1.)\textit{ Client-server} – zahteva ločitev odjemalca od strežnika kar onemogoča odjemalcu direktno povezljivost s podatkovno bazo in s tem poenostavi razširljivost uporabniškega dela. Strežnik ne zanima uporabniški vmesnik ali podatki, tako da je bolj enostaven in prilagodljiv za uporabo. Tako se lahko uporabniški kot strežniški del razvija ali zamenjuje neodvisno. V naši aplikaciji imamo kar dva odjemalca (naročnik, kar pa za strežnik ne predstavlja nobenih omejitev razen na strani podatkovne baze, ki omogoča določeno število hkratnih poizvedb. V naši aplikaciji te omejitve nismo nikoli presegli.

2.) Stateless - vsaka zahteva od odjemalca mora vsebovati vse potrebne podatke, da pridobi odgovor. Strežnik ne shranjuje lokalno nobenih podatkov od odjemalca, vendar jih glede na zahtevek samo prebere ali pa zapiše v podatkovno bazo. To je v programski kodi našega strežnika dobro razvidno, saj ne uporabljamo globalnih spremenljivk. Vse kar odjemalec zahteva se prebere iz podatkovne baze in vrne odjemalcu.

3.) Cachable - vsak odgovor od strežnika more biti označen kot cacheable or non-cacheable. S tem odjemalec ne zahteva od strežnika eno in isto zadevo, razen če v primeru, da je prišlo do sprememb v podatkih.

4.) Uniform interface – zahteva enoten vmesnik (API) med strežnikom in odjemalcem. Vsak vir mora vsebovati povezavo (HATEOAS), ki kaže na svoj relativen URI. Odjemalec te vire pridobi od strežnika v obliki zahtev, ki so lahko GET, POST, PUT ali DELETE. Za predstavitev virov je potrebno uporabiti točno določena podatkovna formata XML in JSON.

Za to zahtevo nam je knjižnica Flask zelo poenostavila izdelavo strežnika. Vsak relativna povezav URI na strežniku predstavlja svoje vir podatkov iz podatkovne baze. Te podatki so odjemalcu na voljo v JSON podatkovnem formatu. Strežnik najprej podatke prebere iz podatkovne baze s pomočjo MySQL knjižnice in jih . vir iz podatkovne baze uporabil svojo po je zajela to zahtevo in sicer za vsak vir smo imeli svoj URI, ki je predstavik podatke v HTTP obliki. Tako je odjemalec sprožil zahtevo GET in prodbil zahtevane podatke. Spodaj je primer za pridobitev vseh pijač/hrane.

5.) Layered system - slojevit sistem, sestavljen iz hierarhičnih slovej z namenom omejevanja komponent.

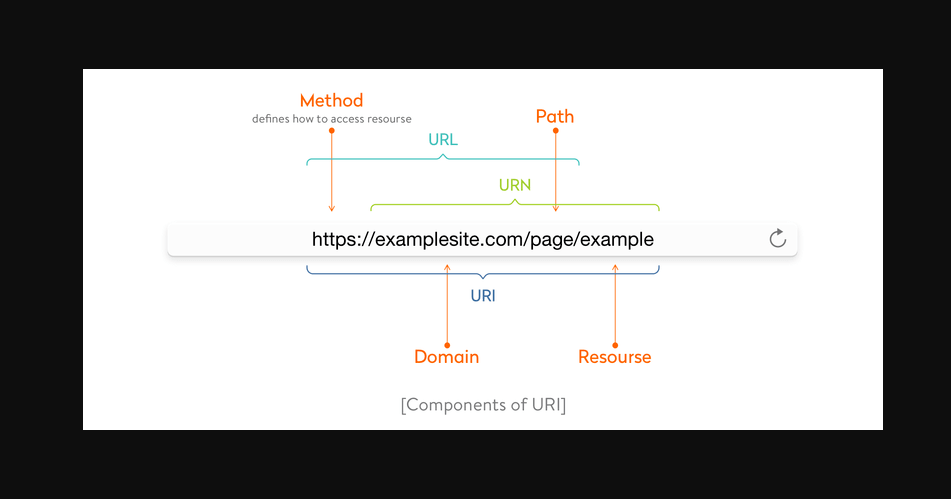
6.) Code on demand (optional) - opcijsko načelo. Strežnik na zahtevo odjemalca pošlje oziroma izvede programsko kodo na strani odjemalca.

