Razvojni okvir Symfony i web-aplikacije u PHP-u

Rajič, Jelena

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Jelena Rajič

RAZVOJNI OKVIR SYMFONY I WEB-APLIKACIJE U PHP-U

Diplomski rad

Voditelj rada:
izv. prof. dr. sc. Zvonimir Bujanović

Zagreb, ožujak, 2022.

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

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

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Slava Njemu jer mi je dao trnje,
jer me držao za ruke,
jer mi je strpljivo srce u ovome tesao
i vječnim ga svjetlom obasjao;
hvala Mu za svaku suzu i svaki osmijeh,
snagu i pokret;
hvala roditeljima jer su bezuvjetno
uz mene bili,
bratu i sestri jer su mi
leđa čuvali
i prijateljima jer su mi sve ovo
s ljubavlju začinili.
Ovo je za baku Milu i pokojnu baku Ivu
bez čije molitve i vjere ne bih ovo pisala.

Sadržaj

Uvod

PHP (eng. *Hypertext Preprocessor*) je skriptni programski jezik otvorenog koda (eng. *open-source*) namijenjen razvoju web-aplikacija. Za razliku od drugih skriptnih jezika, u cijelosti se izvršava na poslužiteljskoj strani. Kako su kroz godine web-aplikacije postale složenije, došlo je do potrebe za izgradnjom razvojnih okvira koji bi olakšavali razvoj web-aplikacija. Jedan od najraširenijih razvojnih okvira za jezik PHP je Symfony. Symfony je otvorenog koda i izuzetno je snažan i fleksibilan okvir. Nudi mogućnost strukturiranja koda pomoću obrasca softverske arhitekture MVC (eng. *Model-View-Controller*), biblioteku za preslikavanja podataka iz baze u objekte, vlastiti predložak za izgradnju korisničkog sučelja te brojne druge komponente koje omogućavaju razvoj složenih web-aplikacija.

Ovaj rad podijeljen je na dva dijela. U prvom dijelu naglasak je na opisu svih bitnih komponenti okvira Symfony, dok drugi dio čini opis razvoja web-aplikacije koja demonstrira njegove mogućnosti.

U prvom poglavlju dajemo općeniti uvod u razvojni okvir Symfony. Potom navodimo zahtjeve i naredbe za instalaciju te opisujemo pokretanje prve web-aplikacije. Nakon toga, objašnjavamo ulogu pojmova rute i kontrolera, a zatim njihovu implementaciju u Symfonyu. Potom predstavljamo predložak *Twig* kojeg Symfony nudi za izgradnju korisničkog sučelja te biblioteku *Doctrine* koja omogućava objektno-relacijsko mapiranje. Slijedi opis alata *Form* i njegovih prednosti prilikom rada s HTML obrascima. Na koncu, dajemo upute kako iskoristiti sve sigurnosne značajke koje Symfony pruža te opisujemo kako biblioteka *Webpack Encore* olakšava dizajniranje korisničkog sučelja.

U drugom poglavlju opisujemo web-aplikaciju *Gažer* koja umrežava glazbene izvođače i ugostiteljske objekte. Na početku prikazujemo osnovne dijelove aplikacije i način na koji se koristi. Potom opisujemo entitete pomoću kojih su bitni podaci pohranjeni u bazu, navodimo rute i opisujemo pripadajuće metode kontrolera. Zatim objašnjavamo implementaciju notifikacija i opisujemo sigurnosne značajke aplikacije. Naposljetku, dajemo uvid u biblioteku *Bootstrap* pomoću koje smo uredili korisničko sučelje.

