznamov in elementov v le-teh. Kot je bilo že omenjeno, se povezava začne na strani čelnega dela aplikacije. Ta je razdeljen na več delov, ki jih bova opisala v nadaljevanju.

4.4.1.1 Čelni del povezave

Za začetek ustvariva primerne povezave na najin Node.js API s pomočjo Axios knjižnice. Za ustvarjanje elementov in seznamov imava definirani dve metodi (glej sliko 34), ki potrebujeta za pošiljanje zahteve tri parametre:

- data (podatki o elementu in seznamu) – V najinem primeru je to le ime elementa in seznama.

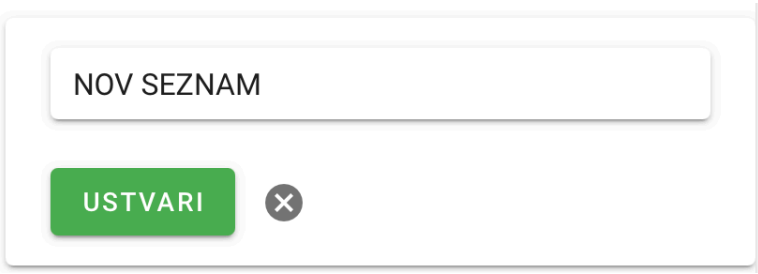
- parentId (identifikacijska številka starševskega dokumenta) – Potrebna je, saj morava povezati seznam s projektom oz. element s seznamom.
- token (JSON-spletni žeton) – Ob vsaki zahtevi je potreben tudi JSON-spletni žeton prijavljenega uporabnika, saj lahko s pomočjo tega na zalednem delu aplikacije varno prebereva podatke o uporabniku

Oba klica opravita enako nalogo in sta tudi enaka v zapisu, vendar sta bila razdeljena na dve metodi, za večjo organiziranost in branje kode.

```
postList(data, parentId, token) {
    return axios.post('item', {
        parentItem: parentId,
        title: data
    }, {
        headers: {
            Authorization: 'Bearer ' + token,
        }
    })
},
postItem(data, parentId, token) {
    return axios.post('item', {
        parentItem: parentId,
        title: data
    }, {
        headers: {
            Authorization: 'Bearer ' + token,
        }
    })
},
```

Slika 34: Metode za ustvarjanje seznamov in elementov s pomočjo knjižnice Axios, lasten vir

Sedaj, ko sva ustvarila metode, morava klicati metode s strani pogleda oz. uporabniškega vmesnika. Na spodnji sliki 35 sta razvidna dva ključna dela ustvarjanja novega seznama. v polje za besedilo se vnese ime seznama in se nato klikne na gumb ustvari, ki izvede funkcijo, v kateri se izvede metoda postList iz prejšnjega odstavka, ki pošlje zahtevo na zaledni del aplikacije.



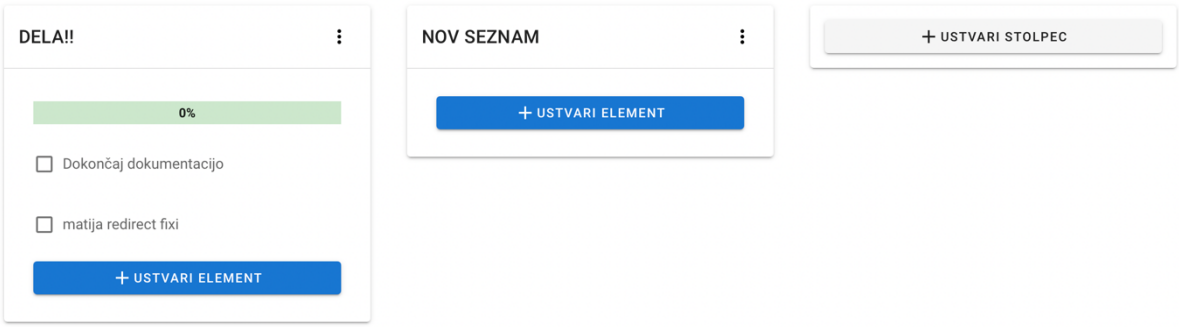
Slika 35: Oblika ustvarjanja novega seznama v aplikaciji, lasten vir