To smo predsvsem uporabljali

<https://stackoverflow.com/questions/25172600/rest-what-exactly-is-meant-by-uniform-interface>

<https://restfulapi.net/rest-architectural-constraints/#uniform-interface>

<https://yalantis.com/uploads/ckeditor/pictures/4238/uri-components.png>



Arhitertura omogoča, da se vsak del razvija neodvisno. Zato so tukaj še štiri podnačela, ki vodijo do enotnega vmesnika. kar pomeni da morajo uporabljeni morajo biti standardi oziroma globalni koncepti npr

Strežnik v naši aplikaciji predstavlja vmesnik med podatkovno bazo in odjemalcem. Ko smo izdelali podatkovno bazo smo začeli z izdelavo strežniškega dela. Najbolj pomembno na je bilo, da je sistem zanesljiv, saj brez njega odjemalec ne more delovati. Zato smo izbrali REST arhitekturo, ki zahteva določena načela, katera so bolj podrobno opisana spodaj. Sama arhitektura omogoča, da odjemalec s pomočjo zahtev pridobiva podatke od strežnika, kateri jih s pomočjo URI povezav oglašuje na relativnih povezavah. Strežnik smo napisali v programskem jeziku Python ter s pomočjo knjižnic Flask, MySQL, SocketIO in CORS, kateri nameni so bolj podrobno opisani spodaj.

\section{Strežnik}

Strežniški del aplikacije smo izdelali po zahtevah RESTful arhitekture, ki vsebuje 6 načel.

1.)\textit{ Client-server} - zahteva ločitev odjemalca od strežnika kar onemogča odjemalcu direktno povezljivost s podatkovno bazo in s tem poenostavi razširljivost uporabniškega dela. Strežnik ne zanima uporabniški vmesnik ali podatki, tako da je bolj enostaven in prilagodljiv za uporabo. Tako se lahko uporabniški kot strežiški del razvijaj ali zamenjujeje neodvisno.

2.) Stateless - vsaka zahteva od odjemalca mora vsebovati vse potrebne podatke, da pridobi odogovor. Strežnik ne shranjuje lokalno nobenih podatkov od odjemalca, vendar jih glede na zahtevek samo prebere ali pa zapiše v podatkovno bazo.

3.) Cachable - vsak odgovor od strežnika more biti označen kot cacheable or non-cacheable. S tem odjemalec ne zahteva od strežnika eno in isto zadevo, razen če v primeru, da je prišlo do sprememb v podatkih.

4.) Uniform interface - zahteva enoten vmensika API med strežnikom in odjemalcem za določene vire, ki imajo lahko samo en logični URI. Arhitertura omogoča, da se vsak del razvija neodvisno. Zato so tukaj še štiri podnačela, ki vodijo do enotnega vmesnika. kar pomeni da morajo uporabljeni morajo biti standardi oziroma globalni koncepti npr. HTTP standart za opis komunikacije - - ali gre za GET, POST,... Mi smo to zagotovili s serviranjem podatkov odjemalcu v JSON formata, ki ga je mogoče uporabiti v vseh programskih jezikih.

5.) Layered system - slojevit sistem, sestavljen iz hierarhičnih slovej z namenom omejevanja komponent.

6.) Code on demand (optional) - opcijsko načelo. Strežnik na zahtevo odjemalca pošlje oziroma izvede programsko kodo na strani odjemalca.

Vsa zgoraj našteta navodila smo morali upoštevati pri izdelavi strežnika. Tako smo dobili vmesnik, ki na zahtevo odjemalca servira podatke v JSON formatu. Primer na sliki~\ref{ServerEX}, ko odjemalec zahteva podatke vseh pijač iz podatkovne baze (HTTP metoda GET). Strežnik omogoča tudi urejanje in brisanje podatkov v podatkovni bazi, vendar zato more biti ustrezna HTTP metoda POST. Spremljanje zahtevkov, ki prihajajo na strežnik, je mogoče preko CLI vmesnika, slika ~\ref{ServerEX2}, ki v primeru nepopolnosti servirajo ustrezno napako.

\begin{figure}

\begin{center}

\includegraphics[width=11cm]{Server\_example.jpg}

\end{center}

\caption{Primer serviranja podatkov na strežniku}

\label{ServerEX}

\end{figure}

\begin{figure}

\begin{center}

\includegraphics[width=10cm]{Server\_example\_2.jpg}

\end{center}

\caption{Primer spremljanja zahtevkov, ki prihajajo na strežnik}

\label{ServerEX2}

\end{figure}

Strežnik smo napisali v programske jeziku Python in knjižnici Flask. Zaradi težav s sinhronizacijo podatkov v realnem času na strani odjemalca smo znotraj Flask uporabil še WebSocket. To je napredna tehnologija, ki omogoča odpiranje dvosmerne interaktivne komunikacijske seje med odjemalcem in strežnikom. Primer uporabe je npr. pogovor preko družbenih omrežjih. V naši aplikaciji so s tem lahko vsi odjemalci hkrati obveščani o spremembah na strani podatkovne baze.

