Razvoj aplikacija pomoću RESTFul web servisa

Keser, Mirna

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Mirna Keser

RAZVOJ APLIKACIJA POMOĆU RESTFUL WEB SERVISA

Diplomski rad

Voditelj rada:
prof. dr. sc. Robert Manger

Zagreb, veljača 2023.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Industrija općenito, pa tako i IT industrija, neumorno raste. Ultimativni je cilj što brže i što jeftinije napraviti što više i prodati što skuplje. Stoga ne čudi kako se konstantno traže načini da se cijeli proces pospiješi. Na razini cijele firme, na razini odjela, na razini pojedinog radnika. Kako da jedna firma što prije privuće kupca, kako da tim razvojnih inženjera (developera) u rekordnom roku isporuči kvalitetan proizvod s kojim će kupac biti zadovoljan i vraćati će se po još? Navedena su pitanja na koja se konstantno traži novi i bolji, efikasniji odgovor.

Tema ovog rada je razvoj web aplikacija koje se sastoje od API-ja rađenih po REST arhitekturi. Cilj rada je pomoći čitatelju da odluči kada mu treba upravo takva arhitektura, te kako da ju sprovede u djelo. U nastavku se opisuje "bottom-up" proces izrade. Od toga na koji način je organiziran tim kojem odgovara i pomaže koristiti takvu arhitekturu, pa sve do ulaska u kod i samih detalja implementacije. Pojašnjavaju se metodologije rada IT firmi, različite aplikacijske arhitekture, građa web servisa, te u konačnici implementacijski detalji izrade REST-a. Primarni naglasak je na objašnjenju same REST strukture i implementiranju iste, ali su isto tako objašnjeni pojmovi vezani uz način razvoja i odabira arhitekture. Pojašnjeno je ne samo što se koristi u kontekstu REST-a nego "what else is out there" kako bi se stvorio kontekst i kako bi se bolje pojasnilo u kojoj situaciji je REST pogodan izbor i zašto. Početak izrade svakog projekta svodi se upravo na odgovaranje na pitanje s početka uvoda - na koji način raditi i što raditi da bi projekt bio brzo gotov i visoko kvalitetan. Zato je vrlo bitno razumjeti sve što se u nastavku navodi i znati odabrati ono što je optimalna opcija za svaki zasebni razvoj.

Vezano za samu implementaciju, neće se samo opisati što sve treba uključiti u dizajn i pisanje koda kako bi se dobio REST, već se sugeriraju neke općenito dobre smjernice za razvoj softvera. Radi se o standardima i praksama koje utječu pozitivno na svojstva aplikacija poput preglednosti, lakšeg korištenja, skalabilnosti, performansi. U konačnici, implementiran je konkretan API čiji dijelovi koda su prikazani i objašnjeni u ovom radu kako bi se dali konkretni primjeri moguće / nužne implementacije određenih segmenata web servisa.

Potrebno je još naglasiti da neki od pojmova koji se tumače u ovom radu zapravo nemaju formalnu definiciju koja je uvažena svugdje u svijetu. Oko mnogo stvari još uvijek

postoje otvorene rasprave i razvojni inženjeri imaju podijeljena mišljenja, budući da se radi o novim principima koji nisu dugo u opticaju. Cilj rada je bio naći činjenice - informacije oko kojih se različiti izvori slažu. Ipak, za neke pojmove ne radi se o definicijama "iz udžbenika" - takvih za puno toga nažalost još uvijek nema. Svijet razvoja softvera (developmenta) se razvija i mijenja sve brže. Teško je pratiti taj tempo stručnom literaturom i mnogo pojmova je još uvijek "u zraku" i svatko se koristi vlastitom interpretacijom i prilagođava svoj posao onome što mu u tom trenutku odgovara. Takav pristup i ima puno više smisla od slijepog pridržavanja pravila samo u svrhu zadovoljavanja forme. Na ovaj način se, ako inženjer zaista zna što radi, dobiva veća kvaliteta u svakom segmentu proizvoda.

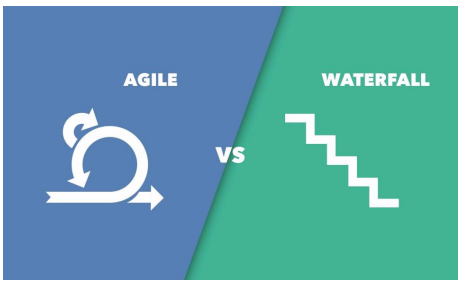
Prvo poglavlje govori o načinima vođenja projekata. Drugo poglavlje je o aplikacijskim arhitekturama, njihovim sličnostima i razlikama. Dalje su ukratko objašnjeni web servisi. Sljedeće poglavlje je uvod u HTTP koji će se koristiti tijekom implementacije praktičnog dijela. Peto poglavlje objašnjava što je REST, koji su mu principi i u kakvom je odnosu sa SOAP-om. Nakon toga imamo poglavlje o tehnologijama i alatima korištenim za implementaciju. Zadnje poglavlje bavi se time kako dizajnirati i implementirati ne samo REST API već kako općenito pisati kvalitetan kod.

Poglavlje 1

Metodologija razvoja

Metodologije razvoja koje se spominju u ovom odlomku odnose se na općenito vođenje projekata, pa tako i na vođenje unutar razvoja softvera. Osnovna podjela metodologija je na klasični i agilni pristup. Navest ćemo primjer modela koji se koristi za klasični razvoj - model vodopada, te ga ukratko objasniti, a zatim slijedi detaljniji pregled agilne metodologije budući da je REST pogodan odabir upravo za takav način razvoja.

Model vodopada [20] svodi se na jasno definirane i odvojene faze razvoja koje se sekvencijalno izvršavaju. Prelazi se u iduću fazu razvoja tek kada je trenutna kompletno završena, nema vraćanja nazad. Cijeli proces se detaljno planira i raspodjeljuje prije samog početka. Ovakav model prigodan je za upotrebu npr. u građevinarstvu. Jasno je da jednom kada smo podigli temelje zgrade, ne možemo se više vraćati na tlocrt i mijenjati ga, jer bi to značilo rušenje i ponovno podizanje temelja što je užasno skupo i ne radi se. U svojim počecima se razvoj softvera također provodio po ovom modelu. Međutim, konkretno za razvoj softvera postoje i nedostaci u ovom modelu. Iako je moguće pomno isplanirati proces, greške se neizbježno događaju tijekom samog procesa i u različitim fazama, no uz ovakav način rada teže ih je otkriti i popraviti.



Slika 1.1: Pristupi: agilni vs vodopad [18]

1.1 Agilni pristup

Agilni način rada po [16] podrazumijeva sljedeće: "Razvoj softvera promatra se kao kontinuirani niz iteracija čiji broj nije moguće predvidjeti. Svaka iteracija usklađuje sustav s trenutnim zahtjevima. Zahtjevi se stalno mijenjaju, ne postoji cjelovita ili konačna specifikacija." Dakle, promjene i dodavanje funkcionalnosti su redovne i nepredvidive, što je olakšano za ostvariti kada imamo podjelu na servise sa zasebnim ulogama - o tome više detalja u slijedećem poglavlju. Slijedi 12 osnovnih principa agilnog rada [15] za bolji uvid to kako se provodi:

1. Zadovoljiti kupce ranom i kontinuiranom isporukom vrijednog softvera
2. Prihvatiti promjene za kompetitivnu prednost kupca, čak i kasno u razvoju
3. Redovita isporuka proizvoda koji radi, u razmacima od nekoliko tjedana do nekoliko mjeseci, preferabilno za kraći period
4. Suradnja na dnevnoj bazi između razvojnih inženjera i poslovnih ljudi na radu na projektu
5. Graditi projekte oko motiviranih individualaca. Kreirati okolinu i podržavati potrebe razvojnih inženjera i vjerovati im da će napraviti posao
6. Prioritizirati razgovor licem u lice kao najučinkovitiju i najdjelotvorniju metodu prenošenja informacija timu i unutar samog tima razvojnih inženjera
7. Mjeriti napredak količinom dovršenog softvera
8. Održavati konstantni i održivi tempo razvoja softvera na neograničeno vrijeme
9. Poboljšavati agilnost kroz kontinuiranu pažnju na tehničku izvrsnost i dobar dizajn
10. Jednostavnost - umjetnost maksimiziranja količine neobavljenog posla - je esencijalna
11. Prepoznati da najbolje arhitekture, zahtjevi i dizajn dolaze od samooragniziranih timova
12. Redovito promišljati i prilagođavati ponašanje za kontinuirani napredak.

Ovaj način rada začet je 2001. godine od skupine razvojnih inženjera koji su smatrali dotadašnji, klasični pristup, previše kompliciranim i usporenim. Tržište i zahtjevi istog se danas sve brže mijenjanju, a tvrtke nastoje redovito izbacivati novitete u svrhu zadržavanja konkurentnosti i povećanja profita. Agilni rad omogućava sve to jer ovim pristupom

Slika 1.2: [13] 12. princip

dobivamo bržu isporuku koja se više poklapa s potrebama klijenata čak i kada se one redovito mijenjanju. Osim toga, razbijanjem posla u više manjih cjelina lakše je planirati kako isporuku, tako i troškove. Lakše je i testirati napravljeni kod, te svaki novi razvoj donosi novo pokriće testovima i ne može proći dalje ako ne prođe sve prijašnje. Time se smanjuje šansa isporuke nekvalitetnog proizvoda.

Poglavlje 2

Arhitektura

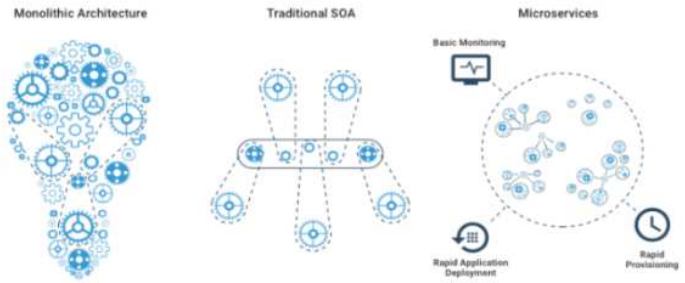
Najprije bi trebalo razdvojiti dva pojma: stil arhitekture i obrazac arhitekture [7]. Obrazac arhitekture je način implementacije stila arhitekture, a stil arhitekture predstavlja način organizacije tog koda. Što to točno znači? Pogledajmo na primjeru. Stil arhitekture može biti recimo klijent - poslužitelj. Mora postojati podjela koda na klijentski dio i na poslužiteljski dio i oni moraju biti nezavisni. Odabir obrasca arhitekture koji će se primijeniti kako bi aplikacija imala stil arhitekture klijent - poslužitelj zapravo je odgovor na pitanja poput koje će se sve klase implementirati u kodu? Obrazac može biti npr. MVC model [6] u kojem je kod podijeljen u tri dijela: "model, view, controller". Svaki od ta tri djela obavlja određeni tip zadaća unutar aplikacije čime su ujedno klijent i poslužitelj odvojeni u zasebne klase i strukture. MVC će kasnije biti detaljnije objašnjen budući da će se upravo taj obrazac koristiti u izradi praktičnog dijela u sklopu ovog rada.

Dalje slijedi nekoliko različitih stilova arhitekture koji se danas najčešće koriste u razvoju softvera. Objašnjeno je kada i zašto je koji optimalan odabir za primjenu. Potrebno je još napomenuti da ovdje nije bitna tehnologija, operacijski sustav, programski jezik. Svaki stil je izvediv (na sebi svojstven način) u proizvoljnim tehnologijama.

Ovdje se opisuju stilovi arhitekture za izradu aplikacija. Kasnije će biti govora o API-ju kao komponenti aplikacije i arhitekturi API-ja u koju spadaju npr. SOAP i primarna tema ovog rada - REST. Zašto onda ulaziti u arhitekturu aplikacije? Koja je važnost tih pojmova za razumjevanje REST-a kao arhitekture za API? Aplikacija može biti podijeljena u nekoliko nezavisnih cjelina - APIja. Dakle, API koji ima REST arhitekturu može biti dio aplikacije koja ima mikroservisnu arhitekturu.

2.1 Monolitna arhitektura

Monolitna arhitektura je upravo ono što naziv govori - monolitni kamen. Sav razvoj je jedna velika, povezana cjelina. To znači da su baze podataka, poslovna logika i korisničko



Slika 2.1: Stilovi arhitekture [10]

sučelje zajedno u jednom sloju. Za male projekte na kojima radi svega nekoliko ljudi, ovo može biti pametan izbor. Performanse su općenito bolje kod ovakve vrste arhitekture budući da ne treba dohvaćati i slati podatke kroz različite servise i API-je već se sve nalazi na jednom mjestu, u jednom sloju. Komunikacija se u ovakvoj aplikaciji odvija samo kroz funkcijske pozive i pozivanje metoda. Ono što može predstavljati problem je popravljanje i proširivanje funkcionalnosti. Mala promjena na sustavu može uzrokovati prestanak rada cijele aplikacije, a ne samo dijela koda koji prepravljamo. Kod monolitne arhitekture je problem i to što se naseljavanje (deploy) svaki puta provodi sa cijelim kodom.

2.2 SOA vs mikroservisi

Budući da se radi o donekle sličnim stilovima arhitekture, ovdje će biti paralelno objašnjeni kako bi došlo do što manje konfuzije oko toga po čemu su specifični i kako se razlikuju. Dan je kratki pregled sličnosti i razlika između ova dva stila arhitekture kako bi se bolje shvatilo kako funkcioniraju i kada se koriste. Iako možda na prvu nije sasvim očito, postoje velike razlike između ove dvije arhitekture. U svakom slučaju, oba stila su u potpunoj suprotnosti u odnosu na monolitnu arhitekturu, ali svejedno imaju puno razlika i među sobom.

SOA (Service-oriented architecture) i mikroservisi su, za razliku od monolita, decentralizirane. To se postiže korištenjem servisa, što je temelj ovih arhitektura. Prije daljnjih detalja o samim arhitekturama, treba pojasniti dijelove istih. Servis je definiran kao komponenta softvera zadužena za točno jednu zadaću unutar aplikacije koja komunicira s ostalim servisima na različite načine. Web servisi primarno koriste HTTP protokol za razmjenu poruka. Može se reći da je općenita softverska komponenta zapravo komad softvera koji

je moguće ukloniti i unaprijediti neovisno o ostatku sustava čiji je dio. Dakle, softver se može graditi na način da se sastoji od niza komponenti - nezavisnih dijelova. Svaka ta komponenta brine o određenoj funkcionalnosti sustava i ne utječe na ostale funkcionalosti koje su dio te cjeline.

Dalje slijedi opis karakteristika koje se poklapaju za SOA i mikroservise. Podjela na servise olakšava razvoj iz nekoliko razloga. Kod korištenja SOA principa i mikroservisa moguće je da nekoliko ljudi u isto vrijeme radi na projektu i da to rade nesmetano jer se svatko bavi konkretnom funkcionalnošću koja se razvija u zasebnom servisu. Ista stvar se događa i kasnije, kod održavanja već gotovog proizvoda. Osim toga, postiže se bolja skalabilnost jer je puno teže "potrgati" postojeći kod. Ako se rade promjene samo na jednom servisu, onda je moguće pokvariti rad isključivo tog dijela funkcionalnosti i sigurni smo da to neće imati nikakav utjecaj na ostatak koda. Osim toga, ovakav kod je lakše kvalitetno pokriti testovima. SOA je vrlo pogodan za agilni način rada budući da je lako mijenjati i nadograđivati ovakve aplikacije te na njima može nesmetano raditi više ljudi. Moguće je i ponovno koristiti postojeće module te ih integrirati u više različitih aplikacija. Ako tvrtka ima više vlastitih proizvoda među kojima postoje sličnosti, tada im ta mogućnost štedi puno vremena i novaca.

Cijena upravo nabrojanih karakteristika servisne arhitekture su lošije performanse. Jedan od izazova implementacije SOA i mikroservisne arhitekture je upravo na koji način što adekvatnije povezati sve servise koji moraju komunicirati kako bi ispunili u potpunosti zahtjev klijenta koji traži određene sadržaje na svom sučelju. Monolitna arhitektura taj dio provodi brzo i efikasno. U ovom slučaju treba osigurati da se sve poruke šalju i primaju na vrijeme te da tijek komunikacije bude smislen.

Razlika koja je najočitija između SOA i mikroservisa jest veličina i finoća komponenti. Kao što i sam naziv govori, mikroservisi su u odnosu na SOA servise građeni od većeg broja manjih servisa s konkretnijim i jednostavnijim ulogama. Također se razlikuju po veličini sustava koje implementiraju. U odnosu na monolite, i SOA i mikroservisi su puno pogodniji za veće aplikacije. Nakon određene razine kompleksnosti postaje puno korisnije koristiti servise. To je slučaj pogotovo kada je krucijalno da sustav ne pada i ne radi greške jer time nastaje financijska šteta, budući da je u ovoj arhitekturi lakše naći bugove i teže srušiti cijeli sustav. S druge strane, ponekad može biti previše komplicirano razbijati velik i kompleksan sustav u sitne i profinjene servise, pa se tada bolje opredijeliti za SOA arhitekturu [2].

Svaki pojedini mikroservis tj. svaka komponenta može biti napisana u drugom programskom jeziku i može koristiti različite tehnologije za pohranjivanje podataka u bazu. Za razliku od toga, svi SOA servisi koriste istu bazu. Ovo nije nužno tako implementirano, i mikroservisi mogu koristiti istu bazu podataka, ali imaju mogućnost nezavisne implementacije. Ono što može predstavljati problem kod korištenja različitih jezika je to što nije moguće proizvoljno prebacivati inženjera s jednog servisa na drugi već postoje ograničenja

ovisno o tehničkom znanju. Ipak, to ograničenje na stranu, budući da je broj komponenti veći, više ljudi može istovremeno raditi na razvoju softvera i aplikacija se može brže isporučiti u odnosu na ostale spomenute stilove.

Iduća razlika odnosi se na način na koji se implementira komunikacija servisa prema van. SOA servisi imaju zajednički međusklop za razmjenu poruka (messaging middleware) koji brine o komunikaciji, dok svaki mikroservis zasebno obavlja tu zadaću.

Dolazimo do veze REST-a i mikroservisa. Naime, ove arhitekture se među ostalim razlikuju po protoklima koje koriste za razmjenu poruka. Kao što je već spomenuto, mikroservisi se većinom oslanjaju na REST. SOA u pravilu koristi SOAP. Ovo nije nužno tako, ali većinom se radi o ovoj podjeli. Osim REST-a, mikroservisi ponekad koriste i MSMQ (Microsoft Message Queuing) što je jedan od jednostavnih protokola za razmjenu poruka.