Poglavlje 1

Razvojni okvir Symfony

Slika 1.1: Logo Symfonya

Symfony je jedan od najraširenijih razvojnih okvira za programski jezik PHP. Sastoji se od skupa PHP komponenti koje nude rješenja za bržu i lakšu izradu web-aplikacija. Objavljen je 2005. godine kao besplatni softver i takav je ostao do danas. Symfony je otvorenog koda (eng. *open-source*) koji vodi zajednica od tisuću suradnika. Ono što ga izdvaja od drugih razvojnih okvira je njegova fleksibilnost jer programerima omogućava da svaki dio razvojnog okvira prilagode svojim potrebama. Aplikacije izrađene pomoću Symfonya se mogu strukturirati s obrascem softverske arhitekture MVC (eng. *Model-View-Controller*), ali nude i mogućnost njegovog izostanka. MVC je obrazac koji pomaže dizajnirati web-aplikaciju na strukturiraniji, slojevitiji i logičniji način [1].

Navedimo uvodno neke od ključnih biblioteka koje nudi Symfony te način na koji u aplikacijama ostvaruje spomenuti obrazac MVC. Za komunikaciju s bazom podataka te preslikavanje podataka iz baze u objekte u PHP-u („model" u MVC-u) koriste se biblioteke Doctrine i Propel. Symfony nudi vlastiti predložak za jezik PHP naziva Twig za prikazivanje podataka („view" u MVC-u) koje dohvaćaju spomenute biblioteke. Za prima-

nje zahtjeva koje aplikacija šalje koristi se PHP klasa Controller („controller" u MVC-u) koja pomoću konfiguracije usmjeravanja definira koja će se članska funkcija pokrenuti za dolazni URL. Klasa Controller služi kao posrednik između Doctrine-a i Twig-a povezujući ih slanjem podataka koje dohvaća Doctrine, njihovim obrađivanjem, ažuriranjem i objašnjavanjem kako trebaju biti prikazani u web-aplikaciji. U ovom radu opisana je verzija Symfony 5.3.10.

1.1 Instalacija Symfonya

Provjera zahtjeva

Prije izrade aplikacije u Symfonyu potrebno je instalirati PHP 8.0.2 ili novije verzije i sljedeće PHP nadogradnje (koje su instalirane prema zadanim postavkama u većini PHP 8 instalacija): Ctype, iconv, PCRE, Session, SimpleXML i Tokenizer. Potrebno je instalirati i Composer koji se koristi za upravljanje PHP paketima [2].

Dodatno, instaliranjem *Symfony CLI* u komandnoj liniji postaje dostupna aplikacija pod nazivom *symfony* koja pruža sve alate koji su potrebni za razvoj i lokalno pokretanje aplikacije u Symfonyu.

Symfony također pruža alat za provjeru ispunjava li računalo sve zahtjeve za instalaciju:

```
$ symfony check:req
```

Instalacija

Za instalaciju u operacijskom sustavu Linux, potrebno je pokrenuti sljedeću naredbu u komandnoj liniji kako bi se stvorila datoteka koja se zove *symfony*:

```
$ wget https://get.symfony.com/cli/installer -O - | bash
```

Da bismo napravili novi Symfony projekt imena *Gazer* potrebno je pokrenuti:

```
$ symfony new gazer
```

Ovdje će *gazer* biti direktorij u kojem će se nalaziti aplikacija. Ovo je također naziv web-aplikacije čija izrada će u ovom radu služiti kao primjer složene aplikacije. Ova naredba klonira *git* repozitorij pod nazivom *symfony/skeleton* i zatim koristi Composer za instaliranje ovisnosti tog projekta.

Za pokretanje Symfony aplikacije potrebno je imati neki od web poslužitelja poput Apache ili Nginx. Najlakši način za instaliranje paketa *apache* za rad sa Symfonyem je pokretanjem naredbe:

```
$ composer require symfony/apache-pack
```

Konfiguracija Apache-a ovisi o tome koja verzija PHP-a i Apache-a se koristi.

Za korištenje lokalnog web poslužitelja Symfony nudi vlastiti web poslužitelj *Symfony Local Web Server*. Ovaj lokalni poslužitelj između ostalog pruža podršku za HTTP/2, istovremene zahtjeve, TLS/SSL i automatsko generiranje sigurnosnih certifikata. Potrebno je u direktoriju projekta pokrenuti lokalni web poslužitelj na sljedeći način:

```
$ symfony server:start
```

Ovaj paket instalira datoteku *.htaccess* u direktoriju *public* koja opisuje pravila pristupa prilikom posluživanja aplikacije u Symfonyu.

