Izrada web aplikacije visokih performansi koristeći Symfony i Pimcore

Picić, Marta Marija

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

VARAŽDIN

Marta Marija Picić

IZRADA WEB APLIKACIJE VISOKIH PERFORMANSI KORISTEĆI SYMFONY I PIMCORE

DIPLOMSKI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Marta Marija Picić

Matični broj: 0016142854

Studij: Informacijsko i programsko inženjerstvo

IZRADA WEB APLIKACIJE VISOKIH PERFORMANSI KORISTEĆI SYMFONY I PIMCORE

DIPLOMSKI RAD

Mentor:

doc. dr. sc. Matija Novak

Varaždin, rujan 2024.

Marta Marija Picić

Izjava o izvornosti

Izjavljujem da je ovaj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autorica potvrdila prihvaćanjem odredbi u sustavu FOI Radovi

Sažetak

Tema ovog rada je izrada aplikacije uz pomoć aplikacijskog okvira Symfony i platforme Pimcore s naglaskom na primjenu tehnika za poboljšanje performansi. Cilj rada je bio provesti mjerenja performansi nad izrađenom aplikacijom kako bi se procijenila učinkovitost odabranih tehnika poboljšanja performansi. Kao alat za mjerenje performansi korišten je Apache JMeter, a testirane su samo tehnike poboljšanja performansi na strani poslužitelja, iako su u radu obrađene i tehnike za poboljšanje performansi na strani klijenta. Rezultati su pokazali da pravilna primjena indeksa može značajno poboljšati performanse aplikacije, dok izvođenje zadataka u pozadini također donosi značajna poboljšanja u situacijama kada je potrebna komunikacija s vanjskim servisima. Iako su slabiji rezultati postignuti primjenom međuspremnika i optimizacijom upita, njihova primjena ne bi trebala biti zanemarena tijekom razvoja aplikacije. Zaključak je da primjena optimizacijskih tehnika može značajno ubrzati rad aplikacije i poboljšati korisničko iskustvo, posebno prilikom izrade složenijih poslovnih aplikacija.

Ključne riječi: Pimcore; Symfony; Apache JMeter; aplikacijski okvir; web aplikacija; performanse;

Sadržaj

1. Uvod

U doba sve veće digitalizacije, web aplikacije kojima svakodnevno pristupamo preko računala i mobilnih uređaja, postaju primarno sredstvo za pretraživanje informacija i multimedijalnih sadržaja. Osim prezentacije podataka, ključna je i brzina kojom oni stižu do korisnika. Sporo učitavanje stranice negativno utječe na korisničko iskustvo i samim time narušava posjećenost web aplikacije. Kod aplikacija gdje je profit izravno povezan s posjećenošću, pad prometa se negativno odražava i na zadovoljstvo klijenata. Takve se situacije mogu izbjeći pridržavanjem određenih pravila o optimizaciji i upravo su ta pravila bit ovog rada.

U sljedećem poglavlju je dan uvod u web aplikacije i osnovne pojmove vezane uz web. Nakon toga slijedi uvod u aplikacijske okvire, specifično Symfony te platformu Pimcore, jer su upravo te tehnologije korištene prilikom izrade aplikacije u praktičnom dijelu. S obradom aplikacijskih okvira, završava upoznavanje s osnovnim pojmovima i tehnologijama i započinje obrada pravila o optimizaciji performansi. Pravila su podijeljena u dvije skupine: optimizacija na strani servera i optimizacija na strani klijenta.

Nakon obrade pravila, započinje praktični dio rada čiji su glavni ciljevi izrada jedne web aplikacije prateći pravila optimizacije i testiranje njezinih performansi uz pomoć programa Apache JMeter. Kod jednog dijela funkcionalnosti namjerno će se izostaviti pravila optimizacije kako bi se analizirao postotak ubrzanja. Na samom kraju rada se nalazi pregled literature i priloga te popisi slika i tablice.

2. Web aplikacije

Prije davanja definicije web aplikacije, potrebno je upoznati sami pojam web. World Wide Web, poznat kao WWW ili W3, predstavlja sustav međusobno povezanih dokumenata kojima se može pristupiti putem interneta [1].

Potrebu za takvim sustavom je primijetio Tim Berners-Lee radeći kao znanstvenik u Europskom vijeću za nuklearna istraživanja (franc. *Conseil européen pour la recherche nucléaire* - CERN). Tijekom 80-ih godina prošlog stoljeća, na velikim projektima unutar CERN-a dolazi do problema pohrane i dijeljenja informacija. Kao moguće rješenje tog problema, Berners-Lee 1989. predlaže projekt pod nazivom Information Management: A Proposal. Upravo je unutar tog prijedloga [2] Berners-Lee opisao neke od glavnih koncepata weba kao što su hipertekst, linkovi, web preglednik i web poslužitelj.

Prema [3], web je od svog nastanka pa do danas prošao kroz nekoliko etapa razvoja. Prva faza se naziva Web 1.0 i njezino glavno obilježje su statične stranice. Korisnici su samo mogli pretraživati i pristupati informacijama, ali to se ubrzo mijenja s dolaskom nove generacije weba. Termin Web 2.0 je prvi put službeno upotrijebio Dale Dougherty 2004. godine. Web 2.0 omogućava da korisnik osim čitanja ima i mogućnost kreiranja sadržaja. Također, dolazi do razvoja web aplikacija korištenjem tehnologija poput AJAX-a (eng. *Asynchronous JavaScript and XML*) i Google Web Toolkita. Kroz Web 3.0, poznat kao semantički web, se nastoji omogućiti da sadržaj na webu, osim ljudima, bude čitljiv i računalnim sustavima. S time se želi postići veća učinkovitost pri automatizaciji i integraciji podataka. Nova generacija Web 4.0 dodatno proširuje cilj povećanja učinkovitosti predlaganjem korištenja umjetne inteligencije, a osim toga, najavljuje i simbiozu između ljudskog uma i računalnih sustava, zbog čega se Web 4.0 naziva i simbiotskim webom.

Kao što je već rečeno, s razvojem Web 2.0 dolazi i do razvoja web aplikacija. Prema [4], web aplikacije su računalni programi koji su pohranjeni na web poslužitelju, a pristupa im se koristeći web preglednike. Kako se program nalazi na poslužitelju, korisnik ne treba izvršiti nikakve instalacijske procedure kako bi mogao koristiti aplikaciju. Web aplikacije omogućuju pristup većem broju korisnika istovremeno, a samo neki od primjera web aplikacija su društvene mreže, online trgovine i alati za uređivanje slika. Jedan od nedostataka web aplikacija je njihova ovisnost o internetskoj vezi. Ako je veza loše kvalitete, aplikacija će se sporo učitavati ili se uopće neće učitati. Taj nedostatak se može djelomično riješiti s progresivnim web aplikacijama o kojima je više rečeno u sljedećem potpoglavlju.

2.1. Vrste web aplikacija

Postoji više vrsta web aplikacija, a u ovom poglavlju su detaljnije objašnjene višestranične (eng. *Multi-Page Applications* - MPA), jednostranične (eng. *Single-Page Applications* - SPA) i progresivne (eng. *Progressive Web Applications* - PWA).

U [5] je objašnjeno kako su **višestranične** web aplikacije specifične po tome što svaka korisnička interakcija (npr. kretanje po navigaciji, otvaranje poveznica itd.) rezultira ponovnim učitavanjem čitave web stranice sa svim njezinim komponentama. Ponovno učitavanje stranice uključuje i ponavljanje svih zahtjeva prema poslužitelju što kao posljedicu ima sporije učitavanje. Jedna od prednosti višestraničnih aplikacija je SEO optimizacija, jer se svaka stranica može indeksirati zasebno.

Za razliku od višestraničnih web aplikacija, [5] navodi kako **jednostranične** učitavaju većinu sadržaja ili ako je moguće čitav sadržaj pri prvom pristupanju aplikaciji. Prednost takvog načina rada je izbjegavanje učitavanja čitave stranice nakon svake korisničke interakcije. Novi zahtjevi se šalju samo ako je potrebno ažurirati dio sadržaja. Iako smo time ubrzali kretanje po aplikaciji i dalje postoje neki nedostaci poput optimizacije SEO-a i trajanja inicijalnog učitavanja.

Progresivne web aplikacije nadilaze ograničenja mobilnih i web aplikacija korištenjem funkcionalnosti servisnih radnika (eng. *service workers*) i manifesta [6] web aplikacija. Prema [7], glavni nedostatak mobilnih aplikacija je njihova instalacija na uređaje čime se zauzima memorija dok je glavni nedostatak web aplikacija njihov manjak responzivnosti na mobilnim uređajima i nemogućnost pristupa u offline načinu rada. Korištenjem servisnih radnika i manifesta, progresivne aplikacije ujedinjuju najbolje karakteristike mobilnih i web aplikacija nudeći pristup sadržaju aplikacije pritiskom ikone na početnom zaslonu uređaja bez zauzimanja sekundarne memorije. Također, omogućava push notifikacije, korištenje u offline načinu rada, responzivnost i spremanje podataka u međuspremnik. Kvaliteta i performans te sam stupanj progresivnosti aplikacije može biti mjeren korištenjem alata Google Lighthouse.

3. Aplikacijski okviri

Prema [8], aplikacijski okviri (eng. *application frameworks*) pružaju osnovnu strukturu projekta, a nerijetko nude i alate za pomoć pri pronalaženju grešaka (eng. *debugging*) i alate za pisanje testova čime se olakšava proces izrade web aplikacije. Okvir se odabire prema području (izrada web aplikacija, mobilnih aplikacija itd.), programskom jeziku i prema funkcionalnostima koje nudi. Na primjer, za programski jezik PHP postoje aplikacijski okviri Laravel i Symfony, a konačna odluka o tome koji će se koristiti na projektu ovisi o funkcionalnostima koje nude. Više je o vrstama okvira rečeno u sljedećem potpoglavlju.

Prema [9, str. 4] postoji pet prednosti korištenja aplikacijskih okvira tijekom razvijanja aplikacija: modularnost, proširivost, jednostavnost, ponovna upotrebljivost i održivost.

Modularnost predstavlja podjelu aplikacije na slojevitu strukturu čime je spriječeno nastajanje prevelike ovisnosti komponenata u projektu. Osim toga, programeri se mogu usmjeriti na specifičan dio aplikacije koji odgovara njihovom iskustvu, čime se povećava produktivnost i smanjuje količina grešaka u kodu [9, str. 4–5].

Zbog raznih varijacija poslovnih logika nemoguće je kreirati aplikacijski okvir koji može zadovoljiti sve zahtjeve poslovanja. Međutim, prema [9, str. 5–6], moguće je izdvojiti osnovne komponente koje su zajedničke svim aplikacijama i omogućiti **proširenje** za komponente koje sadrže specifičnu poslovnu logiku. Pri tome je važno odrediti stupanj proširivosti aplikacijskog okvira. Veći stupanj proširivosti znači da se aplikacijski okvir može upotrijebiti za izradu različitih poslovnih aplikacija, ali to zahtijeva dublje razumijevanje povezanosti među komponentama i utječe na jednostavnost korištenja aplikacijskog okvira.

Jedna od glavnih razlika između aplikacijskog okvira i biblioteke je upravo u **jednostavnosti** korištenja. Biblioteka nudi programeru na raspolaganje brojne komponente kojima može izgraditi aplikaciju. Kako bi ih uspješno upotrijebio, programer treba biti upoznat sa zahtjevima komponenti te načinima na koje ih treba povezati. S druge strane, aplikacijski okvir skriva tehničke detalje od programera i prepušta orkestraciju komponenti dizajneru aplikacijskog okvira. Na taj način se smanjuje krivulja učenja korištenja aplikacijskog okvira i programeru je omogućeno da se usredotoči na razvoj poslovne logike aplikacije [9, str. 6–7]. Osim navedenog, prema [10, str. 26–27], okviri diktiraju arhitekrutu i fokus je na ponovnoj upotrebi dizajna, dok biblioteke pružaju veću fleksibilnost i ciljaju na ponovnu upotrebu koda .

Osim proširivosti, izdvajanje osnovnih komponenti omogućuje **ponovnu upotrebljivost** koda. Prebacivanjem često korištenih komponenti koje ne ovise o poslovnoj logici, u aplikacijski

okvir, povećava se produktivnost, izbjegavaju se različite implementacije komponenti i dupliciranje koda te se olakšava **održavanje** projekta [9, str. 5]. Tijekom životnog ciklusa aplikacije, dolazit će do promjene zahtjeva. Zbog same strukture aplikacijskog okvira, nove promjene neće utjecati na temeljne komponente već samo na one koje se tiču poslovne logike čime se osim produktivnosti postiže i smanjenje troškova nastalih uvođenjem novih izmjena [9, str. 7].

U poglavlju o aplikacijskom okviru Symfony su detaljnije obrađene neke od osnovnih komponenti koje taj okvir nudi, a u nastavku su navedene samo neke od njih [11]:

- ubacivanje zavisnosti (eng. *dependency injection*) - komponenta za centralizirano kreiranje i prosljeđivanje objekta drugim objektima,
- biblioteka Doctrine - komponenta za rad i komunikaciju s bazom podataka,
- forme (eng. *forms*) - komponenta za rad s HTML formama,
- biblioteka Twig - komponenta za rad s HTML predlošcima,
- upravitelj porukama (eng. *messenger*) - komponenta za razmjenu poruka prema/od drugih aplikacija ili putem redova poruka.

3.1. Vrste okvira

Postoje različite vrste okvira dizajniranih za razvoj različitih softverskih proizvoda. U nastavku su navedene neke od vrsta okvira, uz kratak opis i konkretne primjere.

Za izradu web aplikacija koriste se web aplikacijski okviri i oni se dijele na okvire korisničke strane (eng. *front-end frameworks*) i okvire serverske strane (eng. *back-end frameworks*) te su u [8] i [12] dane definicije i primjeri za obje skupine. **Okviri korisničke strane** odgovorni su za izradu komponenti korisničkog sučelja i omogućavanje interakcije korisnika s aplikacijom. Također, ovisno o zahtjevima, okviri korisničke strane su zaduženi i za responzivnost kako bi korištenje web aplikacije bilo omogućeno i na mobilnim uređajima. Neki od primjera okvira korisničke strane su: Angular i Vue.js (JavaScript) i Bootstrap (CSS). **Okviri serverske strane** primaju i obrađuju pristigle zahtjeve (eng. *requests*), šalju odgovor (eng. *response*), omogućavaju komunikaciju s bazom podataka i različitim programskim sučeljima za aplikacije (eng. *Application Programming Interface* - API) te su odgovorni za ispravnu implementaciju i izvođenje poslovne logike. Neki od primjera okvira serverske strane su: Ruby on Rails (Ruby), Django (Python), Spring Boot (Java) i Symfony (PHP).

Za razvoj mobilnih aplikacija koriste se **okviri za izradu mobilnih aplikacija** koji nude alate za izradu korisničkog sučelja i omogućavaju korištenje nativnih funkcionalnosti uređaja (slanje obavijesti, pristup kameri, mikrofon itd.). Okviri za izradu mobilnih aplikacija omogućavaju pokretanje aplikacija na operacijskim sustavima poput iOS-a i Androida, a neki od primjera navedenih u [8] i [12] uključuju Flutter i React Native.

Okviri za testiranje služe za pisanje automatiziranih testova i često podržavaju različite razine testiranja: jedinično testiranje (eng. *unit testing*), funkcionalno testiranje (eng. *functional testing*) i testiranje prihvatljivosti (eng. *acceptance testing*). Osim pisanja testova, omogućavaju i njihovo izvršavanje te generiranje izvještava o uspješnosti samih testova. Neki od poznatijih okvira za testiranje u PHP-u su PHPunit i Codeception, a JUnit je često korišten za testiranje aplikacija pisanih u programskom jeziku Java [12] [13].

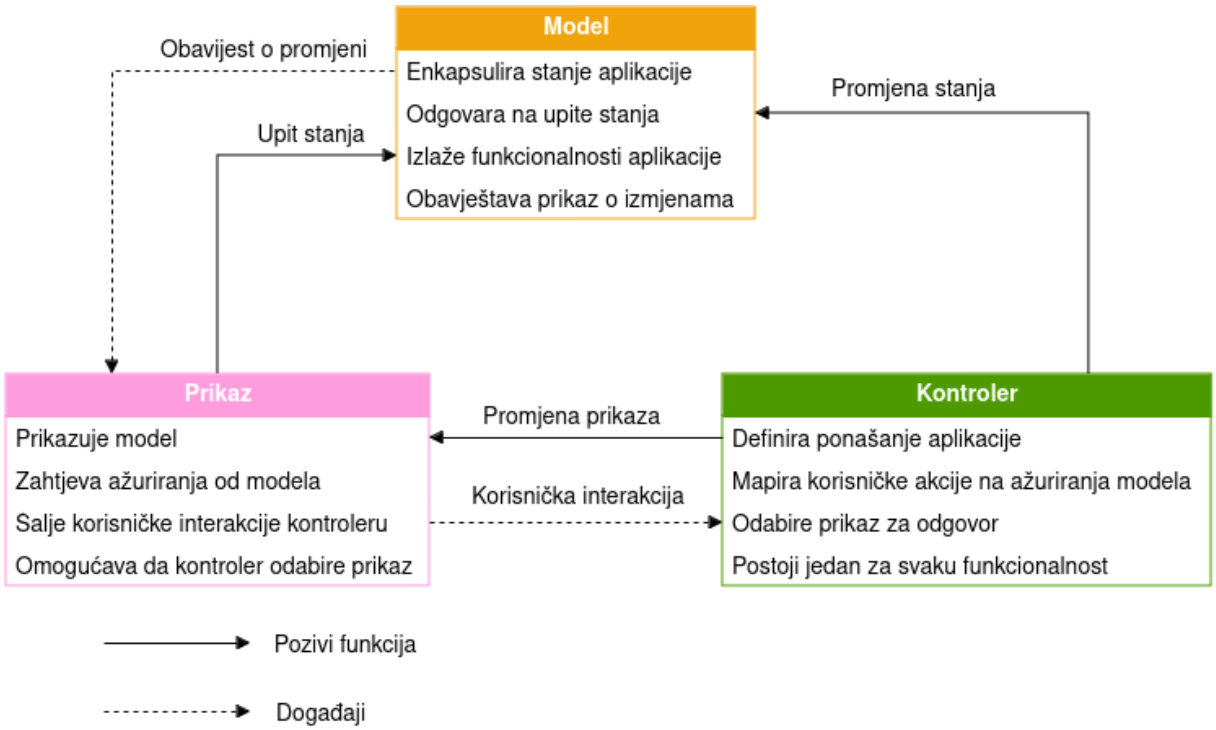
Okviri za rad s podacima i strojno učenje nude različite algoritme i alate za izradu modela za strojno učenje i predviđanje te analizu velike količine podataka. Poznati okviri za strojno učenje su prema [8] i [12] TensorFlow, PyTorch i Apache Spark.

U [8] je objašnjeno kako se za razvoj igara koriste **okviri za izradu računalnih i mobilnih igara** koji između ostalog, nude alate za simulaciju fizike unutar igara i alate za upravljanje resursima (eng. *assets*) kao što su slike i glazba. Poznati primjeri okvira za razvoj igara su Unity i Unreal.

3.2. MVC uzorak dizajna

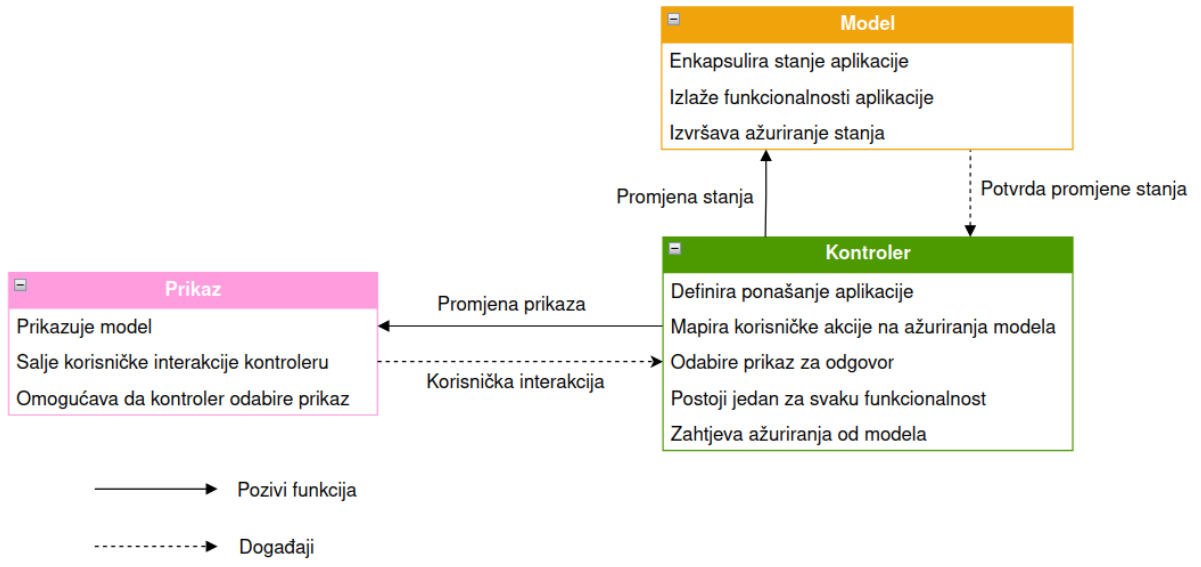
Jedan od poznatih uzoraka dizajna koji se koristi u okvirima je Model-Pogled-Kontroler (eng. *Model-View-Controller* - MVC) te je upravo po tom uzorku implementirana i web aplikacija u praktičnom dijelu rada. U [14] je navedeno kako je MVC uzorak kojega je 1979. godine razvio Trygve Reenskaug, Smalltalk programer iz Xerox Palo Alto Research Centra. Glavna zamisao MVC-a je odvajanje poslovne logike i pristupa podacima od njihove prezentacije korisnicima. Na slici 1 su prikazane komponente MVC uzorka: model, prikaz i kontroler.

Prema [14], **model** predstavlja podatke i logiku koja upravljaju pristupom i izmjenama nad podacima. U kontekstu poslovnog sustava, model reflektira stvarne poslovne entitete i procese. **Pogled** prikazuje podatke iz modela korisniku te određuje na koji način će podaci biti prezentirani. Također, omogućava korisničku interakciju koja se dalje usmjerava kontroleru. **Kontroler** temeljem korisničkih akcija usmjerava model na ažuriranje. Ovisno o kontekstu, kontroler može izvršiti i promjenu pogleda kako bi korisniku prikazao ažurirane ili nove informacije.



Slika 1: MVC uzorak dizajna (tradicionalna verzija) [14]

Na slici 1 se vidi kako su komponente modela i pogleda u izravnoj komunikaciji. U novijoj verziji MVC uzorka dizajna (zvanoj Model 2), model i pogled više nisu direktno povezani već se sva komunikacija odvija preko kontrolera. Može se reći da kontroler ima ulogu medijatora koji upravlja protokom podataka između modela i pogleda. Promjena u komunikaciji između komponenti je nastala kao prilagodba MVC-a web aplikacijama. Na slici 2 se nalazi ažurirana verzija MVC uzorka dizajna prema [14] i [15, str. 557].



Slika 2: MVC uzorak dizajna (Model 2) [14] [15, str. 557]

U MVC uzorku dizajna se ne spominju poslovna pravila, zbog čega se može reći da MVC prvenstveno služi kao arhitektura za korisnička sučelja. Mjesto implementacije poslovnih pravila varira ovisno o aplikacijskom okviru. Na primjer, u Symfony projektu se nastoji izbjegavati pisati poslovnu logiku unutar kontrolera. Ona se smješta u klase poput servisa ili repozitorija koje se zatim ubacuju u kontrolere putem ubacivanja zavisnosti. O ovoj temi je više rečeno u poglavlju o Symfonyu. Prema [16, str. 151], važno je da su poslovna pravila odvojena od prezentacijskog sloja i sloja baze podataka. Time se omogućuje zamjena postojećeg sustava za upravljanje bazama podataka drugim bez potrebe za promjenom same logike, a također se pruža fleksibilnost u korištenju različitih korisničkih sučelja, poput web preglednika ili konzole.

Prema [17], zbog odvajanja odgovornosti na tri komponente, MVC uzorak pojednostavljuje organizaciju koda te olakšava pronalazak i dodavanje novih funkcionalnosti unutar projekta. Neovisnost komponenti omogućava programerima istovremeni rad na različitim dijelovima aplikacije, što ubrzava razvojni proces. Također, promjene u jednoj komponenti ne utječu na cijelu arhitekturu čime se povećava skalabilnost projekta. Među ostalim prednostima MVC uzorka vrijedi spomenuti podršku za razvoj vođen testiranjem (eng. *test-driven development* - TTD), implementaciju različitih pogleda za isti model te razvoj SEO-prijateljskih web aplikacija.

4. Symfony

Prema [18], Fabien Potencier je kreirao aplikacijski okvir Symfony 2004. godine kako bi maksimalno iskoristio mogućnosti PHP-a 5 za potrebe svoje tvrtke. Symfony se ne koristi samo kao PHP okvir za izradu konzolnih i web aplikacija, već se može koristiti i kao skup ponovno upotrebljivih PHP komponenti. Projekt Symfony je otvorenog koda, čija zajednica na GitHub repozitoriju broji više od 3000 suradnika (eng. *contributors*) [19].

Symfony danas prati renomirana reputacija stečena dugotrajnim radom aktivne zajednice. Prema [20], neki od ključnih razloga za korištenje Symfonyja uključuju stabilnost projekta, visoku interoperabilnost s raznim sustavima, upotrebu "Open Source MIT" licence te kontinuirani rad na unapređenju funkcionalnosti. U [21] su navedeni raznovrsni primjeri projekata koji koriste Symfony. Od različitih sustava za upravljanje sadržajem (Drupal, Joomla!, Sulu), kupcima i proizvodima (Pimcore, OroCRM), do platformi poput Yahoo!-a i Dailymotiona. Neke od komponenti koje nudi Symfony se čak koriste za izradu drugih PHP okvira (Laravel, CakePHP).

Od svog nastanka pa do danas, Symfony je prošao kroz nekoliko glavnih (eng. *major*) verzija. Prema [22], trenutna stabilna verzija (eng. *stable version*) je 7.1.1, dok verzija 6.4.8 ima dugoročnu podršku (engl. *Long Term Support* - LTS). Glavne verzije Symfonyja izlaze svake dvije godine, dok manje (eng. *minor*) verzije izlaze svakih šest mjeseci (u svibnju i studenom). Projekt u praktičnom dijelu rada koristi verziju 6.4.8, jer tako zahtjeva korištena Pimcore verzija 11.2.5. Trenutno korištene verzije Pimcora i Symfonyja možemo dobiti koristeći sljedeće naredbe iz isječka koda 1:

Isječak kôda 1: Naredbe za pronalazak trenutne verzije platforme Pimcore i okvira Symfony

```
I have no name!@445d681364b9:/var/www/html\$ php bin/console --version
Pimcore v11.2.5 (env: dev, debug: true)
I have no name!@445d681364b9:/var/www/html\$ php bin/console about
-------------------------------------------------------
Symfony
-------------------------------------------------------
Version 6.4.8
Long-Term Support Yes
```

U sljedećim poglavljima su uz programske primjere opisane neke od glavnih komponenti koje nudi Symfony.