U narednim poglavljima je preciznije opisano kako se mikroservisi uklapaju s REST-om i kako to dvoje zajedno gradi kvalitetne aplikacije koje imaju mnoga korisna svojstva. Također, pobliže su objašnjene značajke SOAP-a i navode se osnovne razlike među protokolima.

Poglavlje 3

Web servisi

Prije dodatnih detalja o REST-u, slijedi objašnjenje web servisa i njihove veze s dosad definiranim pojmovima. Započet ćemo poglavlje s jednim drugim, usko povezanim pojmom, a to je API. Web servis se može definirati kao jedna vrsta API-ja, pa ima smisla započeti s opširnijim pojmom kako bi se u konačnici dala jasnija slika o samim web servisima.

API (Application Programming Interface) [12] odnosno sučelje za programiranje aplikacija odnosi se na skup protokola i funkcija. Taj skup tvori softverske komponente koje omogućavaju komunikaciju između različitih i inače nepovezanih aplikacija. API-ji se povezuju s postojećim aplikacijama i preko svog sučelja pružaju drugim aplikacijma ili korisnicima mogućnost da dohvaćaju podatke i funkcionalnosti drugih aplikacija s kojima inače nebi mogli komunicirati. Naravno, cijela komunikacija je strogo definirana i postoje ograničenja na to čemu API može pristupiti i u kojem obliku.

Upotreba API-ja je učestala pri korištenju mikroservisne arhitekture za izradu aplikacija. Mikroservisne aplikacije sastoje se od nezavisnih mikroservisa povezanih API-jima. Primjeri API-ja viđaju se sve više i više pri korištenju interneta, npr. prijavljivanje u Facebook korištenjem Gmaila - povezivanje dvije aplikacije. Još jedan primjer bi bio API između aplikacije koju turistička agencija koristi za vođenje svog poslovanja i integracijske aplikacije za računovodstvo. API služi kako se ne bi moralo ručno pratiti i pretipkavati sve nove unesene uplate i izdane račune za rezervacije. Automatski se u zadano vrijeme preko API-ja svi novi podaci prenesu u aplikaciju za računovodstvo kako bi informacije redovito bile ažurirane i nema mogućnosti ljudske greške da se nešto zaboravi unijeti. Svi procesi su automatizirani i ubrzani.

Kod kreiranja API-ja najčešće se slijede SOAP i REST arhitekture. Te arhitekture osiguravaju komunikaciju korištenjem određenih standarda. SOAP je popularniji kod web API-ja budući da pruža veću sigurnost, dok je REST praktičniji za korištenje jer nema toliko složen skup protokola koje treba zadovoljiti.

Postoje privatni i javni API-ji. Javni API-ji su većinom web API-ji, dostupni svima

koji im pristupe i može ih se besplatno koristiti. Privatni API-ji se mogu koristiti interno unutar tvrtke, ili npr. kao proizvodi za konkretne kupce i ne može im bilo tko (besplatno) pristupiti.

Web API je softverska komponenta koja šalje podatke internetom. Aplikacija za vremensku prognozu na Androidu komunicira s API-jem koji daje informaciju o vremenu i korisniku se prosljeđuju ti podaci.

Web servis je resurs koji je dostupan na internetu. Vidimo već po definiciji da je web servis zapravo web API, odnosno API koji je dostupan preko interneta. Web servisi kao skup protokola i standarda najčešću upotrebu imaju u razmjeni informacija odnosno u povezivanju nezavisnih aplikacija i sustava. Pritom nije bitno u kojim su jezicima i na kojim platformama rađene aplikacije koje se povezuju. RESTFul web servis je web servis koji koristi REST arhitekturu.

Općenito API-ji, pa tako i web servisi, veoma su korisni zbog mogućnosti ponovne upotrebe. Štede vrijeme i novac jer nije potrebno svaki puta ponovno pisati sav kod već možemo kombinirati i povezivati već postojeće dijelove. Web servisi trebaju mrežu (internet) kako bi mogli komunicirati, a ta komunikacija većinom se odvija pomoću SOAP-a i nešto rjeđe REST-a. Razlog tome je što se SOAP smatra sigurnijim.

Slika 3.1: API vs web servis

Poglavlje 4

HTTP

HTTP protokol je bitno objasniti prije nego se krene u detalje REST-a iz razloga što se ovaj protokol gotovo uvijek koristi za takvu vrstu projekta. Servisi preko API-ja komuniciraju koristeći HTTP. Nije zadano kroz REST principe da je nužno koristiti ovaj protokol, ali je njegovo korištenje općeprihvaćeno. Puno pojmova vezanih za HTTP se spominje i potrebno je razumjeti da bi se bolje shvatilo kako REST radi, ili kako treba raditi. U kasnijim poglavljima vezanim za dizajn i implementaciju, velik dio posla je upravo pravilno implementiranje HTTP-a.

Radi se o protokolu koji definira operacije za razmjenu reprezentacija između klijenta i poslužitelja. To je najkraće rečeno suština ovog protokola. Za lakše razumijevanje, prije dodatnih definicija, slijedi konkretni primjer koji opisuje komunikaciju između klijenta i poslužitelja korištenjem ovog protokola. Primjer za demonstraciju je detaljno objašnjen u [21].

Uzmimo za primjer web stranicu poput Index.hr. Želimo "prolistati" novine. Kako to radimo? Otvaramo Chrome, ili neki drugi web preglednik, od kojih je svaki HTTP klijent. Dalje upućujemo tog klijenta na točno određenu adresu tj. na točno određeni URI. URI je identifikator za resurs koji tražimo. Kada ukucamo adresu web stranice u preglednik, time je poslan HTTP zahtjev prema zadanom URI-ju. Kao odgovor dolazi stranica koju želimo pregledati - Index. Dolazak odgovora od strane poslužitelja se manifestira time da nam se učita stranica koju pokušavamo posjetiti. Ono što je pristiglo je u ovom slučaju kombinacija HTML-a i CSS-a s poslužiteljskim podacima. Učitavanjem pristiglih sadržaja mijenja se stanje klijenta. Zatim prolistamo naslovnicu, nađemo članak koji nas zanima i kliknemo na link koji ga otvara. To je novi HTTP zahtjev. Kliknuti link sadrži novi URI na temelju kojeg poslužitelj šalje novi odgovor, a u Chromu se učitava stranica sa traženim člankom čime se ponovno mijenja klijentsko stanje.

Klijentsko stanje se mijenja ovisno o prikazu traženog resursa i to je zapravo "representational state transfer", odnosno REST. Klijent može biti web preglednik, desktop aplika-

cija ili bilo što drugo što se oslanja na HTTP. REST API korištenjem HTTP protokola može razmjenjivati resurse s bilo kakvim klijentom koji zna slati HTTP zahtjeve i interpretirati odgovore.

Za slanje jednog zahtjeva, klijent mora za početak definirati URI - adresu, mjesto na kojem se nalazi resurs koji ga zanima. Osim toga, zadaje se metoda s kojom se definira željena akcija. HTTP koristi metode poput GET, POST i DELETE koje definiraju različite akcije nad resursima. Time je olakšana izrada aplikacije.

SOAP također koristi HTTP protokol, ali na vrlo primitivnoj razini. U nastavku rada će biti opisani načini za izradu REST API-ja koji detaljno razrađuje protokol, poštuje određena pravila koja protokol nosi sa sobom i samim time pokriva dobar dio principa.

Jedna od bitnih karakteristika HTTP-a je da su interakcije između klijenta i poslužitelja vidljive drugim poslužiteljima, cache-u, i raznim drugim alatima. Ovo omogućava korištenje npr. cache-a čime se može uvelike poboljšati performanse. Za REST je cilj što više povećati vidljivost i to se postiže pravilnim korištenjem HTTP metoda, definiranjem zaglavlja te slanjem odgovarajućih status kodova.

Slijede objašnjenja nekoliko osnovnih pojmova vezanih za HTTP standard.

4.1 URI

"Uniform resource identifier" ili skraćeno URI je u suštini niz znakova koji raspoznaje različite resurse. Obavezan su dio HTTP protokola i na temelju URI-ja klijent daje poslužitelju na znanje kojem točno resursu želi pristupiti. Server ovisno o primljenom URI-ju raspoznaje nad čime provodi zadanu akciju.

Osim što specificira traženi resurs, može biti obogaćen takozvanim query stringom. Query string predstavlja niz upita koji proizvoljno možemo dodati na kraj URI-ja. Najčešće upotrebe su filtriranje, sortiranje, pretraživanje i straničenje. Filtriranje je dohvaćanje resursa ovisno o određenom, zadanom svojstvu. Može biti veoma kompleksno i uvelike smanjiti količinu podataka za dohvat. Sortiranje je promjena poretka dohvaćenih resursa po zadanom parametru, uzlazno ili silazno. Straničenje omogućava podjelu na određeni broj stranica i ograničavanje broja resursa po stranici. Straničenje će dati pregledniji odgovor s manje podataka čime ujedno dobivamo na performansama budući da potencijano nikad nećemo prikazati korisniku API-ja sve rezultate već samo prvu stranicu ili dvije.

Dodajemo u URI dodatne informacije o tome koje filtere i načine sortiranja želimo. Ako recimo želimo listu svih autora poredanih po njihovim imenima, URI kreiramo na sljedeći način: "authors?orderby=name".

4.2 Metode

Prije upuštanja u samu implementaciju HTTP metoda, najprije su objašnjene neke od najčešće korištenih, te dodatna svojstva koja je bitno razumjeti kako bi ih se pravilno primijenilo.

	Kolekcija	**Instanca**
GET	Dohvaćaju se svi resursi iz zadane kolekcije	Dohvaća se jedan resurs iz zadane kolekcije
POST	Kreira se novi resurs unutar kolekcije	
DELETE		Briše se određena instanca iz kolekcije
PUT		Ažurira se jedan određeni resurs - cijeli resurs
HEAD	Dohvaćaju se svi resursi iz zadane kolekcije - samo zaglavlje	Dohvaća se jedan resurs iz zadane kolekcije - samo zaglavlje
PATCH		Ažurira se jedan određeni resurs - dio resursa
OPTIONS	Vraća dostupne metode za kolekciju	Vraća dostupne metode za instancu

Tablica 4.1: HTTP metode

U tablici iznad navedeno je koju funkciju imaju neke od najčešće korištenih HTTP metoda, iako postoje još mnoge koje nisu spomenute. Važno je prije početka svake implementacije odrediti koje će sve akcije biti dozvoljene nad svakim pojedinim resursom.

Dva pojma često se spominju kada govorimo o HTTP metodama i važno ih je razumjeti za pravilno korištenje. Kažemo da je operacija idempotentna ako za više izvršavanja istog zahtjeva daje isti rezultat. Dakle, za rad s istim podacima moramo uvijek dobiti isti poslužiteljski odgovor. GET metoda za dohvaćanje svih pasa u Dumovcu mora uvijek vratiti istu listu pasa, pod uvjetom da u međuvremenu netko nije modificirao tu listu. PUT metoda za jedno te istu instancu i istu promjenu na njoj treba uvijek vratiti isti odgovor - tu instancu s umetnutom promjenom, iako nakon prvog umetanja više ne nastaje promjena u bazi. Ne smijemo dobiti odgovor da npr. nema promjene u zahtjevu ili da se radi o neispravnom zahtjevu jer zapravo nema promjene za unos. POST s druge strane, ne može biti idempotentan. Čak i ako okinemo nekoliko identičnih zahtjeva, svaki resurs ima svoj jedinstveni identifikator po kojem se sve instance uvijek razlikuju, stoga je nemoguće dobiti identični odgovor. Opisani način rada je dio HTTP protokola i stoga REST API mora na taj

način implementirati svoje metode, iako se u zahtjevima REST-a ovo nigdje ne spominje eksplicitno.

Drugi pojam koji je bitno spomenuti je sigurnost. Metoda je sigurna ako se reprezentacija resursa ne mijenja nakon što ju izvršimo, npr. DELETE nije sigurna budući da se određeni resurs briše iz baze.

U tablici vidimo na koji način metode koje se najčešće koriste trebaju funkcionirati po pitanju idempotentnosti i sigurnosti.

	Sigurno?	**Idempotentno?**
GET	Da	Da
POST	Ne	Ne
DELETE	Ne	Da
PUT	Ne	Da
HEAD	Da	Da
PATCH	Ne	Ne
OPTIONS	Da	Da

Tablica 4.2: Sigurnost i idempotentnost osnovnih HTTP metoda

Spomenuta svojstva donekle ograničavaju i lako je prekršiti ih ako se ne koriste metode kako je predviđeno protokolom. Još jedan razlog iz kojeg je bitno pravilno implementirati metode je predmemoriranje. Uključivanje predmemoriranja aplikacije može biti izuzetno učinkovito, ali isto tako treba ga napraviti vrlo oprezno kako bi zaista radilo na predviđeni način i nebi dovelo do pogrešaka u funkcioniranju aplikacije. Više detalja o predmemoriranju biti će kasnije, za sada je samo bitno spomenuti da ga je moguće implementirati specifično za svaku pojedinu metodu i da moramo moći osloniti se na pravilan rad metoda kako bismo dobili željeni učinak.

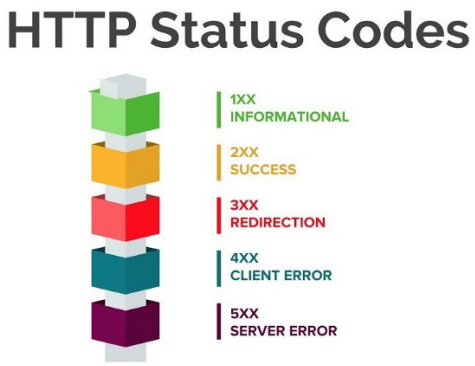
4.3 Greške i status kodovi

Pri komunikaciji između poslužitelja i klijenta može doći do raznih grešaka: od neispravno poslanog zahtjeva do greške pri dohvatu podataka iz baze i nemogućnosti slanja pripadnog odgovora. Kako bi klijent znao je li sve prošlo kako treba ili je došlo do greške, i kakve, potrebno je implementirati slanje odgovora s odgovarajućim porukama koje daju objašnjenje potencijalne greške, te status kodova koji opisuju što se dogodilo. Tada klijent može vidjeti je li potrebno promijeniti zahtjev koji šalje, npr. ne postoji autor s ID-jem kojeg je klijent unio u URI, ili je došlo do greške sa poslužiteljske strane i klijentu ne preostaje drugo nego kasnije pokušati ponovno s istim zahtjevom. Kada nebi bilo status kodova,

klijent nebi imao nikakvu drugu informaciju o tome s koje strane je i koja točno greška nastala.

Obrada nastalih grešaka nije direktni zahtjev REST-a već naprosto dio HTTP protokola. Za svaku pojedinu metodu potrebno je uvijek vratiti i odgovarajuće status kodove te poruke klijentu.

Status kodovi koji se koriste su podijeljeni u pet razina ili kategorija. Na svakoj razini postoji nekoliko kodova različitih značenja, a ovdje su navedeni neki koji se najčešće koriste. Pri implementiranju svake pojedine metode osvrnut ćemo se dodatno na to koje kodove vraćamo i zašto. Sada navodimo one koji su u najčešćoj upotrebi.



Slika 4.1: Grupe status kodova [11]

Razina 100 se sastoji od informativnih kodova koji zapravo nisu dio HTTP standarda i ne koristimo ih u ovom praktičnom primjeru. Razina kodova 200 su kodovi o uspjehu. Za uspješno izvršavanje GET metode odnosno dohvaćanja podataka poslat će se status 200 - ok. Za uspješno dodavanje novog resursa dobit ćemo 201 - created.

200 Success	
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No content

Tablica 4.3: Status kodovi 200

Razina 300 su kodovi za preusmjeravanje koji se koriste ako je recimo neka web stranica uklonjena s dane adrese, no za takve kodove većinom nema nikakve potreba kod

korištenja API-ja. Razina 4 je zato izuzetno bitna: kodovi 400. Oni opisuju klijentove pogreške. Kod 400 označava "bad request" odnosno neispravno poslani zahtjev. 401 označava da klijent nema dozvolu za pristup adresi koju traži. 404 je "not found" tj. adresa ne postoji. Ovo klijent dobiva u slučaju slanja URI-ja s ID-jem koji ne postoji u bazi. Ako pokuša obrisati resurs za koji nema dopuštenje, tada će dobiti kod 405 "method not allowed". Razina 500 označava greške od strane poslužitelja. 500 označava "internal server error".

Kada nebi bilo smislene implementacije kodova na način da se uviek šalje najprikladniji mogući kod, nego se npr. uvijek šalje 200, čak i kada nije sve ok, klijent može pomisliti da je sve prošlo dobro, iako to nije zaista slučaj. SOAP recimo ne koristi status kodove kako je opisano i uz poslužiteljske odgovore se dobiva puno manje informacija. Ako klijent pošalje GET za autora konkretnog ID-ja, i dobije kao odgovor status OK s praznim tijelom odgovora, ne zna je li u pitanju problem sa poslužiteljske strane ili je možda upisao nepostojeći ID autora.

400 Client error	
400	Bad request
401	Unauthorized
403	Forbidden
404	Not Found
406	Not Acceptable
409	Conflict
415	Unsupported Media Type

Tablica 4.4: Status kodovi 400

500 Server error	
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
599	Network Timeout

Tablica 4.5: Status kodovi 500

Poglavlje 5

REST

REST (Representational State Transfer) odnosno prijenos reprezentativnog stanja je arhitektonski stil koji pruža smjernice za razvoj web servisa. Sastoji se od skupa ograničenja, tj. smjernica koje opisuju kako bi komponente međusobno trebale komunicirati. Nema veze sa samim načinom ili stilom implementacije - to su obrasci arhitekture. Samo daje smjernice oko toga na koji bi način neke stvari trebale funkcionirati. Svaki razvojni inženjer ima slobodu na svoj vlastiti način zadovoljiti zadane smjernice i formirati implementaciju principa. REST se može primijeniti u različitim okolinama i jezicima.

Cilj REST-a je pokazati kako se kvalitetno dizajnirana web aplikacija treba ponašati. Jedna web aplikacija je zapravo mreža povezanih web stranica kroz koje se korisnik kreće pomoću linkova koje pronalazi na tim stranicama. Svako selektiranje linka odnosno klik na link će dovesti do promjene aktualnog stanja web aplikacije i do renderiranja druge web stranice. REST pomaže oko toga na koji način napraviti tu izmjenu sadržaja i razmjenu informaciju između klijenta - korisnika, i poslužitelja.