Funkcija, ki ustvari seznam, je asinhronega tipa (glej sliko 36) in kot sem že omenil, izvede funkcijo **postList()**, ki v spremenljivko **id** shrani identifikacijsko številko projekta, ki jo prebere s spletnega naslova. Zatem pa pošlje zahtevo na zaledni del aplikacije in odziv shrani v spremenljivko **response**. V primeru uspešne zahteve se novi podatki shranijo v stanje, imenovano **lists,** in se znova izvede funkcija **loadLists()**, ki pošlje zahtevo **GET** na zaledni del in vrne vse sezname v projektu.

```
async createList() {
  try {
    const id = this.$route.params.id;
    const response = await ItemService.postList(
      this.list_name,
      id,
      this.$store.state.token
    );
    if (response) {
      this.lists = response.data.items;
      this.loadLists();
      this.list_name = "";
    }
  } catch (err) {
    setTimeout(() => (this.waitBeforeClick = false), 3000);
    setTimeout(() => (this.errors = []), 5000);
  }
},
```

Slika 36: Metoda za ustvarjanje seznamov v čelnem delu aplikacije, lasten vir

Po končanem izvajanju funkcije se stran osveži in se prikaže nov seznam, v našem primeru imenovan »NOV SEZNAM«.



Slika 37: Prikaz seznamov po uspešnem ustvarjanju novega, lasten vir

4.4.1.2 Zaledni del povezave in podatkovna baza

Ko pride zahtevek do programskega vmesnika, ga ta najprej prebere in preusmeri na pravilno pot (angl. *routes*), ki so definirane v glavni datoteki **index.js** (glej sliko 38).

```
//ROUTES//

const user = require('./routes/user')
const project = require('./routes/project')
const item = require('./routes/item')
const status = require('./routes/status')

app.use("/api/user", user)
app.use("/api/project", project)
app.use("/api/item", item)
app.use("/api/status", status)
```

Slika 38: Definiranje poti v glavni datoteki index.js, lasten vir

```
router.post("/", passport.authenticate("jwt", {
    session: false
}), ItemController.post)
```

Slika 39: Primer kode, ko API prebere in preusmeri zahtevek, lasten vir

Kot vidimo na zgornji sliki 39 se pri zahtevku najprej preveri spletni naslov in metoda, v tem primeru gre za metodo post (metoda post se uporablja za shranjevanje v podatkovno bazo).

Knjižnica passport potem preveri, če ima uporabnik, ki pošilja zahtevek, dostop do sledečega dela kode, in če ima, ga preusmeri na ustrezen kontroler.

```
async post(req, res) {
    try {
        console.log(req.body)
        const {
            title,
            description,
            tags,
            deadline,
            parentItem,
            status
        } = req.body;

        const item = new Item({
            title,
            description,
            tags,
            deadline,
            dateAdd: new Date().getTime(),
            dateModify: new Date().getTime(),
            parentItem,
            owner: req.user.id,
            status
        });

        await item.save()

        res.status(200).json({
            msg: 'Successfuly added an item'
        })

    } catch (e) {
        res.send({
            message: "Error creating item"
        });
    }
```

Slika 40: Primer kode za kontroler seznama, lasten vir

V kontrolerju najprej preverimo, če smo dobili vse podatke, ki jih zahteva podatkovna baza. Te podatke preberemo iz jedra zahtevka. Če je vse v redu, mongoose funkcija .save() poskrbi, da se seznam shrani v podatkovno bazo in čelni del aplikacije vrne odziv s statusom 200, kar pomeni, da se je seznam uspešno shranil. V primeru napake, oz. če nismo dobili vseh zahtevanih podatkov, programski vmesnik vrne napako.