4.1. Konfiguracija

Kako bi određene funkcionalnosti u narednim poglavljima ispravno radile, potrebno ih je konfigurirati putem posebnih datoteka. Radi lakšeg razumijevanja, u ovom poglavlju su objašnjeni ključni pojmovi vezani uz konfiguraciju Symfony okvira. Prema [23], Symfony za konfiguraciju koristi datoteke koje se nalaze pod direktorijem config. Pod njime se konfiguriraju postavke aplikacije (npr. rute, servisi itd.), ali i postavke instaliranih paketa (eng. *bundles*). Konfiguracijske datoteke mogu biti pisane u PHP-u, XML-u ili YAML-u. Za potrebe primjera u ovom radu je izabran format YAML, a to je ujedno i zadani format koji koristi Symfony.

YAML (eng. *YAML Aint Markup Language*) je jezik za serijalizaciju podataka i često se koristi u konfiguracijskim datotekama koje završavaju s nastavcima .yml ili .yaml. Prema [24], jedna od prednosti korištenja YAML formata je lakoća čitanja sintakse koja nalikuje strukturi liste. Hijerarhija sintakse se zasniva na uvlakama i na definiranim vrijednostima ključeva.

U isječku koda 2 se nalazi primjer konfiguracije koja se nalazi unutar datoteke config/routes.yaml. Konfiguracija iz primjera ukazuje na klase u kojima Symfony može pronaći definirane rute. Pod ključem *type* je postavljena vrijednost *attribute* što znači da su rute definirane kao atributi. O tome kako izgledaju kontroleri i atributi je više rečeno u sljedećem potpoglavlju.

Isječak kôda 2: Primjer YAML sintakse

```
1 controllers:
2     resource:
3         path: ../src/Controller/
4         namespace: App\Controller
5     type: attribute
```

4.2. Kontroler

U poglavlju o MVC uzorku dizajna je dana teorijska podloga za komponentu kontroler, a kroz ovo poglavlje je objašnjeno kako kontroler funkcionira u praksi u Symfony projektu. Prema [25, str. 53], kada stigne HTTP zahtjev, Symfony će prvo pokušati pronaći rutu koja se poklapa s putanjom zahtjeva (eng. *request path*). Na primjer, ako je kreiran HTTP zahtjev za dohvaćanje proizvoda sa sljedećeg URL-a *http://localhost:8080/products*, Symfony će pokušati pronaći rutu koja je jednaka putanji */products*. Rute služe kao poveznica između putanje i PHP funkcije koja vraća HTTP odgovor. Te funkcije se nazivaju kontroleri i u isječku koda 3 je dana implementacija jedne takve funkcije.

Isječak kôda 3: Primjer kontrolera u Symfonyu

```
class ProductController extends AbstractController {
    #[Route('/products', name: 'app_products', methods: ['GET'])]
    public function getProducts(Request $request): Response {
        $sortDirection = $request->query->get('sortDirection', 'asc');
        $products = [
            ['id' => 1, 'name' => 'Phone',  'price' => 1000],
            ['id' => 2, 'name' => 'TV',     'price' => 500 ],
            ['id' => 3, 'name' => 'Laptop', 'price' => 5000]];
        if ($sortDirection === 'asc') {
            usort($products, function ($a, $b) {
                return $a['price'] > $b['price'];
            });
        } else {
            usort($products, function ($a, $b) {
                return $a['price'] < $b['price'];
            });
        }
        return $this->json($products);}}
```

U kodu se nalazi klasa ProductController s funkcijom getProducts. Ta funkcija je ujedno i kontroler, jer joj je pridružen atribut Route s podacima o putanji, nazivu kontrolera i traženoj HTTP metodi. U [25, str. 54] je pokazano kako se kontroleri mogu konfigurirati i kroz YAML dokument, ali taj način se preporuča samo kod izrade paketa (eng. *package*).

4.3. Servis

Kontroler iz prethodnog poglavlja trenutno sadrži logiku za sortiranje proizvoda, ali prema [26, str. 104] preporuča se postavljanje poslovne logike unutar servisa ili repozitorija. Na taj način se pojednostavljuje kod unutar kontrolera te je omogućena ponovna upotreba poslovne logike.

U [26, str. 218] je objašnjeno kako je servis objekt klase čije metode izvršavaju radnje vezane uz poslovnu logiku. Za kreiranje servisa je zadužen poseban servisni kontejner (eng. *Dependency Injection Container*) koji kreira jednu instancu servisa i to tek onda kada je servis zatražen. Drugim riječima, ako servis nikad nije zatražen, kontejner neće potrošiti memoriju na instanciranje servisa, a ako je isti servis zatražen više puta, kontejner neće na svaki zahtjev kreirati novi objekt već će svi zahtjevi raditi s istom instancom servisa. Na taj način se optimiziraju upotreba memorije i performanse aplikacije.

U isječku koda 4 se nalazi novi servis ProductService koji sadrži samo jednu metodu sortProducts. Kako bi se metoda mogla pozvati u kontroleru, potrebno je konfigurirati servis.

Isječak kôda 4: Primjer servisa u Symfonyu

```
class ProductService {
    public function sortProducts(string $sortDirection): array {
        $products = [...];
        if ($sortDirection === 'asc') {
            ...
        } else {
            ...
        }
        return $products;}}
```

Servis se može konfigurirati kroz dokument services.yaml na nekoliko načina. U isječku koda 5 je prikazano definiranje servisa korišenjem ključne riječi resource. Na taj način su sve klase pod direktorijem *src/Service* automatski konfigurirane kao servis. Ako ipak želimo konfigurirati servise zasebno, onda se koristi primjer iz isječka 6. U [27] je objašanjeno da su konfiguracijske opcije autowire i autoconfigure po zadanom postavljene na true čime je omogućeno automatsko injektiranje ovisnosti (eng. *dependencies*) u servise i registriranje servisa kao komandi, pretplatnika događaja i slično.

Isječak kôda 5: Primjer konfiguracije servisa

```
services:
    _defaults:
        autowire: true
        autoconfigure: true
    App\Service\:
        resource: '../src/Service'
```

Isječak kôda 6: Primjer zasebne konfiguracije servisa

```
services:
    _defaults:
        autowire: true
        autoconfigure: true
    App\Service\ProductService:
```

U isječku koda 7 se nalazi refaktorirani kontroler koji sada poziva funkciju sortProducts iz novokreiranog servisa ProductService. S time je postignut pregledniji kod i ponovna upotrebljivost funkcije za sortiranje proizvoda.

Isječak kôda 7: Refaktorirani kontroler

```
1 public function getProducts(Request $request, ProductService $productService):
   Response {
2     $sortDirection = $request->query->get('sortDirection', 'asc');
3     $products = $productService->sortProducts($sortDirection);
4     return $this->json($products);}
```

4.4. Doctrine

Za rad s bazom podataka Symfony koristi Doctrine. Prema [25, str. 69], Doctrine je skup biblioteka koje olakšavaju upravljanje bazama podataka. Doctrine uključuje: Doctrine DBAL (apstrakcijski sloj baze podataka), Doctrine ORM (biblioteku koja mapira PHP objekte na podatke u bazi podataka i obratno) te Doctrine Migrations (za upravljanje migracijama).

Kako bi Doctrine biblioteke mogle raditi s bazom podataka, potrebno je konfigurirati podatke o vezi s bazom podataka. Doctrine ima vlastitu konfiguracijsku datoteku doctrine.yaml i glavna postavka koju očekuje je DSN (eng. *Data Source Name*) baze podataka. DNS je niz koji sadrži informacije o vezi s bazom podataka: korisničko ime, lozinku, port, naziv baze podataka itd. Po zadanom, doctrine.yaml učitava vrijednost varijable okoline (eng. *environment variable*) DATABASE_URL iz .env ili .env.local datoteke. Primjer takve varijable se nalazi u isječku koda 8 [25, str. 70].

Isječak kôda 8: Primjer DSN-a baze podataka u .env datoteci

```
DATABASE_URL="mysql://username:password@host:port/database"
```

4.4.1. Entitet

Entitet (eng. *entity*) je klasa koja predstavlja tablicu u bazi podataka [25, str. 71]. U isječku 9 je dan primjer korištenja naredbe za kreiranje entiteta Product.

Isječak kôda 9: Naredba za kreiranje entiteta

```
symfony console make:entity Product
created: src/Entity/Product.php
created: src/Repository/ProductRepository.php
Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.
New property name (press <return> to stop adding fields):
> Name
```

```
Field type (enter ? to see all types) [string]:
> string
Field length [255]:
> 255
Can this field be null in the database (nullable) (yes/no) [no]:
> no
```

Klasa Product je dodana pod direktorij Entity i sadrži jedan atribut Name. Uz to je kreirana dodatna klasa ProductRepository u koju je moguće dodati funkcije koje će izvršavati CRUD operacije nad entitetom Product. U isječku koda 10 je prikazan dio klase Product. Trenutno je to obična PHP klasa s atributima koje koristi Doctrine prilikom mapiranja podataka.

Isječak kôda 10: Primjer entiteta

```
#[ORM\Entity(repositoryClass: ProductRepository::class)]
class Product {
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;
    #[ORM\Column(length: 255)]
    private ?string $Name = null;
...
```

4.4.2. Migracija

Kako bi se za entitet Product iz prethodnog potpoglavlja kreirala tablica u bazi podataka, potrebno je kreirati i pokrenuti migraciju. Prema [25, str. 79], migracija je klasa koja sadrži nove izmjene s kojima je potrebno ažurirati bazu podataka. Za kreiranje migracije se koristi naredba iz isječka koda 11.

Isječak kôda 11: Naredba za kreiranje migracije

```
symfony console make:migration
created: migrations/Version20240702193740.php
```

U isječku koda 12 se nalazi generirana migracija. Naziv klase je generiran korišenjem riječi Version te datuma i vremena nastanka. Klasa sadrži tri metode: getDescription, up i down. Funkcija getDescription sadrži opisanu namjenu migracije, a funkcija up sadrži SQL kod kojim ćemo dodati nove izmjene u bazu podataka. Funkcija down služi za poništavanje izmjena nastalih migracijom.

Isječak kôda 12: Migracija za kreiranje tablice entiteta Product

```
final class Version20240702193740 extends AbstractMigration {
    public function getDescription(): string {
        return 'Migration for creating product table.';}
    public function up(Schema $schema): void {
        $this->addSql('CREATE TABLE product (id INT AUTO_INCREMENT NOT NULL, name
         VARCHAR(255) NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8mb4
         COLLATE `utf8mb4_unicode_ci` ENGINE = InnoDB');}
    public function down(Schema $schema): void {
        $this->addSql('DROP TABLE product');}}
```

S naredbom u isječku koda 13 će se izvršiti novokreirana migracija. Kako bismo se uvjerili da je migracija stvarno izvršena, osim što možemo pogledati, sadrži li baza podataka tablicu product, možemo pogledati sadrži li tablica iz isječka koda 14 migraciju.

Isječak kôda 13: Naredba za izvršavanje migracije

```
symfony console doctrine:migrations:migrate
```

Isječak kôda 14: Primjer doctrine_migration_versions tablice

```
mysql> SELECT * FROM doctrine_migration_versions;
+----------------------------------------+---------------------+----------------+
| version                                | executed_at         | execution_time |
+----------------------------------------+---------------------+----------------+
| DoctrineMigrations\Version20240702193740 | 2024-07-02 19:38:10 |              9 |
+----------------------------------------+---------------------+----------------+
```

4.4.3. QueryBuilder

Kako se upiti nad bazom ne bi izvršavali koristeći samo direktnu SQL sintaksu, Doctrine omogućava objektno-orijentirani način za kreiranje upita uz pomoć klase QueryBuilder. U poglavlju o entitetima je prikazano kako je prilikom korištenja naredbe za kreiranje entiteta Product, kreirana i klasa ProductRepository. U isječku koda 15 se nalazi primjer navedenog repozitorija s metodom getProductByName koja poziva funkciju za kreiranje QueryBuilder objekt. Nad QueryBuilder objektom se lančano pozivaju metode za kreiranje samog upita, a funkcija getOneOrNullResult na kraju vraća objekt proizvoda ili vrijednost NULL ako izvršeni SQL ne pronađe odgovarajući zapis [28].

Isječak kôda 15: Primjer pisanja upita uz pomoć klase QueryBuilder

```
class ProductRepository extends ServiceEntityRepository {
    public function getProductByName($name): ?Product {
        return $this->createQueryBuilder('product')->andWhere('product.Name = :name')
            ->setParameter('name', $name)->getQuery()->getOneOrNullResult();}
```

4.5. Forma

Za kreiranje proizvoda putem web stranice potrebno je kreirati formu. U isječku koda 16 se nalazi naredba koja će kreirati ProductFormType formu na temelju klase Product [25, str. 131].

Isječak kôda 16: Primjer naredbe za generiranje forme

```
symfony console make:form ProductFormType Product
created: src/Form/ProductFormType.php
```

Nakon što se izvrši naredba pod direktorijem src/Form je nastala nova klasa ProductFormType. U isječku koda 17 se nalazi generirani kod forme koja sadrži samo atribut Name.

Isječak kôda 17: Primjer generirane forme za klasu Product

```
class ProductFormType extends AbstractType {
    public function buildForm(FormBuilderInterface $builder, array $options): void {
        $builder->add('Name');}
    public function configureOptions(OptionsResolver $resolver): void {
        $resolver->setDefaults(['data_class' => Product::class]);}}
```

U [25, str. 135] je navedeno kako su atributi forme automatski konfigurirani na temelju Doctrine atributa unutar klase entiteta. Unatoč tome, moguće je napraviti izmjene nad njima. U isječku koda 18 je atribut Name proširen s dodatnim opcijama. Postavljen je label atributa na "Name" koji će se prikazati uz polje unutar forme. Također, atribut required je postavljen na true kako bi polje bilo obavezno za ispunjavanje. Na kraju je dodan submit gumb kako bi korisnici mogli poslati formu.

Isječak kôda 18: Primjer uređivanja forme

```
$builder
    ->add('Name', TextType::class, ['label' => 'Name', 'required' => true])
    ->add('submit', SubmitType::class);
```

Osim prikazanih tipova atributa u isječku koda 18 postoje i tipovi poput: EmailType, FileType, HiddenType, DateType itd. Prema [25, str. 135], u nekim situacijama neće biti moguće automatski mapirati određenu vrijednost forme u entitet. U tom slučaju se pod atribut u formi dodaje opcija mapped i postavlja se na false, a logika oko spremanja vrijednosti na entitet će biti dodana ručno.

U isječku koda 19 je prikazano kako se forma može upotrijebiti u kontroleru. Na početku se kreira forma uz pomoć metode createForm koja prima tip forme ProductFormType i instancu entiteta klase Product kao parametre. Zatim, forma obrađuje HTTP zahtjev pomoću metode handleRequest, koja popunjava formu podacima iz zahtjeva.

Nakon što se forma popuni podacima, provjerava se je li forma podnesena (isSubmitted) i je li ispravna (isValid). Ako su oba uvjeta zadovoljena, podaci iz forme se spremaju u bazu podataka. Na kraju se korisnika preusmjerava na stranicu uspjeha. Ako forma nije podnesena ili nije ispravna, prikaže se Twig predložak product/form.html.twig s formom kako bi korisnik mogao unijeti potrebne podatke. Više je o Twig predlošcima rečeno u sljedećem potpoglavlju.

Isječak kôda 19: Kreiranje forme u kontroleru

```
1 public function getProducts(Request $request, ProductRepository $productRepository){
2     $productForm = $this->createForm(ProductFormType::class, new Product());
3     $productForm->handleRequest($request);
4     if ($productForm->isSubmitted() && $productForm->isValid()) {
5         $productRepository->save($productForm->getData());
6         return $this->redirectToRoute('task_success');}
7     return $this->render('product/form.html.twig', ['form' => $productForm]);}
```

4.6. Predložak

Prema [29], najbolji način za organiziranje i prikazivanje HTML-a unutar Symfony aplikacije su predlošci. Predlošci se mogu koristiti za vraćanje HTML-a kao odgovor u kontroleru ili za generiranje sadržaja elektroničke pošte. Za kreiranje predložaka Symfony koristi biblioteku Twig. Predlošci se nalaze pod direktorijem templates i završavaju s nastavkom .html.twig, a u isječku koda 20 se nalazi primjer jednog predloška.

Isječak kôda 20: Primjer Twig predloška

```
1 {% block title %}Proizvod{% endblock %}
2 {% block body %}<h1>{{ product.Name }}</h1><p>Primjer!</p>{% endblock %}
```

Twig je u doslovnom prijevodu stroj za predloške (eng. *template engine*), a sintaksa se zasniva na sljedećim trima komponentama [29]:

- {{ ... }} - koristi se za prikazivanje vrijednosti varijable,
- {% ... %} - koristi se za izvršavanje logičkih operacija poput petlji ili uvjeta,
- {# ... #} - koristi se za dodavanje komentara.

U [29] je objašnjeno kako u predlošcima nije moguće pisati PHP kod, ali je moguće koristiti razne funkcije i filtere koje nudi Twig (npr. upper, format_date, trim itd) te je moguće kreirati vlastite funkcije koristeći Twig ekstenzije. Osim što su Twig predlošci lako čitljivi, također su i brzi u produkcijskom okruženju, jer se kompajliraju u PHP i spremaju u međuspremnik. U razvojnom okruženju se predložak kompajlira nakon svake izmjene.

U isječku koda 19 na liniji 8 je već prikazano kako se predložak može prikazati kroz kontroler. U predložak product/form.html.twig je proslijeđena forma koja se može upotrijebiti na nekoliko načina. U isječku koda 21 se nalazi prvi primjer gdje se funkciji form prosljeđuje forma.

Isječak kôda 21: Primjer HTML predloška s formom

```
1 <h1>Unesite podatke za proizvod</h1>
2 {{ form(form) }}
```

Ako želimo imati veći stupanj kontrole nad poljima unutar forme (npr. zbog dodavanja stilova), formu se može upotrijebiti kao što je prikazano u isječku koda 22.

Isječak kôda 22: Primjer HTML predloška (pojedinačno)

```
1  {{ form_start(productForm) }}
2          <div class="inputform">
3                  {{ form_label(form.Name) }}
4                  {{ form_widget(form.Name) }}
5                  {{ form_errors(form.Name) }}
6          </div>
7          <div class="button">
8          {       { form_widget(form.submit) }}
9          </div>
10 {{ form_end(form) }}
```

4.7. Okidanje događaja

U [25, str. 117] je objašnjeno kako Symfony koristi komponentu za okidanje događaja (eng. *event dispatcher component*) koja omogućava da se određeni događaj (eng. *event*) okine u određenom trenutku. U sustav se onda može dodati slušatelj (eng. *listener*) ili pretplatitelj (eng. *subsriber*) koji će čekati na okidanje određenih događaja. Symfony po zadanom nudi neke događaje (npr. kernel.request, kernel.response itd.), ali je moguće kreirati i okidati vlastite događaje.

Na primjer, ako se u svaki odgovor koji vraća aplikacija, želi dodati određeno zaglavlje (eng. *header*), može se upotrijebiti već postojeći događaj kernel.response. Navedeni događaj je zapravo obična klasa ResponseEvent koja sadrži podatke koji se mogu koristiti u slušatelju ili pretplatitelju. Događaj se okida u klasi HttpKernel u metodi filterResponse, a navedeno je prikazano u isječku koda 23.

Isječak kôda 23: Funkcija koja okida kernel.response događaj

```
1 class HttpKernel implements HttpKernelInterface, TerminableInterface {
2     public function __construct(protected EventDispatcherInterface $dispatcher,...
3     private function filterResponse(Response $response, Request $request,int $type){
4         $event = new ResponseEvent($this, $request, $type, $response);
5         $this->dispatcher->dispatch($event, KernelEvents::RESPONSE);
```

Naredba prikazana u isječku koda 24 će kreirati pretplatitelja ResponseSubscriber ili slušatelja ResponseListener. Naredba će tijekom izvršavanja ponuditi dostupne događaje u aplikaciji te će postaviti pitanje o tome želimo li generirati slušatelja ili pretplatitelja [25, str. 117].

Isječak kôda 24: Naredba za kreiranje pretplatitelja

```
1  symfony console make:listener Response
2  Do you want to generate an event listener or subscriber? [Listener]:
3  [0] Listener
4  [1] Subscriber
5  > 1
6  Suggested Events:
7  * kernel.request (Symfony\Component\HttpKernel\Event\RequestEvent)
8  * kernel.response (Symfony\Component\HttpKernel\Event\ResponseEvent)
9  ...
10 What event do you want to listen to?:
11 > kernel.response
12 created: src/EventSubscriber/ResponseSubscriber.php
```

U isječku koda 25 se nalazi primjer generiranog pretplatitelja. Klasa ResponseSubscriber implementira sučelje EventSubscriberInterface i metodu getSubscribedEvents. Metoda getSubscribedEvents daje informaciju EventDispatcher objektu o tome koje događaje očekuje te koja će metoda obraditi određeni događaj. Prema [30], zbog postavljene vrijednosti parametra autoconfigure na true u services.yaml, generirana klasa se automatski prepoznaje kao pretplatitelj.

Isječak kôda 25: Primjer pretplatitelja

```
class ResponseSubscriber implements EventSubscriberInterface {
    public static function getSubscribedEvents(): array {
        return [KernelEvents::RESPONSE => 'onKernelResponse',];}
    public function onKernelResponse(ResponseEvent $event): void {
        $response = $event->getResponse();
        $response->headers->set('X-Subscriber-Header', 'Test');}}
```

U isječku koda 26 se nalazi primjer generiranog slušatelja. Klasa ResponseListener ne mora implementirati sučelje, ali zato sadrži atribut AsEventListener iznad metode koja će obraditi kernel.response događaj [30].

Isječak kôda 26: Primjer slušatelja

```
final class ResponseListener {
    #[AsEventListener(event: KernelEvents::RESPONSE)]
    public function onKernelResponse(ResponseEvent $event): void {
        $response = $event->getResponse();
        $response->headers->set('X-Listener-Header', 'Test');}}
```

Izvršavanjem CURL naredbe prikazane u isječku koda 27 može se vidjeti kako su u zaglavlju odgovora dodani i X-Subscriber-Header i X-Listener-Header.

Isječak kôda 27: Primjer CURL naredbe za dohvaćanje zaglavlja

```
curl --head http://127.0.0.1:8001/login
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
...
X-Listener-Header: Test
X-Subscriber-Header: Test
```

U prikazanim primjerima i slušatelj i pretplatitelj dodavaju zaglavlje u odgovor na poprilično jednostavan način. Postavlja se pitanje, je li bolje koristiti slušatelja ili pretplatitelja? Prije uvođenja atributa AsEventListener, slušatelje je trebalo dodatno konfigurirati kroz konfiguracijsku datoteku. Prema [25, str. 117], kako bi se izbjeglo korištenje konfiguracijske datoteke

za definiranje svakog događaja i pridružene metode, koristili su se pretplatitelji. Kako je i kod pretplatitelja i slušatelja izbjegnuto korištenje konifguracijske datoteke, može se reći da se izbor između njih svodi na odluku programera.

4.8. Upravitelj porukama

Upravitelj porukama (eng. *messenger*) je komponenta koja unutar Symfony aplikacije omogućava asinkrono izvršavanje određenih funkcionalnosti [25, str. 181]. To je ujedno i jedan od načina poboljšanja performansi aplikacije čija je primjena testirana u praktičnom dijelu rada. Na primjer, ako aplikacija pruža mogućnost slanja podataka na e-poštu, može se dogoditi da pružatelj usluge e-pošte nije dostupan ili je slanje usporeno zbog povećanja prometa nad API-em. Kako korisnik ne bi bio izložen sporijem radu aplikacije, moguće je poslati e-poštu korištnjem upravitelja poruka.

Za početak je potrebno kreirati dvije klase. Jedna će služiti kao poruka (eng. *message*), a druga će služiti za obradu poruke (eng. *message handler*). Poruka sadrži samo podatke koji će se serijalizirati i spremiti u red čekanja. U isječku koda 28 se nalazi primjer poruke EmailMessage koja sadrži osnovne podatke poput e-pošte i sadržaja [25, str. 185].

Isječak kôda 28: Primjer poruke

```
1 class EmailMessage {
2     public function __construct(private string $email, private string $content){}
3     public function getEmail(): string {
4         return $this->email;}
5     public function getContent(): string {
6         return $this->content;}}
```

U isječku koda 29 se nalazi primjer rukovatelja poruke EmailMessageHandler označenog s atributom AsMessageHandler. Klasa sadrži i metodu __invoke koja očekuje objekt EmailMessage i služi za obrađivanje primljenog objekta, tj. poruke [25, str. 186].

Isječak kôda 29: Primjer rukovatelja poruke

```
1 #[AsMessageHandler]
2 class EmailMessageHandler {
3     public function __construct(private MailerInterface $mailer){}
4     public function __invoke(EmailMessage $emailMessage){
5         $email = (new Email())
6         ->from('from.email@gmail.com')
```

```
7           ->to($emailMessage->getEmail())
8           ->text($emailMessage->getContent());
9           $this->mailer->send($email);}}
```

U isječku koda 30 se nalazi primjer kontrolera koji će okinuti slanje EmailMessage poruke. Potrebno je preko ubacivanja zavinosti ubaciti MessageBusInterface u kontroler i pozvati funkciju dispatch. U funkciju se proslijeđuje objekt klase EmailMessage [25, str. 187].

Isječak kôda 30: Kreiranje i slanje poruke unutar kontrolera

```
1 #[Route('/email', name: 'email', methods: ['GET'])]
2 public function sendEmail(MessageBusInterface $bus) {
3     $message = new EmailMessage('marta@gmail.com', 'Email content');
4     $bus->dispatch($message);
5     return $this->json(['message' => 'Email successfully sent!']);}
```

Po zadanom će se odmah nakon poziva funkcije dispatch, pronaći odgovarajući upravitelj poruke zadužen za EmailMessage i pozvat će se funkcija __invoke. Prema [25, str. 188], kako bi se poruka izvršila asinkrono, potrebno je doraditi konfiguraciju prema isječku koda 31. Kao transport je postavljen Doctrine, poruke će se spremati u bazu podataka u tablicu messenger_messages. Oznaka async je proizvoljna i potrebna je kako bi se pod routing moglo ispravno usmjeriti poruku na određeni transport, jer je moguće imati više transporta i više poruka.

Isječak kôda 31: Konfiguracija za asinkrono izvršavanje poruke

```
1 framework:
2     messenger:
3         transports:
4             async:
5                 dsn: '%env(MESSENGER_TRANSPORT_DSN)%' #doctrine://default
6         routing:
7             App\Message\EmailMessage: async
```

Prema [25, str. 189], kako bi se poruke iz tablice messenger_messages izvršile, potrebno je pokrenuti komandu u isječku koda 32. Komanda kao argument prima naziv transporta i moguće ju je prebaciti na izvršavanje u pozadini uz opciju '-d'.