Potrebno je naglasiti da REST nije standard nego stil, a tijekom implementiranja REST-a koriste se različiti standardi. Ti standardi nisu nužno uključeni u REST web aplikaciju, ali njihovo korištenje olakšava pridržavanje principa. HTTP protokol nije striktno zadan kao obavezan dio aplikacije, međutim, korištenje tog protokola je jedini način na koji je smisleno implementirati aplikaciju koja se mora pridržavati REST principa.

5.1 Podrijetlo

Prvi spomen pojma REST nalazimo u disertacijskom radu Roya Thomasa Fieldina pod imenom "Architectural Styles and the Design of Network-based Software Architectures". U tom radu Fielding objašnjava kako je REST izveden iz kombinacije nekoliko različitih stilova arhitekture koji se koriste za kreiranje distribuiranih hipermedijskih sustava uz uvođenje dodatnih pravila i restrikcija na implementaciju. Fielding definira REST kako bi

objasnio trenutno funkcioniranje interneta te kako bi definicijom trenutnog stanja olakšao njegov napredak.

Iako je pojam REST definiran u disertaciji 2000. godine, Fielding u svom radu napominje kako se za oblikovanje web aplikacija primjenjuju pravila REST arhitekture još od 1994. godine, međutim, nikada ranije nisu na jedno mjesto skupljena i objašnjena u detalje sva pravila i principi koje je potrebno slijediti.

5.2 Elementi

Prije prelaska na principe ove arhitekture, valja spomenuti nekoliko ključnih pojmova usko vezanih uz razvoj i rad sustava. Trebalo bi objasniti što su i čemu služe resursi, reprezentacije, metapodaci i hipermediji. Svi ovi pojmovi usko su vezani uz HTTP i nužno ih je spomenuti u kontekstu tog protokola. Reusrs i reprezentacija su nešto što moramo, a metadata i hipermediji nešto što možemo koristiti. U sklopu pojedinih principa objašnjena je detaljnije uloga svakog navedenog pojma i potreba za korištenjem istog. Zatim u poglavlju o implementaciji doznajemo kako ove pojmove uklopiti u dizajn i implementaciju. Za sada slijedi samo kratko objašenjenje osnovnog značenja.

Resursi su na neki način baza cijelog klijentsko - poslužiteljskog modela. Resursi su ono što poslužitelj šalje, sprema ili recimo mijenja na zahtjeve klijenta. To je "meta" HTTP zahtjeva. Resursom se u stvari smatra bilo kakav sadržaj koji se može identificirati preko jedinstvenog URI-ja koji mu je pridružen. Može biti tekst, slika, video... Da bi se HTTP zahtjev usmjerio na određeni zahtjev, potrebno je znati identifikator i lokaciju resursa. Te informacije sadržane su u URI-ju. Pojam možda na prvu zvuči dosta apstraktno, ali kasnije će kroz principe i dizajn postati jasnije o čemu se točno radi.

Drugi pojam koji je bitno spomenuti je reprezentacija. Reprezentacija je prikaz resursa tj. može se reći da za reprezentaciju trebamo resurs i format u kojem ga prikazujemo. Kada klijent, npr. web preglednik primi poslužiteljev odgovor, pokušat će ga parsirati. Ako mu to uspije - ako podržava primljeni format, tada krajnji korisnik dobiva prikaz onoga što je zatražio (ili onoga "na što je kliknuo"), dakle otvara se nova web stranica, slika, video, dokument...

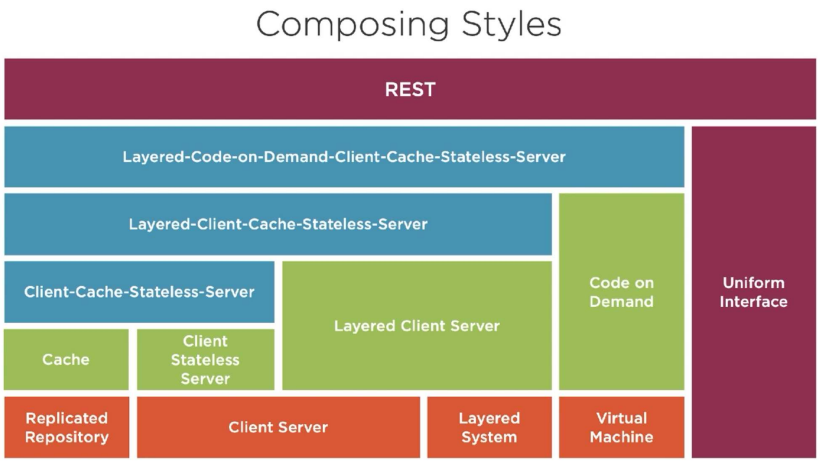
Metapodatke i hipermedije moguće je dodati reprezentaciji resursa. To su dodatne informacije koje poslužitelj može uvrstiti u svoje odgovore klijentu. Termin metapodaci odnosi se na neke dodatne informacije o našim resursima. Termin hipermedija se odnosi na bilo kakav sadržaj koji na sebi sadrži linkove na druge medije poput slika i teksta [9]. Metapodaci su među ostalim, linkovi koje sadrže hipermediji, iako se može raditi i o mnogim drugim dodatnim informacijama za klijenta.

Svaki hiperlink koji nas vodi na neki drugi resurs s neki drugim URI-jem će zapravo poslati GET zahtjev za određeni sadržaj koji ce onda server pomoću web servisa poslati klijentu kao odgovor.

5.3 Principi

RESTFul dizajn temelji se na šest različitih principa odnosno ograničenja koja treba zadovoljavati:

- Klijent - poslužitelj
- Nepostojanje stanja
- Predmemoriranje
- Uniformno sučelje
- Slojevit sustav
- Kod na zahtjev



Slika 5.1: REST principi[21]

Prva tri zahtjeva su zapravo način na koji je internet već funkcionirao prije uvođenja pojma REST i njegove formalne definicije, a ostali zahtjevi su tu u cilju poboljšanja i za budući razvoj interneta i web aplikacija. Naravno svaki od navedenih principa ima dobrih i loših svojstava, ali pozitivni doprinos ipak nadvladava dovoljno da se svaki od zahtjeva uvrstio na listu obaveznih.

Klijent - poslužitelj (client - server): Prvi princip koji treba ispuniti je potpuno odvajanje klijenta i poslužitelja. Između njih se događa razmjena podataka, ali rade sasvim neovisno jedan od drugog. Ovaj princip nije nikakva novost, koristi se redovito u razvoju

softvera i koristio se i prije same definicije REST-a. Zapravo je baza koja mora biti zadovoljena da bismo dalje mogli graditi ostale principe. Naveden je kao zahtjev REST-a kako bi se dokumentiralo na koji način raditi kvalitetan softver.

Klijent šalje poslužitelju zahtjev (request) koji može biti dohvaćanje ili recimo kreiranje novih podataka - resursa. Poslužitelj klijentu šalje odgovor (response) koji sadrži zatražene podatke, informaciju o (ne)uspješnom kreiranju resursa... Opisani proces provodi se korištenjem HTTP protokola. Dakle, korištenje HTTP protokola nije zahtjev REST-a samo po sebi, ali je potreban kako bi se implementiralo klijet - poslužitelj model.

Klijentski dio brine o prikazu podataka koje mu poslužitelj dostavlja na zahtjev. Klijent ni u kojem trenutku ne mora brinuti o tome kako su podaci koje dobiva pohranjeni u bazi u odnosu na to kako se reprezentiraju. S druge strane, poslužitelj ne brine o korisničkom sučelju ni o stanjima. Klijent nije njegov "problem", poslužitelj ima zadatak samo da na pristigli klijentov zahtjev odgovori pravim podacima u pravom formatu. Općenito, neki od najčešće korištenih formata za odgovor klijentu su XML, json i nešto manje korišteni pdf, doc. REST ipak najčešće koristi json format, iako nije ni na koji način ograničen po tom pitanju. Nije postavljeno ograničenje ni na programski jezik izrade klijenta ni poslužitelja jer se HTTP protokol može implementirati u proizvoljnom programskom jeziku.

Zašto koristiti ovaj model? Nekoliko je razloga zašto je korisno provoditi takvo odvajanje. Nezavisnost utječe pozitivno na performanse i na skalabilnost. Nadogradnjom poslužitelja ne utječe se nikako na klijenta, i obrnuto. Više ljudi može nezavisno raditi na svakom zasebno bez da to ima ikakav utjecaj na ovog drugog. Ono što može stvarati problem za skalabilnost je da svaki klijent mora uvijek na istog poslužitelja, zato postoji idući princip.

Iako općenito govoreći u razvoju web aplikacija nije nužno koristiti klijent - poslužitelj model, to je ipak bez daljnjeg u najčešćoj upotrebi. Više o nekim drugim, puno manje korištenim modelima poput: gospodar - rob (master - slave), arhitektura s distribuiranim objektima, model ravnopravnih partnera (peer - to peer) može se pročitati u [16].

Nepostojanje stanja (statelessness): Zahtjeva se neovisnost stanja na klijentu. Neovisnosnt među stanjima se može postići i na poslužitelju, i to pozitivno utjeće na performanse, ali REST ne govori niti zahtjeva nešto po tom pitanju. Da bi se uopće moglo pričati o neovisnosti s klijentske strane, za početak, klijent - poslužitelj model mora biti zadovoljen. Već na drugom koraku vidimo nadogradnju i povezanost principa.

Stanja aplikacije redovito se mijenjaju i međusobno su nezavisna. Svaki trenutni zahtjev ima informacije isključivo o trenutnom stanju i nema potrebu niti mogućnost znati išta o prethodnim (ni budućim) zahtjevima. Sve informacije jednog zahtjeva sadržane su u metodi, URI-ju, zaglavlju i eventualnom tijelu zahtjeva koji se šalje. To sve je dovoljno kako bi se poslao pravilan zahtjev poslužitelju i kako bi se mogao dobiti pravilan i potpuni odgovor. Metoda zahtjeva govori kakva točno promjena je tražena, npr. GET zahtjev dohvaća

podatke, dok POST kreira novi resurs. URI sadrži putanju koja govori nad kojom vrstom entiteta, ili čak nad kojim točno zapisom određenog entiteta (ako se šalje i jedinstveni identifikator) se želi nešto napraviti. Ako je poslan GET zahtjev i URI oblika "...api/shelters", u tom slučaju pokušavaju se dohvatiti informacije o svim skloništima u sustavu. Ako je zahtjev oblika "...api/shelters/12345" tada će se dohvatiti informacije samo o skloništu čiji je identifikator zadan. Za POST zahtjev, potrebno je poslati i tijelo zahtjeva u kojem su ispunjeni podaci o novom zapisu koji se dodaje. Zaglavlje može sadržavati informacije o tipu podataka koji želimo u odgovoru ili npr. zahtjeve za predmemoriranje podataka.

Nedostatak primjene ovog principa je taj što je potrebno slati puno podataka u sklopu svakog klijentskog zahtjeva. To negativno utječe na performanse, ali i na sigurnost. Ako je za sustav bitna velika sigurnost zbog pohrane osobnih podataka ili recimo novčanih transakcija, može se reći da je za inženjere ovaj princip više bug nego funkcionalnost. Potrebno je uložiti u to da se sustav dobro osigura od potencijalnih narušavanja privatnosti i nedozvoljenog dohvaćanja informacija. Naime, preko stanja koja se šalju lako je ući u kod i napraviti "štetu" nad podacima ili ih kasnije zlorabiti.

Predmemoriranje (caching): Ovo nije zahtjev za samim predmemoriranjem, već se traži da se definira je li neki odgovor klijentu "cacheable" ili nije. Radi se o pohranjivanju podataka sa poslužitelja u cache kako se ne bi morali svaki puta iznova slati u odgovoru i kako bi se ubrzao rad aplikacije. Ovim principom se vidno optimizira rad aplikacije.

Predmemoriranje ili caching je vrlo moćan i istovremeno opasan princip. Ima velike prednosti, ali ako se pogrešno koristi može uzrokovati baratanje isteklim i pogrešnim informacijama. Ako se cachira informacija koja se redovito mijenja, tada će dolaziti do pogrešnog rada aplikacije. Razvojni inženjer za svaku informaciju čije predmemoriranje implementira mora biti svjestan koliko često bi se to moglo u upotrebi ažurirati i ima li smisla pohranjivati informacije te koliko dugo.

"Client-cache-stateless server" je već postojao u 90-ima. Ovime se samo opisuje postojeće stanje i tek pravilima koja slijede Fielding opisuje u kojem smjeru vidi širenje weba te opisuje što je drugačije potrebno napraviti da se to postigne.

Uniformno sučelje (uniform interface): Već ovaj princip, koji je prvi Fieldingov novitet, je nešto čega se puno manje pridržava kod izrade web aplikacija. Često se neki API karakterizira kao RESTFul, iako ne slijedi ovaj princip već samo koristi URI i HTTP protokol. Ovaj princip je upravo ono što primarno razlikuje REST od drugih arhitektura.

Za dobivanje uniformnog sučelja u REST arhitekturi, uvedena su 4 zahtjeva: identifikacija resursa, manipulacija resursima kroz reprezentacije, samoopisne poruke i HATEOAS (hypermedia as the engine of application state) odnosno hipermediji kao pokretač aplikacijskog stanja. Ono što se najviše preskače napraviti je HATEOAS.

Zahtjev na identifikaciju resursa znači da se izvor informacija koji se nalazi na poslu-

žitelju razlikuje od odgovora koji dobiva klijent. Neovisno o tome u kojem formatu klijent traži odgovor s npr. GET zahtjevom, svaki puta zapravo dolazi jedan te isti odgovor s jedno te istim informacijama iz baze pripremljenim za klijenta, ali će se na različit način renderirati tj. u drugačijem formatu. URI zadaje koje resurse točno želi i uvijek će dobiti iste informacije, iako prikaz može biti u različitim formatima, odnosno sam resurs nije isto što i reprezentacija tog resursa - razdvajamo ta dva pojma.

Drugi zahtjev je manipulacija resursima kroz reprezentaciju. Dohvaćanje pojedinog resursa tj. svi podaci koje dobivamo o samom resursu te o mogućim akcijama moraju biti dovoljni kako bi se moglo promijeniti ili obrisati taj isti resurs. Navedene akcije nisu uvijek dozvoljene, ali ako nema ograničenja, moraju biti svima jednako omogućene. Metapodaci igraju veliku ulogu u ostvarenju ovog zahtjeva jer sadrže dodatne URI-je koji omogućavaju navedene akcije. Moguće je i samo poslati identifikator u odgovoru, ali to se može smatrati nepotpunom informacijom.

Samodeskriptivne poruke su idući zahtjev. Ovo se odnosi na zadavanje tipa podataka u zaglavlju zahtjeva i služi kako bi se poslani podaci iz tijela zahtjeva uspješno parsirali i kako bi klijent znao koristiti pristigle informacije. Ako ništa nije zadano, REST koristi json. Ako je tijelo za POST poslano u xml formatu, bez da je u zaglavlju zadan taj format, API neće prepoznati poslani zahtjev i vratit će grešku.

HATEOAS je jedan od ključnih dijelova REST-a koji se toliko često zaboravlja i preskače. Odnosi se na hiperlinkove, na povezivanje. Poslužiteljski odgovor ne bi trebao sadržavati samo zatražene resurse, nego i omogućene akcije za te resurse. Odgovor na GET zahtjev treba sadržavati kolekciju hiperlinkova za preostale implemetirane moguće akcije - POST, DELETE... Mijenjamo stanje aplikacije korištenjem danih linkova.

Komponente uniformnog sučelja u osnovi uključuju: URI + HTTP metode + tip podataka. Uniformno sučelje pruža skalabilnost, lako povezivanje različitih programa zajedno - kao i dosadašnji principi. Arhitektura se pojednostavljuje i razdvaja. Već je sada jasno kako se REST uklapa u agilni način rada i korištenje mikroservisa - lako i brzo možemo mijenjati i nadograđivati sustav bez da pritom nastaje šteta na nepovezanim funkcionalnostima. Poslužitelj se može neometano širiti bez da klijent zna išta o tome. Cijena koju plaćamo za to je smanjena efikasnost budući da svaki program mora prebaciti svoje podatke u isti univerzalni format.

Slojevit sustav (layered system): Za slojeviti sustav potrebno je najprije imati implementirano uniformno sučelje i klijentsko-poslužiteljski model. Ideja ovog principa je da svaki sloj koji gradimo zna isključivo za slojeve neposredno prije i nakon sebe. Klijent ni u jednom trenu nema informaciju o tome dolaze li resursi u odgovoru iz njemu najbližeg sloja ili se između nalazi još nešto.

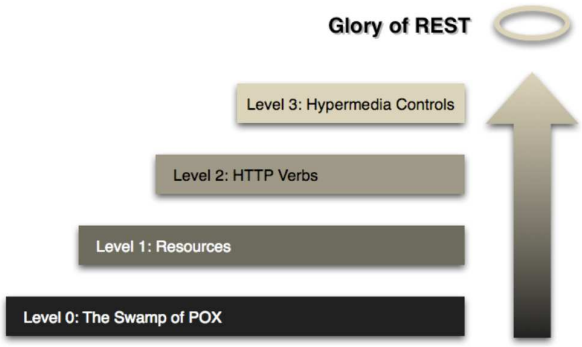
Slojevitost smanjuje cjelokupnu kompleksnost sustava, međutim to ima svoju cijenu. Performanse se smanjuju zbog kompleksnije obrade podataka. Dobro izvedeno predme-

moriranje pomaže oko tog nedostatka i uz njegovo korištenje smanjene performanse su klijentu vidljive u manjem opsegu, budući da se podaci ne moraju svaki puta ponovno dohvaćati.

Kod na zahtjev (code on demand): Kod na zahtjev je jedini opcionalni princip koji nije nužno ispuniti kako bismo određenu arhitekturu mogli karakterizirati kao RESTFul. Na temelju ostalih principa imamo opciju u nekim slučajevima primjeniti ovaj, a s druge strane postoje situacije u kojima nije prakticno koristiti ga. Poanta je da ga se može koristiti kada postoji potreba za to, ali nije nužno. Kod na zahtjev ne smije biti jedini način da klijent napreduje i ne smije biti baza dizajniranja sustava.

5.4 Richardsonov model zrelosti

Richardsonov model govori koliko je web API "daleko" od REST-a po pitanju korištenja HTTP protokola [14]. Naime, postoje 4 razine, od kojih tek API-je koji su na zadnjoj razini možemo karakterizirati kao RESTFul. Dakle ne postoji REST API razine 1, 2... Postoji proizvoljni API proizvoljne razine, a ako je API razine 3, tada ga se može smatrati RESTFul. Slijedi objašnjenje kako od potpuno primitivnog korištenja HTTP protokola doći do nečega što se može smatrati sofisticiranom REST arhitekturom. Iako REST hipotetski može koristiti bilo koji protokol, u praksi se gotovo uvijek odabire HTTP. Richardsonov model određuje koliko se ili bolje rečeno koliko se pravilno koristi taj protokol. Faktori na temelju kojih su konstruirane razine, odnosno faktori koji se mjere uključuju URI, HTTP metode i HATEOAS.