1.2 Prva aplikacija

Ukoliko se koristi neki od web poslužitelja, poput Apache-a ili Nginx-a, potrebno ga je usmjeriti na poddirektorij *public* unutar glavnog direktorija naše web-aplikacije jer se tamo nalazi njen javno dostupni dio. Umjesto postavljanja pravog poslužitelja, na lokalnom računalu moguće je koristiti PHP-ov ugrađeni web poslužitelj. U glavnom direktoriju aplikacije potrebno je pokrenuti:

```
$ php -S 127.0.0.1:8000 -t public/
```

Ukoliko nakon toga otvorimo neki od web preglednika i upišemo *http://localhost:8000*, prikazat će se web-aplikacija kao na Slici 1.2.

Slika 1.2: Početna stranica

Za rad sa Symfonyem, preporuča se korištenje editora PhpStorm. On sadrži dodatke (eng. *plugins*) koji omogućuju sve vrste automatskog upotpunjavanja koda i mnoge druge pogodnosti koje čine pisanje koda lakšim i bržim.

1.3 Route i Controller

Neovisno o jeziku, svaki razvojni okvir ima isti glavni zadatak: omogućiti jednostavno preslikavanje URL-a kojeg zovemo ruta (eng. *route*) na specifični kod koji priprema pripadnu web-stranicu i obrađuje eventualne podatke koje je korisnik unio. Taj specifični kod je metoda unutar komponente zvane kontroler (eng. *controller*), a ono što metoda vraća naziva se odgovor (eng. *response*).

Rute se u Symfonyu ostvaruju putem oznake *Route* definirane iznad metode u kontroleru, dok je kontroler implementiran kao PHP klasa naziva *Controller*. Prikupljajući informacije o dolaznom zahtjevu, klasa *Controller* stvara odgovor koji je u Symfonyu uvijek objekt tipa *Response* koji može biti HTML ili JSON sadržaj ili binarna datoteka poput slike ili PDF-a.

Usmjeravanje

Usmjeravanje (eng. *routing*) je proces preslikavanja HTTP zahtjeva na pripadajuće metode koji određuje što će se prikazati korisniku kada posjeti određenu web-stranicu. Konfiguracija rute pruža korisne mogućnosti kao što je generiranje URL-ova smislenih naziva (npr. */clanak/uvod-u-symfony* umjesto *index.php?clanak-id=24*) [3].

U Symfonyu se rute mogu konfigurirati na različite načine i svaki od njih pruža iste značajke i performanse. Moguće ih je konfigurirati u YAML-u, XML-u, PHP-u, odnosno korištenjem atributa (eng. *attribute*) ili oznaka (eng. *annotation*). Oznake pruža biblioteka *Doctrine Annotations* i njihovo korištenje omogućavamo pokretanjem:

```
$ composer require doctrine/annotations
```

Naredba stvara konfiguracijsku datoteku:

```yaml
# config/routes/annotations.yaml
controllers:
    resource: ../../src/Controller/
    type: annotation

kernel:
    resource: ../../src/Kernel.php
    type: annotation
```

Konfiguracija govori Symfonyu da traži rute definirane kao oznake u svim PHP klasama unutar direktorija *Controller*.

Pokažimo sada kako bismo korištenjem oznaka definirali kontroler, odnosno, njegovu člansku funkciju koja bi se izvršila prilikom pristupa URL-u */homepage*:

```
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class ClanakController extends AbstractController
{
    /**
     * @Route("/homepage", name="app_homepage")
     */
    public function homepage()
    {
        return $this->render('default/homepage.html.twig');
    }
}
```

Ova konfiguracija definira da će prilikom pristupa URL-u */homepage* biti izvršena metoda *homepage()* klase ClanakController. Parametrom *name* dana je pokrata za ovo preslikavanje - korištenjem naziva *app_homepage* kasnije ćemo moći pristupiti bilo URL-u bilo ovoj metodi. Kao što se vidi iz primjera, oznake su u Symfonyu doslovno konfiguracija unutar PHP komentara.