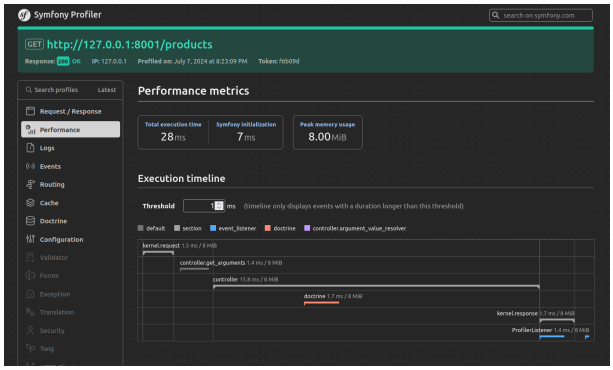
Isječak kôda 32: Naredba za asinkrono izvršavanje poruke

```
symfony console messenger:consume async
```

4.9. Profiler

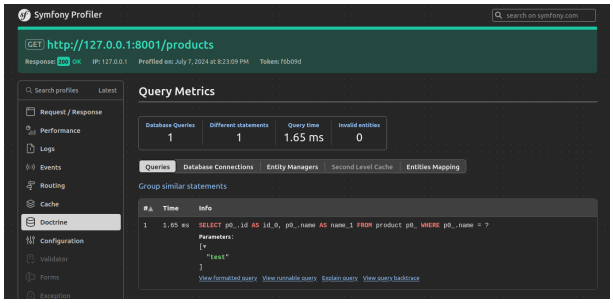
Profiler je komponenta koja prikuplja i prikazuje podatke o svakom HTTP zahtjevu poslanom prema aplikaciji [31]. Podaci koje prikuplja mogu uključivati informacije o vremenu izvršavanja, uhvaćenim greškama, izvršenim operacijama nad bazom podataka, okinutim porukama i slušateljima, sesijama, logovima i mnogim drugim aspektima aplikacije.

Na slici 3 je primjer podataka o performansu izvršavanja zahtjeva. Prikazano je ukupno vrijeme izvršavanja, a moguće je vidjeti i detalje izvršavanja. Na primjer, izvršavanje radnji unutar kontrolera je trajalo 15.8 ms tijekom kojih je izvršen upit nad bazom čije je izvršavanje trajalo 1.7 ms. Smanjivanjem praga (eng. *threshold*) na 0 moguće je dobiti uvid u detaljniji pregled izvršavanja zahtjeva. Prikazana metrika može poslužiti kao pomoć u optimizaciji sustava. Programer može dobiti uvid u problematične dijelove koda i implementirati rješenja koja će ubrzati rad aplikacije.



Slika 3: Primjer kartice Performance unutar alata Profiler

Na slici 4 se nalazi primjer podataka koje generira Doctrine. Profiler pruža uvid u ukupan broj izvršenih operacija nad bazom i ukupno trajanje izvršavanja operacija. Također, moguće je vidjeti sve operacije koje su izvršene tijekom obrade jednog zahtjeva. Kroz detaljnu analizu navedenih podataka, programeri mogu identificirati potencijalne probleme i optimizirati upite, indekse i druge aspekte baze podataka kako bi poboljšali učinkovitost aplikacije.



Slika 4: Primjer kartice Doctrine unutar alata Profiler

5. Pimcore

Prema [32, str. 1, 3–5], Pimcore je platforma otvorenog koda koja omogućuje izradu raznih digitalnih projekata, nudeći pritom različite funkcionalnosti koje su spremne za upotrebu odmah nakon instalacije. Za Pimcore se može reći da je digitalna platforma za iskustvo (eng. *Digital eXperience Platform* - DXP), tj. tehnološko rješenje koje podržava digitalno iskustvo korisnika u interakciji s tvrtkom. DXP stavlja korisnika u središte poslovanja, usmjeravajući tehnologiju na isporuku relevantnog i personaliziranog sadržaja putem portala, web stranica i e-trgovina. Od tehnologija Pimcore koristi Symfony, PHP i ExtJS.

U [32, str. 9] su navedene neke od funkcionalnosti koje nudi Pimcore: upravljanje podacima, sustav za upravljanje sadržajem (eng. *Content Management System* - CMS), upravljanje informacijama proizvoda (eng. *Product Information Management* - PIM), upravljanje glavnim podacima (eng. *Master Data Management* - MDM) i upravljanje digitalnom imovinom (eng. *Digital Asset Management* - DAM).

Upravljanje podacima je funkcionalnost koja omogućava definiranje klasa i njihovih atributa korištenjem administrativnog sučelja. Na taj način, klase se mogu kreirati bez pisanja ijedne linije koda. Pimcore će automatski generirati tablice, polja i upite na temelju odabranih opcija u administrativnom sučelju, a o samom sučelju je više rečeno u sljedećem potpoglavlju [32, str. 10].

CMS predstavlja sustav koji omogućava prikazivanje personaliziranog digitalnog sadržaja korisnicima. Sadržajem se upravlja iz jednog mjesta (platforme Pimcore), a osim personalizacije, omogućena je i višejezičnost i korištenje WYSIWYG (eng. *What You See Is What You Get*) uređivača teksta [32, str. 11].

PIM omogućuje tvrtki učinkovito upravljanje informacijama o proizvodima. Prema [32, str. 12], zahvaljujući mogućnosti integracije s različitim sustavima, nakon kreiranja klase za proizvode, dovoljno je unijeti CSV datoteku s podacima u Pimcore. Pimcore će tada, uz minimalno ili nikakvo prilagođavanje postojeće logike, automatski kreirati objekte proizvoda unutar sustava. Nakon kreacije objekata proizvoda, moguće je uspostaviti procese praćanje statusa (eng. *Status workflow*) i kvalitete (eng. *Data quality management*) te ažurirati njihove podatke.

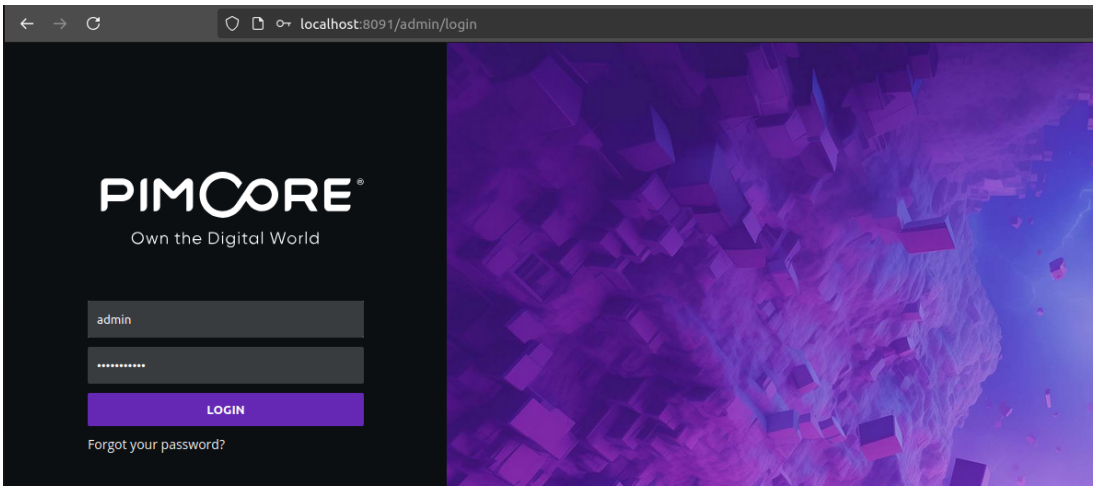
MDM koristi centraliziranu bazu s podacima o raznim entitetima u sustavu te omogućuje upravljanje strukturom, validiranje podataka, verzioniranje te obogaćivanje (eng. *enrichment*) entiteta sadržajem. Umjesto dodavaja slika kroz jedan sustav, prijevoda kroz drugi, a podataka o proizvodu korz treći, moguće je sve to obaviti unutar platforme Pimcore [32, str. 13].

Osim entitetima, Pimcore omogućava upravljanje i digitalnom imovinom (eng. *assets*). **DAM** omogućava upravljanje digitalnom imovnim iz jednog središta, a pod digitalnu imovinu spadaju: slike, videi, PDF datoteke, CSV datoteke itd. [32, str. 15]

Jedna od prednosti Pimcore platforme je njezina fleksibilnost, tj. prilagodba različitim scenarijima bilo da se želi kreirati običan CMS ili poslovna aplikacija. Platforma je izgrađena nad Symfony okvirom što prema [32, str. 18–22] donosi brojne prednosti poput: korištenja paketa, ubacivanje zavisnosti, pristup raznim komponentama i opširnoj dokumentaciji. Pimcore je široko prihvaćen te ga koristi preko 100,000 tvrtki u 56 zemalja, a njegova zajednica otvorenog koda osigurava stalni razvoj i nove funkcionalnosti.

5.1. Administrativno sučelje

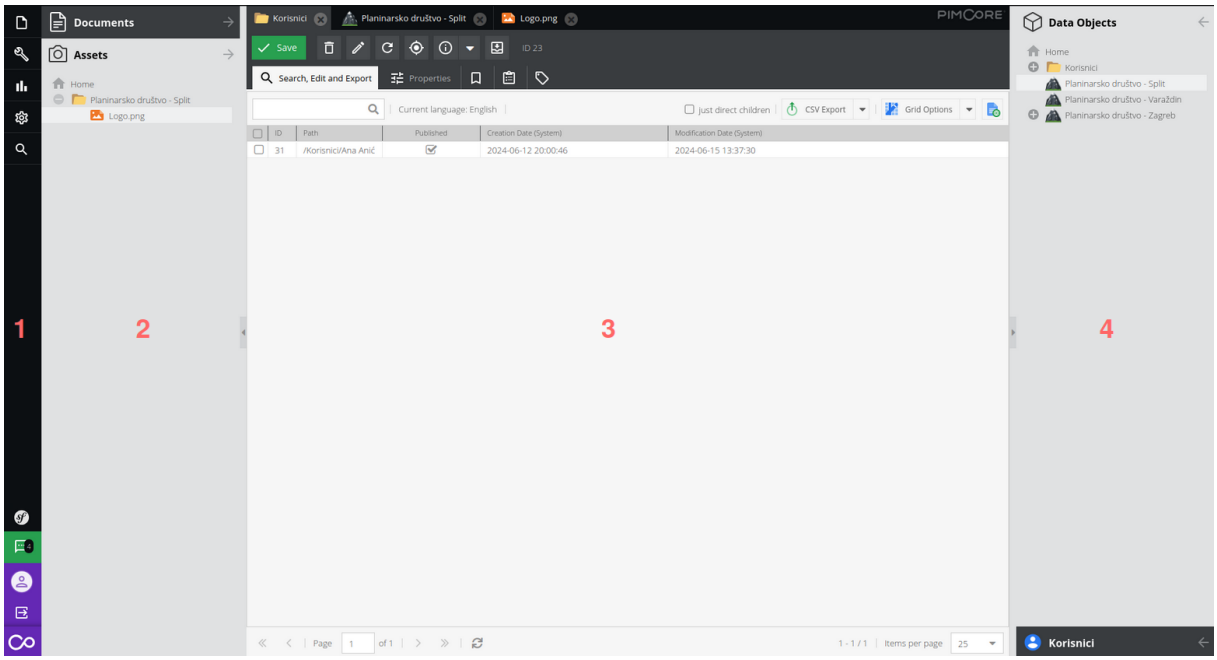
Prema [32, str. 43], lorisničko sučelje u Pimcore aplikaciji se dijeli na sučelje namijenjeno administraciji i sučelje namijenjeno javnosti. Pimcore administraciji se pristupa preko linka nazivstranice.com/admin/ gdje će se prvo učitati stranica sa slike 5.



Slika 5: Stranica za prijavu u Pimcore adminsitraciju

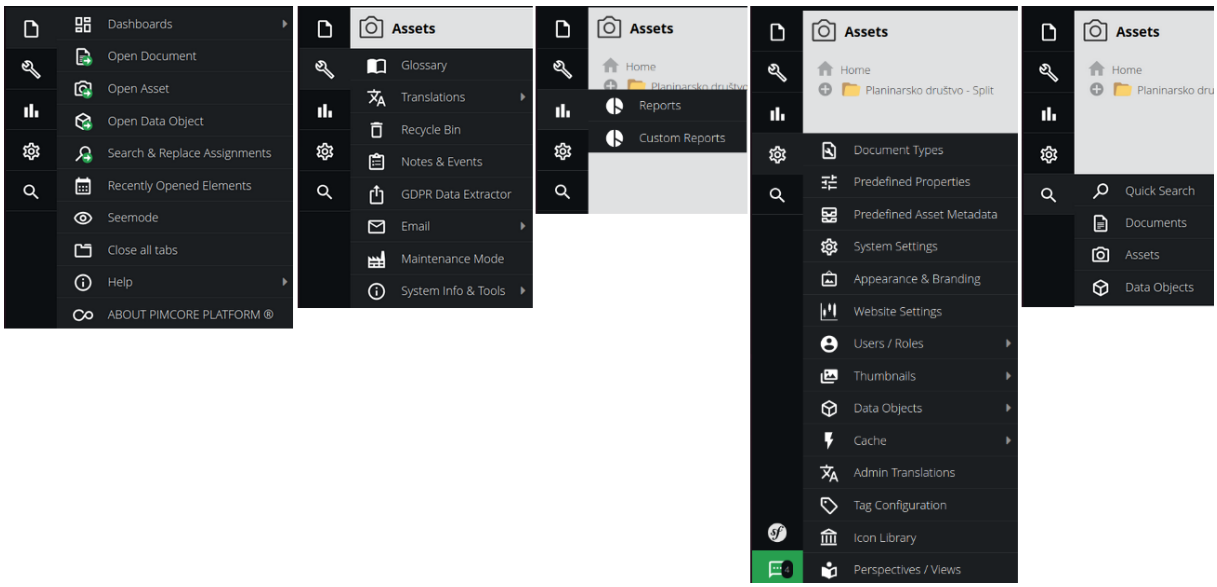
Nakon unosa ispravnih podataka za prijavu, učitat će se stranica sa slike 6 koja je podijeljena na 4 dijela [32, str. 43]:

1. bočnu traku s izbornicima,
2. lijevu bočnu traku,
3. glavni okvir i
4. desnu bočnu traku.



Slika 6: Pimcore administracija

Bočna traka s izbornicima se može podijeliti na dva dijela: na vrhu se nalaze ikone koje će klikom na njih, prikazati dodatne padajuće izbornike sa slike 7, a u donjem dijelu trake se nalaze infomacije o prijavljenom korisniku, poruke, gumb za odjavu i gumb sa Symfony logom. Klikom na Symfony logo će se prikazati alatna traka od komponente Profiler. Padajući izbornici sadrže različite funkcionalnosti. Moguće je pretraživati dokumente, slike i objekte, dodavati prijevode, kreirati izvještaje, upravljati korisnicima, definirati opće postavke, izbrisati podatke u međuspremniku itd. Prema [32, str. 51–54], moguće je dodati nove ikone i proširiti postojeće izbornike.



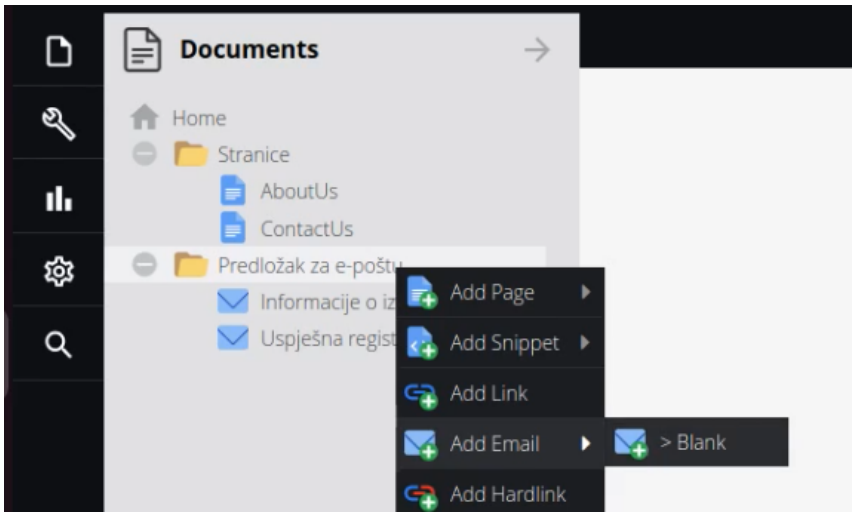
Slika 7: Padajući izbornici u administraciji

Lijeva bočna traka na slici 6 sadrži prečace za pristup dokumentima i digitalnoj imovini. **Desna bočna traka** sadrži podatke o objektima, ali i dodatni prilagođeni izbornik za korisnike. Izbornike je moguće prebacivati s lijeve na desnu stranu i obratno, ali nakon osvježavanja stranice, izbornici će se resetirati. Prema [32, str. 45, 47], za dodavanje novih izbornika i određivanja što će biti na kojoj strani, koristi se uređivač perspektive (eng. *Perspective Editor*). Uređivaču perspektive se pristupa preko padajućeg izbornika, a novokreirane perspektive je moguće dodijeliti ulogama čime se može postići bolja organizacija sučelja. Na primjer, ako korisnik radi samo s digitalnom imovinom, nema potrebe da mu prostor zauzimaju dokumenti i objekti.

Glavni okvir je u središtu Pimcore administracije. Nakon što se klikne na određeni element ili funkcionalnost iz padajućih izbornika, prikazat će se upravo u glavnom okviru. Na slici 6 se u glavnom okviru nalazi otvorena mapa (eng. *folder*) Korisnici. Klikom na mape se može dobiti uvid u elemente koje sadrži, može ih se sortirati, filtrirati, izvesti u CSV itd. Uz karticu (eng. *tab*) mape se nalaze kartice za objekt Planinarsko društvo - Split i digitalnu imovinu Logo.png. Kao i u pregledniku, moguće je istovremeno imati otvoreno više elemenata i izmijenjivati se između njih [32, str. 46].

5.2. Dokumenti (eng. *Documents*)

Prema [32, str. 58], dokumenti u Pimcoru su dio CMS-a i sadrže nestrukturirane informacije kao što su web stranice i e-pošta. Na slici 8 je prikazano kako se kreira dokument tipa Email. Potrebno je kliknuti desnom tipkom miša na mapu (ili Home), odabrati Add Email pa Blank i unijeti naziv dokumenta.



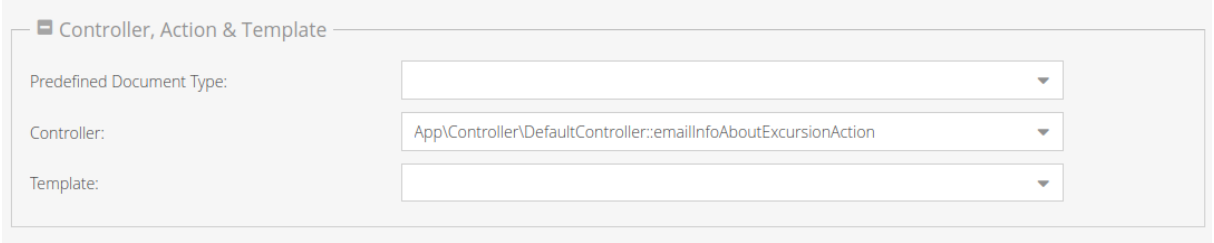
Slika 8: Kreiranje dokumenta tipa Email

Novokreirani dokument će biti prazan, a kako bi dobio sadržaj potrebno je kreirati Twig predložak. U isječku koda 33 se nalazi primjer predloška koji će dodati tekstualno polje za unos naslova e-pošte i polje koje je moguće urediti uz pomoć WYSIWYG-a.

Isječak kôda 33: Primjer predloška za dokument tipa Email

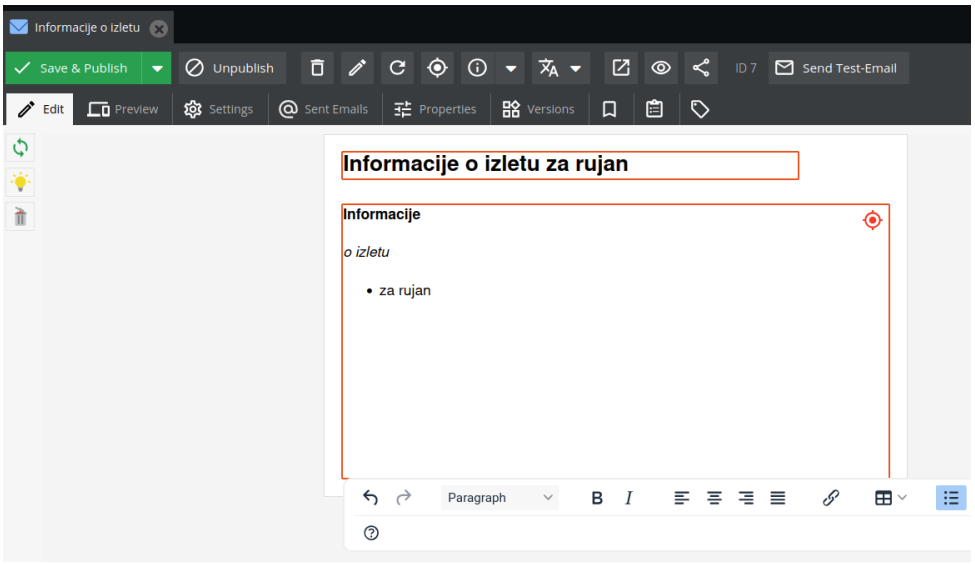
```
<!DOCTYPE html><html lang="en"><head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Email Template</title>
<style>...</style>
</head><body>
<div class="email-container">
<div class="email-title">{{ pimcore_input('EmailTitle', {'width': 500}) }}</div>
<div class="email-body">{{ pimcore_wysiwyg('EmailBody', {'height': 300}) }}</div>
</div>
</body></html>
```

Nakon što je kreiran predložak potrebno je kreirati kontroler koji će ga vratiti. Kontroler je zatim potrebno povezati s dokumentom kao što je prikazano na slici 9.



Slika 9: Primjer povezivanja dokumenta i kontrolera

Rezultat povezivanja dokumenta i kontrolera je prikazan na slici 10. U [32, str. 75] je objašnjeno kako je za kreiranje predloška i kontrolera zadužen programer, ali proces dodavanja i uređivanja teksta u dokumentu može biti prepušten netehničkoj osobi bez straha da će se struktura web stranice ili predložak e-pošte potrgati. Kako bi korisnik provjerio kako će izgledati e-pošta krajnjim primateljima, Pimcore nudi opciju za testno slanje e-pošte odabirom gumba Send Test-Email koji se također može vidjeti na slici 10.



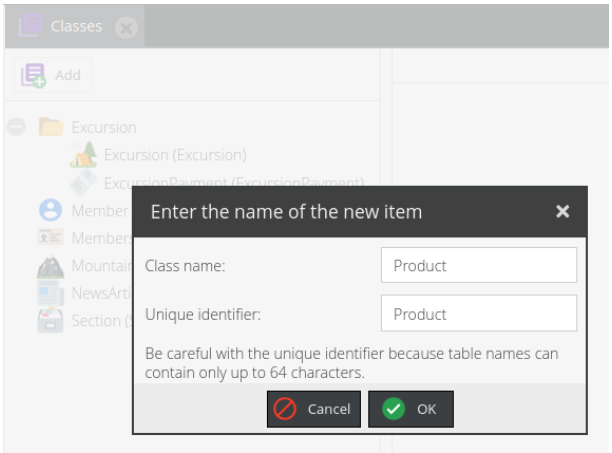
Slika 10: Prikazani predložak dokumenta tipa Email

Uz pomoć dokumenata se mogu kreirati i povezati cijele web stranice, ali prema [32, str. 75], iako su dokumenti dobri u prikazivanju web stranica ipak nisu prilagođeni za rad sa strukturiranim podacima i interaktivnim sadržajem. Za rad sa strukturiranim podacima se koriste objekti, a više je o njima rečeno u sljedećem potpoglavlju.

5.3. Objekti (eng. *Data Objects*)

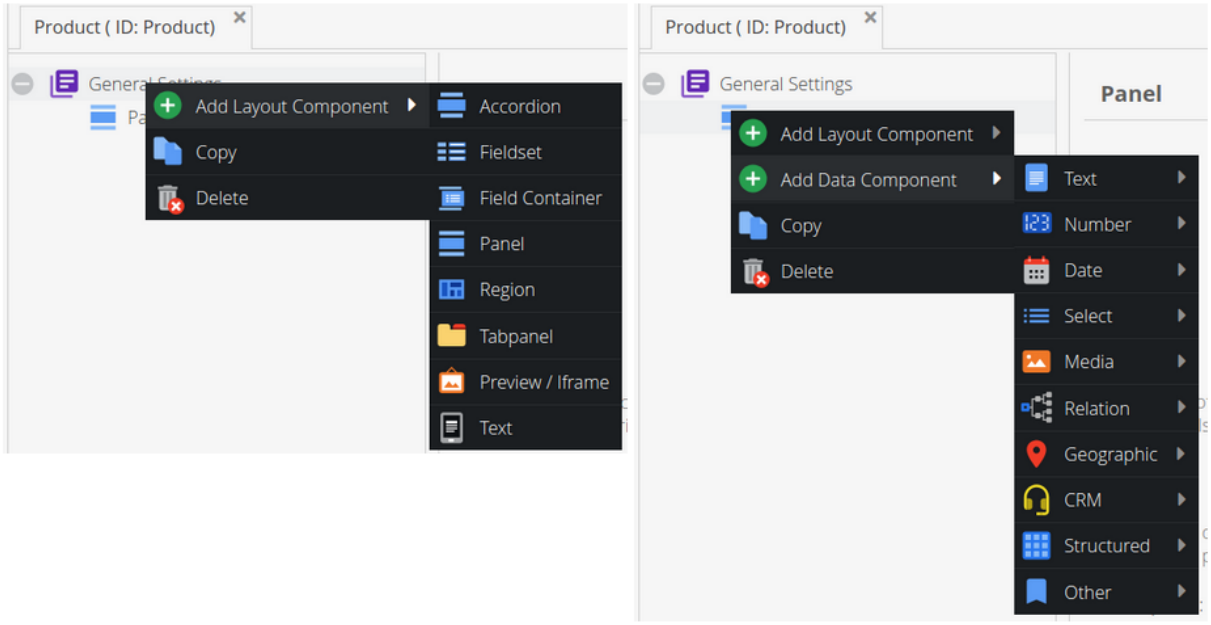
Prema [32, str. 78], klase i objekte se može kreirati kroz administracijsko sučelje bez pisanja ikakvog koda. Potrebno je u padajućim izbornicima pronaći Data Objects i odabrati Classes. Klikom na gumb Add prikazat će se modal sa slike 11 gdje je potrebno unijeti naziv klase.

Nakon klika na gumb OK, kreirat će se tri PHP datoteke: datoteka s definicijom (eng. *definition file*), datoteka s klasom i datoteka s nazivom Listing.php [32, str. 80]. Datoteka s definicijom se nalazi pod direktorijem var/classes i zove se definition_Product.php. U njoj se nalazi kompleksna struktura polja koja predstavljaju sve informacije o atributima na klasi te vizualne aspekte poput CSS pravila. Datoteka s klasom se nalazi pod direktorijem var/classes/DataObject i zove se Product.php. U njoj se nalaze funkcije za dohvaćanje i postavljanje atributa nad objektima koji su instanca te klase. Klasa Listing se nalazi pod direktorijem var/DataObject/Product i sadrži metode za kreiranje upita. Primjer korištenja je dan u sljedećem potpoglavlju.