Slika 5.2: Razine Richardsonovog modela zrelosti [5]

Razina 0 ("Swamp of POX (Plain Old XML)"): Ako smo na razini 0, to znači da se HTTP samo koristi - u najprimitivnijem mogućem obliku i bez pridržavanja određenih pravila. Poanta je samo doći do traženog podatka na najjednostavniji način. Ovdje se HTTP

koristi samo kao protokol za transport. Obično se koristi jedan fiksni URI i samo jedna metoda. Metoda je najčešće POST. I dalje postoje neka pravila kojih bi se trebalo pridržavati vezano za format URI-ja kako bi bio što čitljiviji i kako nebi došlo do zbunjivanja klijenta. SOAP funkcionira od prilike na ovaj način, uz to da se "swamp of POX" šalje kao dio takozvane omotnice (envelope). Više detalja o SOAP-u i njegovom formatu kasnije.

Razina 1 (Resursi): Ono što ostaje isto u odnosu na razinu 0 je da se i dalje koristi jedna metoda, najčešće POST. Napredak u odnosu na prethodni level je složenija upotreba URI-ja. Za različite resurse na ovom levelu počinju se koristiti različiti URI-ji. Postoje dodatna pravila za pisanje URI-ja kojih se treba pridržavati kako bi URI bio jasan i pregledan. Jedno od bitnijih pravila je da znak "/" ne smije biti dio URI-ja. Taj znak se koristi samo za odvajanje kako bi hijerarhija među resursima bila jasna.

Razina 2 (HTTP metode): Na razini 2 se uvode metode. Više ne koristimo samo POST. Počinju se koristiti metode kao što su npr. GET, HEAD, PUT i DELETE. Svaka metoda ima svoju jasnu zadaću i pravilno ju se koristi. GET dohvaća resurse, DELETE ih briše itd. Ono što se također dodaje su status kodovi. To je još jedna povratna informacija klijentu kako bi znao je li sve prošlo kako treba te do koje je greške potencijalno došlo.

Razina 3 (hipermedijske kontrole): Ova razina sastoji se od dva dijela. Prvi dio se odnosi na mogućnost korištenja različitih reprezentacija ("content negotiation"), a drugi dio se odnosi na korištenje hipermedija (HATEOAS). Ova razina najčešće nije implementirana u web aplikacijama. Budući da isti API može biti korišten od strane većeg broja klijenata, dobro je u takvim API-jima omogućiti odabir formata za reprezentaciju resursa kako bi svaki klijent mogao dobiti format koji mu odgovara. Taj odabir je omogućen u zaglavlju klijentskog zahtjeva na način da klijent specificira listu formata koji mu odgovaraju i redoslijed prioritizacije za dane formate. To je opis "content negotiation-a". Klijent ima opciju odabira, ne koristi se jedan jedini format. HATEOAS je već objašnjen u ranijem tekstu, ali da se kratko podsjetimo što su hipermediji: resursi koji sadrže metapodatke koji uključuju moguće akcije za svaki resurs - linkove. Tek API koji ima implementiranu ovu razinu možemo početi razmatrati kao kandidata za RESTFul API.

5.5 REST vs SOAP

Nakon što je dan uvid u REST i sve što tako karakteriziran API mora sadržavati, pravi je trenutak za predstavljanje SOAP-a i usporedbu ovih pojmova. Do sada je opisano od čega se sve mora sastojati aplikacija koja ima REST arhitekturu i istaknute su pozitivne i negativne strane koje sa sobom donosi svaki od osnovnih principa. Sada ćemo promotriti obilježja SOAP-a u odnosu na REST te usporediti njihovu strukturu i pokazati u kojoj situaciji i zašto odabrati svaku od njih.

SOAP (Simple Object Access Protocol) [8] je protokol za razmjenu poruka. Za tu razmjenu koristi primarno HTTP ili SMTP (Simple Mail Transfer Protocol). Poruke koje se

šalju korištenjem SOAP protokola bit će u XML formatu. Cijela SOAP poruka sastoji se od nekoliko dijelova koji zajedno tvore takozvanu omotnicu. Ako je klijent primio cijelu omotnicu, to znači da je poruka uspješno poslana u cijelosti i da se može parsirati te prikazati korisniku. SOAP je protokol dizajniran prije postojanja REST-a, krajem 90-ih. Cilj kreiranja SOAP-a je bio omogućiti jednostavnu razmjenu informacija i dobiti neovisnost o platformi i programskom jeziku. Primarno aplikacije SOA arhitekture se oslanjaju na SOAP. Ovo je standard koji se pridržava strogog protokola, i za formalni oblik komunikacije među web servisima je idealan odabir. Isto tako, pogodno ga je koristiti kod kreiranja privatnih API-ja gdje se zahtjeva visoka razina sigurnosti. Pogotovo novije verzije ovog protokola pružaju visoku dozu sigurnosti.

Zašto su REST i SOAP pojmovi koji se tako često uspoređuju iako se uopće ne mogu jednako kategorizirati? [19] Jedan pojam je stil arhitekture a drugi je "samo" protokol kojim se razmjenjuju informacije. Iako je REST arhitektonski stil, u srži se njegovi zahtjevi svode na definiranje načina na koji će se vršiti razmjena poruka. U nastavku je dan pregled razlika između SOAP-a i protokola generiranog REST-om. Ni jedan nije bolji ili lošiji, svaki ima svoje prednosti i mane u odnosu na drugog i bitno je razumjeti kada je potrebno koristiti SOAP, a kada REST. Iako hipotetski REST može koristiti SOAP (dok SOAP ne može koristiti REST), to je u praksi izuzetna rijetkost i u velikom broju slučajeva razvojni inženjeri koriste ili jedno, ili drugo.

Primarno se SOAP koristi za SOA, kao što je već i spomenuto, dok je REST u upotrebi za mikroservisnu arhitekturu. Rečeno je već da SOAP koristi XML te da se poruke šalju u formatu omotnice (koja se sastoji od zaglavlja i tijela). Zbog toga poruke brzo mogu postati veoma glomazne i lakše je izgubiti dio informacije jer je potrebno poslati i primiti cijelu veliku poruku. S druge strane, REST najčešće koristi json i manje je formalan pa se u konačnici šalje manja količina podataka svakom porukom. Velike SOAP poruke mogu potencijalno loše utjecati na performanse, i kada imamo ograničenu propusnost, tada je puno bolje opredjeliti se za REST i njegove kraće poruke.

Osim toga, budući da SOAP koristi samo jedan format, dok ih REST redovito implementira nekoliko i omogućava korisniku da sam odabere u kojem točno želi dobiti odgovor, veće su šanse da neće biti kompatibilan s nekim web servisom, u odnosu ne REST koji je fleksibilniji po tom pitanju. Iz tog razloga je REST pogodniji za javne API-je koji će imati puno različitih korisnika, dok je SOAP pogodniji za privatne API-je rađene za nekog konkretnog klijenta. Još jedan od razloga za takvu podjelu je sigurnost: SOAP je puno sigurniji i time također pogodniji za privatne API-je.

Ako nam ne trebaju informacije o prethodnim stanjima, tada je REST pravi odabir. Ako nam trebaju informacije od ranije, npr. trebamo sadržaj košarice u prelasku na stranicu za plaćanje, tada nam je potreban SOAP koji može zadržavati takve informacije, dok REST zbog nepostojanja stanja ovdje nije primjenjiv.

Kada se očekuje da će klijent više puta tražiti iste resurse, tada je dobro implementirati predmemoriranje, za što je REST idealan odabir, a korištenje SOAP-a ovdje nije pogodno jer on stalno šalje velike zahtjeve poslužitelju. Drugačije ćemo odabrati i ovisno o veličini aplikacije za izradu te vremenu koje imamo na raspolaganju. REST je vidno brže i lakše implementirati pa je pogodan za projekte koji trebaju biti dovršeni u kratkom roku.

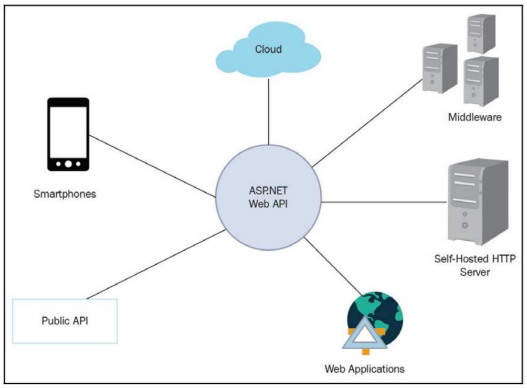
Vidimo da ni jedan protokol nije dobar ili loš, svaki ima svoje karakteristike i pogodni su za različite primjene. Na svakom developeru je da procjeni što mu je potrebno i koju cijenu može platiti za to, npr. može li si dopustiti da izgubi na sigurnosti kako bi uspio što prije dostaviti proizvod itd.

Poglavlje 6

Tehnologije i alati

Za izradu praktičnog dijela ovog rada koristi se nekoliko različitih tehnologija i alata. Nije ih nužno upotrijebiti kako bi se dobile sve funkcionalnosti koje se u nastavku opisuju, ali su u ovom slučaju korišteni jer pružaju dobru podršku za REST.

6.1 ASP.NET Core



Slika 6.1: [6]

Za samu implementaciju koristi se ASP.NET Core. Pogodan je za korištenje na bilo kojem operacijskom sustavu. Besplatan je i otvorenog koda. Pogodan je za mikroservise. ASP.NET je predviđen za izradu dinamičkih web stranica. Podržava korištenje svih jezika koje .NET podržava, među kojima je C# i njega upravo koristimo za ovaj rad.

U odnosu na ASP.NET uvedene su promjene koje olakšavaju i automatiziraju neke procese vezane za implementaciju REST-a. Pojednostavljeno je korištenje u svrhu izrade

REST API-ja. Naime, spojeni su modeli za MVC i za Web API u jedan obrazac. Olakšano je implementiranje HTTP protokola. I dalje postoje stvari za koje se trebamo sami pobrinuti kako bi HTTP protokol bio u potpunosti implementiran i REST zadovoljen, ali neke osnovne stvari su olakšane. Što se tiče prikaza za klijenta, ako ne želimo koristiti ugrađene opcije za view, moguće je jednostavno implementirati neki postojeći ili novi mehanizam. Ovo sve može se koristiti pomoću Visual Studija koji podržava korištenje spomenutih obrazaca.

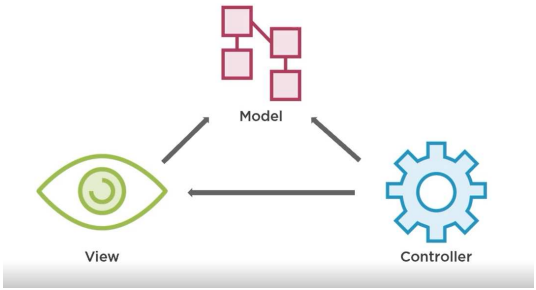
6.2 MVC

Slijedi malo detalja o obrascu arhitekture koji je korisno koristiti za ovakve aplikacije i koji je korišten za praktični dio ovog rada. MVC (Model-View-Controller) je već spomenut ranije kao primjer obrasca arhitekture kod implementacije aplikacije. Ovo je jedan od najčešće korištenih obrazaca za izradu web aplikacija. Popularnost duguje tome što je ovako građena aplikacija praktična za održavanje i proširivanje.

Model dohvaća i pohranjuje podatke. Model se često poklapa s resrusima koje klijent dobiva, što se može podosta razlikovati od samog modela podataka koji se pohranjuje u bazu. Model sadrži podatke koji se prikazuju klijentu, a može sadržavati neke popratne, pomoćne informacije.

Pogled (view) služi za prikaz podataka, većinom se koristi json reprezentacija. Ovo je jedini sloj kojeg je klijent svjestan i zapravo nema nikakva saznanja što se sve događa u pozadini, koji su aplikacijski slojevi i odakle dolaze podaci koji se prikazuju.

Općenito, upravitelj (controller) povezuje model i pogled. Kada klijent pošalje zahtjev preko sloja pogled, to dolazi u upravitelj koji se pobrine da se iz modela dohvate pravi podaci i pošalju u view. U ovom sloju se implementira poslovna logika aplikacije.



Slika 6.2: MVC model [3]

Ovakvo razdvajanje logike i prikaza olakšava održavanje. Kao što se vidi na slici, pogled je ovisan o modelu. Ako dodamo novi stupac u tablicu, recimo želimo dodati

u tablicu s podacima o skloništima pasa podatak o godini osnivanja skloništa, događa se promjena u sloju modela. Da bi se taj podatak počeo prikazivati klijentu, promjena se treba dogoditi i u sloju pogled - trebamo u naš prikaz podataka dodati godinu osnivanja. U ovom slučaju je također potrebno prilagoditi i upravitelj - on određuje da se nadodani podatak iz baze prikazuje u novom elementu na pogledu. Ovisno o promjeni, moguće je da treba napraviti promjenu i u upravitelju. U slučaju dodavanja sasvim nove tablice, potrebno je kreirati i novi pogled i novi upravitelj. S druge strane, ako se odlučimo na promjenu u sloju pogled, nije nužno potrebno raditi promjene i u drugim slojevima. Promjena redoslijeda, boja, pozicije na prikazu za klijenta ne utječe na sami rad i logiku.

Praktični primjer za ovaj rad također će slijediti ovaj obrazac i na opisani način razdvajati slojeve.

6.3 Entity Framework Core

Entity Framework Core je nova verzija Entity Framework-a koja je otvorenog koda. Entity Framework je okvir koji služi za mapiranje objektno-relacijskih baza. Prilagođena je za .NET jezike. Pomoću ovog okvira olakšano je slanje upita na bazu, migracije podataka, korištenje instance baze unutar Visual studija. Kod koji ćemo graditi se oslanja na repozitorij čije metode odnosno dijelove metoda ćemo kasnije vidjeti u sklopu implementacije. Na početku implementacije zadaje se "seed data", početni podaci, koji se dodaju migracijom. Upiti na bazu se ne moraju ručno pisati za svaki pristup bazi već korištenjem entiteta koje kreiramo na početku i "contexta" lako dohvaćamo i spremamo podatke u tražene tablice. Kontekst koji kreiramo nasljeđuje od Entity Framework-ovog DbContext-a.

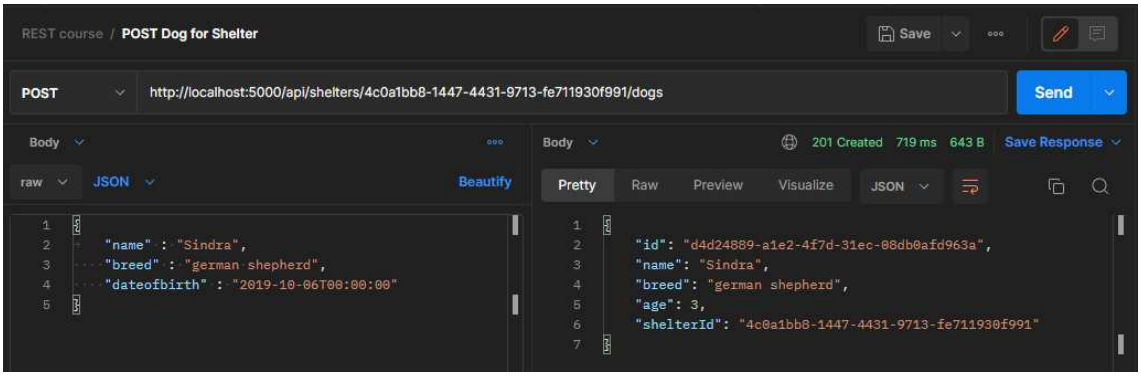
6.4 Postman

Prije detalja o samoj implementaciji, treba spomenuti jedan korisni alat: Postman. To je platforma koja pomaže u izgradnji i testiranju API-ja. Koristi se za slanje HTTP zahtjeva i iščitavanje odgovora. U nastavnim poglavljima koristi se kako bi se demonstrirao rad API-ja koji je implementiran u sklopu ovog rada, vidjet ćemo na koji način treba strukturirati zahtjeve za različite scenarije te kako treba izgledati odgovor koji poslužitelj vraća.

Kod slanja zahtjeva od klijenta prema poslužitelju, šalju se tražena metoda, URI, zaglavlje i eventualno tijelo zahtjeva, ovisno o tome traži li ga korištena metoda.

Kao poslužiteljski odgovor dolazi sam sadržaj koji je zatražen, zaglavlje te status kod. Odgovor sadrži podatke koje smo tražili, npr. ispis podataka o svim resursima nekog tipa. Status kod daje informaciju o tome je li zahtjev uspješno odrađen, i ako nije, o kakvoj vrsti problema se radi. U zaglavlju su informacije o tipu podataka, metapodaci i podaci u cache-u.

Na slici niže je sučelje Postman-a koje prikazuje slanje POST zahtjeva za dodavanje novog resursa, te odgovor poslužitelja koji je uspješno pohranio podatke.



Slika 6.3: Postman

Poglavlje 7

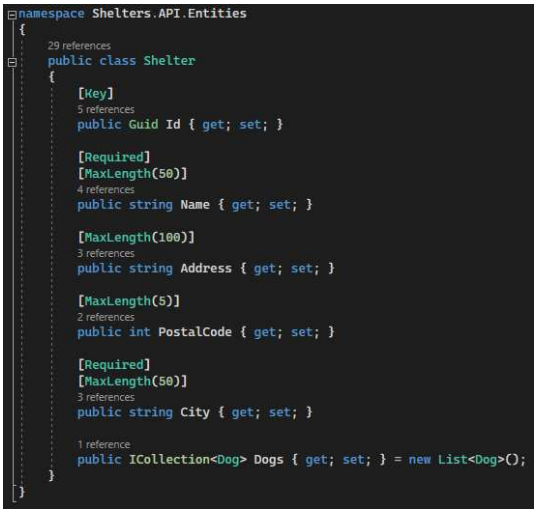
Dizajn i implementacija

U ovom poglavlju je objašnjeno što je sve nužno uzeti u obzir pri izradi REST API-ja. U principu se radi o pravilnoj implementaciji HTTP protokola i svega što uključuje njegovo korištenje. Ovdje se spominje i dosta stvari koje nisu direktno zahtjev REST-a, ali u praksi je potrebno pobrinuti se za pravilno korištenje svega navedenog. Isto tako, spomenuti ćemo neke naprosto korisne prakse u pisanju koda koje nemaju veze s REST-om ni HTTP-om direktno, ali ih je preporučljivo slijediti. U prethodnim poglavljima objašnjeno je što je sve potrebno zadovoljiti kako bi dobili REST API. Ovo poglavlje govori kako to dobiti. Na konkretnom primjeru objasnit će se implementacija određenih komponenti iz prethodnih poglavlja.