Drugi način definiranja ruta je da se sve popišu u jednu YAML, XML ili PHP datoteku. Glavna prednost ovog načina je što ne zahtijeva nikakvu dodatnu ovisnost poput oznaka iz biblioteke *Doctrine Annotations*, a nedostatak je što je potrebno raditi s više datoteka kada se provjerava ruta neke radnje kontrolera. Sljedeći primjer pokazuje kako istu rutu */homepage* definirati u YAML/XML/PHP-u:

```
# config/routes.yaml
app_homepage:
    path: /homepage
    controller: App\Controller\ClanakController::homepage
```

Kontroler je potrebno napisati u formatu *controller_class::ime_metode*. Ukoliko je metoda unutar kontrolera implementirana kao __invoke(), može se izostaviti dio s imenom metode:

```
# config/routes.yaml
app_homepage:
    path: /homepage
    controller: App\Controller\ClanakController
```

U daljnjim primjerima u radu korištena je konfiguracija pomoću oznaka.

Prema zadanim postavkama, ruta odgovara svim HTTP metodama (GET, POST, PUT, itd.) te je zato dobro koristiti opciju *method* koja će ograničiti rute na koje će odgovoriti. Metode se pišu unutar rute, odvojene zarezom od URL-a. Ukoliko se, na primjer, želi ograničiti na metode GET i POST, rutu bismo napisali u sljedećem formatu:

```
/**
* @Route("/homepage", methods={"GET","POST"})
*/
```

Symfony pruža alat za ispisivanje svih ruta koji može biti koristan u otklanjanju pogrešaka pri izradi velikih aplikacija gdje postoji velik broj njih. Alat omogućujemo pokretanjem naredbe:

```
$ php bin/console debug:router
```

```
-------------  --------  -------- ------  ------------------------
 Name           Method    Scheme   Host    Path
-------------  --------  -------- ------  ------------------------
app_homepage    ANY       ANY      ANY     /homepage
clanak_lista    ANY       ANY      ANY     /clanak/lista
clanak_show     ANY       ANY      ANY     /clanak/{naslov}
clanak_edit     GET|POST  ANY      ANY     /clanak/{naslov}/edit
----------------------------------------------------------------
```

Ukoliko se naredbi proslijedi i naziv rute, ispisuju se njezini detalji.

Najčešće se koriste rute u kojima je jedan dio promjenjiv. Na primjer, URL za prikaz nekog članka vjerojatno će uključivati naslov ili oznaku (npr. */clanak/moj-prvi-clanak* ili */clanak/uvod-u-symfony*). Ti varijabilni dijelovi su napisani u vitičaste zagrade i moraju imati jedinstveno ime. Na primjer, ruta za prikaz sadržaja članka definirana je kao /clanak/{naslov}. Ruta */clanak/uvod-u-symfony* će se kao argument u obliku stringa u varijabli naslov proslijediti kao parametar kontroleru koji izvršava metodu *show()*:

```
$naslov='uvod-u-symfony'
```

Param converter je značajka koju Symfony nudi ukoliko je potrebno pretvoriti varijable koje ruta prosljeđuje. Ona omogućuje slanje parametara preko rute koje pripadajuća

funkcija kontrolera onda može dohvatiti. Tako se na jednostavan način od, na primjer, prosljeđenog integera *id*-a članka stvori objekt, u Symfonyu zvan *Entity*, koji sadrži sve značajke članka. Kako bismo omogućili tu značajku, potrebno je pokrenuti:

```
$ composer require sensio/framework-extra-bundle
```

Objekt *$clanak* čiji naslov odgovara parametru rute, sada je argument metode *show()*:

```
namespace App\Controller;

use App\Entity\Clanak;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class ClanakController extends AbstractController
{
    /**
     * @Route("/clanak/{naslov}", name="clanak_show")
     */
    public function show(Clanak $clanak): Response
    {
        // ...
    }
}
```