5. RAZPRAVA

Poznamo veliko aplikacij za načrtovanje opravil, kot so Asana, Trello, ClickUp in podobne aplikacije, ki nudijo najrazličnejše funkcije, kot so npr. razdelitev opravil med uporabniki, pogledi nad opravili in možnosti komunikacije znotraj aplikacije. Midva pa sva se odločila, da se bova osredotočila na tiste funkcije, ki sva jih pri že obstoječih rešitvah pogrešala, in to je predvsem zasebnost podatkov in dvosmerna sinhronizacija med opravili in oblačnim skladiščem. Ker sva se odločila aplikacijo povezati z oblačno storitvijo, sva sklenila, da bova uporabnikom omogočila tudi nalaganje velikih datotek na najino aplikacijo oz. na njihovo oblačno skladišče. Zasebnost njihovih podatkov bova zagotoviva tako, da nimava na aplikaciji shranjene nobene datoteke, ampak vse shraniva na njihovo oblačno skladišče.

Najina aplikacija omogoča dvosmerno sinhronizacijo med opravili in oblačnim skladiščem Google Drive. Meniva, da to uporabniku omogoča več svobode pri njegovih opravilih in načrti, saj ni vezan le na najino spletno aplikacijo. Če pa bi se radi vrnili na uporabo aplikacije, je to zelo preprosto, saj omogoča aplikacija sinhronizacijo s storitvijo Google Drive.

Najina aplikacija se od ostalih razlikuje tudi v tem, da ima uporabnik vse podatke, ki jih je zaupal najini aplikaciji, varno shranjene na njegovem oblačnem skladišču Google Drive. To mu daje popoln nadzor nad njegovimi podatki, kar meniva, da je dandanes še posebej pomembno.

5.1 Pregled hipotez

Po končani raziskovalni nalogi in izdelavi spletne aplikacije z dvosmerno sinhronizacijo med opravili in oblačnim skladiščem sva si nabrala dovolj znanja, da sva lahko potrdila oz. ovrgla zadane hipoteze.

1. **S spletno aplikacijo se lahko povežemo preko programskega vmesnika (API) na poljubno oblačno skladišče.**

Spletno aplikacijo sva prek Google Drive API povezala na oblačno skladišče Google Drive. Ni pa nama uspelo povezati spletne aplikacije na poljubno oblačno skladišče, oz. nama je za to zmanjkalo časa. Ugotovila sva tudi, da vsako oblačno skladišče uporablja svoj API in zato je bila hipoteza, da se bova lahko z aplikacijo povezala na poljubno oblačno skladišče preveč

optimistična. Vseeno bi bilo aplikacijo dobro povezati še s kakšnim oblačnim skladiščem, kot sta AWS in Microsoft OneDrive. Hipotezo sva delno potrdila, saj nama je uspelo aplikacijo povezati na oblačno skladišče Google Drive, ne pa tudi na poljubno oblačno skladišče.

2. **Preko programskega vmesnika (API) lahko v oblačnih skladiščih ustvarjamo mape in nalagamo datoteke.**

Preko programskega vmesnika Google Drive API ustvarjava mape za potrebe dvosmerne sinhronizacije med aplikacijo in oblačnim skladiščem, prav tako na oblačno skladišče Google Drive shranjujeva datoteke, ki jih uporabnik naloži na spletno aplikacijo, zato lahko hipotezo potrdiva.

3. **Preko programskih vmesnikov (API) lahko ustvarimo dvosmerno sinhronizacijo med opravili in mapami v oblačnem skladišču.**

Preko programskega vmesnika Google Drive API narediva za vsako opravilo svojo mapo, prav tako pa lahko s programskim vmesnikom iz spletnega skladišča uvoziva mape in datoteke v aplikacijo. Torej sva dosegla dvosmerno sinhronizacijo za oblačno skladišče Google Drive, in zato sva hipotezo potrdila.

4. **Skriptni programski jezik JavaScript ni dovolj zmogljiv za strežniški del.**

Skriptni programski jezik v osnovi ni bil narejen za strežniški del aplikacij, ampak mu to omogoča Node.js, ki JavaScript kodo poganja zunaj spletnega brskalnika, za katerega je bil prvotno narejen programski jezik JavaScript, in sicer jo poganja v V8 motorju in prav to nam omogoča, da lahko uporabljamo skriptni programski jezik JavaScript za strežniški del aplikacij. Zato sva hipotezo ovrgla.