Najprije nekoliko osnovnih informacija o radu. Radi se o API-ju koji pohranjuje podatke o skloništima i psima koji su tamo smješteni. To su entiteti koje imamo. Između njih postoji hijerarhija - svaki pas pripada točno jednom skloništu. Niže su prikazane pripadne klase za te entitete i jednako ovako su kreirani i zapisi u bazi.

Ovo poglavlje će pokazati što sve treba uzeti u obzir kod osmišljavanja dobrog dizajna, kako implementirati neke protokole te kojih se ne nužnih praksi korisno pridržavati. Postepeno se gradi i proširuje kod koji kreće od ova dva osnovna entiteta, a završava kodom koji na njih primjenjuje nekoliko različitih metoda, vraća odgovarajuće kodove, kreira metapodatke i linkove.

Dizajn kakav se ovdje kreira moguće je primjeniti u proizvoljnim programskim jezicima, dok se za dijelove implementacije koriste specifične funckionalnosti omogućene ovim specifičnim tehnologijama, što ne znači da ih nije moguće izvesti drugačije.



Slika 7.1: Entitet skloniste

Slika 7.2: Entitet pas

Pogledajmo kratko ranije spomenuti repozitorij. ShelterContext koji se koristi u ShelterRepository ima svojstva jednaka definiranim entitetima. Vidimo odmah na slici ispod i primjer pristupanja bazi gdje vrlo jednostavno dohvaćamo podatke. Zadajemo koji entitet (koja tablica) iz repozitorija nas zanima, a zatim pomoću FirstOrDefault() zadajemo uvjet za koji tražimo prvi odgovarajući zapis iz bazi.

```
public class ShelterRepository : IShelterRepository, IDisposable
{
    private readonly ShelterContext _context;
    private readonly IPropertyMappingService _propertyMappingService;

    0 references
    public ShelterRepository(ShelterContext context, IPropertyMappingService propertyMappingService)
    {
        _context = context ?? throw new ArgumentNullException(nameof(context));
        _propertyMappingService = propertyMappingService ?? throw new ArgumentNullException(nameof(propertyMappingService));
    }

    1 reference
    public Shelter GetShelter(Guid shelterId)
    {
        if (shelterId == Guid.Empty)
        {
            throw new ArgumentNullException(nameof(shelterId));
        }

        return _context.Shelters.FirstOrDefault(a => a.Id == shelterId);
    }
```

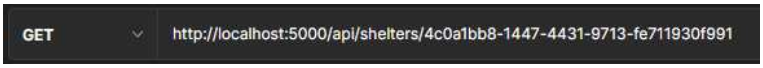
Slika 7.3: repozitorij

7.1 Imenovanje resursa

Prije nego krenemo na samu implementaciju, nekoliko riječi o tome kakav dizajn prema klijentu ima najviše smisla kad je u pitanju URI - jedan od osnovnih dijelova koji služi za komunikaciju između poslužitelja i klijenta. REST ni na koji način ne zahtijeva i ne ograničava dizajn imenovanja resursa. Ipak, postoje uvažene inženjerske prakse koje se primjenjuju kod izrade općenitih API-ja. Konzistentnom primjenom određenih smjernica dobiva se na preglednosti i na jednostavnosti korištenja za klijenta. Svaki resurs ima svoj URI, ali nema standarda koji opisuje kako ga treba dodijeliti.

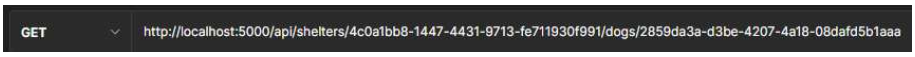
Prva stvar koju je bitno istaknuti je korištenje imenica, ne glagola. Budući da koristimo HTTP standard, za svaki zahtjev poslužitelju šaljemo nekoliko osnovnih informacija od kojih je jedna glagol za opis željene akije nad resursom - GET, POST, DELETE... To je dovoljan opis radnje samo za sebe i nema potrebe da se URI kreira na način npr. "api/get-shelters". Šaljemo višak informacija koji nam ni na koji način ne doprinosi. S druge strane, ako URI oblikujemo korištenjem imenica i ako zahtjev za dohvat svih skloništa ima oblik "api/shelters", vidimo da se isti URI može iskoristiti za sve implementirane radnje na tom resursu. Klijentu je na ovaj način jednostavnije koristiti API, a s druge strane inženjer ima jednostavniji i pregledniji kod koji sve radnje nad istim resursom može smjestiti u isti upravitelj.

Druga stvar je korištenje množine. Postoje debate oko toga trebaju li imenice biti u jednini ili množini, no preporuća se množina jer je tada jasnije što se dohvaća. Čak i kada je u pitanju dohvaćanje samo jednog skloništa, neovisno o količini instanci određenog resursa, URI bi trebao uvijek biti jednak - "api/shelters" umjesto "api/shelter". Kada nas zanima samo jedno, konkretno sklonište, dohvaćamo ga tako da dohvaćanju svih skloništa pridružimo jedinstveni identifikator konkretnog skloništa koje nas zanima. U svakom slučaju, neovisno o odabranoj praksi, najbitnije je ostati konzistentan i pridržavati se istog kroz cijeli razvoj kako bi API bio praktičan i jasan za korištenje.



Slika 7.4: URI za dohvat konkretnog skloništa

Hijerarhija također korisniku olakšava korištenje API-ja i navigiranje po njemu. Ako API pohranjuje informacije o skloništima i psima, jasno je da je svaki pas koji je unesen u bazu smješten u točno jedno sklonište kojem pripada. Zbog toga moramo prvo odabrati traženo sklonište da bismo zatim došli do pasa koji su tamo zbrinuti. Kada nas zanima konkretan pas, dodajemo na već izgrađeni URI još i jedinstveni ID: "shelters/id/dogs/id".



Slika 7.5: URI za dohvat konkretnog psa

Preporučaju se i određene prakse za kreiranje URI-ja koji sadrže query stringove. Koriste se drugačija pravila da bude jasno da se ne radi o novoj vrsti resursa. Nakon što odlučimo koje resurse želimo, stavljamo upitnik i onda redamo query string, npr. "api/shelters?name=dumovec&pageNumber=2". U nastavku će biti objašnjeno među ostalim i kako u implementaciju uključiti filtritanje i pretraživanje, a zatim i straničenje. Korištenje query stringova ne čini aplikaciju više ili manje RESTFul, niti je nužan dio HTTP-a. Međutim, spada pod dobre programerske prakse i olakšava korištenje aplikacije, a isto tako nije komplicirano za izvesti. Iz tih razloga, uključena je implementacija istoga.

7.2 Usmjeravanje

Sada kada znamo kako ćemo kreirati URI-je, vrijeme je da se objasni usmjeravanje (routing). Najkraće rečeno, usmjeravanje spaja URI iz zahtjeva s pripadnom akcijom u kodu. Svaki zahtjev klijenta prema poslužitelju sadrži URI i metodu. Te dvije informacije koriste se za odabir metode u upravitelju koja će obraditi dani zahtjev. Za API koji gradimo potrebna su za početak dva upravitelja: jedan za sklonište i jedan za pse. Upravitelj za skloništa odgovara na sve zahtjeve oblika "api/shelters" i "api/shelters/id". Ekvivalentno je napravljeno za pse. Upraviteljima i pojedinim metodama možemo pridružiti atribute koji omogućavaju prepoznavanje ciljanih zahtjeva. U ovom slučaju najprije zadajemo "Route" atribut na razini cijelog upravitelja kako bi svi zahtjevi istog oblika došli na isto mjesto. Zatim dodatnim atributom pojedinim metodama naznačavamo koja će obraditi koju akciju i koje dodatne informacije može primiti iz URI-ja.

Za atribute Route i npr. HttpPut u vitičastim zagradama su promjenjive informacije, u našem slučaju ID koji možemo dodatno zadati ako želimo npr. dohvatiti ili ažurirati

```
namespace Shelters.API.Controllers
{
    [ApiController]
    [Route("api/shelters/{shelterId}/dogs")]
    [HttpCacheExpiration(CacheLocation = CacheLocation.Public)]
    [HttpCacheValidation(MustRevalidate = true)]
    1 reference
    public class DogsController : ControllerBase
    {
        private readonly IShelterRepository _shelterRepository;
        private readonly IMapper _mapper;

        0 references
        public DogsController(IShelterRepository shelterRepository, IMapper mapper)
        {
            _shelterRepository = shelterRepository
                ?? throw new ArgumentNullException(nameof(shelterRepository));
            _mapper = mapper
                ?? throw new ArgumentNullException(nameof(mapper));
        }

        [HttpGet(Name = "GetDogsForShelter")]
        0 references
        public ActionResult<IEnumerable<DogDto>> GetDogsForShelter(Guid shelterId)
        {
```

Slika 7.6: upravitelj s metodama za sve akcije nad entitetom pasa

konkretnog psa. Metoda poput one sa slike - UpdateDogForShelter poziva se ako je poslan zahtjev čiji URI započinje kao što je navedeno na razini upravitelja u Route atributu te je nakon toga navedeno ono što imamo na razini same metode - ti komadi se spajaju i zajedno daju potpunu informaciju.

Vidimo još da GetDogsForShelter prima ID skloništa čije pse dohvaćamo. Taj ID dolazi iz URI-ja. U atributu Route na početku upravitelja definiran jer oblik URI-ja i sadrži jedan promjenjivi parametar koji treba iščitati i koristiti pri dohvatu resursa za GET. S druge strane, GetDogsForShelter prima shelterId koji se također treba od nekuda iščitati i za to postoji nekoliko mogućnosti kao što je iz URI-ja, iz query stringa... Da bi se naznačilo odakle dolazi neki parametar, inače je potrebno ispred njegove definicije dodati atribut oblika FromRoute, FromQueryString itd. Atribut ApiController postavlja neke vrijednosti, i za zadani shelterId je postavljena početna vrijednost FromRoute što zaista i jest izvor te informacije pa nema potrebe posebno to naglasiti.

```
[HttpPut("{dogId}")]
0 references
public IActionResult UpdateDogForShelter(Guid shelterId, Guid dogId, DogForUpdateDto dog)
{
```

Slika 7.7: UpdateDogForShelter() poziva se za URI oblika api/shelters/id/dogs/id

Na taj način se povezuju klijentovi zahtjevi i njihova obrada u kodu. Primjetimo ovdje još jedan atribut: "ApiController" na početku upravitelja.

Postoje višestruke korisne posljedice korištenja tog atributa, a jedna od njih je upravo omogućavanje usmjeravanja korištenjem atributa. To naravno nije jedino čemu ovo služi,

```
namespace Shelters.API.Controllers
{
    [ApiController]
    [Route("api/shelters")]
    0 references
    public class SheltersController : Controller
    {

        [HttpGet(Name = "GetShelters")]
        0 references
        public IActionResult GetShelters()
        {
```

Slika 7.8: atribut ApiController

omogućeno je još mnoštvo toga kao što je automatsko vraćanje odgovarajućeg status koda na klijentov neispravan zahtjev, o čemu više detalja kasnije. Primjetimo ovdje još kako se za neke od zahtjeva koristi isti URI a razlika je samo u metodi koja se primjenjuje - recimo DELETE i GET. Sada postaje jasnije zašto je praktično - i poslužitelju i klijentu, da se korisite ranije spomenute smjernice za kreiranje URI-ja.

7.3 Dohvaćanje resursa

Da bi se zadovoljio HTTP protokol potrebno je pravilno implementirati metode. Počnimo s dohvatom podataka.

Bitno je na početku razmisliti o tome koje sve informacije želimo izložiti klijentima i u kojem obliku. Ranije je spomenuto da je poželjno razmisliti o ograničenjima na određene akcije, no jednom kada ustanovimo koje resurse je klijentu dozvoljeno vidjeti a koje obrisati, bitno je i razmisliti o detaljima koje prikazujemo. Model podataka koji se pohranjuje u bazi ne mora biti isti kao onaj koji prikazujemo krajnjem korisniku. Budući da su slojevi odvojeni, klijent ne mora imati nikakvu informaciju o tome što točno pohranjujemo u bazu, zna samo ono što upravitelj šalje nakon što smo podatke dohvatili iz baze i zatim ih prilagodili. Npr. iako za pse imamo zapise o datumu rođenja, klijentu ćemo prikazati samo dob psa. Isto tako, nekada je bolje smanjiti opseg poslanih podataka zbog preglednosti i praktičnosti. Ako klijent zatraži listu svih skloništa, nema previše smisla poslati za svako sklonište i listu svih pasa koje zbrinjava te detalje o njima. Takav odgovor bi bio veoma nepregledan i klijent bi teško razaznao informacije vezane za sama skloništa i izgubio bi se u moru dodatnih podataka.

Sve navedeno potrebno je implementirati na jednostavan, razumljiv način i konzistentno za svaki resurs. Moguće je da klijent bude drugi razvojni inženjer koji povezuje svoj API s našim. Potrebno je olakšati snalaženje svakome tko bi mogao koristiti API, pogotovo omogućiti inženjeru da može odmah raspoznati o čemu se radi kada dobije poslužiteljski odgovor.

S gore navedenim na umu, prelazimo na detalje same implementacije. Na slici vidimo vrlo jednostavan kod, no ipak je potrebno objasniti nekoliko detalja.

```
[HttpGet(Name = "GetDogsForShelter")]
0 references
public ActionResult<IEnumerable<DogDto>> GetDogsForShelter(Guid shelterId)
{
    if (!_shelterRepository.ShelterExsists(shelterId))
    {
        return NotFound();
    }

    var dogsForShelterFromRepo = _shelterRepository.GetDogs(shelterId);
    return Ok(_mapper.Map<IEnumerable<DogDto>>(dogsForShelterFromRepo));
}
```

Slika 7.9: metoda za dohvat podataka

Prvi korak je provjeriti postoji li uopće traženi roditeljski resurs - sklonište. Ne moramo ga dohvaćati, samo nas zanima postojanje kako bismo mogli ili dohvatiti njegove podatke, ili vratiti odgovarajući status kod 404 koji klijentu daje na znanje da traženi resurs nije pronađen. Kada se uvjerimo da resurs postoji, dohvaćamo podatke. Nema potrebe najprije provjeravati postojanje, a zatim dohvatiti podatke. Štoviše, nije ni preporučljivo. Iako je vremenski razmak između te dvije radnje vrlo kratak, moguće je da se između te dvije operacije dogodi promjena. Recimo da želimo dohvatiti jedno sklonište. Ako provjerimo da resurs zaista postoji, zatim ga netko obriše drugim API pozivom, i nakon toga ga pokušamo dohvatiti - nakon što smo već zaključili da postoji, završiti ćemo s odgovorom koji vraća status kod ok i prazno tijelo odgovora, umjesto da pošaljemo 404 "Not found". Treba paziti, kada vraćamo cijelu kolekciju i ta kolekcija je prazna, tada vraćamo prazan odgovor i status kod ok. Tražena kolekcija je pronađena, međutim prazna.

Dolazimo do koraka koji je ranije spomenut, odlučiti koje informacije prikazujemo klijentu. Neovisno o tome hoće li se klijentov odgovor i zapis u bazi poklapati ili ne, uvijek je preporučljivo imati različite klase za različite upotrebe. U tom slučaju, ako dođe do promjene u strukturi tablice u bazi, ako preimenujemo ili dodamo stupac, to ne mora imati nikakav utjecaj na klijenta i na način na koji koristi aplikaciju. Uz ovakvu implementaciju puno je lakše održavati i nadograđivati kod, te je korištenje olakšano.

U ovom konkretnom slučaju, dohvaćeni podaci su u obliku različitom od onog koji želimo poslati klijentu: u bazi spremamo datum rođenja, a u odgovoru šaljemo dob. Prebacivanje iz jednog oblika u drugi (računanje dobi na temelju datuma rođenja) moguće je pisati ručno za svako pojedino svojstvo, međutim kada imamo entitet s trideset različitih svojstava, to postaje vrlo nepraktično i nepregledno. Ovo rješavamo korištenjem mapiranja kao što je prikazano na slici. Klasa Profile implementira metodu za jednostavno mapiranje. Za svojstva klasa koja se poklapaju nije potrebno ništa navoditi, već samo za svojstva koja se na neki način razlikuju definiramo njihov odnos i kako prebacujemo informacije iz

jednog oblika u drugi. U slučaju dodavanja novog svojstva za entitet, kako bismo ga mogli dodati u prikaz koji GET metoda dohvaća, nije potrebno dodavati to svojstvo u mapiranje (ako se poklapaju) kao što bi trebalo da smo ručno pisali odnose među svim svojstvima.

```
public class DogsProfile : Profile
{
    0 references
    public DogsProfile()
    {
        CreateMap<Entities.Dog, Models.DogDto>()
            .ForMember(
                dest => dest.Age,
                opt => opt.MapFrom(src => src.DateOfBirth.GetCurrentAge()));
        CreateMap<Models.DogForCreationDto, Entities.Dog>();
        CreateMap<Models.DogForUpdateDto, Entities.Dog>();
        CreateMap<Entities.Dog, Models.DogForUpdateDto>();
        CreateMap<Entities.Shelter, Models.ShelterFullDto>();
    }
}
```

Slika 7.10: mapiranje

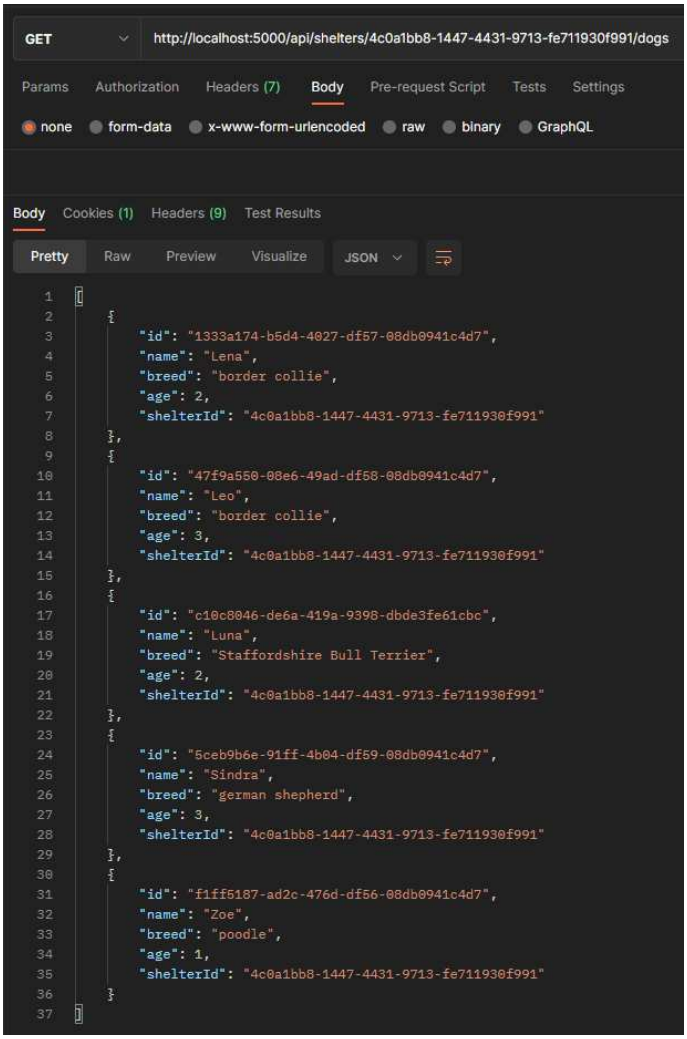
Dakle, nakon što dohvatimo podatke iz baze i mapiramo ih u prikladni oblik, možemo ih vratiti klijentu. U slučaju dohvaćanja skloništa situacija je još jednostavnija jer odmah dohvaćamo željeni resurs, mapiramo ga i vraćamo. Metoda Ok će se pobrinuti za to da se podaci prikažu u pravilnom formatu i da se vrati status kod 200 - ok. Za druge metode ćemo uvesti još neke status kodove, no za sada nam je ovo dovoljno.