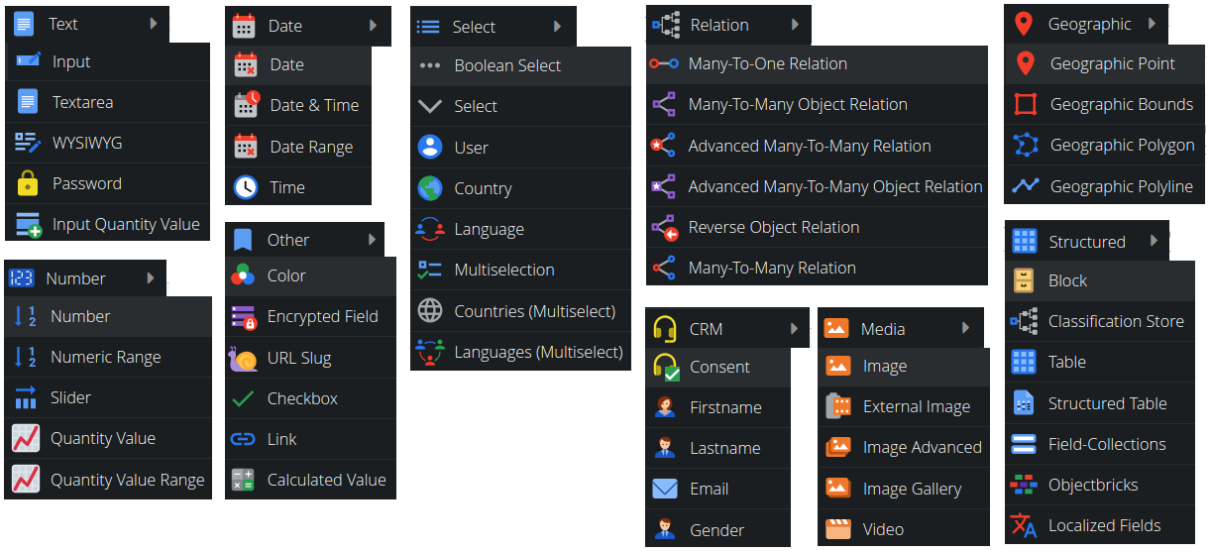
Slika 11: Kreiranje klase u Pimcoru

Na novokreiranu klasu je moguće dodati dvije vrste komponenti: komponente za raspored i komponente za podatke. Navedeno je prikazano i na slici 12. U [32, str. 83–84] je objašnjeno kako komponente za raspored služe za organizaciju atributa klase u prostoru. Na primjer ako klasa ima puno atributa, preglednije je razdvojiti ih u više kartica ili ih podijeliti u dva stupca umjesto da se atributi izlistavaju jedan iza drugoga vertikalno.



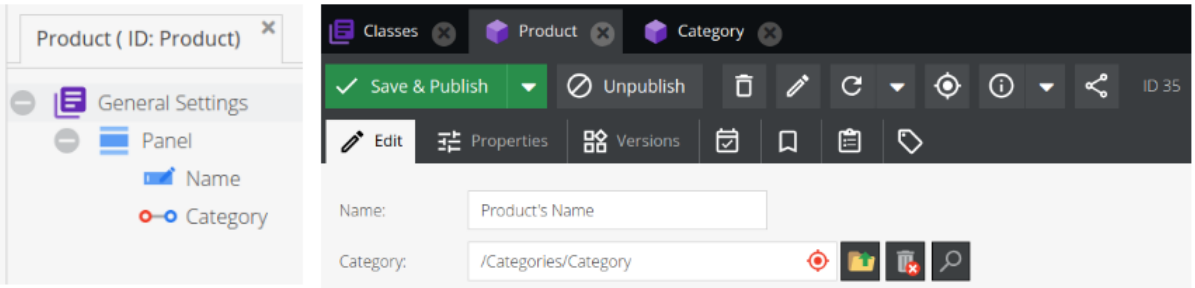
Slika 12: Moguće komponente

Prema [32, str. 88], komponente za podatke su atributi za koje će se generirati funkcije za dohvaćanje i postavljanje vrijednosti. Po zadanom postoji 10 skupina atributa: tekst, brojevi, podaci o datumu i vremenu, padajući izbornici, mediji, relacije, geografske oznake, podaci za upravljanje korisnicima, struktuirani podaci i ostalo. Na slici 13 su detaljnije prikazani svi mogući atributi unutar navedenih skupina, ali Pimcore omogućava i dodavanja vlastitih atributa.



Slika 13: Zadani atributi

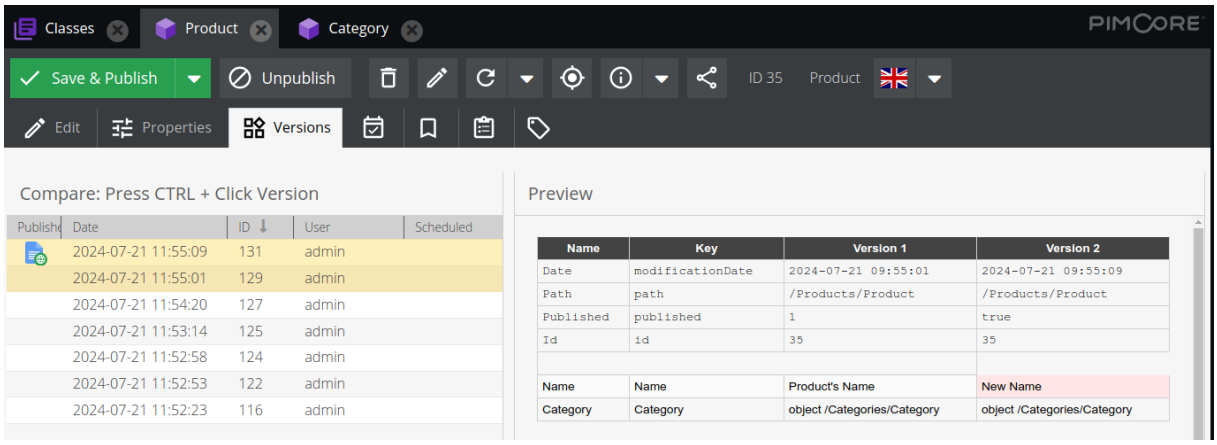
Na slici 14 se nalazi definicija klase Product koja na sebi ima jedan teskstualni atribut Name (Input) i jedan relacijski atribut Category (Many-To-One Relation). Uz definiciju klase je prikazan i primjer jednog objekta klase Product. Objekt se kreira slično kao i dokument. Potrebno je kliknuti desnom tipkom miša nad mapom, odabrati Add Object i kliknuti na Product. Nakon toga će se kreirati objekt i prikazat će se u glavnom okviru gdje se onda mogu dodati vrijednosti u atribute.



Slika 14: Atributi na klasi i objektu

5.3.1. Verzioniranje

Nakon svakog spremanja objekta, Pimcore će kreirati novu verziju tog objekta. Nakon klika na karticu Versions sa slike 15, izlistat će se zadnjih 10 verzija objekta. Broj verzija je moguće kontrolirati kroz postavke. Na slici su označene zadnje dvije verzije kako bi se vidjele nastale izmjene.



Slika 15: Primjer verzija

U isječku koda 34 se nalazi upit s kojim se mogu dohvatiti spremljene verzije za objekt koji ima ID 35. U tablici je ID od objekta spremljen pod stupac cid, a pod stupcem id se nalazi ID verzije. U tablici se nalazi i jedan stupac stackTrace u koji se spremaju pozivi funkcija koji su nastali kada se pozvala save funkcija nad objektom. U samoj tablici se ne nalaze vrijednosti atributa već su one spremljene u binarni dokumenti koji se nalazi pod direktorijem var/versions/object/g0/35, a sam naziv dokumenta je jednak ID-u verzije. Osim što verzije mogu pomoći korisnicima da vide koje su izmjene napravljene nad objektom, također mogu pomoći i programerima u procesu traženja grešaka (eng. *debugging*).

Isječak kôda 34: Primjer versions tablice

```
1 mysql> SELECT * FROM versions WHERE cid = 35 \G
2 *************************** 1. row ***************************
3 id: 116
4 cid: 35
5 userId: 2
6 date: 1721555573
7 stackTrace: #0 /var/www/html/vendor/pimcore/pimcore/models/Element/AbstractElement.
  php(596): Pimcore\Model\Version->save()
```

5.3.2. CRUD operacije

Osim kroz administrativno sučelje, objekte je moguće kreirati i kroz kod. U isječku koda 35 je dan primjer kreiranja objekta Product. Potrebno je postaviti "roditelja" koji će u ovom slučaju biti mapa s putanjom /Products i potrebno je postaviti "ključ". Ključ je naziv objekta koji je vidljiv u adminsitrativnom sučelju kada se otvori mapa te je dio putanje objekta. Puna putanja objekta bi u ovom slučaju bila /Products/Product's Name (moguće je da ključ sadrži razmake).

Isječak kôda 35: Kreiranje objekta kroz kod

```
$folder = \Pimcore\Model\DataObject\Service::createFolderByPath('/Products');
$category = \Pimcore\Model\DataObject\Category::getById(37);
$product = new \Pimcore\Model\DataObject\Product();
$product->setParent($folder);
$product->setKey('Product\'s Name');
$product->setName('Product\'s Name');
$product->setCategory($category);
$product->save();
```

Kod dohvaćanja objekata se koristi već spomenuta klasa Listing. Svaka klasa ima vlastitu Listing klasu, jer će funkcija getObjects na kraju vratiti objekte te klase. U isječku koda 36 se nalazi primjer korištenja klase Listing. Listing će na kraju napraviti upit koji iz pogleda object_Product dohvaća 10 zadnje kreiranih proizvoda koji su povezani s kategorijom čiji je ID jednak 37. Ako takav proizvod ne postoji, funkcija getObjects vraća prazno polje.

Isječak kôda 36: Dohvaćanje objekata kroz kod

```
$listing = new \Pimcore\Model\DataObject\Product\Listing();
$listing->addConditionParam('category__id = :categoryId', ['categoryId' => 37]);
$listing->setOrderKey('creationDate')->setOrder('DESC');
$listing->setLimit(10)->setOffset(0);
$products = $listing->getObjects();
```

Ažuriranje objekta je prikazano u isječku koda 37. Potrebno je samo izmijeniti atribute i pozvati save funkciju nad objektom.

Isječak kôda 37: Ažuriranje objekta kroz kod

```
$category = \Pimcore\Model\DataObject\Category::getById(39);
$product->setCategory($category);
$product->save();
```

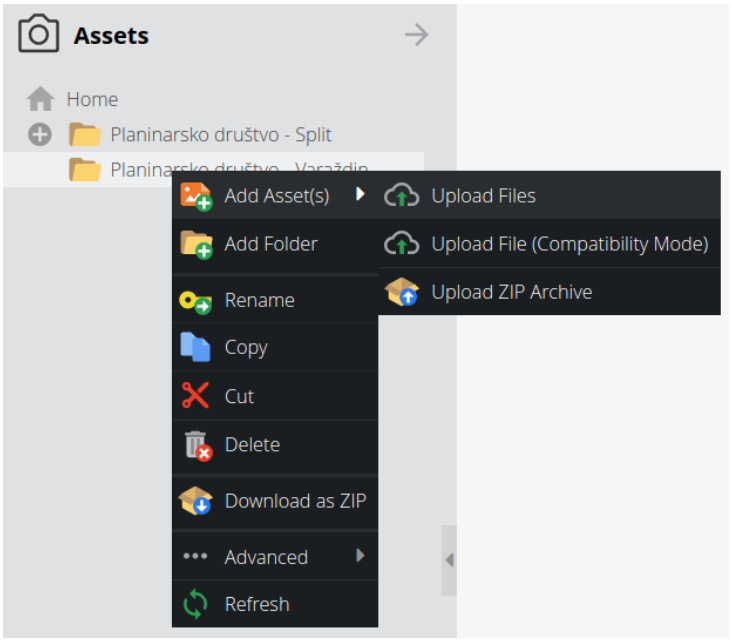
Brisanje objekta je prikazano u isječku koda 38. Potrebno je samo pozvati funkciju delete nad objektom.

Isječak kôda 38: Brisanje objekta kroz kod

```
$product->delete();
```

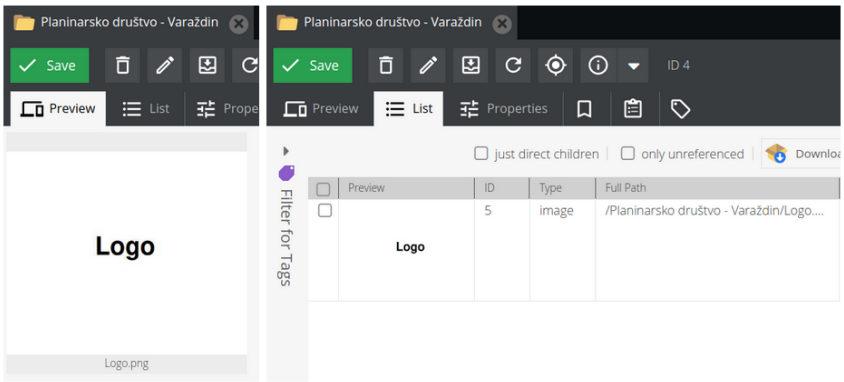
5.4. Digitalna imovina (eng. *Assets*)

Digitalna imovina se kreira u Pimcoru slično kao dokumenti i objekti. Potrebno je kliknuti desnom tipkom miša na mapu u koju se želi dodati datoteka, odabrati Add Asset(s) i odabrati Upload Files (ili Upload ZIP Archive ako se radi o arhivi). Prema [32, str. 111], Pimcore podržava preko 200 formata i moguće je odjednom prenijeti više datoteka različitih formata.



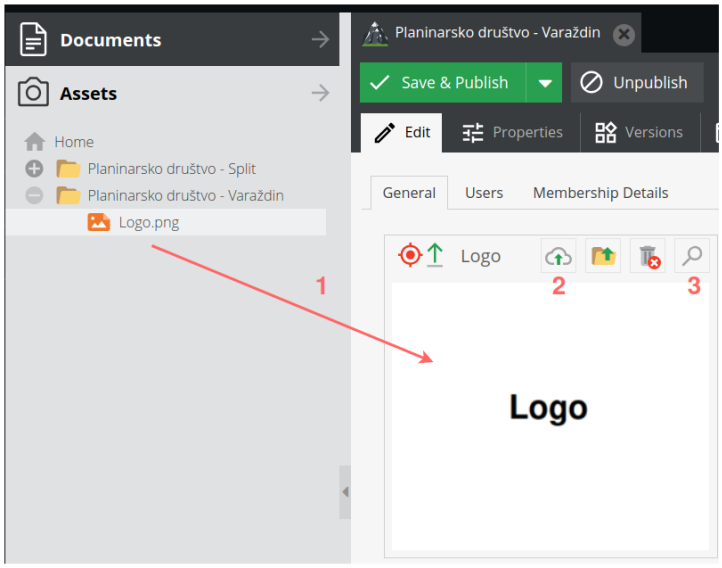
Slika 16: Opcije prijenosa digitalne imovine u Pimcore

Kod pregleda digitalne imovine je moguće odabrati dva prikaza koja su prikazana na slici 17. U prvom prikazu (Preview) se prikazuju datoteke samo sa svojim nazivom, a drugi prikaz (List) pruža detaljniji uvid u datoteke s infomacijama o tipu, putanji, datumu kreacije itd. Osim navedenog prikaz u obliku liste omogućava izvoz slika odabirom opcije Export selected items as ZIP [32, str. 112].



Slika 17: Mogućnosti pregleda digitalne imovine

Objekte je moguće povezati s digitalnom imovinom ako na sebi imaju jedan od atributa iz skupine medija ili relacija. Na slici 18 se nalazi objekt s atributom Image i moguće je spremiti sliku u njega na 3 načina. Prvi način je da se klikne na sliku i povuče u atribut. Drugi je direktni prijenos s računala, a treći je uz pomoć pretraživanja.

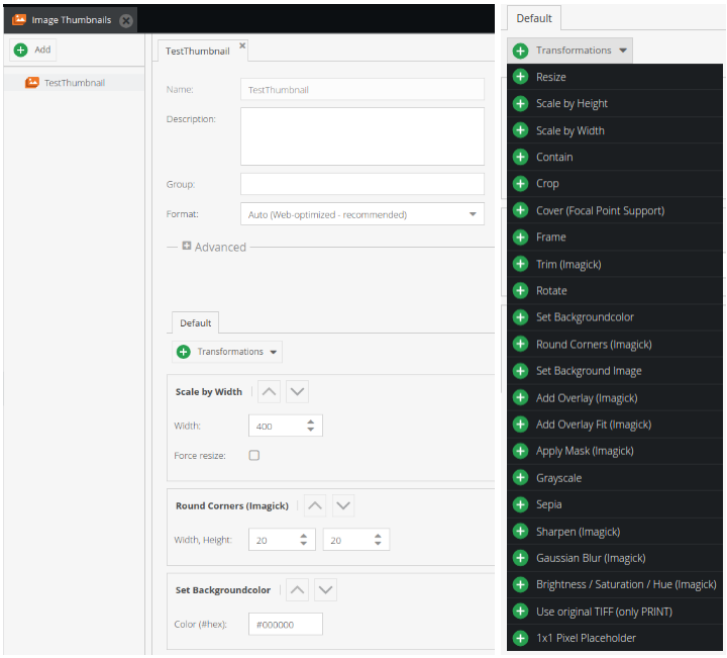


Slika 18: Povezivanje objekta i digitalne imovine

5.4.1. Umanjene slike (eng. *Thumbnails*)

Slike je često potrebno prikazati na različitim rezolucijama (monitor, tablet, mobitel) zbog čega je potrebno primijeniti različite veličine slika. U [32, str. 120] je objašnjeno kako je jedno od mogućih rješenja pripremiti sliku u različitim veličinama, ali to nije najbolje rješenje jer ako će se nešto izmijeniti na originalnoj slici, to treba promijeniti i na duplikatima. Drugo rješenje je korištenje responzivnih tehnika u HTML-u i CSS-u. Pimcore nudi treće rješenje, a to su umanjene slike koje omogućavaju puno više od prilagođavanja slika u različitim ekranima.

Do funkcionalnosti za kreiranje umanjenih slika se dolazi preko padajućeg izbornika odabirom opcije Thumbnails pa odabirom opcije Image Thumbnails (ili Video Thumbnails ako se radi o videima). Na slici 19 se nalazi primjer sučelja gdje se umanjena slika dodaje preko gumba Add. U postavkama se zatim mogu definirati određena transformacijska pravila. Na slici su odabrana tri pravila: prvo će se širina slike postaviti na 400 piksela, zatim će se zaobliti kutevi slike i na kraju će se pozadinska boja postaviti na crnu. To su samo neka od mogućih pravila, a na slici 19 se mogu vidjeti i ostala pravila poput rotacije, promjene visine slike itd. [32, str. 120].



Slika 19: Kreiranje umanjene slike

U isječku koda 39 je prikazano kako se novokreirana konfiguracija za umanjenu sliku može primijeniti nad slikom. Prema [32, str. 124], potrebno je prvo dohvatiti digitalnu imovinu i zatim pozvati funkciju getThumbnail nad njom. U funkciju se prosljeđuje naziv umanjene slike. Rezultat primjene umanjene slike nad digitalnom imovinom je vidiljiv na slici 20.

Isječak kôda 39: Primjer korištenja umanjene slike u kodu

```
#[Route('/')]
public function getThumbnail(): Response {
$logo = \Pimcore\Model\Asset\Image::getById(5);
$thumbnail = $logo->getThumbnail('TestThumbnail');
// HTML u predlo\v{s}ku: <img src="{{ thumbnail }}" alt="image" >
return $this->render('thumbnail.html.twig', ['thumbnail' => $thumbnail]);}
```

Slika 20: Primijenjena umanjena slika

Umanjene slike pomažu i u poboljšanju performanca, ali više je o tome rečeno u poglavlju o optimizaciji performansi na strani klijenta.

5.5. Paketi (eng. *Bundles*)

Paketi sadrže komponente s kojima se dodatno mogu proširiti funkcionalnosti Pimcore aplikacije. U [32, str. 130] je pokazano kako se paketi za Pimcore se mogu pronaći na stranici Pimcore Marketplace, a moguće je i tijekom razvoja izdvojiti određene funkcionalnosti u paket i dodati ih u postojeći ili novi projekt iz privatnog repozitorija. Paketi inače dolaze s instalacijskim uputama i ako nisu po zadanom dohvaćeni s Pimcorom, potrebno ih je dohvatiti uz composer require naredbu. Kod nekih paketa će navedeno biti dovoljno, ali neki će tražiti dodatne radnje poput Custom Reports paketa. Custom Reports paket dolazi s Pimcorom, ali nije po zadanom aktiviran u sustavu. U sljedećem potpoglavlju je nešto više rečeno o njegovim funkcionalnostima i procesu instalacije.

5.5.1. Custom Reports paket

Custom Reports paket služi za kreiranje izvještaja u tabličnom i/ili grafičkom obliku [32, str. 338]. Kao što je već rečeno, ovaj paket nije po zadanom uključen u projekt. Kako bi paket postao dostupan u administrativnom sučelju, potrebno je dodati kod iz isječka koda 40 u config/bundles.php datoteku.

Isječak kôda 40: Aktiviranje paketa unutar bundles.php datoteke

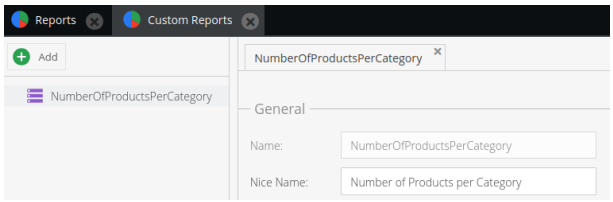
```
Pimcore\Bundle\CustomReportsBundle\PimcoreCustomReportsBundle::class=>['all'=>true]
```

Nakon toga je potrebno izvršiti naredbu iz isječka koda 41. Pregledom klase Pimcore/Bundle/CustomReportsBundle/Installer može se dobiti uvid u to što će naredba napraviti. U ovom slučaju će naredba samo dodati par korisničkih prava (eng. *user permissions*), ali neki drugi paketi si na taj način mogu dodati tablice u bazu podataka, izvršiti migracije i sl.

Isječak kôda 41: Naredba za instaliranje paketa

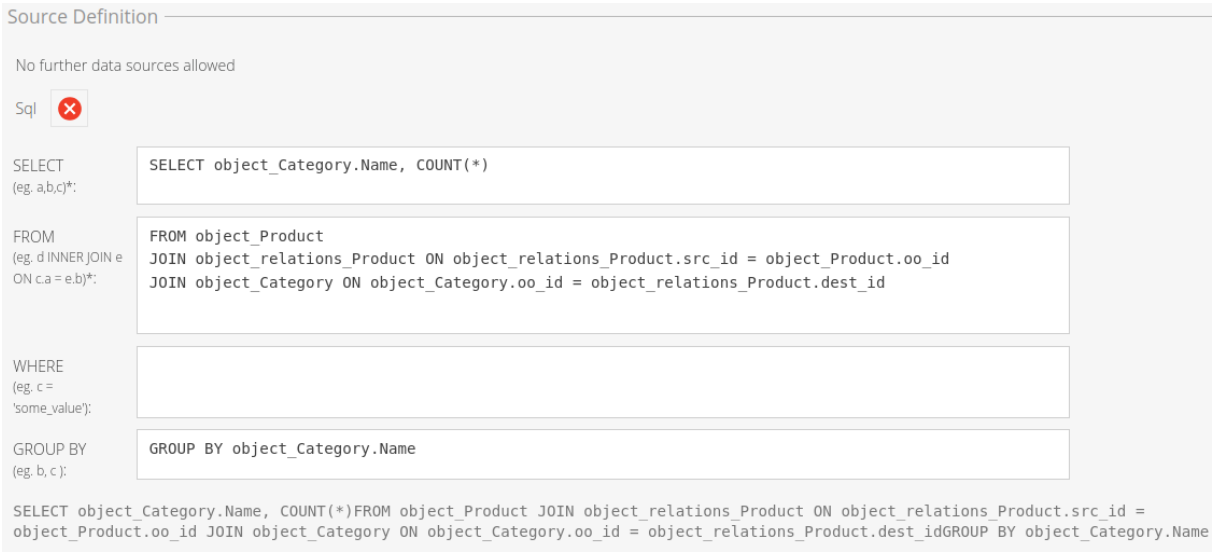
```
bin/console pimcore:bundle:install PimcoreCustomReportsBundle
```

Izvještaj se može kreirati kroz bočnu traku s izbornicima odabirom opcije Custom Reports. Nakon toga je potrebno kliknuti na gumb Add sa slike 21 i unijeti naziv izvještaja. Naknadno je moguće dodati prilagođeno ime za korisnika kroz polje Nice Name [32, str. 339].



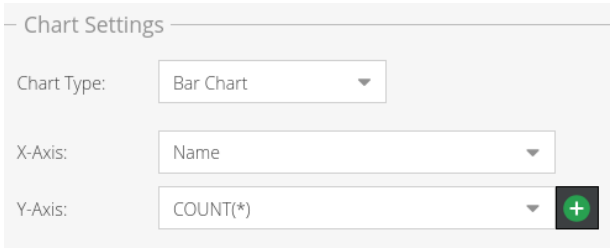
Slika 21: Kreiranje izvještaja

Na slici 22 se nalazi primjer jednog upita kojeg je potrebno unijeti kako bi izvještaj generirao broj proizvoda po kategoriji [32, str. 339].



Slika 22: Dodavanje upita za izvještaj

Za kraj je potrebno definirati izgled izvještaja. Na slici 23 je dan primjer sa stupčastim dijagramom. Na osi x će se nalaziti naziv kategorije, a na osi y broj proizvoda. Osim stupčastog dijagrama, moguće je odabrati kružni graf ili tablični prikaz [32, str. 339].



Slika 23: Definiranje izgleda grafa

6. Performanse

Brzina s kojom sadržaj i informacije dolaze i prikazuju se korisniku, je jedan od ključnih faktora koji utječu na korisničko iskustvo. Kao što je već rečeno u uvodu, loše performanse mogu narušiti posjećenost web aplikacije. Za aplikaciju čiji uspjeh ovisi o visokoj posjećenosti, pad prometa može utjecati na profitabilnost i zadovoljstvo klijenta. Kako do takvih scenarija ne bi došlo, preporuča se slijediti određena pravila o optimizaciji performansa. Performans web aplikacije se može poboljšati i na strani klijenta i na strani servera.

U nastavku su prvo obrađeni načini poboljšanja performansa na strani klijenta. Dane su smjerenice oko smanjivanja broja HTTP zahtjeva te lijenog učitavanja slika u web pregledniku. Obrađena je kompresija slike te je pokazano kako se navedeno može postići u Pimcoru uz pomoć umanjenih slika. Objašnjena je minifikacija JavaScripta te je ukazano na ulogu pozicioniranja JS skripti i CSS stilova unutar HTML datoteke. Na strani servera je obrađen pojam međuspremnika te su prikazana dva moguća načina rada s međuspremnikom u Pimcoru. Dan je uvod u indekse u bazi podataka te su dane smjernice oko pisanja upita. U poglavlju "Upravitelj porukama" je već objašnjeno kako se može implementirati aisnkron način rada te ga se preporuča pročitati zbog razumijevanja prednosti asinkronog pristupa.

6.1. Optimizacija performansi na strani klijenta

Prema [33, str. 1–3] kada se želi poboljšati performans, fokus je inače na optimizaciji servera (indeksi u bazi podataka, upravljanje memorijom). U stvarnosti, na većini web stranica, samo 10-20% vremena odaziva otpada na dohvaćanje HTML dokumenta sa servera. Preostalih 80-90% vremena korisnik provodi čekajući da završi prijenos komponenti. Pod komponente spadaju preusmjeravanja te slike, skripte i stilovi koji nisu spremljeni u međuspremnik itd.