5.2 Možne izboljšave

Sprva sva si zamislila, da bi najina aplikacija omogočala povezavo na poljubno oblačno skladišče, kar sva kasneje ugotovila, da je prezahtevno, saj uporablja vsako oblačno skladišče svoj programski vmesnik, s katerim se je mogoče povezati nanj. Vseeno pa bi bilo dobro aplikacijo povezati vsaj še s kakšnim oblačnim skladiščem, kot sta Microsoft OneDrive in AWS. Aplikacijo sva izdelala takšno, kot bi jo midva uporabljala in upoštevala mentorjeve predloge. Zavedava pa se, da nisva pomislila na vse, kar bi uporabnik najine aplikacije potreboval oz. želel, zato bi bilo dobro dati aplikacijo testirati in bi tako izvedela več o

funkcijah, ki jih aplikacija nima, bi pa si jih uporabniki želeli, to je o pomanjkljivostih in potencialnimi napakami.

6. ZAKLJUČEK

Spletna aplikacija omogoča dvosmerno sinhronizacijo med opravili in oblačnim skladiščem, ker lahko iz aplikacije podatke shraniva na oblačno skladišče Google Drive. Prav tako pa lahko iz njega uvoziva podatke v najino aplikacijo. Zelo pomembno je tudi preverjanje, kateri podatki že obstajajo na Google Drive oz. na aplikaciji, da ne pride do podvajanja podatkov, kar seveda ni v redu.

Najin model aplikacije omogoča uporabniku popoln nadzor nad svojimi podatki, saj vse datoteke shranjujeva na njegovo oblačno skladišče, kar pomeni tudi, da najina podatkovna baza ni tako obremenjena, kar pripomore k boljšemu delovanju aplikacije in zmanjša potencialne stroške.

Med izdelovanjem raziskovalne naloge sva se soočila z mnogimi izzivi in problemi, a sva se pri tem veliko naučila, predvsem sva dopolnila svoje znanje o integraciji knjižnic, kot sta passport in mongoose. Največje težave nama je povzročala povezava spletne aplikacije z Google Drive API, saj sva morala prebrati celo njihovo dokumentacijo, da sva ga znala uporabljati. Sva pa zelo vesela, da sva se soočila s takšnim problemom, saj sva se naučila uporabljati že napisan programski vmesnik za reševanje svojih problemov.

Ena od pomembnejših stvari, ki sva se jih naučila med izdelovanjem spletne aplikacije, je skupinsko delo, ki je še posebej pomembno v današnjem svetu individualizma. Naučiti sva se morala biti zanesljiva, da sva naredila delo, ki sva si ga porazdelila, da nama ni bilo treba čakati drug na drugega.

7. POVZETEK

Načrtovanje je eden od najpomembnejših delov vsakega dogodka, izdelka, storitve ..., saj lahko z dobrim oz. slabim načrtovanjem naredimo ogromno razliko v končnem produktu. Na spletu obstaja veliko aplikacij, ki nam pomagajo pri načrtovanju, zato sva se osredotočila na opravila, za katera je potrebno pripraviti veliko dokumentov (poročila za vaje, evidenca prisotnosti ...) ter v ta opravila vključiti veliko ljudi.

Velik poudarek sva dala tudi na varnost spletne aplikacije, kar sva rešila s knjižnico passport. Poleg varnosti aplikacije sva se odločila poskrbeti tudi za varnost podatkov, kar sva dosegla s povezavo aplikacije na oblačno skladišče Google Drive. Pri tem sva naredila tudi velik korak na področju uporabnikove zasebnosti, saj se vsi podatki shranijo na njegovo oblačno skladišče, kar se nama zdi dandanes še posebej pomembno.

Ker sva hotela najino aplikacijo narediti uporabniku prijazno, sva Google Drive še dodatno izkoristila, in namesto da bi služil le kot nekakšna varnostna kopija, sva aplikacijo naučila brati mape in datoteke iz njega. Tako sva dobila aplikacijo z dvosmerno sinhronizacijo s spletnim skladiščem Google Drive.