Query string

Dohvat podataka moguće je proširiti omogućavanjem filtriranja i pretraživanja. Filtriranjem dohvaćamo samo podatke koji za zadano svojstvo imaju zadanu vrijednost. Pretraživanje radi na razini cijelog resursa, dohvaćaju se svi resursi čije proizvoljno svojstvo sadrži traženu vrijednost.

Da bi ovo funkcioniralo, potrebno je napraviti nekoliko izmjena u kodu. Za početak moramo nekako učitati vrijednosti zadane query stringom. To omogućavaju atributi. Metodi u upravitelju koja obrađuje GET zahtjeve ćemo dodati atribut FromQuery ispred parametara koje šaljemo. Da nije naglašeno da se radi o query stringu, pokušalo bi se iščitati informacije iz početnog URI-ja što bi rezultiralo greškom.

Ovdje uvodimo novu klasu koja sadrži sve parametre koje učitavamo iz query stringa. Što time dobivamo? Sve vrijednosti koje se učitaju iz URI-ja moraju se dalje proslijediti metodi koja će izvršiti upit na bazu. Dakle, u ovom trenutku je potrebno adaptirati metodu i proslijediti joj dodatna dva parametra - jedan za filtriranje i jedan za pretraživanje. Recimo da u nekom trenutku odlučimo proširiti funkcionalnost aplikacije i uvesti straničenje. Tada ponovno treba mijenjati metodu u upravitelju te metodu za dohvat na bazi. Kod mora biti



Slika 7.11: GET metoda

```
[HttpGet(Name = "GetShelters")]
0 references
public IActionResult GetShelters([FromQuery] SheltersResourceParameters sheltersResourceParameters)
{
```

Slika 7.12: klasa za parametre query stringa

pisan na način da je otvoren za proširivanje, a zatvoren na modificiranje - ovo je jedan od SOLID principa [17].

Dakle, GetShelters parametre iz query stringa smještene u novu klasu proslijeđuje u metodu koja radi dohvat iz baze. Tamo će se dodatno uvrstiti vrijednosti query stringova u

```
public class SheltersResourceParameters
{
    5 references
    public string Name { get; set; }
    5 references
    public string SearchQuery { get; set; }
```

Slika 7.13: klasa za parametre query stringa

upit. Ovdje treba paziti na redoslijed izvršavanja naredbi budući da to uvelike može utjecati na performanse. Što se točno misli time? Ovdje imamo dvije opcije za implementaciju. Najprije filtrirati podatke po zadanom parametru pa ih tako filtrirane dohvatiti iz baze, ili najprije dohvatiti sve iz baze i zatim na dohvaćeno primijeniti filter. Prva opcija je puno učinkovitija. Nepotrebno dohvaćanje svih podataka i tek onda njihovo odbacivanje je nepotrebno i rezultira lošijim performansama. Na maloj količini podataka to nije osjetno, ali kada su pohranjeni milijuni zapisa, tada se radi o velikoj razlici.

Ovakav redoslijed izvođenja omogućen je korištenjem IQueryable kolekcija. IQueryable omogućuje takozvano odgođeno izvršenje (deffered execution). Upit možemo graditi u koracima, nadodavati parametre. Izvršavanje se događa tek kada na neki način zatražimo prolazak po svim elementima koje treba dohvatiti. To se može postići na način da upit pretvorimo npr. u listu ili da zatražimo broj elemenata u dohvaćenoj kolekciji. Vidi se u kodu niže kako se na upit - collections - nadodaju zahtjevi i svo to vrijeme sve što imamo je samo upit. Na kraju je samo potrebno pozvati ToList() čime će se izvršiti dohvat podataka, i imamo listu traženih objekata koju u takvom obliku možemo vratiti.

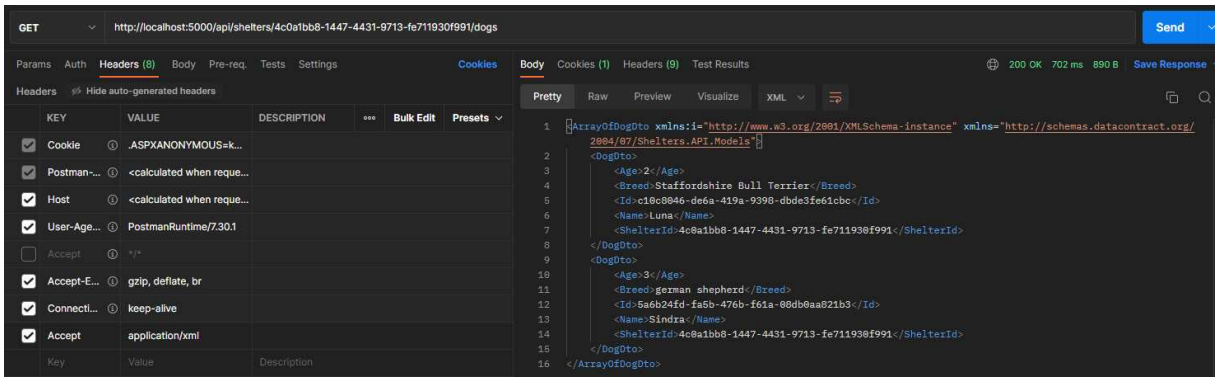
```
public PagedList<Shelter> GetShelters(SheltersResourceParameters sheltersResourceParameters)
{
    if (sheltersResourceParameters == null)
    {
        throw new ArgumentNullException(nameof(sheltersResourceParameters));
    }

    var collection = _context.Shelters as IQueryable<Shelter>;

    if (!string.IsNullOrWhiteSpace(sheltersResourceParameters.Name))
    {
        var location = sheltersResourceParameters.Name.Trim();
        collection = collection
            .Where(x => x.Name == location);
    }

    if (!string.IsNullOrWhiteSpace(sheltersResourceParameters.SearchQuery))
    {
        var searchQuery = sheltersResourceParameters.SearchQuery.Trim();
        collection = collection
            .Where(x => x.Address.Contains(searchQuery)
                || x.City.Contains(searchQuery)
                || x.Name.Contains(searchQuery));
    }
```

Slika 7.14: filtriranje i pretraživanje

7.4 Formatiranje

Prije početka implementacije treba razmisliti tko će sve koristiti naš API i s obzirom na to, koje formate bi trebalo omogućiti za slanje i primanje zahtjeva. "Content negotiation" je proces odabira najbolje reprezentacije za dani odgovor kada postoji više dostupnih opcija [1]. Ovo je dio HTTP protokola. Kako izabrati koje sve formate omogućiti? Ako se radi o proizvodu za jednog, specifičnog klijenta, tada je potencijalno dovoljno omogćiti jedan jedini format koji će odgovarati tom kupcu. Međutim, ako to nije slučaj, dobro je omogućiti više opcija. Format koji se najčešće koristi za REST je json i to je minimum koji je gotovo uvijek omogućen. Osim json-a, često se koristi i xml. Postoji još formata, ali nisu toliko popularni. Željeni format najbolje je zadati preko zaglavlja zahtjeva koji se šalje poslužitelju, u polju Accept. Moguće je koristiti i query stringove u te svrhe, ali taj pristup je bolje izbjegavati. Ne treba pretjerivati sa količinom stavki koje se šalju u query stringovima i općenito je uvaženo zadavanje formata preko zaglavlja.

U slučaju kada klijent u zaglavlju zatraži format koji nije podržan, aplikacije često vraćaju odgovor u nekom zadanom formatu, najčešće json. Ako ne napravimo u svom kodu ništa po pitanju biranja odgovarajućeg formata, točno tako će i raditi. Međutim nije preporučljivo vratiti odgovor u nekom tako zadanom formatu budući da to može uzrokovati problem za klijentsku stranu. Klijent može biti druga aplikacija i u ovakvom scenariju naprosto neće znati prevesti vraćene podatke. U takvoj situaciji treba vratiti prazan odgovor i status kod 406 (not acceptable) kako bi klijent znao da traženi format nije podržan.



Slika 7.15: GET odgovor u xml formatu

Osim što je moguće zatražiti odgovor u određenom formatu, štoviše, preporučljivo je da se ta vrijednost zaglavlja uvijek zadaje, također je moguće napisati kod na način da prihvaća tijelo zahtjeva u više različitih formata. Format koji se šalje također treba definirati u zaglavlju, u polju Content-type, kako bi se na temelju te informacije dalje u kodu podaci mogli parsirati na odgovarajući način.

U Startup klasi se mogu riješiti neke od ovih situacija. Startup klasa je automatski kreirana pri korištenju obrasca za MVC web API i u njoj možemo konfigurirati mnoge druge elemente potrebne za REST, kao što će biti još opisano kasnije. Prvo što ćemo definirati je da se pri slanju zahtjeva koji u polju Content-type ima nepodržani format, vrati odgovarajući status kod. Vrijednost ReturnHttpNotAcceptable je zadana na "false" i zbog toga se klijentu vraća json kada pokuša dobiti odgovor u drugom formatu. Kada vrijednost stavimo na true, šalje se prazno tijelo i 406 (not acceptable).

```
services.AddControllers(setupAction =>
{
    setupAction.ReturnHttpNotAcceptable = true;

}).AddXmlDataContractSerializerFormatters()
```

Slika 7.16: podržavanje "content negotiation-a" u Startup klasi

Ovdje se također može dodati podržavanje xml formata. Dovoljno je samo nadodati AddXmlDataContractSerializerFormatters. ASP.NET na temelju toga zna podržati xml format za podatke koje šalje klijentu i kada je potrebno vratiti taj format. Osim što je podržan xml format za odgovor, također je omogućeno slanje tijela u xml formatu. Dakle, sada je podržan unos resursa korištenjem POST-a s podacima zadanim u xml formatu. Postoje naravno načini i za dodavanje dodatnih tipova podataka, ali za sada se zaustavljemo na xml-u.

7.5 Dodavanje resursa i validacija

Već kod dohvaćanja resursa je spomenuto kako bi svaki model podataka trebao imati jednu funkciju. Kreirana je klasa DogDto koja se koristi za prikaz u GET metodi. Za kreiranje novih resursa ćemo koristiti novu klasu DogForCreationDto. Postoji nekoliko razloga zašto je to ovdje nužno.

```
[Required(ErrorMessage = "You should fill out a name.")]
[MaxLength(100, ErrorMessage = "The name should not have more than 100 characters.")]
1 reference
public string Name { get; set; }

[MaxLength(100, ErrorMessage = "The breed should not have more than 100 characters.")]
4 references
public virtual string Breed { get; set; }

[Required]
0 references
public DateTimeOffset DateOfBirth { get; set; }
```

Slika 7.17: POST model za kreiranje

Prikaz podataka GET metodom uključuje u ovom slučaju dob, a ne datum rođenja koji je pohranjen u bazu. Kod unošenja novih resursa moramo zadati datum rođenja, dob bi bila nepotpuna informacija i ne bismo to mogli točno mapirati u datum rođenja (kao što je to moguće napraviti u obrnutoj situaciji). Dakle, imamo različita svojstva. Svojstvo koje ovdje možemo izostaviti za svaki resurs je identifikator roditeljskog entiteta. Moramo ga svakako navesti u URI-ju i to je dovoljno da znamo kojem točno resursu pristupamo. Kada bismo identifikator slali i u tijelu zahtjeva, bilo bi potrebno provoditi dodatne provjere i odlučiti o prigodnim odgovorima u slučajevima kada informacije nisu usklađene ili neka od njih nedostaje.

Validacija

Drugi razlog za kreiranje nove klase je validacija. Početna klasa za pohranu podataka u bazi ima zadana ograničenja na određena svojstva - neka polja je nužno unijeti, neka imaju npr. ograničenu duljinu. Iste te zahtjeve moramo prenijeti na unos od strane korisnika. Spomenimo još kako za GET nije bilo potrebe za validacijom budući da podatke uzimamo direktno iz baze u koju su već pohranjeni na pravilan način. Međutim, klijent ne mora nužno znati prije korištenja koja su ograničenja i zahtjevi ili može naprosto napraviti grešku te je potrebno provjeriti svaki novi unos.

Vidimo na gornjoj slici kako su pojedinim svojstvima pridružene anotacije. Automatski će se vršiti provjere te vraćati odgovarajući status kod i poruka u slučaju pada validacije. Ovakve anotacije su dio ASP.NET-a i uvelike olakšavaju provjere te smanjuju količinu posla za pisanje pojedinih pravila i traženog ponašanja. Postoji jednostavan način i za kreiranje dodatnih provjera koji ćemo ovdje implementirati. Proširenjem ValidationAttribute koji je također dio ASP.NET-a i nadjačavanjem IsValid metode kreiramo pravila prilagođena poslovnim potrebama kao što je prikazano na slici. Proširena pravila možemo primjenjivati na razini klase ili pojedinog svojstva. Jednostavno kreiramo pravilo validacije i povratnu poruku u slučaju pada istog. Odgovarajući status kod 400 se šalje automatski. Ono što zapravo okida slanje poruke i koda je ApiController atribut iz upravitelja koji smo spomenuli na početku poglavlja. ErrorMessage koji se ispisuje zadajemo posebno na svakom korištenju ove konkretne validacije, kojih može biti proizvoljno mnogo, što je još jedna prednost pisanja ovakvih validacija.

Dodavanje

Pogledajmo sada implementaciju metode za dodavanje novog resursa. Kao i kod dohvaćanja podataka, razlika između metoda za "roditelja" i "dijete" je u tome što kod djeteta dodatno provjeravamo postoji li uopće roditelj. S mapiranjem smo se upoznali do sada. Najprije model za unos novog resursa mapiramo u onaj za unos u bazu. Zatim ga spre-

```
public class DogNameMustBeDifferentFromBreedAttribute : ValidationAttribute
{
    0 references
    protected override ValidationResult IsValid(object value, ValidationContext validationContext)
    {
        var dog = (DogForManipulationDto)validationContext.ObjectInstance;
        if (dog.Name == dog.Breed)
        {
            return new ValidationResult(ErrorMessage, new[] { nameof(DogForManipulationDto) });
        }
        return ValidationResult.Success;
    }
}
```

Slika 7.18: prilagođena validacija

```
[DogNameMustBeDifferentFromBreed(ErrorMessage = "Name must be different from breed.")]
```

Slika 7.19: prilagođena poruka o grešci

mamo u bazu. Nakon toga ga mapiramo u model za prikaz korisniku kako bismo ga mogli poslati korisniku u odgovoru. Preostaje još objasniti CreatedAtRoute. Želimo nakon kreiranja resursa vratiti status kod 201 (Created) koji će se poslati upravo korištenjem ove metode. Također, u zaglavlju odgovora će biti zapisana ruta odnosno URI za dohvaćanje upravo kreiranog resursa. Sada možemo objasniti davanje atributa Name pojedinim metodama, kao što smo dali i ovoj metodi. Na temelju tih atributa se kreiraju pripadni URI-ji koje šaljemo u zaglavlju i zato ovdje CreatedAtRoute prima naziv metode za dohvat.

```
[HttpPost(Name = "CreateDogForShelter")]
0 references
public ActionResult<DogDto> CreateDogForShelter(Guid shelterId, DogForCreationDto dog)
{
    if (!_shelterRepository.ShelterExsists(shelterId))
    {
        return NotFound();
    }

    var dogEntity = _mapper.Map<Dog>(dog);
    _shelterRepository.AddDog(shelterId, dogEntity);
    _shelterRepository.Save();

    var dogToReturn = _mapper.Map<DogDto>(dogEntity);
    return CreatedAtRoute("GetDogForShelter", new { shelterId, dogId = dogToReturn.Id }, dogToReturn);
}
```

Slika 7.20: POST metoda za kreiranje resursa

Potrebno je napomenuti još kako POST smije biti korišten samo nad kolekcijama. Ne smijemo slati zahtjeve za URI oblika "api/shelters/id". U slučaju da resurs sa zadanim identifikatorom ne postoji, trebalo bi vratiti 404 (not found). U slučaju da resurs postoji, javlja se konflikt. Poslužitelj bi trebao biti onaj koji kreira identifikatore.

7.6 Ažuriranje

Pogledajmo dalje kako ažurirati resurse. U ovom odlomku biti će opisane dvije metode: PUT i PATCH. Obije služe za ažuriranje, ali postoje velike razlike u načinu na koji funkcioniraju odnosno rezultatima koje postižu. PUT je predviđena za ažuriranje cijelog resursa tj. svih njegovih svojstava, dok je PATCH u upotrebi za promjenu određenih, odabranih svojstava. PUT radi na način da mijenja sva svojstva, čak i kada nisu zadani podaci za sve. Ono što je u zahtjevu će se promijeniti kako je traženo, dok će ostala svojstva poprimiti početne vrijednosti svojih tipova. Tako bi PUT trebao raditi, i potrebno je dobro razmisliti želimo li dopustiti takvu razinu destruktivnosti te zaista izvesti implementaciju da funckionira na taj način. Ako želimo da neka svojstva ostanu nepromijenjena, potrebno je u zahtjev unijeti već postojeće vrijednosti. S druge strane, PATCH mijenja samo zadana svojstva, dok vrijednosti ostalih, nenavedenih, ostaju netaknute. PATCH poprima sve veću popularnost u odnosu na PUT. Zamislimo da pohranjujemo resurse koji imaju po trideset svojstava i moramo ih mijenjati često, ali pojedinačno. Kako se postojeće vrijednosti nebi izgubile, potrebno je uložiti puno dodatnog posla.