Steve Souders je u svojoj knjizi izdvojio 14 pravila kojih se treba pridržavati kako bi se poboljšao performans, a u ovom radu je izdvojeno njih nekoliko. Prema autoru, pridržavajući se pravila koja su primjenjiva na određenoj web stranici, brzina se može povećati za čak 20-25% čime je u konačnici poboljšano korisničko iskustvo. Osim navedenog, optimizacija na strani klijenta obično zahtjeva manje vremena i resursa, jer optimizacija na serveru nerijetko uključuje redizajniranje arhitekture, izmjene u kritičnim dijelovima koda, izmjene na hardveru i sl. [33, str. 4–5]. Uz pravila iz knjige, dodatno su obrađene teme kompresije slika te tehnika lijenog učitavanja (eng. *lazy loading*).

6.1.1. Smanjiti broj HTTP zahtjeva

Kako bi se smanjio broj HTTP zahtjeva potrebno je ukloniti određene komponente poput slika ili skripti iz HTML dokumenta. Navedeni prijedlog ide u korist poboljšanja performansi, ali i na štetu željenog dizajna. Na sreću, prema [33, str. 10] postoje određene tehnike kojima se može eliminirati određen broj HTTP zahtjeva, a da se pritom ne mora mijenjati dizajn. Navedene tehnike uključuju korištenje mapiranih slika (eng. *image map*), CSS sličica (eng. *CSS sprites*), URL sheme data: i kombiniranje JS skripti i CSS stilova.

Kada se slike koriste za navigaciju ili kao gumbi na stranici, potrebno je <img> element okružiti <a> elementom kako bi nastao hiperlink. Ako se u navigaciji koriste dvije slike, svaka slika će generirati zaseban HTTP zahtjev. Prema [33, str. 11], ovaj problem se može izbjeći korištenjem **mapiranih slika**. U isječku koda 42 nalazi se primjer upotrebe <map> elementa. Potrebno je dvije slike spojiti u jednu i učitati je pomoću <img> elementa. Zatim se unutar <map> elementa definiraju dva područja na slici uz pomoć koordinata, pri čemu svako područje sadrži drugačiji hiperlink. Da bi se slika i mapirana slika povezali, potrebno je dodati atribut name <map> elementu i upisati ga u atribut usemap unutar <img> elementa. Iako upotreba mapiranih slika poboljšava performanse, postoje nedostaci. Definiranje koordinata područja slike je zahtjevno i sklono greškama te se može primijeniti samo za pravokutne površine.

Isječak kôda 42: Primjer korištenja mapirane slike

```
<img usemap="#navigation" border=0 src="/images/navigation.png">
<map name="navigation">
    <area shape="rect" coords="0,0,31,31" href="home.html" title="Home">
    <area shape="rect" coords="36,0,66,31" href="profile.html" title="Profile">
</map>
```

Prema [33, str. 11–13], **CSS sličice** su slične mapiranim slikama jer omogućavaju grupiranje više slika u jednu kako bi se izbjegli dodatni HTTP zahtjevi. Kod mapiranih slika slike moraju biti u nizu dok kod CSS sličica slike mogu ići u više redova. U isječku koda 43 je dan primjer korištenja CSS sličica. Na sprites.png se nalazi devet ikona veličine 30x30 piksela. Ako se želi prikazati druga ikona u drugom redu, potrebno je otići 30px udesno i 30px dolje uz pomoć CSS svojstva background-position. Nakon toga je potrebno postaviti width i height na 30px i na stranici će se prikazati samo ta jedna ikona. Osim što su korištenjem CSS sličica izbjegnuti dodatni HTTP zahtjevi, smanjena je i veličina preuzimanja jer umjesto dohvaćanja devet slika sa svojim metapodacima, informacijama o formatu i tablici boja, sada se ti podaci dohvaćaju samo za jednu sliku.

Isječak kôda 43: Primjer korištenja CSS sličica

```
<div style="background-image: url('sprites.png'); background-position: -30px -30px;
 width: 30px; height: 30px;"></div>
```

Slike je moguće prikazati i na treći način uz pomoć **URL sheme data:**. Neki od malo poznatijih URL shema uključuju mailto:, ftp:, file: itd. U isječku koda 44 je prikazan primjer <img> element koji u atributu src sadrži URL shemu data:. URL shema data: se sastoji od formata (image/png), teksta base64 i podataka koji predstavljaju sliku kodiranu u base64 formatu. Nedostatak ovog pristupa je taj što se slika neće spremiti u međuspremnik. Srećom, taj problem je moguće izbjeći tako da se URL shema uvrsti unutar CSS stila kojeg će preglednik spremiti u međuspremnik [33, str. 13–14].

Isječak kôda 44: Primjer korištenja URL sheme data:

```
<img alt="" src="..."
 style="width:36pt;height:36pt" />
```

Osim smanjivanja broja HTTP zahtjeva koji se tiču slika, preporuča se smanjiti i broj HTTP zahtjeva kod **dohvaćanja JS skripti i CSS stilova**. Jedan od mogućih rješenja je pisanje svog JS koda u jednu skriptu i svih stilova u jednom CSS stilu. Takav način rada se kosi s mnogim principima pisanja čistog i modularnog koda i prema [33, str. 15–16], trebalo bi se držati određenih pravila i razdvajati JS kod i CSS u više datoteka. Zatim se može analizirati koje se skripte i stilovi najčešće koriste zajedno te kreirati proces koji će ih grupirati u jednu skriptu ili stil.

Srećom, postoje paketi poput Laravl Mixa koji već nude navedenu funkcionalnost. U isječku koda 45 je pokazano kako se u Laravel Mixu mogu kombinirati dvije JS skripte u jednu. Navedeno je moguće i za CSS stilove [34]. Osim za kombiniranje, Laravel Mix se može koristiti i za minifikaciju, a više je o samom procesu rečeno u sljedećem potpoglavlju.

Isječak kôda 45: Primjer kombiniranja JS skripti

```
let mix = require("laravel-mix");
mix.combine(['one.js', 'two.js'], 'merged.js');
```

6.1.2. Minifikacija

Prema [33, str. 69, 75], minifikacija je proces uklanjanja nepotrebnih znakova iz koda s ciljem smanjenja veličine dokumenta. Nepotrebni znakovi uključuju razmake, nove redove i tabulatore, a uklanjaju se i komentari. Minificirani kod, posebno kod JavaScript datoteka,

značajno doprinosi bržem učitavanju stranica. Iako je moguće minificirati i CSS stilove, učinak nije toliko drastičan jer CSS općenito sadrži manje komentara i nepotrebnih znakova u usporedbi s JavaScriptom. Kod CSS stilova se veličina dokumenta može dodatno smanjiti korištenjem skraćenica. Na primjer dovoljno je napisati samo #606 umjesto #660066. Prema [33, str. 73], uz minifikaciju je moguće ubrzati učitavanje manjih JS datoteka za 17-19%, a velikih za 30-31%.

Uz minifikaciju, postoji jedna agresivnija metoda optimizacije koda zvana obfuskacija. Prema [33, str. 70], obfuskacija također uklanja komentare i razmake, ali dodatno mijenja imena funkcija i varijabli u kraće nizove. Na taj način se može dodatno smanjiti veličina datoteke, ali uz obfuskaciju dolaze i određeni nedostaci. Prvi od njih su greške u radu zbog same složenosti procesa, a zbog smanjene čitljivosti je otežano pronalaženje uzroka grešaka na produkciji. Uz to je otežano i održavanje jer je potrebno kroz konfiguraciju izostaviti određene simbole i nizove iz procesa obfuskacija (na primjer API funkcije).

U isječku koda 46 je dan primjer koda prije minifikacije. Kod je pisan u JS-u i sadrži jedno polje i petlju koja ispisuje sadržaj polja.

Isječak kôda 46: Primjer koda prije minifikacije

```
let array = [’jabuka’, ’banana’, ’ananas’];
array.forEach((element, index) => {
console.log(‘Fruit at index ${index}: ${element}‘);
});
```

U isječku koda 47 je dan primjer koda nakon minifikacije. Proces je sažeo cijeli kod u jednu liniju te je uklonio nepotrebne razmake. Razmaci su ostali između riječi let i array te unutar funkcije log.

Isječak kôda 47: Primjer koda nakon minifikacije

```
let array=["apple","banana","cherry"];array.forEach((a,r)=>{console.log(‘Element at
 index ${r}: ${a}‘)});
```

6.1.3. Pozicioniranje JS skripti i CSS stilova

Kod učitavanja web stranica poželjno je progresivno učitavanje, tj. sadržaj i komponente bi se trebale prikazati čim prije. U [33, str. 37–38] je objašnjeno kako je navedeno pogotovo bitno kod stranica koje imaju puno sadržaja ili kod korisnika koji imaju sporiju internetsku vezu. Ako korisnik ne dobiva vizualnu povratnu informaciju kroz progresivno učitavanje stranice, može pomisliti da nešto nije u redu s njom i napustiti je.

Prema [33, str. 38], CSS stilovi bi se trebali postaviti unutar <head> elementa korištenjem <link> elementa kao što je prikazano u isječku koda 48. Naime, ako bi se CSS stil postavio na kraj HTML dokumenta, korisniku bi se prvo prikazao sadržaj bez primijenjenog stila. Nakon što se učita CSS stil, sadržaj će se ponovno prikazati s primijenjenim stilom. Takvo ponašanje se naziva bljesak nestiliziranog sadržaja (eng. *flash of unstyled content*) i trebalo bi ga izbjegavati.

Isječak kôda 48: Učitavanje CSS stilova

```
<head><link rel="stylesheet" href="style.css"></head>
```

Za razliku od CSS stilova, JS skripte se treba pozicionirati na kraj HTML dokumenta. Kada web preglednik tijekom učitavanja komponenti naiđe na skriptu, obustavlja se preuzimanje svih nadolazećih komponenti dok se ne preuzme skripta što utječe na performanse. Prema [33, str. 48], razlozi zbog kojih web preglednici rade na opisani način su sljedeći:

- moguće je da skripta pozove document.write - document.write je funkcija koja omogućava skripti da izravno uvrsti dodatni sadržaj u HTML tijekom učitavanja stranice. Izmijena sadržaja koji se nije do kraja učitao, može prouzročiti nepredvidive pogreške u samom prikazu,
- ispravan redoslijed izvršavanja skripti - moguće je da stranica treba učitati više skripti koje ovise jedna o drugoj. Ako skripta a.js ovisi o b.js i ako se a.js skripta zbog manje veličine učita prije nego li završi učitavanje skripte b.js, nastat će pogreške u JS-u i stranica neće ispravno raditi. Zbog toga će preglednik prvo pričekati da se u potpunosti preuzme skripta b.js i tek će onda započeti preuzimanje skripte a.js.

Pravilo pozicioniranja JS skripti na kraj HTML dokumenta se može zaobići upotrebom atributa defer u script elementu. Bitno je napomenuti da se defer atribut može upotrijebiti samo nad vanjskim JS skriptama [33, str. 50].

6.1.4. Tehnika lijenog učitavanja (eng. *Lazy loading*) slika

Web preglednici rade na način da čim naiđu na element <img>, odmah započinju proces preuzimanja slike. Prema [35, str. 93, 95], samo je 38% sadržaja prosječne stranice vidljivo nakon učitavanja, dok čak 80% dohvaćenih slika nije vidiljivo. Problem je izraženiji na mobilnim uređajima zbog manjih ekrana koji prikazuju manje sadržaja u usporedbi s monitorima. Također, većina korisnika neće listati (eng. *scroll*) do kraja stranice, što dodatno povećava nepotrebno opterećenje resursa.

Rješenje problema je u atributu **loading** u elementima <img> i <iframe> koji može imati vrijednost eager ili lazy. Razlika je u tome što će loading="eager" dohvatiti sliku odmah, a loading="lazy" će odgoditi dohvaćanje slike dok korisnik ne dođe blizu nje listanjem. U isječku koda 49 je dan primjer kako se može postaviti atribut loading unutar elementa <img> [36].

Isječak kôda 49: Korištenje loading atributa u elementu <img>

```
<img loading="lazy" src="image.jpg" />
```

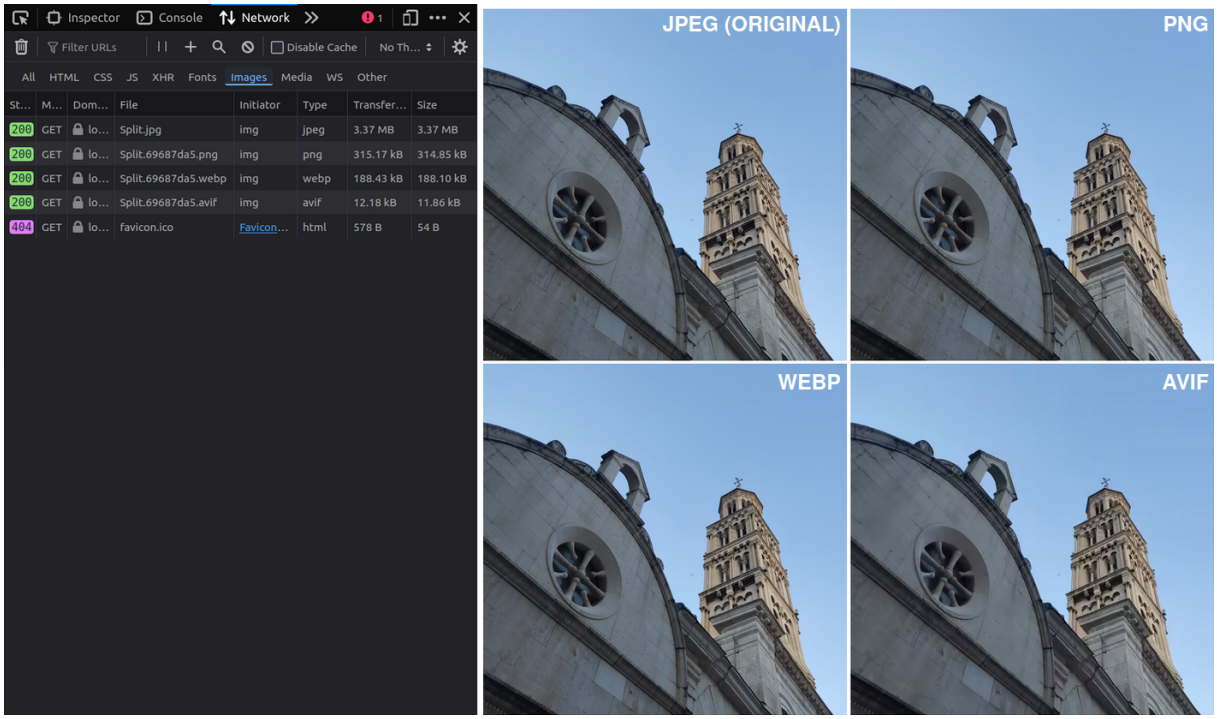
6.1.5. Kompresija slika

Kompresija slike može biti gubitna (eng. *lossy*) ili bezgubitna (eng. *lossless*). Gubitna kompresija smanjuje veličinu slike uklanjanjem informacija koje utječu na kvalitetu slike, pazeći pritom da se uklone samo one informacije koje nisu lako primjetne ljudskom oku. Bezgubitna kompresija smanjuje veličinu slike uklanjanjem metapodataka (npr. podaci o vremenu i lokaciji snimanja slike), što omogućuje očuvanje kvalitete slike. Formati za slike se mogu podijeliti prema tome uzrokuju li gubitak informacija tijekom procesa kompresije ili ne. Na primjer, JPEG format spada pod gubitnu kompresiju dok PNG spada pod bezgubitnu. Postoje i formati koji podržavaju oba tipa kompresije poput WEBP-a i AVIF-a [35, str. 22–23].

Formati WEBP i AVIF spadaju pod novije formate. Prema [35, str. 63], jedini nedostatak novih formata je njihova podrška u web preglednicima. Knjiga je pisana 2016. godine i situacija se od tada poboljšala. Godine 2016. je postojala podrška kod određenih verzija preglednika Chrome, Android i Opera. Podrška nije postojala za Safari, Firefox, Internet Explorer i Edge. Prema [37], na dan 27. sprnja 2024. 96.91% korisnika je koristilo preglednik koji podržava WEBP format. Jedini preglednik koji ga ne podržava je Internet Explorer koji čini samo 0.53% tržišnog udjela, a u 2022. godini je najavljen prestanak njegove podrške. Najveća prednost novih formata je podrška za transparentnost uz mogućnost gubitne kompresije. Jedini tradicionalni format koji omogućava transparentnost je PNG, no on ne podržava gubitnu kompresiju, što rezultira većim slikama [35, str. 53].

Pimcore pruža funkcionalnost optimiziranja i konverzije formata slike korištenjem umanjenih slika. Na slici 24 je prikazano učitavanje četiriju slika. Prva slika je u originalnom JPEG formatu i njezina veličina bez korištenja umanjene slike iznosi 3.37 MB. Druga slika je nastala primjenom umanjene slike koja je promijenila format slike u PNG. Veličina slike se drastično smanjila s 3.37 MB na 314.45 kB. Treća slika je nastala primjernom umanjene slike koja je promijenila format slike u WEBP čime je veličina slike smanjena na 188.10 kB. Posljednja slika koristi AVIF format i ima najmanju veličinu od 11.86 kB.

Usporedbom kvalitete, slike u PNG i WEBP formatima najviše nalikuju originalnoj slici, dok se kod AVIF formata može primijetiti smanjenje kvalitete. Odabir formata ovisi o cilju aplikacije. Ako je prioritet zadržavanje veće kvalitete slike, WEBP format je bolji izbor, dok je AVIF odličan izbor kada je brzina učitavanja slike važnija od same kvalitete.



Slika 24: Veličine slike nakon korištenja umanjene slike

6.2. Optimizacija performansi na strani servera

Unatoč tome što je u kasnijem razvoju projekta jednostavnije poboljšati performanse na strani klijenta, ne smiju se zanemariti moguća poboljšanja na serveru. Optimizacija performansi na strani servera je ključna za osiguravanje bržeg odziva aplikacije, učinkovitije iskorištavanje resursa i smanjenje opterećenja. U nastavku su obrađene neke od tehnika kojima se mogu optimizirati performanse na strani servera: spremanje podataka u međuspremnik, korištenje indeksa u bazi podataka i optimizacija upita.

6.2.1. Međuspremnik

Međuspremnik je sustav pohrane podataka koji se često koristi za brzo dohvaćanje podataka. Cilj je smanjiti vrijeme odgovora na zahtjeve i poboljšati perfromanse aplikacije [38]. Prema [39], Pimcore po zadanom koristi Doctrine za spremanje podataka u tablicu cache_items, ali moguće je konfigurirati da se umjesto Doctrine koristi Redis.

Redis je baza podataka koja koristi radnu memoriju za pohranu podataka. Pruža cloud i on-premise rješenja za spremanje podataka u međuspremnik, vektorsko pretraživanje i NoSQL baze podataka. Spremanjem podataka u radnu memoriju postižu se bolje performanse jer je dohvaćanje podataka brže nego s diska ili udaljene pohrane [38] [40].

Redis se jednostavno može instalirati i postaviti, pogotovo ako se koristi Docker. U isječku koda 50 se nalazi konfiguracija koju je potrebno dodati u docker-compose.yaml kako bi se kreirao kontejner koji će pokrenuti Redis.

Isječak kôda 50: Konfiguracija za Redis u docker-compose.yaml datoteci

```
services:
    redis:
        image: redis:alpine
        command: [ redis-server, --maxmemory 128mb, --maxmemory-policy volatile-lru,
          --save "" ]
        ports:
            - "6379:6379"
```

Zatim je potrebno u config.yaml dodati konfiguraciju iz isječka koda 51 kako bi Pimcore kao međuspremnik koristio Redis umjesto Doctrine.

Isječak kôda 51: Konfiguracija za Redis u config.yaml datoteci

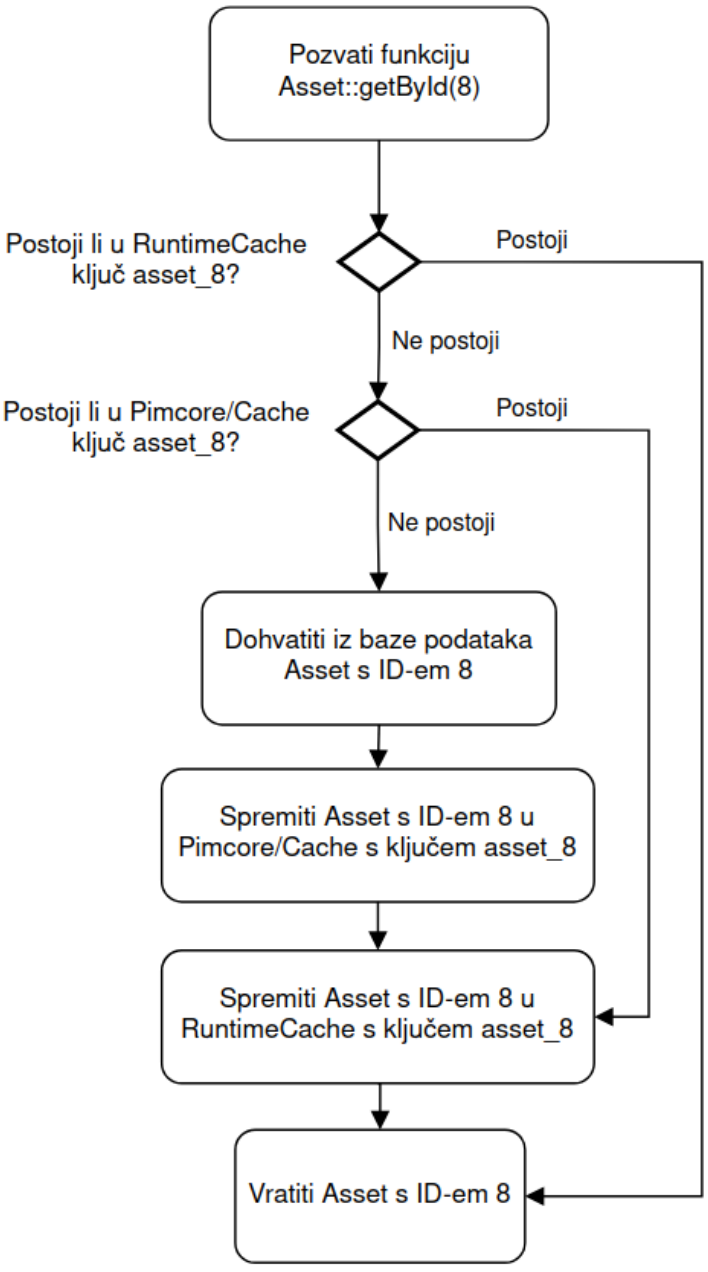
```
framework:
    cache:
    pools:
        pimcore.cache.pool:
            public: true
            default_lifetime: 31536000  # 1 year
            adapter: cache.adapter.redis_tag_aware
            provider: 'redis://redis'
```

Na slici 25 je prikazan pojednostavljeni proces dohvaćanja objekata iz međuspremnika unutar Pimcora. Za rad s međuspremnikom je bitno spomenuti dvije klase:

- Pimcore/Cache - poziva funkcije iz drugih klasa (kao što su CoreCacheHandler, RedisTagAwareAdapter itd.) koje pohranjuju podatke u međuspremnik koristeći Doctrine ili Redis. Također, služi za dohvaćanje podataka iz međuspremnika.
- Pimcore/Cache/RuntimeCache - instancira se pri dolasku zahtjeva i postoji samo za vrijeme njegovog izvršavanja. Slijedi Singleton dizajn i sprema podatke vraćene iz Pimcore/Cache u međuspremnik.

Kada se prvi put pokuša dohvatiti digitalna imovina s ID-om 8, potrebno ju je dohvatiti iz baze podataka. Nakon toga se s digitalne imovine dohvaćaju vrijednosti atributa type i id, te se generira ključ asset_8. Zatim klasa Cache poziva funkcije koje pohranjuju digitalnu imovinu u Redis pod ključem asset_8. Digitalna imovina se pohranjuje i unutar RuntimeCache objekta.

Ako se tijekom izvršavanja istog zahtjeva ponovno pozove funkcija Asset::getById(8), digitalna imovina će se dohvatiti iz objekta RuntimeCache. Ako se funkcija Asset::getById(8) pozove tijekom izvršavanja novog zahtjeva, ovaj put će se digitalna imovina dohvatiti iz Redisa, jer ju se već spremilo u međuspremnik tijekom prethodnog zahtjeva.



Slika 25: Proces dohvaćanja objekta iz međuspremika

Pimcore omogućava programerima da koriste funkcije iz klase Pimcore/Cache unutar koda aplikacije za pohranu podataka u međuspremnik ili njihovo dohvaćanje iz međuspremnika [39]. Primjer je prikazan u isječku koda 52.

Isječak kôda 52: Spremanje podataka u međuspremnik

```
$cacheKey = 'products_for_caching';
if (!$data = \Pimcore\Cache::load($cacheKey)) {
    $data = $this->productRepository->getProductsForCaching();
    \Pimcore\Cache::save($data, $cacheKey, ['tag_1'], 259200);}
```

Za brisanje podataka iz međuspremnika se može pozvati komanda 'bin/console pimcore:cache:clear', ali je važno napomenuti da ona neće izbrisati podatke koji su spremljeni u međuspremnik na način prikazan u isječku koda 52.

6.2.2. Indeksi

Prema [41, str. 155], indeks je struktura podataka koja se koristi za brže pretraživanje redova u bazi podataka. Ključni su za dobar performans i postaju sve važniji kako se podaci u bazi povećavaju. U isječku koda 53 se nalazi primjer naredbe za kreiranje indeksa;

Isječak kôda 53: Naredba za kreiranje indeksa

```
CREATE INDEX index_name ON table_name (column1, column2, ...);
```

U isječku koda 54 se nalaze naredbe za kreiranje tablice customers i indeksa customers_data. Indeks je kreiran za sve stupce u tablici i jako je bitan njihov poredak jer o tome ovisi kod kojih upita će se moći primijeniti indeks, a kod kojih neće.

Isječak kôda 54: Kreiranje tablice i indeksa

```
CREATE TABLE customers (first_name varchar(50) not null, last_name varchar(50) not
 null, date_of_birth date not null);
CREATE INDEX customers_data ON customers (last_name, first_name, date_of_birth);
```

U isječku koda 55 se nalaze primjeri upita kod kojih se indeks neće primijeniti. U [41, str. 159–160] je objašnjeno pravilo gdje se indeks ne može primijeniti ako upit u WHERE dijelu sadrži samo stupac date_of_birth bez stupaca first_name i last_name ili ako sadrži date_of_birth i first_name bez stupca last_name. Mora se slijediti točan redoslijed stupaca iz naredbe CREATE INDEX s lijeva na desno bez preskakanja.

Isječak kôda 55: Upiti kod kojih se neće primijeniti indeks customers_data

```
1 SELECT * FROM customers WHERE date_of_birth = '2000-01-01'
2 SELECT * FROM customers WHERE first_name = 'Hrvoje' AND date_of_birth = '2000-01-01'
```

U isječku koda 56 se nalaze primjeri kod kojih će se u potpunosti ili djelomično primijeniti indeks. Na primjer, u zadnjem upitu se pretražuje po stupcima last_name i date_of_birth bez stupca first_name. U tom slučaju se indeks primjenjuje samo nad stupcem last_name, a date_of_birth se pretražuje kako bi se pretražio i bez indeksa. Također, u trećem upitu je dan primjer s LIKE 'H%' gdje će se također u potpunosti primijeniti indeks [41, str. 159–160].