Flask je eno izmed najbolj popularnih spletno aplikacijskih vmesnikov (angl. Freamwork) \cite{Flask}. Zasnovan je tako, da omogoča hiter in enostaven začetek z možnostjo razširitve na zapletene aplikacije. V primerjavi z Django spletnim vmesnikov je za enak primer veliko bolj ekspliciten. Flask je prvotno zasnoval in razvil Armin Ronacher kot prvoaprilsko šalo leta 2010. Kljub taki predstavitvi je Flask postal izjemno priljubljen kot alternativa projektom narejenih v Django.

Zahtevek:

\begin{verbatim}

C:\Users\lukah>curl -I http://192.168.1.13:5000/drinks

HTTP/1.0 200 OK

Content-Type: application/json

Content-Length: 5066

Access-Control-Allow-Origin: \*

Server: Werkzeug/0.16.0 Python/3.7.3

Date: Tue, 02 Mar 2021 20:23:57 GMT

\end{verbatim}

\begin{verbatim}

C:\Users\lukah>curl http://192.168.1.13:5000/drinks

{

"product\_id": 1,

"name": "Somersby Apple",

"price": 3,

"size": "0.33L",

"calorie": "207",

"picture": "somersby-apple.png",

"description": null,

"type\_id": 1

},

{

"product\_id": 2,

"name": "Heineken",

"price": 2,

"size": "0.25L",

"calorie": "110",

"picture": "heineken.png",

"description": null,

"type\_id": 2

},

{

"product\_id": 3,

"name": "Heineken",

"price": 3.5,

"size": "0.5L",

"calorie": "220",

"picture": "heineken.png",

"description": null,

"type\_id": 2

},

\end{verbatim}

\section{Odjemalec}

Odjemlaca bi načeloma lahko implementirali v spletnih tehnologijah (HTML, CSS, JS) ali pa v namenski mobilni aplikaciji. Mi smo se odločili za programskem jezik JavaScript in knjižnico Vue. Naredili smo odziven in reaktiven vmesnik, ki deluje v realnem času. Vue je eden izmed mnogih kot npr. Angular, Ember, React,… poznan pa je predvsem zaradi enostavnosti za upravljanje in izvajanje testov. Vsem je skupna točka reaktivnost, vendar v drugačnem pomenu besede. Gre za to, da se aplikacija postopno prilagaja glede na vrednosti podatkov. To prednost sem izkoristil in dodal še knjižnico Vuex, ki služi kot centralizirana baza podatkov za vse komponente v aplikaciji. Podatke i

Da bi to dosegli ne bi potreboali JavaScript zaradi RESTful arhitekutre, ki omogoča katerokoli platformo na strani odjemalca.

\subsection {Kaj je reaktivnost?}

Reaktivnost \cite{reaktivnost}, je programska paradigma, ki nam omogoča, da se na deklarativni način prilagodimo spremembam. Dober primer reaktivnosti je npr. funkcija SUM, ki jo uporabljamo v Excelu. Slika~\ref{Excel-SUM} prikazuje primer v Excelu.

\begin{figure}[h]

\begin{center}

\includegraphics[width=7cm]{Excel-SUM.png}

\end{center}

\caption{Primer fukcije SUM v programu Excel}

\label{Excel-SUM}

\end{figure}

Če vstavimo številko 2 v prvo celico in številko 3 v drugo celico ter izberemo funkcijo SUM teh dveh celic. Kot rezultat dobimo vsoto obeh številk skupaj, kar ni nič posebnega. Vendar če bomo spremenili vrednost prve celice, bo funkcija SUM avtomatsko posodobila skupno vrednost. Tako deluje tudi reaktivnost v aplikacijah za razliko, da je podatek lahko vezan na več funkciji oziroma delov programske kode, ki se ob spremembi vrednosti posodobijo.

\subsection {Delovanje reaktivnost v Vue}

Vue se torej v primerjavi z navadnim JavaScriptom sprehodi skozi podatke in njihove lastnosti (angl. Properties) pretvori v fukciji Getter in Setter, ki sta nevidni uporabniku \cite{delovanjeReaktivnost}. Poglejte si sliko~\ref{VueReacitivity} za lažjo predstavo.

Torej funkcija Getter pokliče instanco Watcher z namenom odvisnosti do drugih komponent. To pomeni, če je podatek označen kot odvisen (angl. Dependency) bodo nekateri deli programske kode oziroma funkcije poklicane vsakič, ko se spremeni vrednost podatka. Funkcija Setter pa obvesti instanco Watcher, vsakič ko se podatku spremeni lastnost. Ta poskrbi, da se pokliče funkcija upodabljanja (angl. Render) tiste komponente, ki potem prikaže spremembe v samem pogledu aplikacije.

\begin{figure}[h]

\begin{center}

\includegraphics[width=13cm]{Vue-reactivity.jpg}

\end{center}

\caption{Kateri dialekt uporabljati?}

\label{VueReacitivity}

\end{figure}