Prvo što je potrebno napraviti je novi model predviđen samo za ažuriranje. Kao i kod kreiranja resursa, ovdje također ne želimo roditeljski identifikator kako bismo smanjili mogućnost greške i količinu posla. Za kreiranje i ažuriranje nam trebaju ista svojstva, čemu onda raditi novu, identičnu klasu? U ovom konkretnom trenutku zaista nije potrebno raditi više klasa, međutim to se u kasnijim fazama može promijeniti i puno je praktičnije u početku napisati klasu više i ostaviti više prostora za fleksibilnost uvođenja promjena. Scenariji poput poslovnih pravila koja dopuštaju kreiranje ali ne i mijenanje istog svojstva ili potreba za različitim validacijama su realne situacije koje se često događaju i najpametnije je u startu razdvajati modele kako bi kasnije promjene bile olakšane i manje utjecale na klijentovo korištenje. Moguće je npr. da tijekom registracije na nekoj web stranici moramo popuniti samo osnovne podatke, a tek na plaćanju se traže podaci o kartici koji su u tom trenutku i obavezni. Općenita programerska praksa je da svaka klasa treba imati samo jednu funkciju i ovdje je rađeno u skladu s time. Ono što se može napraviti da se izbjegne dupliciranje koda jest bazna klasa koja sadrži sva zajednička svojstva i anotacije i koja je apstraktna kako bi služila samo za nasljeđivanje.

Obije implementacije su slične. Provjera o postojanju roditeljskog resursa, mapiranje između modela - kao što je već implementirano za prethodne metode. Obije metode vraćaju 204 (no content). Razlika je u podacima koje zahtjevi šalju, točnije u njihovom formatu. Za PUT kreiramo tijelo zahtjeva na način kao i kod unosa novih resursa. PATCH je specifičan po tome što koristi json patch standard. Šalje se lista operacija koje želimo izvršiti. U upotrebi je šest različitih operacija: add, remove, replace, copy, move, test. Osim operacije, zadaju se imena svojstava na koja se odnosi i nova vrijednost, ako je potrebna za traženu operaciju. Add dodaje novu vrijednost, remove postavlja početnu vrijednost

tipa podataka, replace mijenja postojeću vrijednost novom, copy kopira postojeću vrijednost nekog drugog svojstva, move premješta vrijednost, a test provjerava podudaraju li se zadana svojstva.

Slika 7.21: zahtjev za dodavanje i kopiranje pomoću PATCH-a

Budući da PATCH prima ovako specifičan oblik, potrebno ga je na neki način prevesti da možemo pravilno napraviti promjene na resursima. Na temelju identifikatora iz URI-ja dohvaćamo resurs iz baze, te na njega primjenjujemo tzv. "patch document". Također je potrebno provesti validaciju nakon što primjenimo dokument na "dto" budući da to ne radi automatski. Sve je ovo omogućeno gotovim metodama u ASP.NET pa je implementacija zapravo prilično jednostavna.

Gornji kod prikazuje još nešto. Osim za ažuriranje, PATCH i PUT mogu se koristiti i za kreiranje, ako odlučimo da to ima smisla za naš proizvod. Kreiranje s ovim metodama naziva se "upserting" (umetanje). Da bi to bilo ostvarivo, mora se omogućiti da osim poslužitelja, klijent također može kreirati identifikatore. Kod kreiranje koritenjem PATCH se također radi apliciranje dokumenta i validacija. Nakon toga imamo postupak kao u POST. U slučaju kreiranja vraća se 201, kao i kod POST-a.

7.7 Brisanje resursa

Brisanje resursa prilično je jednostavno i nema potrebe za uvođenjem novih pojmova i alata kako bi se implementiralo. Općenito, možemo pričati o brisanju pojedinog resursa ili cijele kolekcije. U ovom primjeru omogućeno je brisanje samo pojedinog resursa. To je također preporučena praksa s obzirom na destruktivnost brisanja i potencijalni gubitak velike količine podataka.

Zahtjev za brisanjem sadrži samo metodu i URI s identifikatorom resursa koji želimo ukloniti, dok je tijelo prazno. Tijelo odogova je također prazno i vraća se 204 (no content), budući da je obrisan resurs i nemamo što vraćati osim statusa koji daje na znanje da je

```
[HttpPatch("{dogId}")]
0 references
public IActionResult PartiallyUpdateDogForShelter(Guid shelterId, Guid dogId, JsonPatchDocument<DogForUpdateDto> patchDocument)
{
    if (!_shelterRepository.ShelterExsists(shelterId))
    {
        return NotFound();
    }

    var dogForShelterFromRepo = _shelterRepository.GetDog(shelterId, dogId);
    if (dogForShelterFromRepo == null)
    {
        var dogDto = new DogForUpdateDto();
        patchDocument.ApplyTo(dogDto, ModelState);
        if (!TryValidateModel(dogDto))
        {
            return ValidationProblem(ModelState);
        }

        var dogToAdd = _mapper.Map<Dog>(dogDto);
        dogToAdd.Id = dogId;

        _shelterRepository.AddDog(shelterId, dogToAdd);
        _shelterRepository.Save();

        var dogToReturn = _mapper.Map<DogDto>(dogToAdd);
        return CreatedAtRoute("GetDogForShelter", new { shelterId, dogId = dogToReturn.Id }, dogToReturn);
    }

    var dogToPatch = _mapper.Map<DogForUpdateDto>(dogForShelterFromRepo);
    // add validation
    patchDocument.ApplyTo(dogToPatch, ModelState);

    if(!TryValidateModel(dogToPatch))
    {
        return ValidationProblem(ModelState);
    }

    _mapper.Map(dogToPatch, dogForShelterFromRepo);
    _shelterRepository.UpdateDog(dogForShelterFromRepo);
    _shelterRepository.Save();
    return NoContent();
```

Slika 7.22: PATCH metoda

```
[HttpDelete("{shelterId}", Name = "DeleteShelter")]
0 references
public ActionResult DeleteShelter(Guid shelterId)
{
    var shelterFromRepo = _shelterRepository.GetShelter(shelterId);
    if (shelterFromRepo == null)
    {
        return NotFound();
    }

    _shelterRepository.DeleteShelter(shelterFromRepo);
    _shelterRepository.Save();

    return NoContent();
}
```

Slika 7.23: brisanje resursa

operacija uspješno izvršena. Svako iduće izvršavanje nad identičnim, obrisanim resursom, mora rezultirati s 404 (not found).

Između skloništa i pasa postoji hijerarhija - svaki pas pripada određenom skloništu i svako sklonište sadrži svoju kolekciju pasa. Kreiranjem novog skloništa dodajemo mu i kolekciju njegovih pasa. U slučaju kada brišemo sklonište, brišemo i njegove kolekcije, jer one samostalno ne postoje. Zbog upravo navedenog, brisanju resursa koji sadrže kolekcije također treba pažljivo pristupati zbog potencijalno velikog gubitka informacija. Entity framework je ovdje praktičan jer sam brine o tome da brisanje pojedinog resursa automatski

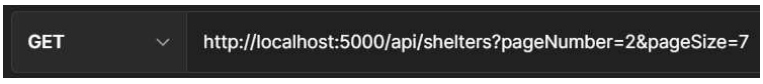
ukloni i sve njegove kolekcije.

U ovom trenutku je objašnjeno implementiranje osnovnih HTTP metoda. Iako postoje još mnoge, za potrebe ovog rada se zaustavljamo i prelazimo na rješavanje preostalih REST zahtjeva, odnosno principa.

7.8 Označavanje stranica

Takozvani "paging" odnosno označavanje stranica znači da ne prikazujemo sve postojeće instance određenog resursa odjednom već su razlomljene u više stranica. Postoji nekoliko razloga da se implementira takav prikaz. Ono što dobivamo označavanjem i prijelomom resursa u stranice su za početak bolje performanse. Ako se barata malom količinom podataka moguće je da razlika u brzini bude dovoljno mala da korisnik to ni ne primjeti. Međutim, ako u bazi postoje ogromne količine zapisa, što često zaista i jest slučaj, označavanje može biti itekako korisno i poboljšati korisničko iskustvo. Ako u Google upišemo pojam "restful", dobit ćemo odgovor u oko pola sekunde i prikazat će se prvih desetak izvora. Ono što također doznajmo je da ukupno postoji 90.800.000 izvora za traženi pojam. Što god da pretražujemo, vjerojatno ćemo otvoriti prvih nekoliko ponuđenih web stranica, i na iduću stranicu pretrage nikada nećemo ni stići. Da u ovom konkretnom slučaju nema označavanja stranica i ograničavanja veličine stranica, čekali bi jako dugo da se učitaju informacije koje na kraju ne dodaju na vrijednosti pretrage koja nas zanima.

Za svaki pojedini resurs u svakom pojedinom API-ju potrebno je prije početka implementacije procijeniti koje količine podataka bi mogle dospjeti u bazu te koliko instanci bi korisniku moglo biti potrebno po stranici. Na temelju toga je potrebno napraviti označavanje stranica, ali isto tako i dopustiti klijentu da navigira po stranicama, za što se preporuča korištenje query stringova.



Slika 7.24: označavaje stranica

Dakle, na način na koji se zadaju filtriranje i pretraživanje, možemo također zadati željenu stranicu i veličinu stranica. Sukladno tome, potrebno je proširiti klasu koja pohranjuje svojstva za filtriranje i pretraživanje. Upravo u ovakvom scenariju je puno praktičnije da postoji odvojena klasa za dijelove query stringa kojoj ćemo samo dodati još svojstava i nema potrebe za modifikacijama postojećeg koda, samo nadodajemo nove funkcionalnosti.

Zadaju se početne vrijednosti koje se koriste kada query string ne sadrži informacije o broju i veličini stranice. Početna stranica je prva stranica. Broj resursa prikazan na jednoj stranici je zadan na 5 i ograničen na 20. Zadajemo prvu stranicu i mali broj resursa po

```
public class SheltersResourceParameters
{
    5 references
    public string Name { get; set; }
    5 references
    public string SearchQuery { get; set; }

    const int maxPageSize = 20;
    4 references
    public int PageNumber { get; set; } = 1;
    private int _pageSize = 5;
    4 references
    public int PageSize
    {
        get => _pageSize;
        set => _pageSize = (value > maxPageSize) ? maxPageSize : value;
    }
```

Slika 7.25: proširivanje klase s vrijednostima query stringa

stranici kako nebi poslali prevelik broj informacija. Osim što to narušava performanse, također je jako nepregledno i nepraktično za korištenje.

Gornju granicu je bitno zadati kako klijent nebi zadavanjem prevelike vrijednosti negativno utjecao na performanse. Zamislimo da Google dopusti da se zatraži stranica s nekoliko milijuna rezultata. Poslužitelju bi trebale minute da obradi zahtjev i vrati odgovor i bilo bi jako otežano koristiti takvu aplikaciju. Klijent možda nije ni svjestan koju štetu može prouzročiti takvim korištenjem i zato se bitno ograditi po tom pitanju. Svaki novi unos se validira i uspoređuje s gornjom granicom koja će biti upotrebljena u slučaju da korisnik unese preveliku brojku.

Svaki resurs treba imati vlastitu klasu s ovim parametrima kako bismo mogli svaki prilagoditi poslovnim potrebama i svojstvima pojedinih resursa. Dobra je praksa implementirati straničenje ako ne uvijek, onda barem za slučajeve kada se dopušta kreiranje novih resursa i kada njihov broj može vidno narasti.

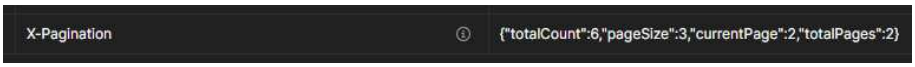
Označavanje stranica je nužno primjeniti nakon što se dohvate svi traženi podaci, isfiltrirani prije toga. Ako najprije podijelimo podatke na stranice i tek onda na dohvaćenu stranicu primijenimo npr. filtriranje to će rezultirati potpuno krivim podacima. Prvo filtriramo, dohvatimo sve što je traženo, i zatim to dijelimo na stranice. Spomenuto odgođeno izvođenje omogućava da dohvat podataka za određenu stranicu izvršimo točno traženim redoslijedom.

Osim same primjene podjele podataka na stranice, potrebno je uz to klijentu omogućiti kretanje kroz stranice. Svaki poslužiteljski odgovor mora sadržavati ukupan broj vraćenih podataka, ukupan broj stranica, trenutnu stranicu. Ove informacije spadaju pod takozvane metapodatke. Također mora biti omogućeno preći na prethodnu i iduću stranicu, ako postoje. To nisu po definiciji metapodaci i vraćamo ove dvije skupine informacija na različite načine. Brojke će biti smještene u zaglavlje odgovora. Na slici ispod je pokazano kako jednostavno dodati u zaglavlje odgovora novo polje i zadati mu naziv i napuniti sadržajem. Linkovi na prethodnu i / ili iduću stranicu će se slati u tijelu odgovora, na dnu svake stranice. Iako ih na temelju poslanog URI-ja klijent može i sam kreirati, ipak je preciz-

nije poslati odgovarajuće URI-je. Ovo je jedan od REST principa i iako neki tvrde da je ispunjen i bez dodavanja linkova, ovo je najispravniji pristup. Dodavanje informacija od prethodnoj i sljedećoj stranici je implementirano i argumentirano u idućem odlomku.

```
var sheltersFromRepo = _shelterRepository.GetShelters(sheltersResourceParameters);

var paginationMetaData = new
{
    totalCount = sheltersFromRepo.TotalCount,
    pageSize = sheltersFromRepo.PageSize,
    currentPage = sheltersFromRepo.CurrentPage,
    totalPages = sheltersFromRepo.TotalPages
};
Response.Headers.Add("X-Pagination", JsonSerializer.Serialize(paginationMetaData));
```

Slika 7.26: kreiranje metapodataka



Slika 7.27: zaglavlje odgovora s označavanjem stranica

7.9 HATEOAS

HATEOAS je akronim za "hypermedia as the engine of application state". U prijevodu, hipermedija kao pokretač stanja aplikacije. Ovo je jedan od kompleksnijih REST principa koji se često izostavlja, iako uvelike olašava klijentu tako što opisuje mogućnosti i načine korištenja. Daje klijentu na znanje koje sve akcije su moguće u određenom trenutku i kako ih izvršiti. To postižemo dodavanjem linkova u odgovore klijentu. Za svaki resurs koji dohvatimo dodane su informacije o tome koje metode su dozvoljene te pomoću kojih URI-ja. Poveznice za prethodnu i iduću stranicu također spadaju pod ovaj princip. Implementacija ovog principa je zapravo dio HTTP protokola.

Uz ovakvu implementaciju, klijent ne mora unaprijed znati koje sve mogućnosti mu pruža aplikacija koju koristi već u hodu, tokom samog korištenja, otkriva koje su mogućnosti implementirane. Zamislimo da prije korištenja svake web stranice najprije moramo detaljno proučiti priručnik za korištenje. Zamislimo sada da izađe nova verzija i da moramo nanovo čitati priručnike i dokumentacije kako bi doznali što je ostalo isto, što je uklonjeno, te koje su nove opcije dostpne. Zvuči jako nepraktično, u odnosu na to da jednostavno otvorimo stranicu koja nas interesira i po njoj se krećemo pomoću poveznica koje su ponuđene.

Linkovi koji se u ovu svrhu konstruiraju sastoje se od tri komponente: method, rel, href. Method je HTTP metoda, rel je opisno polje koje kratko opisuje akciju, href je URI.

Ove linkove je potrebno uključiti na razini pojedinog resursa, te na razini cijele kolekcije. Za pojedine resurse navodimo linkove koji nas usmjeravaju na sve dozvoljene radnje s resursom - dohvaćanje, brisanje, kreiranje, ovisno što je implementirano za pojedini resurs. Linkovi na razini kolekcije uključuju prelaske na iduću i prethodnu stranicu te na trenutnu. Oni također utječu na stanje aplikacije, a ovi linkovi upravo tome i služe - kontroliranje stanja aplikacije. To je razlog zašto podatke o prethodnoj i sljedećoj stranici ne treba držati u zaglavlju zajedno s metapodacima. Ovo je potrebno uključiti kako kod dohvata, tako i kod kreiranja resursa.

Kod kreiranja URI-ja za prethodnu i iduću stranicu treba uzeti u obzir ne samo traženi resurs već i sve parametre query stringa. Novi URI mora sadržavati identične parametre, s razlikom u oznaci stranice.