Isječak kôda 56: Upiti kod kojih će se primijeniti indeks customers_data

```
1 SELECT * FROM customers WHERE last_name = 'Horvat' AND first_name = 'Hrvoje' AND
   date_of_birth = '2000-01-01';
2 SELECT * FROM customers WHERE last_name = 'Horvat' AND first_name = 'Hrvoje':
3 SELECT * FROM customers WHERE last_name = 'Horvat' AND first_name LIKE 'H%':
4 SELECT * FROM customers WHERE last_name = 'Horvat';
5 SELECT * FROM customers WHERE last_name LIKE 'H%';
6 SELECT * FROM customers WHERE last_name LIKE 'H%' AND date_of_birth = '2000-01-01';
```

Iako indeksi mogu značajno ubrzati dohvaćanje podataka, dodavanjem prevelikog broja indeksa se može stvoriti kontra efekt kod ažuriranja podataka. Prema [41, str. 190], prilikom dodavanja novih ili poboljšavanja postojećih indeksa, važno je analizirati vrijeme izvršavanja korištenih upita. Također, treba razmotriti broj redaka koje upit zahvaća tijekom pretraživanja kako bi se donijela ispravna odluka o kreiranju indeksa.

6.2.3. Optimizacija upita

U prethodnom poglavlju su pokazane prednosti korištenja indeksa, ali čak i uz pravilno postavljene indekse, neefikasno napisani upiti mogu rezultirati značajnim smanjenjem performansi. U nastavku je dano nekoliko jednostavnih, ali učinkovitih savjeta za pisanje upita. Iako se mogu činiti trivijalnim, primjena ovih savjeta može značajno ubrzati vrijeme izvršavanja upita.

U [41, str. 193] je objašnjena važnost upotrebe ključne riječi LIMIT. Na primjer, ako tablica filmovi sadrži 100 redaka i izvrši se upit koji dohvaća sve filmove bez korištenja ključne riječi LIMIT, performanse neće biti značajno narušene. Problem nastaje kada tablica sadrži na tisuće redaka. Bez dodavanja ključne riječi LIMIT, iz baze podataka će se dohvatiti svi redovi, što ne opterećuje samo bazu podataka, već i server na kojem se izvodi aplikacija i internetsku vezu ako je baza na udaljenom serveru.

Zatim, ako se želi provjeriti postoji li neki podatak u tablici, moguće je koristiti kombinaciju COUNT(*) u SELECT dijelu s uvjetom u WHERE dijelu. Ako upit vrati veći broj od nula, traženi podatak postoji u tablici. Iako je cilj postignut, ovo rješenje nije optimalno jer funkcija COUNT mora pregledati sve redove u tablici koji zadovoljavaju uvjet prije nego što vrati konačan rezultat. Bolji pristu bi bio dodati LIMIT 1 u upit čime se postiže da se rezultat vrati čim se pronađe prvo podudaranje u tablici. U tom slučaju je dovoljno provjeriti je li upit vratio bilo kakav rezultat ili nije vratio ništa.

Sljedeći savjet se odnosi na SELECT dio. Prilikom dohvaćanja podataka, važno je razmotriti koji su stupci doista potrebni i je li potrebno dohvatiti sve stupce uz SELECT *. Ovo je posebno važno kada se povezuje više tablica korištenjem ključne riječi JOIN. Umjesto korištenja SELECT *, preporuča se specificirati točno one stupce koji su potrebni, navodeći tablicu i stupce [41, str. 193].

U ovom poglavlju su navedena samo neka od pravila koja mogu znatno poboljšati performanse upita. Naravno ovdje nisu nabrojani svi mogući načini optimizacije, te se preporuča dodatno proučavanje literature, poput knjige [41], za detaljnije informacije i savjete o optimizaciji upita.

7. Izrada aplikacije

U nastavku poglavlja je dan pregled korištenih tehnologija pri izradi aplikacije, opisane su funkcionalnosti po ulogama te je prikazan ERA model. Nakon toga je istaknuto par zanimljivih dijelova implementacije te je na kraju napravljena analiza mjerenja performansa aplikacije. Cilj analize je usporediti ubrzanje rada nakon primjene nekih od tehnika navedenih u prethodnom poglavlju.

7.1. Korištene tehnologije i arhitektura aplikacije

Na strani servera su korišteni aplikacijski okvir **Symfony** i platforma **Pimcore**, a na strani klijenta je korišten programski jezik **JavaScript** u kombinaciji s alatom **Webpack** koji olakšava minifikaciju JS datoteka. Od korištenih biblioteka vrijedi spomenuti **Navigo** i **Leaflet**. Navigo je biblioteka koja pojednostavljuje upravljanje rutama i o njoj je više rečeno u kasnijim poglavljima, a biblioteka Leaflet[1] pruža jednostavan način za interakciju s mapama unutar aplikacije.

Za upravljanje relacijskom bazom podataka odabran je **MySQL**, a cijeli projekt je organiziran korištenjem **Dockera**, što omogućava jednostavnije pokretanje i prenosivo razvojno okruženje. Docker je programski alat koji se može pokrenuti na Linux i Windows operacijskim sustavima, a služi za kreiranje, upravljanje i orkestraciju kontejnera (eng. *containers*). Kontejneri su slični virtualnim mašinama, ali ne zahtijevaju čitav operacijski sustav, već svi kontejneri na jednom domaćinu dijele isti operacijski sustav. Time se štedi na resursima poput procesora, radne memorije i prostora za pohranu, smanjuju se troškovi licenciranja i održavanja sustava. One koji žele znati više se upućuje na knjigu Docker Deep Dive od Nigela Polutona [42]. Za pokretanje Docker kontejnera potrebno je u korijenu projekta pokrenuti komandu 'docker-compose up'.

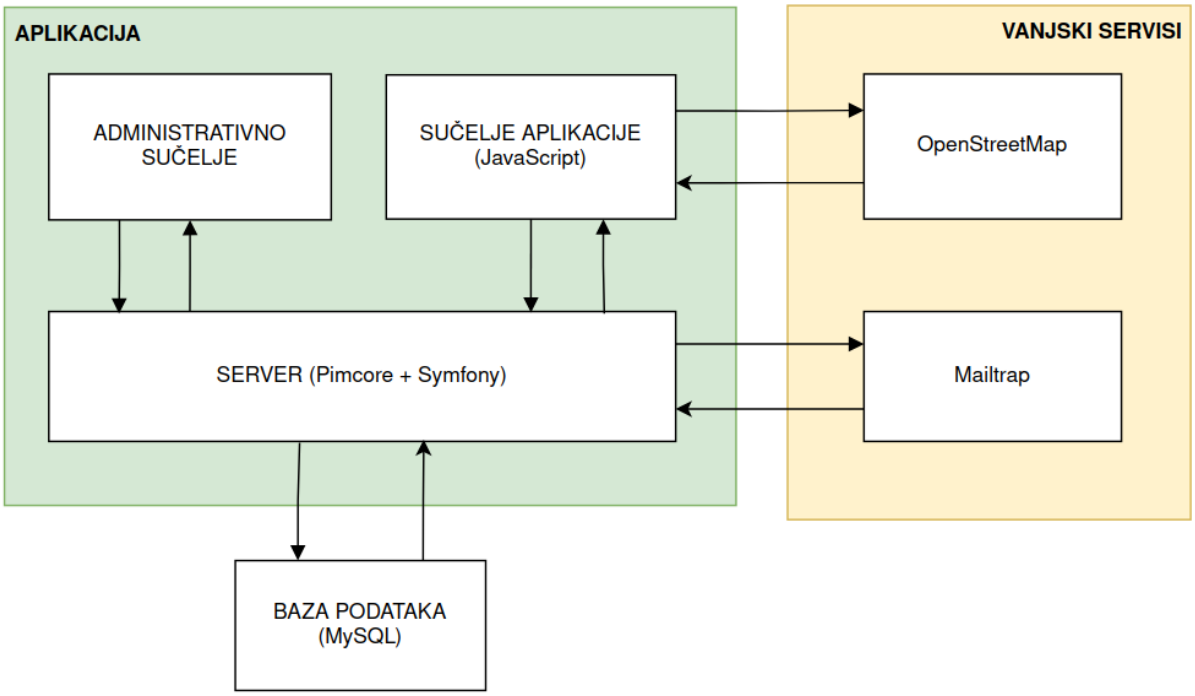
U isječku koda 57 se nalaze naredbe za rad s alatom Webpack koje su definirane u package.json datoteci. Naredba watch služi za kontinuirano praćenje nastalih promjena u JS i CSS datotekama, a naredbe development i production za minifikaciju. Naredba development će grupirati odabrane JS i CSS datoteke, ali ih za razliku od naredbe production, neće minificirati. Na taj način se olakšava pronalaženje grešaka i ubrzava vrijeme kompajliranja datoteka tijekom razvoja.

[1] Na stranici `https://leafletjs.com/` se nalaze dokumentacija i smjernice za instalaciju Leaflet biblioteke

Isječak kôda 57: Naredbe za rad s alatom Webpack

```
1 "watch": "npm run development -- --watch",
2 "development": "cross-env NODE_ENV=development node_modules/webpack/bin/webpack.js
   --progress --config=node_modules/laravel-mix/setup/webpack.config.js",
3 "production": "cross-env NODE_ENV=production node_modules/webpack/bin/webpack.js --
   config=node_modules/laravel-mix/setup/webpack.config.js"
```

Na slici 26 se nalazi arhitektura aplikacije s nekim od korištenih tehnologija. Aplikaciju čine administrativno sučelje koje dolazi s instalacijom platforme Pimcore, sučelje izrađene aplikacije u JavaScriptu te poslužitelj za čiju su izradu korišteni Pimcore i Symfony. Poslužitelj je zadužen za komunikaciju sa sučeljima, bazom podataka i vanjskim servisom Mailtrap za slanje e-pošte. Sučelje izrađene aplikacije, osim što komunicira s poslužiteljom, šalje zahtjev za prikazom mape na vanjski servis OpenStreeMap uz pomoć već spomenute bibioteke Leaflet.



Slika 26: Arhitektura aplikacije

7.2. Funkcionalnosti

U nastavku su opisane funkcionalnosti koje nudi izrađena aplikacija po ulogama. Glavna namjena aplikacije je omogućiti korisnicima pretraživanje i učlanjivanje u planinarska društva te prijavljivanje za izlete. Administrativnom sučelju mogu pristupiti korisnici s ulogom administratora, vodiča ili moderatora planinarskog društva, a aplikaciji mogu pristupiti neregistrirani i registrirani korisnici pri čemu registrirani korisnici imaju pristup većem broju funkcionalnosti.

Bitno je za napomenuti da administracija i aplikacija koriste različit autentifikacijski proces. Na primjer, ako vodič želi pristupiti aplikaciji kao registrirani korisnik, mora se registrirati iako već ima profil kojim može pristupiti administraciji.

Neregistriranim korisnicima je omogućeno pretraživati planinarska društva, pregledati osnovne informacije o planinarskom društvu, pregledati izlete i detalje izleta, registrirati se i prijaviti se. U nastavku su detaljnije objašnjene pojedinosti svake funkcionalnosti:

- **Pretraživanje planinarskih društava** - korisnik može pretražiti planinarska društva po nazivu i filtrirati ih po gradu. Ponuđeni su samo gradovi dostupni u aplikaciji. Nakon odabira grada ili unesenog pojma u tražilicu, prikazat će se loga i nazivi planinarskih društava s omogućenom paginacijom. Klikom na logo ili naziv društva, korisnika se vodi na prikaz s izletima odabranog duštva.
- **Pregled osnovnih informacija planinarskog društava** - korisnik može pregledati dostupne sekcije i izlete planinarskog društva te pristupiti informacijama o učlanjenju. Također, može pregledati kontakt podatke i novosti koje je objavilo planinarsko društvo.
- **Pregled izleta planinarskog društva** - korisnik može pregledati izlete koje nudi planinarsko društvo. Izlete je moguće pretražiti po nazivu i filtirati po sekciji, vrsti izleta i datumu polaska. Pod vrstom izleta se može odabrati opcija Nadolazeći ili Završeni. Također, moguće je sortirati izlete po datumu polaska, uzlazno ili silazno. U tablici s izletima se nalaze podaci o nazivu izleta, sekciji, datumu polaska i lokaciji koju se planira posjetiti, a omogućena je i paginacija.
- **Pregled detalja izleta** - korisnik može pristupiti detaljima kliknom na naziv izleta unutar tablice s izletima. Nakon toga će se učitati podaci o datumu polaska i povratka, opis izleta, detalji o prijevozu i popunjenosti kapaciteta izleta te popis vodiča zaduženih za izlet. Prikazana je i mapa s lokacijom koju se planira posjetiti te forma za plaćanje s detaljima o cijeni izleta.
- **Registracija** - tijekom procesa registracije, korisnik treba unijeti ime, prezime, OIB, e-poštu i lozinku. Nakon uspješne registracije, korisnika će se preusmjeriti na formu za prijavu.
- **Prijava** - korisnik se može prijaviti u aplikaciju korištenjem e-pošte i lozinke koju su unijeli na registracijskoj formi. Moguće je odabrati opciju Zapamti me, koja će

spremiti korisničku sesiju na 7 dana. Ako korisnik ne pristupi aplikaciji nijednom u 7 dana, morat će se ponovno prijaviti.

Registriranom korisniku je omogućeno učlaniti se u planinarsko društvo, prijaviti se na izlet te pregledati vlastiti profil, prijavljene izleta te kreirane članarine.

- **Učlanjenje u planinarsko društvo** - korisnik može ispuniti formu za učlanjenje u planinarsko društvo. Forma nudi dvije opcije plaćanja: Uživo ili Online. Ako je odabrana opcija Online, korisnik mora priložiti potvrdu o plaćanju. Kraj forme su izlistane cijene članarine koje postavlja planinarsko društvo kroz administraciju.
- **Prijava na izlet** - kod pregleda detalja izleta, korisnik se može zabilježiti za izlet ako još ima mjesta. Korisnik se može prijaviti za izlet iako nije član planinarskog društva. Proces plaćanja za izlet je identičan kao i kod učlanjivanja.
- **Moj profil** - korisnik može pristupiti svom profilu i ažurirati svoje podatke (ime, prezime, OIB). Također, putem svog profila može pregledati svoje trenutne i prošle članarine te izlete na koje su se prijavili.

Korisnik s ulogom **vodiča** može pristupiti administrativnom sučelju i pregledati objekte pridružene društvu. Vodič može kreirati nove izlete koji će se prikazati u aplikaciji nakon što ih odobri moderator. Vodič ne može brisati izlete.

Moderator je zadužen za odobravanje članstava i prijava za izlet te ažuriranje glavnih informacija planinarskog društva. Pod glavne informacije spadaju cijene članarina, dostupne sekcije, novosti, kontaktni podaci i izleti. Moderator ima pristup sljedećim izvještajima u administraciji:

- **Broj članova po godinama (zadnje 4 godine)** - izvještaj uzima u obzir zadnje 4 godine i prikazuje koliko se ukupno članova pridružilo tijekom svake godine.
- **Broj članova po mjesecima (zadnjih 12 mjeseci)** - izvještaj uzima u obzir zadnjih 12 mjeseci i prikazuje koliko se ukupno članova pridružilo tijekom svakog mjeseca.
- **Broj prijava na izlete po mjesecima (zadnjih 12 mjeseci)** - izvještaj uzima u obzir zadnjih 12 mjeseci i prikazuje koliko je ukupno prijava nastalo za bilo koji izlet tijekom svakog mjeseca.

Admin je zadužen za dodavanje moderatora i vodiča u administraciju. Nakon dodavanja, admin će dodijeliti uloge korisnicima i postaviti dodatna prava ovisno o planinarskom društvu. Osim navedenog, admin je zadužen i za kreiranje početne strukture objekata planinarskih društava te za dodijeljivanje izvještaja planinarskim društvima.

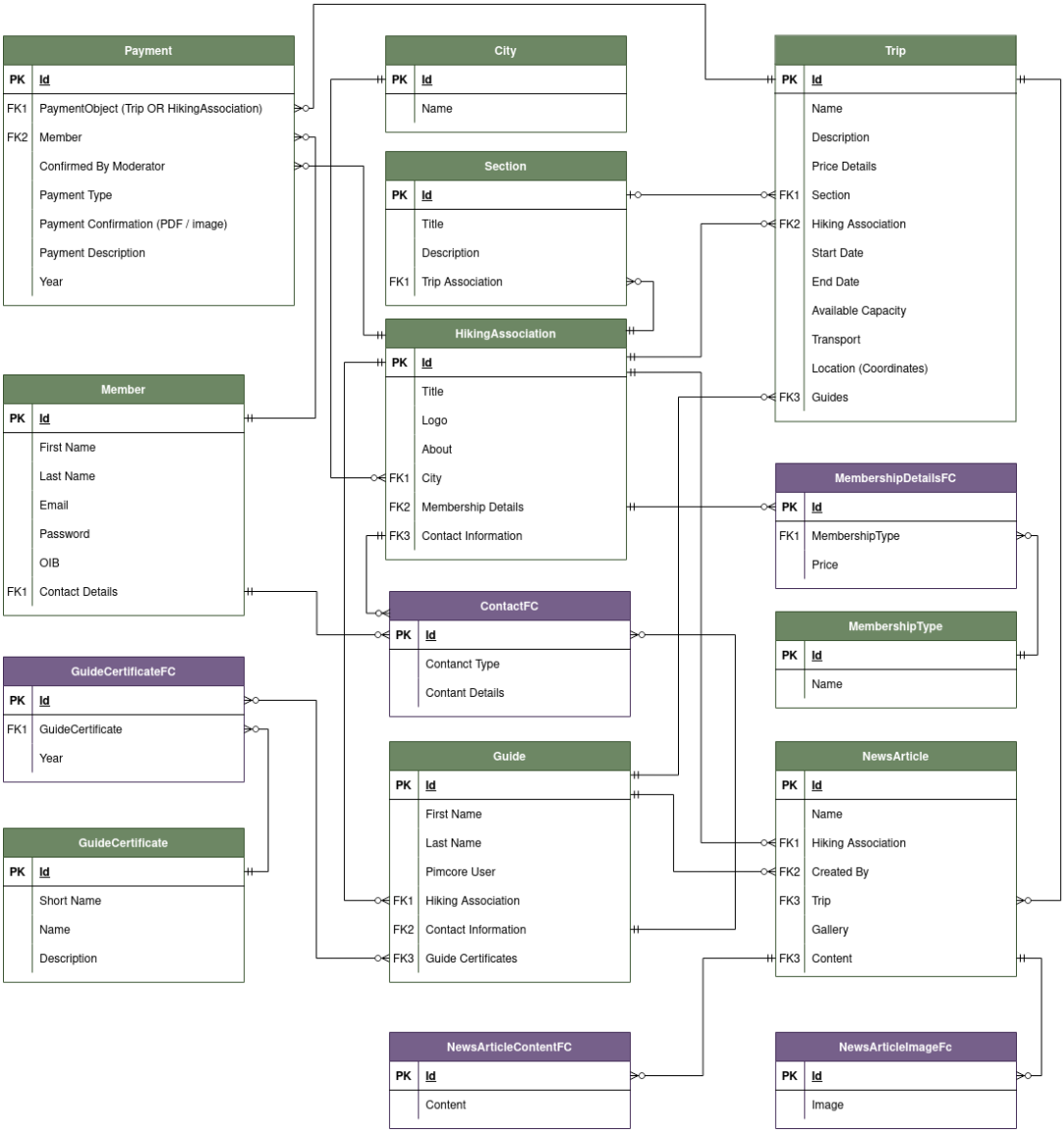
7.3. ERA model

Na slici 27 je prikazan pojednostavljeni ERA model aplikacije gdje su glavne tablice obojane zelenom bojom, a kolekcije ljubičastom. ERA model je pojednostavljen na način da je na slici prikazana jedna tablica za svaku klasu, ali Pimcore će u pozadini za svaku klasu kreirati minimalno tri tablice. Na primjer, za klasu HikingAssociation će se kreirati [43]:

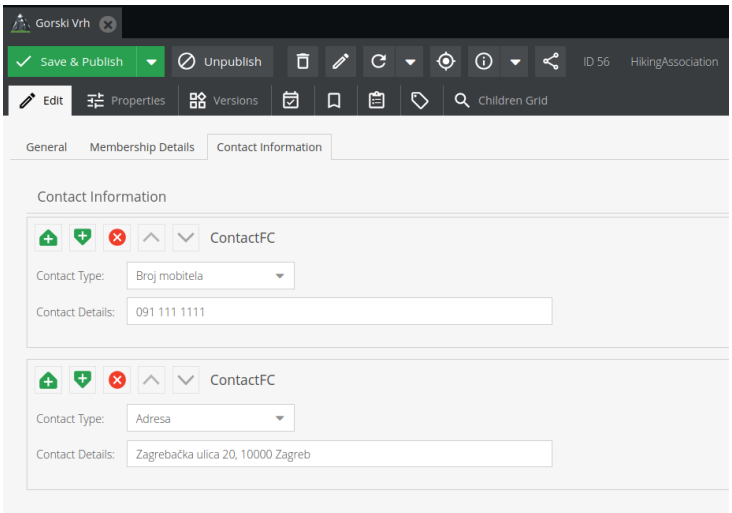
- object_store_HikingAssociation - sadrži samo jednostavne podatke poput teksta, brojeva i sl.
- object_relation_HikingAssociation - sadrži podatke o relacijama.
- object_query_HikingAssociation - sadrži jednostavne podatke i relacije, pri čemu su relacije spremljene u serijaliziranom obliku.

Prema [43], dodatno će se kreirati i pogled object_HikingAssociation koji će se iskoristiti za dohvaćanje podataka uz pomoć objekta klase Listing. Pogled čini spoj query i objects tablice. Tablica objects sadrži osnovne podatke svih podatkovnih objekata u aplikaciji.

Na slici 27 se vidi kako je tablica HikingAssociation povezana s dvije kolekcije: ContactFC i PaymentDetailsFC, a na slici 28 se nalazi primjer kolekcije ContactFC na objektu HikingAssociation. Prema [44], kolekcije su objekti koji se mogu kreirati unutar objekta u proizvoljnoj količini te se spremaju u posebne tablice prema strukturi object_collection_NAZIV-KOLEKCIJE_OBJECT-ID. U primjeru s klasom HikingAssociation, kreirat će se dvije tablice object_collection_ContactFC_HikingAssociation i
object_collection_PaymentDetailsFC_HikingAssociation.



Slika 27: ERA model aplikacije



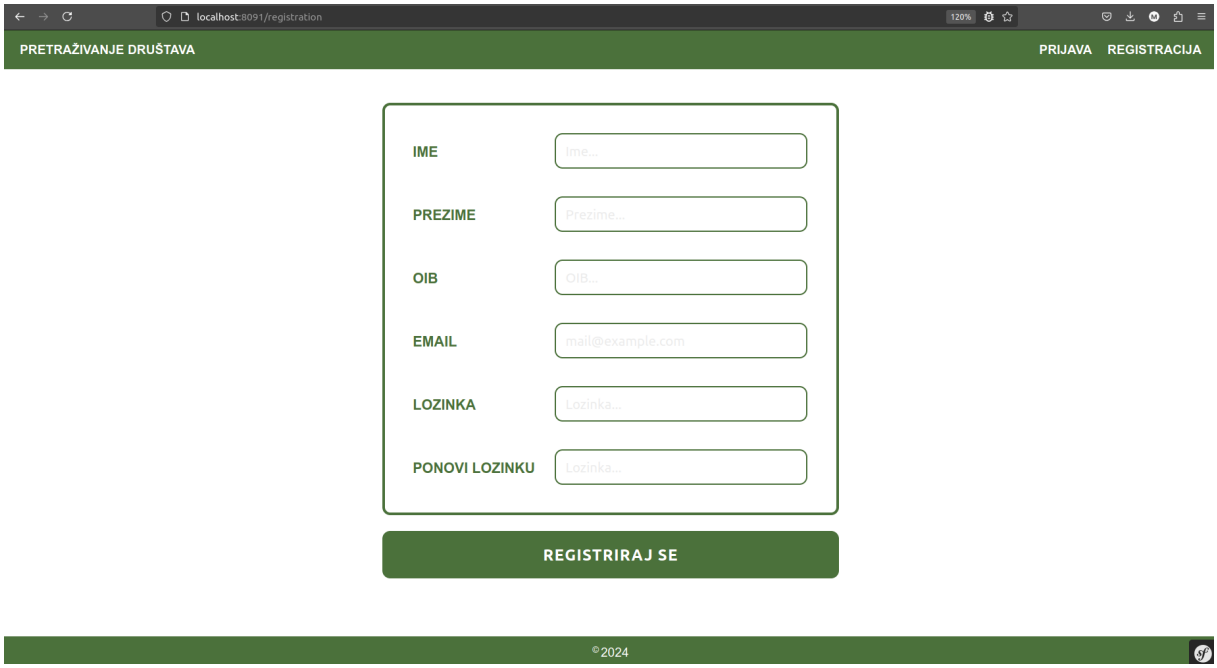
Slika 28: Primjer kolekcije (eng. *field collection*)

7.4. Istaknuti dijelovi implementacije

U nastavku je dan pregled triju izdvojenih dijelova implementacije. Prikazano je kako je postignuta jednostraničnost aplikacije, objašnjena je konfiguracija koja omogućava korisnicima prijavu u aplikaciju, te je na kraju pokazano kako je kod forme za plaćanje postignuta ponovna upotrebljivost bez kršenja SOLID principa.

7.4.1. Jednostraničnost

Kako bi se postigao bolji performans rada aplikacije, aplikacija je izrađena slijedeći principe jednostraničnosti uz pomoć JS biblioteke Navigo. U nastavku je dan primjer korištenja biblioteke kod prikaza registracije sa slike 29.



Slika 29: Prikaz registracije unutar aplikacije

Prikazu sa slike 29 se može pristupiti preko putanje /registration koja je povezana s kontrolerom iz isječka koda 58.

Isječak kôda 58: Kontroler za učitavanje forme za registraciju

```
#[Route('/registration', name: 'registration')]
public function registration(Request $request) {
    if (!$this->isAjaxRequest($request))
        return $this->getMainFrameView();
    $form = $this->createForm(MemberFormType::class, new Member());
```

```
6       $form->handleRequest($request);
7       if ($form->isSubmitted() && $form->isValid()) {
8           /** @var Member $member */
9           $member = $form->getData();
10          $this->memberRepository->createMember($member);
11          return $this->respondWithSuccess(['redirectUrl' => $this->generateUrl('login
             ')]);}
12      $htmlString = $this->renderView('auth/registration.html.twig', ['form' => $form
         ->createView()]);
13      return $this->respondWithSuccess(['html_string'=>json_encode($htmlString)]);}
```

U slučaju da korisnik osvježi stranicu dok se nalazi na URL-u iz slike 29, kontroler neće primiti parametar ajax i pozvat će se funkcija getMainFrameView. Funkcija će vratiti predložak default/default.html.twig koji sadrži glavne elemente poput <html>, <head>, <body> itd. Osim navedenog, predložak sadrži i navigaciju te jedan glavni element . Navedeno je prikazano u isječku koda 59.