Metodi *show()* proslijedili smo argument *$clanak* tipa *Clanak* koji je entitet biblioteke Doctrine. Proces kojim ParamConverter traži objekt iz rute odvija se u dva koraka. Na početku, sve vrijednosti iz rute stavljaju se u niz *$request−>attribute*, zatim klasa *RequestAttributeValueResolver* provjerava postoji li u nizu *$request−>attribute* argument koji je predan metodi te ako ga pronađe vraća njegovu vrijednost. Upravo klasa *RequestAttributeValueResolver* je ono što nam daje ovu fundamentalno važnu funkcionalnost Symfonya. Klasa provjerava postoji li u bazi podataka, unutar tablice *Clanak*, članak čija vrijednost svojstva iz rute odgovara danoj vrijednosti. U ovom slučaju, članak koji ima naslov istovjetan naslovu iz rute. Ukoliko ga pronađe, proslijedit će entitet s traženim svojstvom metodi *show()*, inače će generirati odgovor o greški naziva *404 Error*.

Ponekad je potrebno pomoću biblioteke Doctrine koristiti prilagođeni upit pa je korisno onemogućiti automatsku konverziju argumenata metode. Onemogućavanje automatske konverzije se ostvaruje konfiguracijom u datoteci *sensio_framework_extra.yaml*:

```
# config/packages/sensio_framework_extra.yaml
sensio_framework_extra:
    request:
        converters: true
        auto_convert: false
```

Moguće je eksplicitno onemogućiti pojedine Convertere:

```
# config/packages/sensio_framework_extra.yaml
sensio_framework_extra:
    request:
        converters: true
        disable: ['doctrine.orm', 'datetime']
```

Kontroler

Kontrolor je odgovoran za logiku kontrole toka kojom korisnik stupa u interakciju s aplikacijom koja koristi obrazac MVC. Kontroler određuje koji će se odgovor poslati natrag korisniku kada korisnik uputi zahtjev pregledniku. U obrascu MVC kontroleri su zapravo veza između modela i pogleda [4].

U Symfonyu je kontroler definiran kao PHP klasa naziva *Controller* koja čita informacije iz objekta tipa *Request* i stvara i vraća objekt tipa *Response*. U svjetlu obrasca MVC, klasa *Controller* služi kao posrednik izmedu Doctrine-a i Twig-a.

Kontroleri se u Symfonyu spremaju u poddirektorij *Controller* unutar direktorija *src*. Svaka klasa koja se kreira u direktoriju *src* treba *namespace* koji mora biti *App* i zatim direktorij u kojem je klasa pa je tako dio svakog kontrolera namespace *App\Controller*. Jedino pravilo funkcije kontrolera je da mora vratiti objekt tipa *Symfony Response*. Kontroleru se tipično daje ime prema entitetu kojeg u njemu koristimo te sufiksa Controller pa je tako ClanakContoller nazvan prema kontroleru koji prikazuje listu svih članaka metodom homepage(), pojedini članak metodom show() te metodom edit() omogućava uređivanje članka.

Symfony dolazi s opcionalnom baznom klasom kontrolera koja se zove *AbstractController*. Kao što je na prošlim primjerima vidljivo, kod definicije klase potrebno je navesti *extends AbstractController*.

Na početku datoteke u kojoj je implementiran kontroler pomoću ključne riječi *use* potrebno je navesti sve objekte koji se koriste. Tu dolazi do izražaja editor koji se koristi jer ukoliko ima dodatke s upotpunjavanjem, definicijom objekta automatski će se dodati sve potrebne *use* naredbe.

1.4 Twig

U obrascu softverske arhitekture MVC, pogled (eng. *view*) služi za izgradnju korisničkog sučelja. Pogled čini kod napisan u HTML-u koji prikazuje podatke koji dolaze iz modela i određuje na koji će ih način korisnik vidjeti. Predložak je najbolji način organiziranja i generiranja tog koda [5].