Ovdje se uvodi početna, "root" stranica, koja predstavlja početnu stranicu aplikacije od koje dalje navigiramo. U kodu niže je prikazano kreiranje linkova korištenjem metode Link - ovo je još jedan alat koji je ugrađen u ASP.NET.

```
[Route("api")]
[ApiController]
0 references
public class RootController : ControllerBase
{
    [HttpGet(Name = "GetRoot")]
    0 references
    public IActionResult GetRoot()
    {
        var links = new List<LinkDto>
        {
            new LinkDto(Url.Link("GetRoot", new { }), "self", "GET"),
            new LinkDto(Url.Link("GetShelters", new { }), "shelters", "GET"),
            new LinkDto(Url.Link("CreateShelter", new { }), "create_shelter", "POST")
        };
        return Ok(links);
    }

}
```

Slika 7.28: upravitelj za početnu stranicu

Dodavanje linkova dovodi do sljedećeg problema: gubi se json format. Poslužiteljski odgovor je sada u omotnici koja se sastoji od liste resursa i liste linkova. Time je prekršen princip o samoopisnim porukama. Klijent može imati problema s parsiranjem ovakvog odgovora, pogotovo ako se radi o drugoj aplikaciji. Tome se može doskočiti na način da se uvede novi format koji će sadržavati vrijednosti i linkove, dok će za pravi json odgovor biti poslan bez dodatnih informacija tj. linkova. Uvođenje takozvanih "vendor specific" tipova podataka ima još prednosti. Možemo uvesti više tipova koji će vraćati različite reprezentacije u smislu prikaza samo dijela svojstava. Recimo da aplikacija koju radimo ima trideset svojstava na jednom resursu. Aplikacija se koristi prema više drugih aplikacija i svaka od njih treba samo dio informacija. Jednoj su možda potrebna samo dva svojstva postojećeg entiteta, a drugoj treba sve. Nema smisla da obije aplikacije primaju svih trideset budući da smanjena količina informacija koja se šalje može imati pozitivan učinak

```
"value": [
    {
        "id": "0a2247d6-627f-491f-9553-770f3c581178",
        "name": "Dumovec",
        "location": "Franjčevićeva 43, 10361 Dumovec",
        "links": [
            {
                "href": "http://localhost:5000/api/shelters/0a2247d6-627f-491f-9553-770f3c581178",
                "rel": "self",
                "method": "GET"
            },
            {
                "href": "http://localhost:5000/api/shelters/0a2247d6-627f-491f-9553-770f3c581178",
                "rel": "delete_shelter",
                "method": "DELETE"
            },
            {
                "href": "http://localhost:5000/api/shelters/0a2247d6-627f-491f-9553-770f3c581178/dogs",
                "rel": "create_dog_for_shelter",
                "method": "POST"
            },
            {
                "href": "http://localhost:5000/api/shelters/0a2247d6-627f-491f-9553-770f3c581178/dogs",
                "rel": "dogs",
                "method": "GET"
            }
        ]
    },
```


Slika 7.29: odgovor za GET za skloništa

na performanse. U tom slučaju se uvede tip koji prikazuje samo ta dva konkretna svojstva. Svaki od navedenih klijenata će slati zahtjeve s različitim vrijednostima polja Accept u zaglavlju i dobivati različite reprezentacije. Također, možda jedan klijent treba samo dob, a drugi cijeli datum rođenja. Još jedan razlog može biti promjena nekog entiteta, uvođenje novih svojstava. Recimo, želimo dodati entitetu psa datum udomljenja. Ne želimo "potrgati" postojeću funkcionalnost, već ju proširiti, pa se kreira novi tip koji će uključivati dodatna svojstva na resursima. Kreiramo novi model koji je identičan početnom osim što sadrži novo svojstvo koje nije nužno unositi. Ovisno o tome koji je tip poslan u zaglavlju, pozivamo različite metode koje primaju različite dto klase i pohranjuju dane podatke. Poslovna pravila i odluke mogu jako varirati iz velikog broja razloga i "vendor specific media types" uvelike pomažu oko toga te su sve više u upotrebi.

7.10 Caching

Caching odnsno predmemoriranje istovremeno je vrlo moćan i opasan alat. Ako ga se pravilno implementira, može uvelike utjecati na performanse, ali ako ga se ne napravi dobro tada može dovesti do pogrešnog rada aplikacije i baratanja pogrešnim, promijenjenim, nepostojećim podacima i dovoditi do grešaka. Predmemoriranje ima vrlo čestu upotrebu. Ovo nije dio HTTP protokola, a nije ni obavezni dio REST-a. Fielding, začetnik REST-

a, u svojoj disertaciji piše: "Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests." [4] Dakle, nije nužno implementirati cache, već samo naznačiti u odgovorima može li se koristiti cache ili ne. Iako nije obavezan zahtjev, vrlo je koristan i posvetit ćemo se ovoj temi. Treba još napomenuti da je ovo korisno samo u situacijama kada se učestalo šalju velike količine podataka, u suprotnom ili se na brzini neće primjetiti rad predmemoriranja, ili će naprosto svaki puta neiskorišteno isticati zbog prevelikih vremenskih razmaka u zahtjevima. Možemo reći da je cache "middle man of request response communication"[21].

Postoje brojne opcije za uvrštavanje cache-a, i svaka ima svoje prednosti i mane. Ovdje ulazimo u detalje takozvanog HTTP cache-a koji se smatra vrlo učinkovitim. Općenito, cache može smanjti kako broj zahtjeva koji se šalju, tako i broj odgovora.

Cache čini sponu između klijenta i poslužitelja, pohranjuje dio informacija koje razmjenjuju i time smanjuje potrebu za razmjenom zahtjeva i odgovora od klijenta do poslužitelja i obrnuto. Klijent šalje zahtjev, cache ga prosljeđuje poslužitelju, prima i pohranjuje odgovor sa poslužitelja, šalje ga dalje klijentu. Kada klijent ponovno pošalje isti zahtjev, cache može vratiti odgovor koji je već pohranio, bez da ponovno kontaktira poslužitelja, ili u nekim slučajevima može kontaktirati poslužitelja kako bi provjerio jesu li pohranjene informacije još uvijek validne i ako poslužitelj odgovori da jesu, cache vraća klijentu odgovor koji ima od ranije.

Postoje dva tipa cache-a: privatni i dijeljeni. Privatni cache je klijentski cache koji se pohranjuje za svakog klijenta posebno. Dijeljeni cache se dijeli među klijentima. Kada jedan klijent pošalje zahtjev i cache ga pohrani, može ga proslijediti svakom drugom klijentu koji šalje isti zahtjev. Kod privatnog se to ne smije događati i jedan od primjera privatnog cache-a bi bio onaj na mobilnim aplikacijama. Dijeljeni cache dijelimo na gateway i proxy. Gateway cache živi na poslužitelju, dok je proxy na mreži. Obije vrste dijeljenog cache se koriste npr. za aplikacije koje su korištene od velikog broja korporacijskih zaposlenika. Moguće je međusobno kombinirati sve tri vrste.

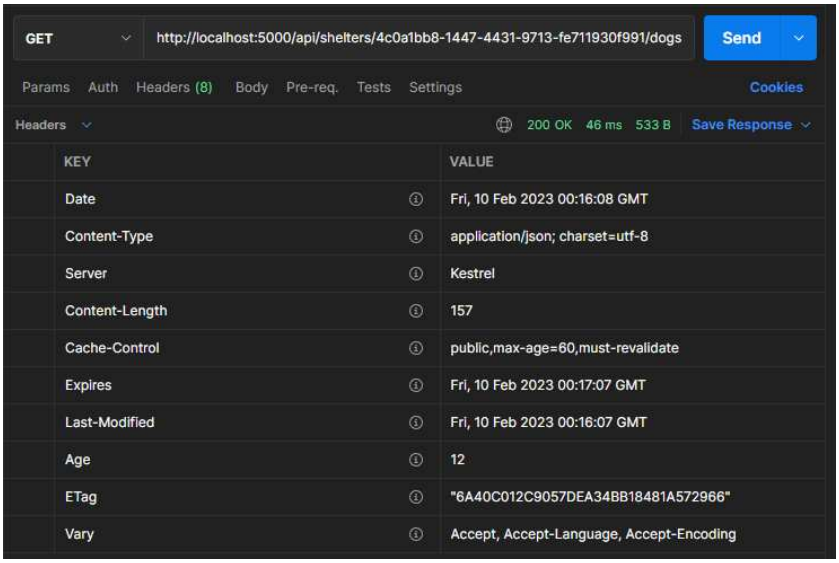
Da bi se ispunio REST princip o predmemoriranju, potrebno je na neki način u odgovoru klijentu dati na znanje može li se informacija cacheirati ili ne. To se radi kroz zaglavlje. U zaglavlju možemo npr. zadati koliko dugo cache pohranjuje odgovor ili koliko je isteklo od kada je odgovor pohranjen u cache. Potrebno je odgovore negdje zaista i pohraniti. Taj dio se implementira u Startup klasu dodavanjem takozvanog "cache store-a".

Pravila predmemoriranja se mogu zadati na razini metode, upravitelja ili cijele aplikacije. Ovo zadnje također radimo u Startup klasi. Na slici niže je pokazano kako zadati aplikacijski cache koji traje 240 sekundi i primjenjuje se na sve resurse. Ovo može biti nadjačano pravilima zadanim na razini upravitelja ili metode, što se postiže korištenjem atributa kojima se može definirati veliki broj pravila za cache.

```
services.AddResponseCaching();

services.AddControllers(setupAction =>
{
    setupAction.ReturnHttpNotAcceptable = true;
    setupAction.CacheProfiles.Add(
        "240SecondsCacheProfile",
        new CacheProfile()
        {
            Duration = 240
        });
```

Slika 7.30: aplikacijski cache

Dalje slijede kompliciraniji, ali i efikasniji načini za korištenje predmemoriranja. Spomenimo dva modela: model isteka i model validacije. Model isteka je jednostavniji i njega smo zapravo već uvrstili u kod: odgovor se iščitava iz cache-a dok god mu ne istekne zadani "rok trajanja". Nakon toga se odgovor ponovno učitava s poslužitelja i pohranjuje u cache. Validacijski model provjerava koliko je pohranjeni odgovor svjež. U ovom modelu server je kontaktiran svaki puta, a ne tek po isteku, ali da provjeri je li došlo do promjene u traženom resursu. Ako nije, server šalje kod 303 (not modified) i cache proslijeđuje odgovor koji već ima prema klijentu. Ako je pak resurs promijenjen, poslužitelj šalje novi odgovor. U zaglavlju odgovora su vidljive informacije kao što su last-modified i ETag. Na temelju tih informacija cache store provjerava s poslužiteljem je li došlo do promjene - provjerava se jesu li ti podaci i dalje isti. Time je smanjen protok informacija i odlazak na bazu koji je najskuplja operacija. Ova dva modela moguće je i kombinirati da imamo funkcionalnost takvu da se tek po isteku vremena provjerava je li cache još uvijek dovoljno svjež. Dakle, i nakon isteka vremena potencijalno još ne uzimamo nove podatke od poslužitelja.



Slika 7.31: zaglavlje odgovora s cache informacijama

7.11 Sigurnost

Kako odlučiti treba li uložiti vrijeme u sigurnost tijekom izrade aplikacije? Korištenje privatnih podataka, slanje osjetljivih podataka, korištenje korisničkih imena i lozinki, neki su od kriterija za odabir da se posveti ovom segmentu.

Kada pričamo o sigurnosti, ne misli se nužno samo na prijave s korisničkim podacima. Za početak donosimo odluku o tome hoće li API biti privatan ili javan. Ako je u pitanju proizvod za unutarnju upotrebu tvrtke i nitko izvan internog poslužitelja ne bi trebao imati pristup, tada je potrebno držati ga privatnim.

Spomenimo dalje autentifikaciju: postupak kojim se provjeravaju nečiji podaci za pristup odnosno doznajemo tko je korisnik koji pristupa API-ju ili određenom dijelu API-ja. To se provodi korištenjem korisničkih imena i lozinki i povećava sigurnost API-ja. Sigurnost dodatno varira o načinu na koji se provodi spomenuta provjera.

Autorizacija je postupak kojim se utvrđuju ovlaštenja autentificiranog korisnika, odnosno doznajemo što sve korisnik smije raditi. Možda neki web servis recimo dopušta svim korisnicima dodavanje resursa (korištenje POST metode), ali pravo na brisanje resursa (korištenje DELETE metode) ima samo admin. To za implementaciju znači da pri obradi svakog zahtjeva potencijalno treba provjeriti i podatke o korisniku koji želi obaviti određenu radnju, osim same provjere smije li se općenito ta radnja primjeniti na dane resurse.

Autentikaciju možemo jednostavno kreirati korištenjem takozvane bazične ili npr. "di-

gest" autentikacije. Obije metode koriste razmjenu informacija za autentikaciju kroz zaglavlje. U slučaju kada se pokuša pristupiti sadržajima zaštićenim na ovaj način, bez da se pošalju traženi podaci, potrebno je umjesto traženih informacija s odgovarajućim status kodom poslati 401 koji daje na znanje klijentu da je autentikacija potrebna. Kod implementacije je potrebno pripaziti i na pravilnu upotrebu predmemoriranja kako ne bismo podijelili zaštićene podatke s ostalim klijentima.

Vrijedno je još spomenuti OAuth. Radi se o protokolu koji se koristi za autorizaciju. Točnije, tronožni OAuth. Odakle takav naziv? Tronožni iz razloga što se koristi treća strana za pohranu povjerljivih podataka, umjesto da ih samostalno pohranjujemo u vlastitu bazu podataka. Jedan od primjera bi bio logiranje u aplikacije pomoću Gmail-a. U takvim situacijama svoje podatke ne šaljemo aplikaciji koju koristimo već samo Gmail-u koji je u ovom slučaju treća strana od povjerenja. Na taj način skidamo dio odgovornosti i rizika sa sebe, a korištenje ovakvog protokola smatra se vrlo sigurnim.

U sklopu praktičnog rada nećemo ulaziti u detalje implementiranja sigurnosti, ali to je svakako segment aplikacije o kojem je potrebno razmisliti i ponekad ga uključiti. Mogu nastati negativne i skupe posljedice ako ne implmentiramo neku vrstu sigurnosti za određene vrste proizvoda. S druge strane, ako procijenimo da ne postoji potreba za time, onda je gubitak vremena i novaca upuštati se u izradu nepotrebne funkcionalnosti. Potrebno je pri odabiru optimalne opcije voditi računa o usklađenosti s HTTP protokolom i naravno REST principima.

Zaključak

U ovom radu predstavljen je REST, njegova primjena, pravila, način implementacije. Pokazane su pozitivne i negativne strane, objašnjene arhitekture u koje se uklapa i za koje nije toliko pogodan. REST nije nužno ni dobar ni loš, naprosto za svaki novi razvoj softvera treba pojedinačno procijeniti što je najbolje primijeniti. Također, nema smisla uvijek forsirati potpunu implementaciju REST-a samo da bi mogli reći da smo ga implementirali. Ponekad je možda REST najbolja opcija, ali i dalje nije idealna. Dobar je kao skup smjernica, ali ponekad treba biti pragmatičan. Koristiti principe koji doprinose potrebama koje je potrebno zadovoljiti razvojem, i odbaciti ideje koje će donijeti više štete nego koristi. Svaka tvrtka, svaki tim, svaki proizvod treba pažljivo isplanirati i odlučiti što im točno najbolje odgovara i kako to postići.

Bibliografija

[1] Howard Dierking, *REST Fundamentals*, https://app.pluralsight.com/course-player?clipId=3e723248-a651-47ed-a0ec-083ad32185d0, 2020.

[2] Anastasia D. Dmytro H., *Best Architecture for an MVP: Monolith, SOA, Microservices, or Serverless?*, https://rubygarage.org/blog/monolith-soa-microservices-serverless, 2019.

[3] Kevin Dockx, *Building a RESTful API with ASP.NET Core 3*, https://app.pluralsight.com/library/courses/asp-dot-net-core-3-restful-api-building/table-of-contents, 2019.

[4] Roy Thomas Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, Disertacija, University of California, 2000.

[5] Martin Fowler, *Richardson Maturity Model*, https://martinfowler.com/articles/richardsonMaturityModel.html, 2010.

[6] Tadit Dash Gaurav Aroraa, *Building RESTful Web Services with .NET Core*, Packt Publishing Ltd., Birmingham, 2018.

[7] GeeksforGeeks, *Difference Between Architectural Style, Architectural Patterns and Design Patterns*, https://www.geeksforgeeks.org/difference-between-architectural-style-architectural-patterns-and-design-pa 2022.

[8] Alexander S. Gillis, *SOAP (Simple Object Access Protocol)*, https://www.techtarget.com/searchapparchitecture/definition/SOAP-Simple-Object-Access-Protocol, 2022.

[9] Lokesh Gupta, *HATEOAS Driven REST APIs*, https://restfulapi.net/hateoas/, 2021.

[10] Aqib Javed, *Monolithic vs. SOA vs. Microservices*, https://www.linkedin.com/pulse/monolithic-vs-soa-microservices-aqib-javed-/, 2019.

[11] Jose, *API calls and HTTP Status codes*, https://itnext.io/api-calls-and-http-status-codes-e0240f78f585, 2019.

[12] Jamie Juviler, *Web Service vs. API, Explained*, https://blog.hubspot.com/website/web-services-vs-api, 2022.

[13] Kanbanize, *What Are the 12 Principles of Agile Project Management?*, https://kanbanize.com/agile/project-management/principles.

[14] Guy Levin, *4 Maturity Levels of REST API Design*, https://blog.restcase.com/4-maturity-levels-of-rest-api-design/, 2018.

[15] Lucidspark, *Agile methodology: What it is, how it works, and why it matters*, https://lucidspark.com/blog/what-is-agile-methodology.

[16] Robert Manger, *Softversko inženjerstvo*, Element, Zagreb, 2016.

[17] Samuel Oloruntoba, *SOLID: The First 5 Principles of Object Oriented Design*, https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design, 2020.

[18] Jakub Ryba, *Waterfall vs. Agile: What Is the Best Approach For a Software Development Project?*, https://www.easyredmine.com/news/waterfall-vs-agile-what-is-the-best-approach-for-a-software-development-pr 2021.

[19] Alyssa Walker, *SOAP vs REST API: Difference Between Web Services*, https://www.guru99.com/comparison-between-web-services.html, 2022.

[20] Jason Westland, *Top 10 Project Management Methodologies: An Overview*, https://www.projectmanager.com/blog/project-management-methodology, 2021.

[21] Shawn Wildermuth, *Designing RESTful Web APIs*, https://app.pluralsight.com/library/courses/designing-restful-web-apis/table-of-contents, 2019.

Sažetak

Ovaj diplomski rad se bavi aplikacijama koje koriste RESTFul web servise. Najprije se opisuje u kakvim uvjetima su takve aplikacije poželjne i kakva aplikacijska arhitektura je pogodna. Upoznajemo se s alternativnim opcijama kako bi dobili bolji uvid u to kada je pogodno raditi ovakvu vrstu aplikacije, a kada je pogodnije odlučiti se za drugačije opcije. Zadan je uvod u HTTP koji ima veliku ulogu u izgradnji RESTFul web servisa. Dalje napokon ulazimo u detalje toga što REST zapravo znači i uspoređujemo ga sa SOAP-om. Zatim ulazimo u praktični dio rada, u tehnologije koje se koriste i alate koji to omogućavaju. Opisane su smjernice koje je potrebno pratiti u dizajniranju i načini na koje ih implementirati.

Summary

This paper is about applications which use RESTFul web services. First, we describe the appropriate conditions for applications using RESTFul web services that and the suitable architecture. Alternative options are also described to get a better sense of when this kind of application should be made, and when it is better to use other options. The introduction to HTTP is given because it is a big part of building REST web services. After that, we finally get into details of what REST really means and it is compared to SOAP. Then we get into the practical part, the technologies which are used are tools helping with it. Guidelines are given which need to be followed in designing and implementing the code.

Životopis

Rođena sam 18. rujna 1994. u Zagrebu. Upisala sam osnovnu školu Matije Gupca u Zagrebu. Po završetku petog razreda selim u Sv. Ivana Zelinu gdje završavam osnovnu školu te krećem u srednju. Završila sam opću gimnaziju u Srednjoj školi Dragutina Stražimira. 2013. godine upisujem studij matematike na Prirodoslovno-matematičkom fakultetu, te na istom fakultetu 2016. upisujem studij Računarstvo i matematika. S početkom 2022. započinjem raditi u Lemaxu kao junior software developer.