Isječak kôda 59: Isječak koda iz Twig predloška default/default.html.twig

```
<div id="menu">{% include "segments/header.html.twig" with {'hikingAssociation':
 hikingAssociation|default(null)} %}</div>
<div id="main-content"></div>
```

Nakon što se učita JavaScript skripta, izvršit će se kod iz isječka 60 gdje se na početku kreira objekt klase Navigo. Zatim se pomoću funkcije on definira ruta, a na kraju se nalazi uvjet koji provjerava poklapa li se trenutni URL s definiranim rutama unutar instance Navigo. Ako se URL poklapa s putanjom /registration, pozvat će se funkcija init iz klase RegistrationController.

Isječak kôda 60: Primjer korištenja Navigo objekta

```
const Navigator = new Navigo('/')
Navigator.on({
    '/registration': {
    as: 'RegistrationController',
    uses: (match) => {
    Routing.switchRoute(RegistrationController)
    Routing.onInit(match.url)...
if (Navigator.match(Navigator.getCurrentLocation().url)) {
    Navigator.resolve(Navigator.getCurrentLocation().url)}
```

Funkcija init će pozvati funkciju loadRegistrationForm iz isječka koda 61 koja će dohvatiti podatke s rute /registration. Funkcija će poslati i parametar ajax=true zbog kojeg će se u kontroleru umjesto poziva funkcije getMainFrameView, kreirati forma i poslati predložak aut-

h/registration.html.twig. Nakon što stigne odgovor sa servera, sadržaj predloška će se ubaciti u glavni element .

Isječak kôda 61: Funkcija loadRegistrationForm koja ažurira sadržaj HTML-a

```
1 loadRegistrationForm() {
2     this.registrationService.loadRegistrationForm().then((response) => {
3     let html = JSON.parse(response.data.data.html_string)
4     this.replaceView(html)})}
```

Ako je korisnik pristupio registraciji putem navigacije, umjesto učitavanje čitave stranice kao kod osvježavanja, pozvati će se funkcija loadRegistrationForm iz isječka koda 61 koja će ažurirati sadržaj glavnog elementa . Navedeno je moguće, jer poveznica koja vodi na registraciju u sebi sadrži atribut data-navigo. Primjer je da u isječku koda 62.

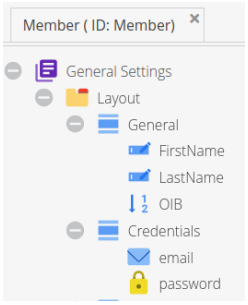
Isječak kôda 62: Primjer korištenja atributa data-navigo

```
1 <a href="{{ path('registration') }}" data-navigo>REGISTRACIJA</a>
```

Atribut data-navigo se dodaje elementima <a> kod kojih se želi spriječiti osvježavanje cijele stranice. Nakon inicijalizacije, Navigo će registrirati sve linkove koji su definirani u elementima <a> s atributom data-navigo. Svaki put kad se ažurira DOM i dodaju novi elementi <a> s atributom data-navigo, treba se pozvati metodu updatePageLinks kako bi Navigo ponovno pregledao DOM i ažurirao linkove. Linkovi s target="_blank" atributom bit će ignorirani čak i ako imaju data-navigo [45].

7.4.2. Prijava

Kako bi prijava mogla funkcionirati, potrebno je kreirati klasu koja će predstavljati korisnika. Na slici 30 se nalazi primjer jedne takve klase nazvane Member. Bitno je da klasa koja predstavlja korisnika sadrži jedinstveni atribut poput korisničkog imena ili e-pošte i atribut u kojem će biti pohranjena lozinka.



Slika 30: Klasa Member

Zatim je potrebno klasu nadjačati i implementirati funkcije iz sučelja UserInterface i PasswordAuthenticatedUserInterface kako bi se objekti klase Member mogli integrirati sa konfiguracijom za prijavu koju nudi Symfony. Navedeno je odrađeno u isječku koda 63 [46].

Isječak kôda 63: Nadjačavanje klase Member

```
class Member extends \Pimcore\Model\DataObject\Member implements UserInterface,
 PasswordAuthenticatedUserInterface {
    public function getRoles(): array {
        return ['ROLE_USER'];}
    public function eraseCredentials(): void {
        /** @var Password $field */
        $field = $this->getClass()->getFieldDefinition('password');
        $field->getDataForResource($this->getPassword(), $this);}
        public function getUserIdentifier(): string {
        return $this->getEmail();}}
```

U isječku koda 64 je prikazana konfiguracija koja omogućava da Pimcore ubuduće koristi novokreiranu klasu Member, koja nadjačava standardnu Pimcorovu klasu Member [46].

Isječak kôda 64: Konfiguracija za nadjačavanje klase Member

```
pimcore:
    models:
        class_overrides:
            'Pimcore\Model\DataObject\Member': 'App\Model\Member'
```

U sljedećem koraku je potrebno definirati servis koji će poslužiti kao pružatelj korisnika (eng. *user provider*). Pružatelj korisnika je objekt koji pronalazi odgovarajući korisnički objekt na temelju korisničkog imena ili e-pošte. Pimcore već nudi klasu ObjectUserProvider koja može poslužiti za traženje korisnika, a potrebno joj je proslijediti klasu koja predstavlja korisnika i naziv jedinstvenog identifikacijskog atributa kao što je prikazano u isječku koda 65 [46].

Isječak kôda 65: Definiranje servisa za pružanje korisnika

```
services:
    security.member_provider:
        class: Pimcore\Security\User\ObjectUserProvider
        arguments: [ 'App\Model\Member', 'email' ]
```

Prema [46], potrebno je definirati i servis koji će se koristiti za provjeravanje i sažimanje korisnikove lozinke. Servis treba implementirati sučelje PasswordHasherInterface pri čemu Symfony po zadanom kreira po jednu instancu objekta koji implementira navedeno sučelje za svaki tip korisnika. U ovom slučaju, cilj je delegirati provjeru i sažimanje lozinke na objekte klase

Member, jer atribut password već sadrži logiku koju zahtjeva sučelje PasswordHasherInterface. Kako bi se logika delegirala na objekte, potrebno je definirati tvornicu kao što je prikazano u isječku koda 66.

Isječak kôda 66: Definiranje tvornice za delegaciju provjere i sažimanja lozinke na objekte klase Member

```
services:
    security.pimcore.password_hasher_factory:
        class: Pimcore\Security\Hasher\Factory\UserAwarePasswordHasherFactory
        arguments:
            - Pimcore\Security\Hasher\PasswordFieldHasher
            - [ 'password' ]
    pimcore:
        security:
            password_hasher_factories:
                App\Model\Member: security.pimcore.password_hasher_factory
```

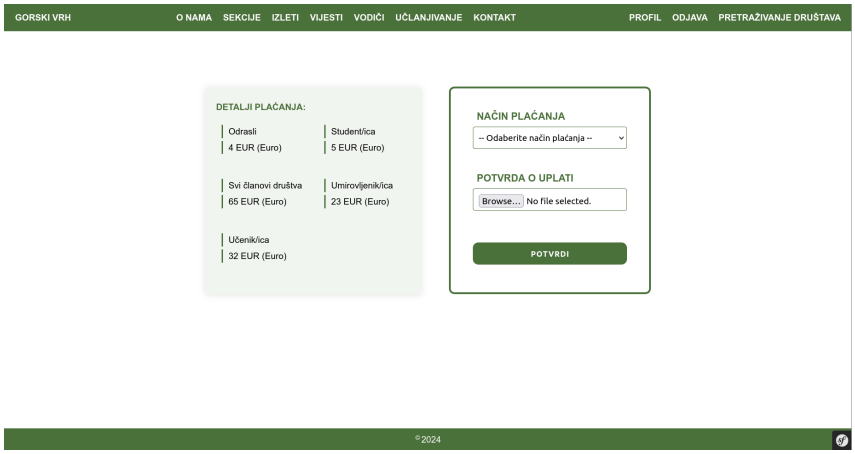
Na kraju je potrebno dodati definiranog pružatelja korisnika u konfiguraciju za sigurnost i povezati ga s određenom konfiuguracijom vatrozida (eng. *firewall*). Vatrozid u Symfony aplikaciji služi za izvršavanje postupka autentifikacije, a u ovom slučaju, autentifikacija se vrši upotrebom forme. U isječku koda 67 je definirano korištenje rute login za prikaz forme za prijavu, rute login_check za podnošenje forme i rute logout za odjavljivanje iz aplikacije.

Isječak kôda 67: Primjer konfiuguracije vatrozida

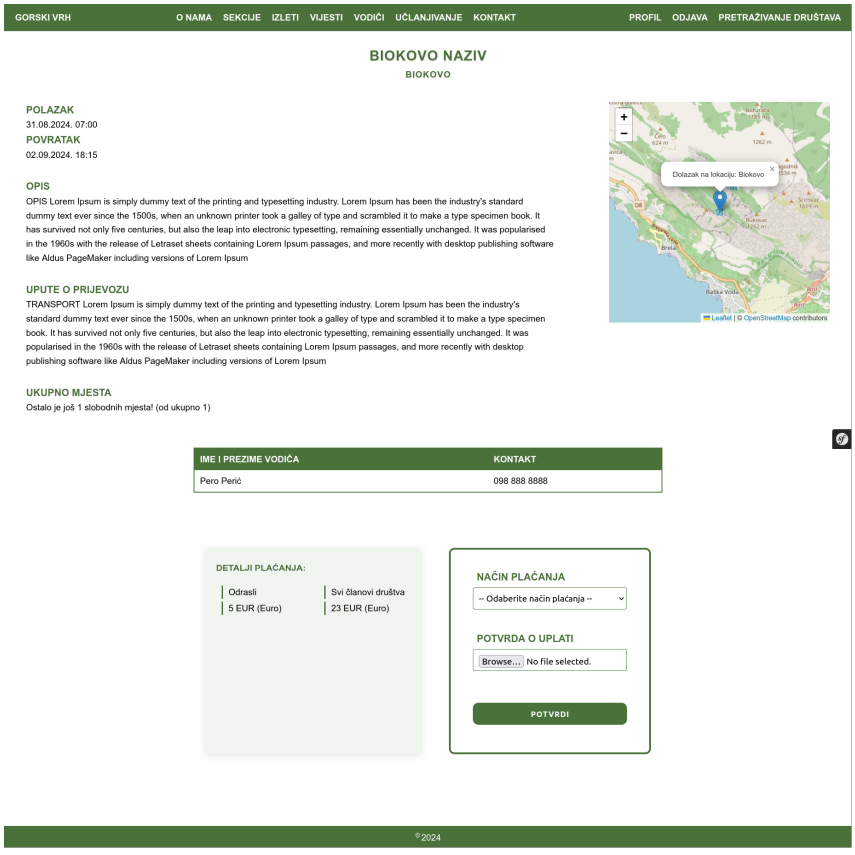
```
security:
    providers:
        public_application_provider:
            id: security.member_provider
    firewalls:
        public_application:
            provider: public_application_provider
            form_login:
                login_path: login
                check_path: login_check
                default_target_path: default
                username_parameter: email
                password_parameter: password
                always_use_default_target_path: true
            logout:
                path: logout
                target: login
```

7.4.3. Forma za plaćanje

Na slici 31 se nalazi prikaz za učlanjivanje, a na slici 32 se nalazi prikaz za pregled detalja izleta. Ono što im je zajedničko je da oba prikaza sadrže formu za plaćanje. Iako forme na prikazima izgledaju isto, logika oko njihovog spremanja na strani servera se poprilično razlikuje. Na primjer, kod prijave za izlet, potrebno je provjeriti ima li još slobodnih mjesta, dok navedena provjera nije potrebna kod učlanjivanja.



Slika 31: Prikaz za učlanjivanje u planinarsko društvo



Slika 32: Prikaz za pregled detalja izleta

Za početak je kreirano sučelje PaymentServiceContract koje zahtjeva da klase implementiraju funkcije iz isječka koda 68. Zatim su kreirani servisi MembershipPaymentService i TripPaymentService koji implementiraju navedeno sučelje.

Isječak kôda 68: Sučelje PaymentServiceContract

```
interface PaymentServiceContract {
    public function getPaymentDetails(): ?array;
    public function createPayment(Payment $payment, ?UploadedFile $file): Payment;
    public function getSuccessfulPaymentMessage(): string;
    public function canMemberCreatePayment(?Member $member): ?string;
    public function setPaymentObject($paymentObject): void;
    public function setHikingAssociation(HikingAssociation $hikingAssociation): void
     ;}
```

Kako bi se tijekom izvođenja zahtjeva mogao dohvatiti ispravan servis za plaćanje, iskorišten je lokator servisa (eng. *service locator*) kojeg nudi Symfony. Prema [47], lokator servisa sadrži set servisa, ali ih instancira na zahtjev. U isječku koda 69 je dan primjer kreiranja lokatora servisa uz pomoć atributa AutowireLocator koji kao prvi argument prima polje servisa. U ovom slučaju ključevi polja su nazivi klasa, a vrijednosti servisi.

Isječak kôda 69: Primjer konfiuguracije lokatora servisa

```
use Psr\Container\ContainerInterface;
use Symfony\Component\DependencyInjection\Attribute\AutowireLocator;
class PaymentController extends BaseController {
public function __construct(
    #[AutowireLocator([
        HikingAssociation::class => MembershipPaymentService::class,
        Trip::class => TripPaymentService::class
    ])]
    private ContainerInterface $handlers,
```

U isječku koda 70 se nalazi primjer dohvaćanja servisa iz lokatora. Na rutu se šalje parametar paymentObject koji može predstavljati ID od izleta ili planinaskog društva. Uz pomoć ID-a se dohvaća objekt i na temelju klase objekta će se dohvatiti potrebni servis nad kojim se mogu pozivati metode iz sučelja PaymentServiceContract.

Isječak kôda 70: Primjer dohvaćanja servisa iz lokatora servisa

```
#[Route('/payment/{hikingAssociation}')]
public function payment(Request $request, HikingAssociation $hikingAssociation):
 Response {
    $paymentObjectId = $request->get('paymentObject');
```

```
4       $paymentObject = DataObject::getById($paymentObjectId);
5       /** @var PaymentServiceContract $service */
6       $service = $this->handlers->get($paymentObject::class);
7       $service->setPaymentObject($paymentObject);
8       $service->setHikingAssociation($hikingAssociation);
9       $paymentDetails = $service->getPaymentDetails();
10      $message = $service->canMemberCreatePayment($this->security->getUser());
```

Opisanim postupkom je zadovoljen otvoren-zatvoren princip (eng. *open-closed principle*), jer je izbjegnuto korištenje uvjeta koji provjeravaju o kojem se objektu radi prije nego li se pozove ispravna funkcija za obradu plaćanja. Također, ako u budućnosti nastane potreba za dodavanjem novog predmeta plaćanja, samo je potrebno kreirati novi servis koji implementira sučelje PaymentServiceContract i dodati ga u atribut AutowireLocator u klasi PaymentController.

7.5. Mjerenje performansa

Za potrebe testiranja tehnika poboljšanja performansi odabrano je pet funkcionalnosti: dohvaćanje vodiča, dohvaćanje forme za plaćanje izleta i članarine, dohvaćanje broja prijava za izlet te slanje detalja o izletu na e-poštu. Navedene funkcionalnosti se nalaze u tablici 1 zajedno s tehnikama poboljšanja koje su primijenjene.

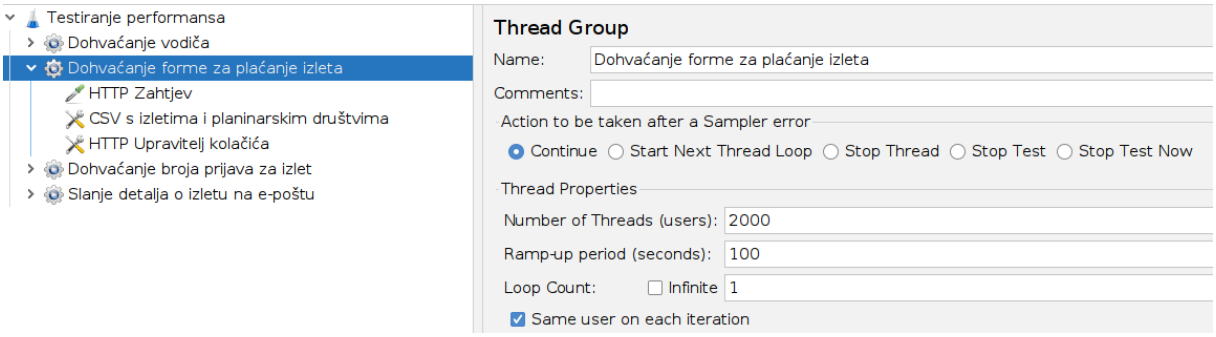
Tablica 1: Funkcionalnosti s izabranim tehnikama poboljšanja performansa

	MEĐUSPREMNIK	**INDEKS**	**OPTIMIZACIJA UPITA**	**ASINKRONOST**
Dohvaćanje vodiča planinarskog društva	+	+	-	-
Dohvaćanje forme za plaćanje izleta	-	+	-	-
Dohvaćanje forme za plaćanje članarine	-	+	-	-
Dohvaćanje broja prijava za izlet	-	+	+	-
Slanje detalja o izletu na e-poštu	-	-	-	+

Za mjerenje performansa odabran je Apache JMeter, alat otvorenog koda koji služi za testiranje performansi i funkcionalnog ponašanja aplikacija pod različitim opterećenjima. Iako je prvotno kreiran za testiranje web aplikacija, danas je s njime moguće testirati različte tipove aplikacija, servera i protokola. Važno je za napomenuti da Apache JMeter nije preglednik, već radi na razini protokola. Iako Apache JMeter može simulirati ponašanje preglednika u pogledu

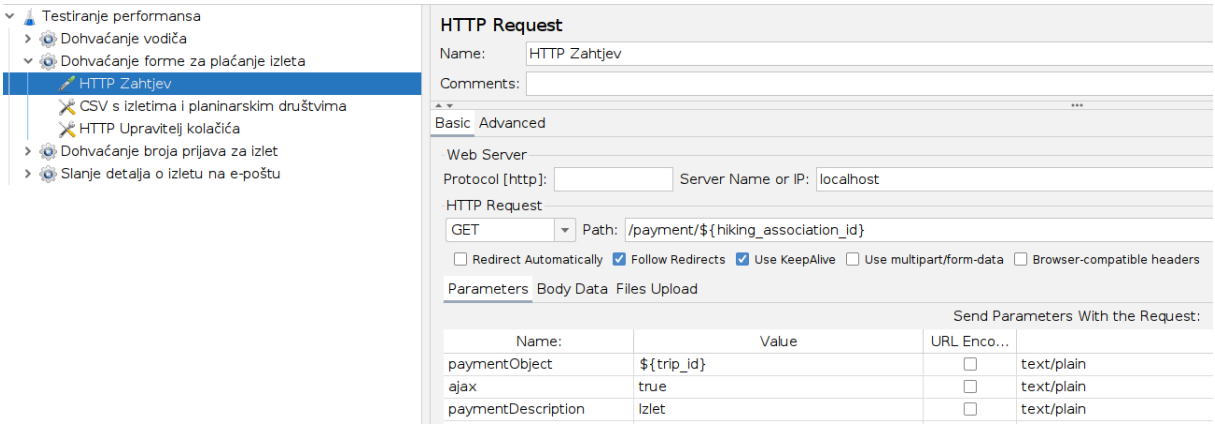
web servisa, nema mogućnost izvršavanja Javascripta koji se nalazi na HTML stranicama niti prikazuje HTML stranice kao što to čini preglednik. Zbog toga su u ovom poglavlju testirane samo tehnike poboljšanja performansa na strani servera, ali preporuka je da se inače testiraju i performanse na strani klijenta [48].

Na početku svakog testiranja u alatu Apache JMeter je potrebno kreirati testni plan čije se izmjene spremaju u datoteci s ekstenzijom JMX. U testni plan se zatim dodavaju grupe dretvi (eng. *thread group*), a na slici 33 se nalazi primjer konfiguracije jedne grupe. Svaka grupa određuje broj dretvi (eng. *number of threads*) koji označava koliko će se paralelnih korisnika simulirati u testu. Razdoblje povećanja (eng. *ramp-up period*) određuje koliko će vremena alatu Apache JMeter trebati da dosegne puni broj odabranih dretvi. Na primjer, ako je broj dretvi 10, a razdbolje povećanja 100 sekundi, svakih 10 sekundi će se kreirati jedna nova dretva [49].



Slika 33: Primjer grupe dretvi u alatu Apache JMeter

Na slici 34 se nalazi primjer postavki HTTP zahtjeva. Potrebno je definirati naziv servera, port, HTTP metodu i putanju uz mogućnost dodavanja parametara. Kako bi testiranje bilo dinamičnije, uz pomoć CSV konfiguracije pod nazivom "CSV s izletima i planinarskim društvima" su učitani ID-evi izleta i planinarskih društava. ID-evi su zatim mapirani u varijable trip_id i hiking_association_id te su iskorišteni u HTTP zahtjevu.



Slika 34: Primjer HTTP zahtjeva u alatu Apache JMeter

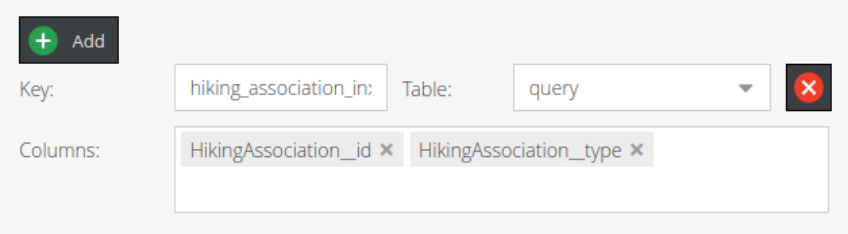
Ovom radu je priložena JMX datoteka za one koji žele vidjeti ostatak konfiguracije testnog plana. Osim testnog plana, priloženi su i HTML izvještaji svih mjerenja koje je generirao alat Apache JMeter.

7.5.1. Dohvaćanje vodiča planinarskog društva

U isječku koda 71 se nalazi upit kojim se dohvaćaju vodiči za planinarsko društvo. Postavljanjem ključne riječi EXPLAIN ispred upita, dobiva se uvid u to da će se prilikom izvršavanja upita provjeriti 28,612 redova unutar tablice object_query_Guide. Dodavanjem indeksa sa slike 35, taj broj će se smanjiti na 2 reda.

Isječak kôda 71: Upit za dohvaćanje vodiča planinarskog durštva

```
1 SELECT object_Guide.id AS id, object_Guide.type AS `type` FROM object_Guide WHERE ((
   HikingAssociation__id = 56)  AND  object_Guide.type IN ('object','variant','folder'
   )) AND object_Guide.published = 1;
```



Slika 35: Kreiranje indeksa *hiking_association_inx* nad klasom Guide

Osim indeksa, pretpostavka je da će se spremanjem vodiča unutar međuspremnika također poboljšati performanse. U isječku koda 72 se nalazi funkcija koja prvo provjerava jesu li vodiči već spremljeni u međuspremnik. Ako nisu, dohvaća ih iz baze podataka i sprema ih u međuspremnik na 7 dana.

Isječak kôda 72: Funkcija za dohvaćanje vodiča iz međuspremnika

```
1 public function getCachedGuidesByHikingAssociation(HikingAssociation
   $hikingAssociation): array{
2     $cacheKey = self::getGuidesCacheKey($hikingAssociation->getId());
3     $guides = Cache::load($cacheKey);
4     if (!empty($guides)) {
5         return $guides;}
6     $data=$this->guideRepository->getGuidesByHikingAssociation($hikingAssociation);
7     Cache::save($data, $cacheKey, [], 604800); // 604800 is 7 days in seconds
8     return $data;}
```

U konfiguraciji HTTP zahtjeva broj dretvi je postavljen na 2000, a razdbolje povećanja je 100. Grupi dretvi je pridružena i CSV konfiguracija koja učitava CSV datoteku s 50 ID-eva planinarskih društava. U bazi podataka se u trenutku testiranja nalazilo 735 planinarskih društava i 30,946 vodiča. Prije pokretanja svakog mjerenja, obrisani su međuspremnik od aplikacijskog okvira Symfony i podaci koji su spremljeni u Redisu.

U tablici 2 se nalaze rezultati četiriju mjerenja. U prvom mjerenju nije korištena nijedna tehnika optimizacije. U drugom je testirana samo primjena međuspremnika, a u trećem samo primjena indeksa. U zadnjem mjerenju su korišteni međuspremnik i indeks zajedno. Iz rezultata mjerenja su prikazani podaci o prosječnom, najkraćem i najdužem trajanju izvođenja zahtjeva u milisekundama.

Tablica 2: Rezultati mjerenja brzine dohvaćanja vodiča planinarskog društva

	PROSJEČNO TRAJANJE (u ms)	**NAJKRAĆE TRAJANJE (u ms)**	**NAJDUŽE TRAJANJE (u ms)**
Bez optimizacije	36.21	13	654
Međuspremnik	34.42	13	506
Indeks	31.72	13	438
Međuspremnik i indeks	32.36	13	463

Kroz sva četiri mjerenja, prosječno trajanje izvršavanja je iznosilo oko 33 ms, dok je najkraće trajanje bilo jednako u sva četiri mjerenja i iznosilo je 13 ms. Najveća razlika je vidiljiva u najdužem trajanju. Upotrebom samog međuspremnika, perfromans se poboljšao za 148 ms. Upotrebom indeksa performans se poboljšao za 216 ms, a kombinacijom međuspremnika i indeksa se poboljšao za 191 ms.

Važno je za napomenuti da se u ovom slučaju radi o jednostavnom upitu koji dohvaća relativno malu količinu podataka. Svako planinarsko društvo ima između 20-30 vodiča, zbog čega optimizacijske tehnike kao što su indeksi i međuspremnici, ne rezultiraju značajnim poboljšanjima. Unatoč tome, preporuča se primijeniti indekse i međuspremnik u situacijama gdje se podaci često čitaju, a rijetko ažuriraju. Kod složenijih upita koji uključuju povezivanje većeg broja tablica ili rad s većim količinama podataka, indeksi i međuspremnik se mogu pokazati znatno korisnijima.

7.5.2. Dohvaćanje forme za plaćanje izleta

Kod dohvaćanja forme za plaćanje izleta se provjerava je li korisnik prijavljen u aplikaciju, je li korisnik već prijavljen na izlet te ima li izlet još slobodnih mjesta. Kritični dijelovi funkcionalnosti koje bi trebalo optimizirati se nalaze u isječcima koda 73 i 74. U isječku koda

73 se nalazi funkcija koja provjerava je li korisnik prijavljen na izlet, a u isječku koda 74 se nalazi funkcija koja dohvaća broj prijava za izlet.