Symfony ima vlastiti predložak za jezik PHP naziva *Twig*. Izrazito je fleksibilan jer omogućuje definiranje vlastitih prilagođenih oznaka i filtera, a moguće ga je koristiti i za aplikacije u kojima korisnici mogu sami mijenjati dizajn predloška. Budući da jezik predloška *Twig* nema mogućnost pristupa bazi podataka te time isključivo služi za njihovo prikazivanje, provodi se jasno razdvajanje između kontrolera i pogleda jer se u kontroleru podaci moraju prikupiti i spremiti za prosljeđivanje. Jezik predloška Twig omogućuje pisanje sažetih i čitljivih predložaka koji su laki za upotrebu za web dizajnere.

Predlošci u Symfonyu imaju dvije ekstenzije za nazive datoteka (npr. *index.html.twig* ili *index.xml.twig*) pri čemu je prva ekstenzija (*html, xml* ili neka druga) konačni format koji će predložak generirati. Iako predlošci obično generiraju sadržaj u HTML-u, mogu generirati bilo koji tekstualni format. Konvencija o pisanju dviju ekstenzija zato pojednostavljuje način na koji se predlošci stvaraju i prikazuju za više formata.

Prema zadanim postavkama predlošci se nalaze u direktoriju *templates*, a konvencija je da se povezani predlošci unutar njega grupiraju u odvojene poddirektorije.

Za instaliranje predloška *Twig* u komandnoj liniji je potrebno pokrenuti:

```
$ composer require template
```

Nakon instalacije automatski se kreira direktorij *templates* i datoteka *base.html.twig*:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        {% block stylesheets %}{% endblock %}
    </head>
    <body>
        {% block body %}{% endblock %}
        {% block javascripts %}{% endblock %}
    </body>
</html>
```

Datoteka *base.html.twig* definira zajedničke elemente svih predložaka aplikacije kao što su *<head>, <header>, <footer>* ili neki drugi element koda pisanog u jeziku HTML.

Koncept nasljeđivanja predloška Twig sličan je nasljeđivanju klase u PHP-u. Definira se roditeljski predložak iz kojeg se drugi predlošci mogu proširiti. Dijelovi koda koji su grupirani u blokove (eng. *block*) definiraju odjeljke stranice koji se mogu izmijeniti u podređenim predlošcima. Blokovi služe kao „rupe" u koje pojedini predložak stavlja svoj sadržaj. Ostatak blokova roditeljskog predloška prikazat će svoj zadani sadržaj.

Jezik predloška *Twig* ima tri glavna pravila: kod koji ide unutar oznake {% %} treba biti izvršen: pokretanje neke logike, kao što je uvjet ili petlja, kod unutar {{ }} ispisan, a kod unutar oznake {# #} predstavlja komentar. Predložak Twig nudi zaštitu od XSS-a (eng. *cross-site scripting*) što znači da se prilikom ispisivanja radi automatsko „*escape-anje*" ispisanog sadržaja (zamjena znakova sa specijalnim značenjem za HTML, poput <, njihovim kodovima, poput <).

Predložak *Twig* nema opciju pisanja koda u jeziku PHP, ali nudi pomoćne funkcije za logičke izraze. Filteri, koji su zapravo funkcije koje su jednostavne za korištenje, mijenjaju sadržaj prije nego što se generiraju. Na primjer, filter *length* broji elemente niza ili duljinu stringa, ovisno o tipu varijable na kojoj je pozvan:

```
{{ text|length }}
```

Korisna značajka su i naredbe koje olakšavaju provjeru poput je li varijabla *text* definirana:

```
{% if text is defined %}
```

ili na primjer, je li broj spremljen u varijabli *number* paran:

```
{% if number is even %}
```

Interakcija kontrolera i predloška Twig

Metodom *render()* kontroler šalje podatke prema predlošku. Obično metoda kontrolera vraća *this->render()* i prosljeđuje dva argumenta. Prvi je naziv datoteke predloška koji se imenuje prema metodi kontrolera koji ga šalje, a drugi je niz svih varijabli koje će se prikazivati u predlošku.

U sljedećem jednostavnom primjeru varijabla *$text_array* čuva niz od dva stringa koji se u metodi *render()* kao argument prosljeđuje predlošku *show.html.twig*. U pravilu ćemo podatke koji se prosljeđuju dohvaćati iz baze.