K vsem funkcionalnostim spletne aplikacije sva se odločila dodati tudi uporabniku prijazen uporabniški vmesnik, za kar se da najboljšo uporabniško izkušnjo. Tega sva se lotila tako, da sva se zgledovala po dobrih praksah že narejenih spletnih strani, kot sta Asana in Trello.

8. ZAHVALA

Rada bi se zahvalila za vso pomoč pri razvijanju in nastajanju raziskovalne naloge: mentorjema Islamu Mušiču, prof., in Samu Železniku, inž., za vso spodbudo in pomoč, Lidiji Šuster, prof., za lektoriranje, Vlasti Leban, prof., za lektoriranje angleškega avtorskega izvlečka, recenzentom raziskovalne naloge, staršem in vsem ostalim, ki so prispevali k razvoju raziskovalne naloge.

9. VIRI

1. https://www.voidcanvas.com/describing-node-js/ (22. 2. 2021)
2. https://www.techomoro.com/what-are-the-benefits-of-using-express-js-for-backend-development/ (25.3.2021)
3. https://dk.um.si/IzpisGradiva.php?id=55437 – Analiza zmogljivosti podatkovne baze MongoDB (22. 3. 2021)
4. https://dk.um.si/IzpisGradiva.php?id=54848&lang=eng – Uporaba Node.js in MongoDB pri izdelavi spletnega socialnega omrežja (16. 3. 2021)
5. https://www.theseus.fi/bitstream/handle/10024/263205/Pham_An.pdf?sequence=2 (18. 3. 2021)
6. https://jwt.io/introduction (29. 3. 2021)
7. https://martinfowler.com/articles/newMethodology.html (18. 3. 2021)
8. https://www.freecodecamp.org/news/introduction-to-mongoose-for-mongodb-d2a7aa593c57/ (29. 3. 2021)
9. https://github.com/ (29. 3. 2021)
10. https://vuex.vuejs.org (21. 3. 2021)
11. https://repozitorij.uni-lj.si/IzpisGradiva.php?id=110078&lang=slv (30. 3. 2021)
12. https://mentormate.com/blog/what-is-heroku-used-for-cloud-development/ (31. 3. 2021)
13. https://swagger.io/docs/specification/authentication/bearer-authentication/ (31. 3. 2021)
14. https://developers.google.com/drive/api/v3/about-sdk (31. 3. 2021)
15. https://www.cmswire.com/digital-workplace/trello-vs-asana-battle-of-the-freemium-project-management-tools/ (23. 3. 2021)
16. https://tallyfy.com/what-is-asana-how-does-it-work/ (23. 3. 2021)
17. https://project-management.com/pros-and-cons-of-using-clickup/ (23. 3. 2021)

9.1 Viri slik

Vir [1] https://luna1.co/7c29c0.png (20. 3. 2021)

Vir [2] https://www.cmswire.com/-/media/c3e2a7cbd2c7432591188a92e2e393ba.ashx?h=542&w=1686 (23. 3. 2021)

Vir [3] https://clickup.com/landing/images/views/team_tab.png (30. 3. 2021)

Vir [4] https://vuejs.org/images/components.png (28. 3. 2021)

Vir [5] https://db-engines.com/en/ranking (16. 3. 2021)

Vir [6] https://jwt.io/introduction (29. 3. 2021)

Vir [7] https://cdn-media-1.freecodecamp.org/images/0*b5piDNW1dqlkJWKe. (29. 3. 2021)

Vir [8] https://1000logos.net/wp-content/uploads/2018/11/GitHub-logo.png (29. 3. 2021)

Vir [9] https://vuex.vuejs.org (21. 3. 2021)

Vir [10] https://miro.medium.com/max/3600/1*fIjRtO5P8zc3pjs0E5hYkw.png (22. 3. 2021)

10. PRILOGE

10.1 PRILOGA A

Spletna aplikacija z dvosmerno sinhronizacijo med opravili in oblačnim skladiščem imenovana Checky, ki se nahaja na spletni povezavi https://checky-app.herokuapp.com.