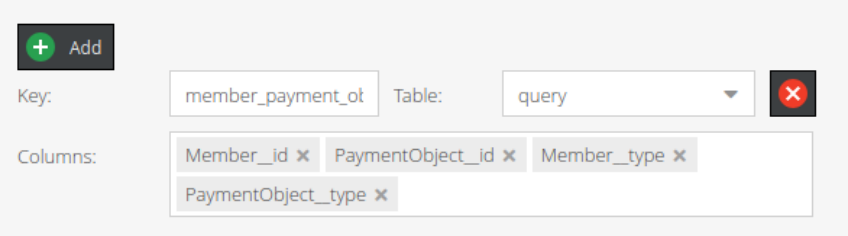
Isječak kôda 73: Funkcija koja provjerava je li korisnik prijavljen na izlet

```
public function isMemberAppliedForTrip(Member $member, Trip $trip): bool {
    $queryBuilder = Db::getConnection()->createQueryBuilder();
    $queryBuilder
        ->select('oo_id')->from('object_Payment')
        ->where('Member__id = :memberId')->andWhere('PaymentObject__id = :tripId')
        ->setParameters(['memberId' => $member->getId(),'tripId' => $trip->getId()])
        ->setMaxResults(1);
    $result = $queryBuilder->executeQuery()->fetchOne();
    return !empty($result);}
```

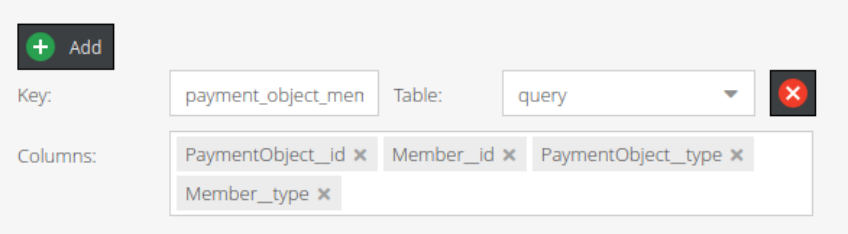
Isječak kôda 74: Funkcija koja dohvaća broj prijava za izlet

```
public function getNumberOfApplicantsForTrip(Trip $trip): int {
    $queryBuilder = Db::getConnection()->createQueryBuilder();
    $queryBuilder->select('COUNT(*)')->from('object_Payment')
        ->where('PaymentObject__id = :tripId')
        ->setParameters(['tripId' => $trip->getId()]);
    return $queryBuilder->executeQuery()->fetchOne();}
```

Zbog velikog broja zapisa klase Payment u bazi podataka koji je u trenutku mjerenja iznosio 176,927, primjena indeksa je prepoznata kao učinkovita metoda poboljšanja performansa. Za potrebe mjerenja su testirana dva indeksa. Na slici 36 se nalazi indeks koji će prvo sortirati podatke po stupcu Member__id pa po stupcu PaymentObject__id, a na slici 37 se nalazi indeks koji prvo sortira po stupcu PaymentObject__id pa po stupcu Member__id. Navedeni stupci su odabrani, jer se koriste u WHERE dijelu SQL upita iz isječaka koda 73 i 74. Bitno je za napomenuti da indeksi nisu postojali istovremeno već je prvo kreiran i testiran indeks member_payment_object_inx koji je prije kreacije i testiranja indeksa payment_object_member_inx, uklonjen.



Slika 36: Kreiranje indeksa *member_payment_object_inx* nad klasom Payment



Slika 37: Kreiranje indeksa *payment_object_member_inx* nad klasom Payment

U konfiguraciji HTTP zahtjeva broj dretvi je postavljen na 2000, a razdbolje povećanja je 100. Grupi dretvi je pridružena i CSV konfiguracija koja učitava CSV datoteku s 500 zapisa koji sadrže ID-eve izleta s pridruženim ID-em planinarskog društva. Prije pokretanja svakog mjerenja, obrisani su međuspremnik od aplikacijskog okvira Symfony i podaci koji su spremljeni u Redisu.

U tablici 3 se nalaze rezultati triju mjerenja. U prvom mjerenju nije korištena nijedna tehnika optimizacije. U drugom je testirana primjena indeksa member_payment_object_inx, a u trećem je testirana primjena indeksa payment_object_member_inx. Iz rezultata mjerenja su prikazani podaci o prosječnom, najkraćem i najdužem trajanju izvođenja zahtjeva u milisekundama.

Tablica 3: Rezultati mjerenja brzine dohvaćanja forme za plaćanje izleta

	PROSJEČNO TRAJANJE (u ms)	**NAJKRAĆE TRAJANJE (u ms)**	**NAJDUŽE TRAJANJE (u ms)**
Bez optimizacije	84,298.58	176	167,718
Indeks (Member, PaymentObject)	38,683.16	171	75,417
Indeks (PaymentObject, Member)	53.71	28	270

Prvo mjerenje bez upotrebe indeksa je imalo najdulja trajanja. Prosječno trajanje je iznosilo skoro 85 sekundi, dok je najduže trajanje iznosilo skoro duplo više. Kod primjene indeksa member_payment_object_inx prosječno i najduže trajanje su se poboljšali, ali su i dalje bili iznad granica primjerenog čekanja na odgovor. Primjenom indeksa payment_object_member_inx su postignuti daleko najbolji rezultati u sve tri kategorije.

Ovim mjerenjem je pokazano da sama kreacija indeksa ne mora nužno dati i najbolje rezultate. Razlog zbog kojeg indeks payment_object_member_inx nudi bolji performans od member_payment_object_inx je upravo u poretku stupaca koji je detaljnije objašnjen u teorijskom dijelu. Dok će se indeks payment_object_member_inx primijeniti nad oba upita iz isječaka koda 73 i 74, indeks member_payment_object_inx će se primijeniti samo nad upitom iz isječka koda 73.

7.5.3. Dohvaćanje forme za plaćanje članarine

Kod dohvaćanja forme za plaćanje članarine provjerava se je li korisnik prijavljen u aplikaciju te je li korisnik već član planinarskog društva. Kritični dio funkcionalnosti koji bi trebalo optimizirati se nalazi u isječku koda 75. U isječku koda se nalazi funkcija koja provjerava je li se korisnik već učlanio u planinarsko društvo tekuće godine.

Isječak kôda 75: Funkcija koja provjerava je li korisnik član društva

```
public function isUserMember(Member $member, HikingAssociation $hikingAssociation) {
    $queryBuilder = Db::getConnection()->createQueryBuilder();
    $queryBuilder->select('oo_id')->from('object_Payment')
        ->where('Member__id = :memberId')->andWhere('Year = :year')
        ->andWhere('PaymentObject__id = :hikingAssociationId')
        ->setParameters(['memberId' => $member->getId(), 'hikingAssociationId' =>
         $hikingAssociation->getId(), 'year' => date('Y')])->setMaxResults(1);
    $result = $queryBuilder->executeQuery()->fetchOne();
    return !empty($result);}
```

U konfiguraciji HTTP zahtjeva broj dretvi je postavljen na 2000, a razdbolje povećanja je 100. Grupi dretvi je pridružena i CSV konfiguracija koja učitava CSV datoteku s 500 zapisa koji sadrže ID-eve izleta s pridruženim ID-em planinarskog društva. Prije pokretanja svakog mjerenja, obrisani su međuspremnik od aplikacijskog okvira Symfony i podaci koji su spremljeni u Redisu.

U tablici 4 se nalaze rezultati dvaju mjerenja. U prvom mjerenju nije korištena nijedna tehnika optimizacije, a u drugom je testirana primjena indeksa payment_object_member_inx sa slike 37. Iz rezultata su prikazani podaci o prosječnom, najkraćem i najdužem trajanju izvođenja zahtjeva u milisekundama.

Tablica 4: Rezultati mjerenja brzine dohvaćanja forme za plaćanje članarine

	PROSJEČNO TRAJANJE (u ms)	**NAJKRAĆE TRAJANJE (u ms)**	**NAJDUŽE TRAJANJE (u ms)**
Bez optimizacije	65,656.69	373	131,025
Indeks (PaymentObject, Member)	59.15	34	273

Rezultati su slični kao i kod mjerenja brzine dohvaćanja forme za plaćanje izleta. Bez indeksa je prosječno trajanje iznosilo oko 65 sekundi, dok je primjena indeksa skratila vrijeme prosječnog čekanja na 59.15 ms. Ovime je pokazano kako se samo s jednim indeksom može ubrzati performans većeg broja funkcionalnosti.

7.5.4. Dohvaćanje broja prijava za izlet

Kod dohvaćanja broja prijava za izlet se koristi funkcija getNumberOfApplicantsForTrip iz isječka koda 74. Kod mjerenja brzine dohvaćanja forme za plaćanje je već pokazano kako primjena ispravnog indeksa može uvelike poboljšati performans, a u ovom poglavlju je testiran utjecaj načina pisanja upita. U isječku koda 76 se nalazi izmijenjena funkcija getNumberOfApplicantsForTrip koja bi trebala narušiti performans. Glavni nedostaci funkcije su nepotrebno dohvaćanje vrijednosti svih stupaca iz tablice object_Payment te računanje ukupnog broja prijava korištenjem PHP funkcije count, umjesto korištenjem funkcije COUNT(*) direktno u upitu.

Isječak kôda 76: Funkcija koja dohvaća broj prijava za izlet (bez optimizacije)

```
1 public function getNumberOfApplicantsForTrip(Trip $trip): int {
2     $queryBuilder = Db::getConnection()->createQueryBuilder();
3     $queryBuilder->select('*')->from('object_Payment')
4         ->where('PaymentObject__id = :tripId')
5         ->setParameters(['tripId' => $trip->getId()]);
6     $results = $queryBuilder->executeQuery()->fetchAllAssociative();
7     return count($results);}
```

U konfiguraciji HTTP zahtjeva broj dretvi je postavljen na 1000, a razdbolje povećanja je 20. Grupi dretvi je pridružena i CSV konfiguracija koja učitava CSV datoteku s 500 zapisa koji sadrže ID-eve izleta s pridruženim ID-em planinarskog društva. Prije pokretanja svakog mjerenja, obrisani su međuspremnik od aplikacijskog okvira Symfony i podaci koji su spremljeni u Redisu.

U tablici 5 se nalaze rezultati dvaju mjerenja. U prvom mjerenju se poziva funkcija iz isječka koda 76 s već opisanim nedostacima, a u drugom mjerenju se poziva funkcija koja je optimizirana. Obje funkcije bi pod manjim opterećenjem mogle dati isti rezultat, a to se vidi po najmanjem trajanju koje iznosi 12 ms. Problem nastaje kad se opterećenje sustava poveća. U tom slučaju će loše napisan upit u proječnom trajanju iznositi 635.31 ms, a u najdužem 3.37 sekundi. Korištenjem optimiziranog upita prosječno trajanje se poboljšalo za pola sekunde, a najduže trajanje je iznosilo 875 ms što je skoro 4 puta brže od prvog mjerenja.

Tablica 5: Rezultati mjerenja brzine dohvaćanja broja prijava za izlet

	PROSJEČNO TRAJANJE (u ms)	**NAJKRAĆE TRAJANJE (u ms)**	**NAJDUŽE TRAJANJE (u ms)**
Loše napisan upit	635.32	12	3369
Optimizacija upita	115.24	12	875

7.5.5. Slanje detalja o izletu na e-poštu

Kod slanja detalja o izletu na e-poštu, može se omogućiti slanje e-pošte u pozadini dodavanjem konfiguracije iz isječka koda 77.

Isječak kôda 77: Konfiguracija za upravitelja poruka

```
framework:
    messenger:
        buses:
            messenger.bus.default:
        transports:
            # MESSENGER_TRANSPORT_DSN=doctrine://default
            async: "%env(MESSENGER_TRANSPORT_DSN)%"
        routing:
            'Symfony\Component\Mailer\Messenger\SendEmailMessage': async
```

U konfiguraciji HTTP zahtjeva broj dretvi i razdoblje povećanja su postavljeni na 30. Za testiranje slanja e-pošte je korištena platforma Mailtrap koja omogućava slanje testnih e-pošti, ali uz određen mjesečni limit zbog čega je za ovo mjerenje korišten puno manji broj dretvi nego kod drugih mjerenja. Prije pokretanja svakog mjerenja, obrisani su međuspremnik od aplikacijskog okvira Symfony i podaci koji su spremljeni u Redisu.

U tablici 6 se nalaze rezultati dvaju mjerenja. U prvom mjerenju nije korištena nijedna tehnika optimizacije, a u drugom mjerenju je konfigurirano da se e-pošta šalje u pozadini. U prvom mjerenju prosječno trajanje je iznosilo oko 18 sekundi, najmanje trajanje je iznosilo 2.2 sekunde, a najduže čak 34 sekunde. U mjerenju gdje se e-pošta šalje asinkrono sva trajanja su iznosila između 62 i 268 ms što je veliko poboljšanje u usporedbi s prethodnim mjerenjem. Ovim mjerenjem je pokazano da se performans funkcionalnosti koje zahtjevaju komunikaciju s vanjskim servisima, može uvelike povećati izvršavanjem logike u pozadini.

Tablica 6: Rezultati mjerenja slanja detalja o izletu na e-poštu

	PROSJEČNO TRAJANJE (u ms)	**NAJKRAĆE TRAJANJE (u ms)**	**NAJDUŽE TRAJANJE (u ms)**
Bez optimizacije	18,155.83	2282	34,232
Asinkronost	103.17	62	268

7.6. Osvrt na implementaciju

Tijekom izrade aplikacije utvrđene su određene mogućnosti i ograničenja u korištenju aplikacijskog okvira Symfony i platforme Pimcore te je u ovom poglavlju dan detaljniji osvrt na uočene nedostatke i prednosti.

Jedno od uočenih ograničenja korištenja Pimcora odnosi se na nasljeđivanje klasa. Na primjer, ako je potrebno dodati klase Student, Nastavnik i Zaposlenik, koje dijele zajedničke atribute, nije moguće kreirati apstraktnu klasu Osoba koja bi sadržavala te atribute. Umjesto toga, svaki se zajednički atribut mora ručno dodati u svaku pojedinu klasu, što otežava održavanje koda, posebice u situacijama kada treba dodati novi zajednički atribut. Na sreću, Pimcore opisanom problemu doskače sa strukturom podataka blokovi objekta (eng. *object bricks*). Blokovi objekta omogućavaju dodavanje novih zajedničkih atributa klasama bez proširivanja. U primjeru s klasama Osoba, Student i Nastavnik, blok objekta nazvan OsnovniPodaciOsobe bi sadržavao zajedničke atribute poput imena, prezimena itd., a klase Student i Nastavnik bi sadržavale njima specifične atribute i blok objekta OsnovniPodaciOsobe. Oni koji žele detaljnije proučiti blokove objekata, mogu pronaći dodatne informacije u Pimcore dokumentaciji [50].

Još jedan od uočenih nadostataka je u paketu Custom Reports, koji iako nudi jednostavno rješenje za kreiranje izvještaja, ima određena ograničenja kada je potrebno generirati veći broj sličnih izvještaja. Na primjer, u slučaju aplikacije koja upravlja s 20 planinarskih društava, za svaki izvještaj o broju korisnika u posljednjih 12 mjeseci mora se kreirati zaseban izvještaj za svako društvo, iako je jedina razlika između njih ID društva. To dovodi do nepreglednog izlistavanja i dupliciranja rada prilikom dodavanja svakog novog izvještaja.

Osim navedenog, Pimcore ima manju zajednicu u usporedbi s popularnijim tehnologijama kao što su Symfony ili Laravel, što otežava pronalaženje rješenja za specifične probleme. Zbog toga se često mora uložiti dodatno vrijeme u istraživanje i rješavanje problema.

Unatoč navedenim nedostacima, tijekom rada s platformom Pimcore su uočene i mnoge prednosti. Jedna od najvećih prednosti Pimcore platforme je njezino administrativno sučelje, koje značajno ubrzava proces izrade aplikacija. Administracija dolazi s već postojećim funkcionalnostima koje programerima olakšavaju rad, a neke od njih su izvoz i uvoz podataka, upravljanje korisnicima i digitalnom imovinom, unaprijeđenje kvalitete podataka te mnoge druge. Osim navedenog, omogućava i kreiranje klasa putem sučelja, što dodatno pojednostavljuje razvoj.

Kao platforma otvorenog koda, Pimcore omogućava zajednici da aktivno doprinosi njegovom razvoju i unapređenju, a otvorenost platforme omogućava integraciju s različitim sustavima, što je čini fleksibilnim rješenjem za složenije poslovne aplikacije. Također, moguće je razviti vlastitu aplikaciju uz Pimcore, a osobe koje već imaju iskustva u radom sa okvirom Symfony lako će se prilagoditi platformi Pimcore.

U usporedbi s razvojem aplikacija koristeći samo programski jezik PHP, Symfony i Pimcore nude značajne prednosti pogotovo kod izrade složenih poslovnih aplikacija. Razvijanje od nule često zahtijeva ručnu izradu svih komponenata, što može biti vremenski zahtjevno i podložno greškama. S druge strane, korištenje Symfonya i Pimcora omogućava programeru da se usredotoči na logiku aplikacije i prilagodbe, umjesto na izgradnju osnovnih funkcionalnosti. Unatoč navedenim prednostima Symfonya i Pimcora, bilo bi praktičnije izbjegavati ih za jednostavne ili statične web stranice, te umjesto njih upotrijebiti jednostavnija rješenja.

8. Zaključak

U teorijskom dijelu rada su kao uvod u aplikacijski okvir Symfony i platformu Pimcore, obrađeni osnovni pojmovi te prednosti i nedostaci vezani uz web aplikacije i aplikacijske okvire. Kod aplikacijskog okvira Symfony su prikazane ključne komponente koje olakšavaju razvoj složenih poslovnih aplikacija, a platforma Pimcore je predstavljena kroz svoje glavne funkcionalnosti te je pokazano kako se korisnici mogu služiti njezinim administrativnim sučeljem. Na kraju teorijskog dijela rada, detaljno su obrađene tehnike za poboljšanje performansi na strani klijenta i poslužitelja. Od tehnika poboljšanja performansi na strani klijenta su obrađene teme poput minifikacije JS i CSS datoteka, kompresije slika, pozicioniranja JS i CSS datoteka u HTML dokumentu i sl., a od tehnika poboljšanja performansi na strani poslužitelja su obrađene teme poput korištenja međuspremnika i primjene indeksa nad tablicama u bazi podataka.

U praktičnom dijelu rada je razvijena web aplikacija koristeći aplikacijski okvir Symfony i platformu Pimcore. Nad aplikacijom su provedena mjerenja s ciljem testiranja performansi, a zbog ograničenja alata Apache JMeter, mjerenja su se fokusirala na tehnike poboljšanja performansi na strani poslužitelja. Provedbom mjerenja je utvrđeno da pravilna primjena indeksa može značajno poboljšati performanse aplikacije, dok izvođenje zadataka u pozadini također donosi značajna poboljšanja, osobito u slučajevima kada je potrebna komunikacija s vanjskim servisima. U testiranoj aplikaciji su mjerenja korištenja međuspremnika i optimizacije upita imala manji učinak na poboljšanje performansa, ali njihov se doprinos u razvoju aplikacija također ne smije zanemariti.

Nedostatak provedenih mjerenja leži u izostavljanju testiranja performansi na strani klijenta. Iako aplikacija već implementira tehnike optimizacije na strani klijenta, kao što su minifikacija JS i CSS datoteka, kompresija i lijeno učitavanje slika, bilo bi korisno istražiti njihov stvarni učinak na performanse aplikacije u nekom budućem radu.

Popis literature

[1] M. W. Docs. „World Wide Web.” (23. 6. 2023.), adresa: https://developer.mozilla.org/en-US/docs/Glossary/World_Wide_Web (pogledano 18. 5. 2024.).

[2] T. Berners-Lee, *Information Management: A Proposal*, CERN, 1989.

[3] S. Aghaei, M. A. Nematbakhsh i H. K. Farsani, „Evolution of the World Wide Web: From Web 1.0 to Web 4.0,” *International Journal of Web Semantic Technology (IJWesT)*, sv. 3, br. 1, str. 1–10, 2012. DOI: 10.5121/ijwest.2012.3101.

[4] A. Volle. „Web application.” Encyclopedia Britannica. (6. 10. 2022.), adresa: https://www.britannica.com/topic/Web-application (pogledano 18. 5. 2024.).

[5] T. Bhatt. „Web Applications Types Examples of Web Apps + Benefits and Challenges.” Intelivita. (27. 11. 2023.), adresa: https://www.intelivita.com/blog/types-of-web-applications/ (pogledano 20. 5. 2024.).

[6] M. W. Docs. „Web app manifests.” (19. 4. 2024.), adresa: https://developer.mozilla.org/en-US/docs/Web/Manifest (pogledano 6. 7. 2024.).

[7] S. Tandel i A. Jamadar, „Impact of Progressive Web Apps on Web App Development,” 2018. DOI: 10.15680/IJIRSET.2018.0709021.

[8] A. Mathur. „What is a Framework?” GeeksforGeeks. (10. 5. 2024.), adresa: https://www.geeksforgeeks.org/what-is-a-framework/ (pogledano 24. 5. 2024.).

[9] X. Chen, *Developing Application Frameworks in .NET*. Apress, 2004.

[10] E. Gamma, R. Helm, R. Johnson i J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st. Addison-Wesley Professional, 1994.

[11] Symfony. „Symfony Components.” (2024.), adresa: https://symfony.com/components (pogledano 27. 5. 2024.).

[12] C. BasuMallick. „What Is a Framework? Definition, Types, Examples, and Importance.” Spiceworks. (31. 10. 2023.), adresa: https://www.spiceworks.com/tech/tech-general/articles/what-is-framework/ (pogledano 27. 5. 2024.).

[13] S. Baskaran. „Top 10 PHP Testing Frameworks for 2024.” Lambdatest. (1. 3. 2024.), adresa: https://www.lambdatest.com/blog/php-testing-frameworks/ (pogledano 30. 5. 2024.).

[14] T. Dey, „A Comparative Analysis on Modeling and Implementing with MVC Architecture,” *Proceedings of the International Conference on Web Services Computing (ICWSC)*, International Journal of Computer Applications (IJCA), 2011.

[15] E. Freeman, E. Robson, B. Bates i K. Sierra, *Head First Design Patterns: A Brain-Friendly Guide*, First Edition. O'Reilly Media, 2004., str. 692.

[16] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, First Edition. Pearson, 2017., str. 432.

[17] A. Verma. „Benefit of using MVC.” GeeksforGeeks. (15. 4. 2023.), adresa: https://www.geeksforgeeks.org/what-is-a-framework/ (pogledano 30. 5. 2024.).

[18] Symfony. „About the Authors.” (2024.), adresa: https://symfony.com/legacy/doc/more-with-symfony/1_4/en/authors (pogledano 23. 6. 2024.).

[19] Symfony, *Symfony*, https://github.com/symfony/symfony, GitHub repozitorij, 2024.

[20] Symfony. „Six good reasons to use Symfony.” (2024.), adresa: https://symfony.com/six-good-reasons (pogledano 23. 6. 2024.).

[21] Symfony. „Projects using Symfony.” (2024.), adresa: https://symfony.com/projects (pogledano 23. 6. 2024.).

[22] Symfony. „Symfony Releases.” (2024.), adresa: https://symfony.com/releases (pogledano 23. 6. 2024.).

[23] Symfony. „Configuring Symfony.” (2024.), adresa: https://symfony.com/doc/current/configuration.html (pogledano 6. 7. 2024.).

[24] R. Hat. „What is YAML?” (3. 3. 2023.), adresa: https://www.redhat.com/en/topics/automation/what-is-yaml (pogledano 6. 7. 2024.).

[25] F. Potencier, *Symfony 6: The Fast Track*. Symfony SAS, 2022.

[26] S. Salehi, *Mastering Symfony*. Packt Publishing, 2016.

[27] Symfony. „Service Container.” (2024.), adresa: https://symfony.com/doc/current/service_container.html (pogledano 29. 6. 2024.).

[28] Symfony. „Databases and the Doctrine ORM.” (2024.), adresa: https://symfony.com/doc/current/doctrine.html (pogledano 6. 7. 2024.).

[29] Symfony. „Creating and Using Templates.” (2024.), adresa: https://symfony.com/doc/current/templates.html (pogledano 7. 7. 2024.).

[30] Symfony. „Events and Event Listeners.” (2024.), adresa: https://symfony.com/doc/current/event_dispatcher.html (pogledano 7. 7. 2024.).

[31] Symfony. „Profiler.” (2024.), adresa: https://symfony.com/doc/current/profiler.html (pogledano 29. 7. 2024.).

[32] D. Fontani, M. Guiducci, F. Mina i D. Dietz Rietsch, *Modernizing Enterprise CMS Using Pimcore: Discover techniques and best practices for creating custom websites with rich digital experiences*. Packt Publishing Ltd, 2021., str. 412.

[33] S. Souders, *High Performance Web Sites*, First Edition. Sebastopol, CA: O'Reilly Media, Inc., 2007.

[34] L. Mix. „Concatenation and Minification.” (), adresa: https://laravel-mix.com/docs/6.0/concatenation-and-minification (pogledano 25. 7. 2024.).

[35] C. Bendell, T. Kadlec, Y. Weiss, G. Podjarny, N. Doyle i M. McCall, *High Performance Images: Shrink, Load, and Deliver Images for Speed*, First Edition. O'Reilly Media, 2016., str. 351.

[36] M. W. Docs. „Lazy loading.” (21. 11. 2023.), adresa: https://developer.mozilla.org/en-US/docs/Web/Performance/Lazy_loading (pogledano 27. 7. 2024.).

[37] „Can I use WebP image format.” (), adresa: https://caniuse.com/webp (pogledano 27. 7. 2024.).

[38] Redis. „Caching.” (), adresa: https://redis.io/solutions/caching/ (pogledano 27. 7. 2024.).

[39] Pimcore. „Cache.” (2024.), adresa: https://pimcore.com/docs/platform/Pimcore/Development_Tools_and_Details/Cache/ (pogledano 27. 7. 2024.).

[40] Redis. „About.” (), adresa: https://redis.io/about/ (pogledano 27. 7. 2024.).

[41] S. Botros i J. Tinley, *High Performance MySQL: Proven Strategies for Operating at Scale*, Fourth Edition. O'Reilly Media, 2021., str. 386.

[42] N. Poulton, *Docker Deep Dive*. Leanpub, 2023.

[43] Pimcore. „Database Model.” (2024.), adresa: https://pimcore.com/docs/platform/Pimcore/Development_Tools_and_Details/Database_Model/ (pogledano 27. 8. 2024.).

[44] Pimcore. „Authenticate Against Pimcore Objects.” (2024.), adresa: https://pimcore.com/docs/platform/Pimcore/Objects/Object_Classes/Data_Types/Fieldcollections/ (pogledano 27. 8. 2024.).

[45] K. Tsonev. „Navigo Documentation.” (), adresa: https://github.com/krasimir/navigo/blob/master/DOCUMENTATION.md (pogledano 27. 8. 2024.).

[46] Pimcore. „Fieldcollection.” (2024.), adresa: https://pimcore.com/docs/platform/Pimcore/Development_Tools_and_Details/Security_Authentication/Authenticate_Pimcore_Objects/ (pogledano 27. 8. 2024.).

[47] Symfony. „Service Subscribers and Locators.” (2024.), adresa: https://symfony.com/doc/current/service_container/service_subscribers_locators.html (pogledano 27. 8. 2024.).

[48] A. S. Foundation. „Apache JMeter.” (2024.), adresa: https://jmeter.apache.org/ (pogledano 31. 8. 2024.).

[49] A. S. Foundation. „Elements of a Test Plan.” (2024.), adresa: https://jmeter.apache.org/usermanual/test_plan.html (pogledano 31. 8. 2024.).

[50] Pimcore. „Objects Bricks.” (2024.), adresa: https://pimcore.com/docs/platform/Pimcore/Objects/Object_Classes/Data_Types/Object_Bricks/ (pogledano 5. 9. 2024.).

Popis slika

Popis tablica

Popis isječaka koda

9. Prilog

GitHub repozitorij: `https://github.com/MartaMarija/Diplomski`

Na GitHub repozitoriju se nalazi sljedeće:

- kod aplikacije,
- HTML izvještaji generirani u alatu Apache JMeter (izvjestaji.zip),
- SQL skripta s podacima iz baze podataka nad kojom su provedena mjerenja (database_dump.sql.gz).

Navedeno je dostupno i u sustavu FOI radovi kao ZIP datoteka.