```
class ClanakController extends AbstractController
{
    /**
     * @Route("/clanak/{naslov}", name="clanak_show")
     */
    public function show(Clanak $clanak): Response
```

```
    {
        $text_array = [
            'Tekst koji čini prvi odlomak',
            'Tekst koji čini drugi odlomak',
        ];
        return $this->render('clanak/show.html.twig', [
            'text_array' => $text_array,
        ]);
    }
}
```

Promotrimo kako izgleda pripadni predložak *show.html.twig*:

```
{% extends 'base.html.twig' %}

{% block body %}
<a href="{{ path('app_homepage') }}">Naslovnica</a>
<h1>Clanak</h1>
<div>
    {% for text in text_array %}
        <p>{{ text }}</p>
    {% endfor %}
</div>
{% endblock %}
```

Predložak proširuje datoteku *base.html.twig*. Sadržaj koji je stavljen nakon oznake {% *block body* %} i završava s {% *endblock* %} predstavlja sadržaj s kojim će se zamijeniti odgovarajući blok u datoteci *base.html.twig*.

Kako bismo se klikom na oznaku *Naslovnica* preusmjerili na naslovnu stranicu, koristi se funkcija *path()*. Funkcija kao prvi argument prima dodijeljeni naziv rute, a kao neobavezni drugi argument parametre rute. Budući da je varijabla *text_array* niz, potrebno je proći kroz njega i ispisati svaki pojedini element. Primijetimo da u slučaju petlje s kojom se ide kroz niz koristimo zagrade za uvjete ili petlje, a zagrade za ispis u slučaju ispisa varijable samog elementa niza.

Povezivanje predložaka s datotekama pisanim u CSS-u i JavaScript-u

Ako se predložak treba povezati sa statičnim materijalom, na primjer slikom, Symfony nudi funkciju *asset()* koja pomaže u generiranju URL-a. Prvo je potrebno instalirati paket pokretanjem naredbe u komandnoj liniji:

```
$ composer require symfony/asset
```

Datoteke do kojih funkcija *asset()* omogućava pristup stavljaju se u direktorij *public* pa nije potrebno navoditi rutu do njega. Na primjer, ako bismo željeli prikazati sliku naziva *logo.png*, smjestili bismo je u direktorij *images* unutar direktorija *public* i u predlošku je povezali na sljedeći način:

```
<img src="{{ asset('images/logo.png') }}" alt="Symfony!"/>
```

Ukoliko bismo u predloške željeli uključiti datoteke pisane u CSS-u i JavaScript-u, na isti bismo način definirali odgovarajući poddirektorij unutar direktorija *public* i povezali ih pomoću funkcije *asset()*:

```
<link href="{{ asset('css/blog.css') }}" rel="stylesheet"/>

<script src="{{ asset('bundles/acme/js/loader.js') }}"></script>
```

Međutim, u slučaju integracije predloška s datotekama pisanim u CSS-u i JavaScript-u, Symfony preporuča korištenje biblioteke *Webpack Encore* koja integraciju čini jednostavnijom. O njoj će više govora biti u Potpoglavlju 1.8.

1.5 Doctrine

Bitna stavka pri izradi svake web-aplikacije je baza podataka. Ona omogućuje pohranjivanje, ažuriranje, kontroliranje i organiziranje podataka koji su bitni za aplikaciju.

Symfony nudi sve alate koji su potrebni za korištenje baza podataka u aplikacijama zahvaljujući *Doctrine*, jednom od najboljih skupa biblioteka za jezik PHP za rad s bazama podataka. *Doctrine* podržava relacijske baze podataka poput *MySQL* i *PostgreSQL* te baze podataka *NoSQL* poput *MongoDB*.

Doctrine se instalira pomoću paketa *orm* uz kojeg je korisno instalirati i paket *MakerBundle* koji će pomoći generirati kod:

```
$ composer require symfony/orm-pack
$ composer require --dev symfony/maker-bundle
```

Instalacija mijenja datoteku *.env* i stvara nove direktorije *src/Entity*, *src/Repository* i *migrations*.

U datoteku *.env* dodana je definicija varijable *DATABASE_URL*. To je varijabla koju biblioteka *Doctrine* koristi za povezivanje s bazom podataka i potrebno ju je podesiti ovisno o tome koju će vrstu baze podataka aplikacija koristiti. Ukoliko bismo koristili bazu podataka *MySQL*, definicija bi bila:

```
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?
serverVersion=5.7"
```

Varijable *db_user*, *db_password* i *db_name* je potrebno zamijeniti odgovarajućim podacima korištene baze podataka. Potrebno je konfigurirati i verziju poslužitelja, u datoteci *.env* varijablom *serverVersion* ili u datoteci *config/packages/doctrine.yaml* varijablom *server_version*.

Kada su parametri veze baze i aplikacije postavljeni, sljedećom naredbom biblioteka *Doctrine* stvara bazu podataka imena koje je postavljeno na mjesto *db_name* u konfiguraciji:

```
$ php bin/console doctrine:database:create
```

Klasa Entity

Biblioteka *Doctrine* koristi objektno-relacijsko mapiranje (eng. *object–relational mapping*). To znači da svaka tablica u bazi podataka ima pripadajuću PHP klasu i svaki stupac u toj tablici ima definirano svojstvo (člansku varijablu) u toj klasi. Kada se postavlja upit za redak u tablici, *Doctrine* će kreirati objekt na temelju podataka iz tog retka.

Za stvaranje nove klase i svih potrebnih varijabli koristi se naredba:

```
$ php bin/console make:entity
```

Kako bi olakšala i ubrzala izradu klase, naredba postavlja pitanja o potrebnim svojstvima: naziv, tip, veličina i mogućnost da varijabla bude bez vrijednosti.

Uzmimo za primjer da gradimo web-aplikaciju za čitanje znanstvenih članaka te nam je potrebna tablica s popisom svih naslova članaka:

```
$ php bin/console make:entity

Class name of the entity to create or update:
> Clanak

New property name (press <return> to stop adding fields):
> naslov

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

New property name (press <return> to stop adding fields):
>
(press enter again to finish)
```

Ovime se stvara nova datoteka *Clanak.php* unutar direktorija *Entity*:

```
namespace App\Entity;

use App\Repository\ClanakRepository;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass=ClanakRepository::class)
 */
class Clanak
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=255)
     */
    private $naslov;

    public function getId(): ?int
    {
        return $this->id;
    }
    // ...
}
```

Ova vrsta klase se zove entitet (eng. *entity*). Kako bi biblioteka *Doctrine* mapirala ovu klasu i njezina svojstva u tablicu baze podataka, koristi se oznaka *@ORM* u komentarima iznad svakog svojstva. Na primjer, *@ORM\Entity()* iznad klase je ono što govori *Doctrine* da to nije obična PHP klasa nego da je entitet: klasa koja se može pohraniti u bazu podataka, a oznaka *@ORM\Column()* iznad objekta je način na koji *Doctrine* zna koja svojstva trebaju biti pohranjena u tablici i njihove vrste.

Iza definicije varijabli slijedi kod kojeg je naredba generirala. To su metode dohvaćanja i postavljanja (eng. *getter and setter methods*) definiranih varijabli, poput *getNaslov()* i *setNaslov()*.

Primijetimo da naredba *make:entity* na početku pita za ime klase koju želimo stvoriti ili ažurirati što znači da bismo na isti način dodavali nova svojstva već postojećoj klasi.

Naredba *make:entity* pored spomenute datoteke stvara datoteku *ClanakRepository.php*. U njoj je definirana PHP klasa čiji je glavni zadatak dohvaćanje entiteta određene klase. Ukoliko bismo trebali kompliciranije SQL naredbe, unutar spomenute klase definirali bismo metodu i u njoj pomoću funkcije *createQuery()* napisali bismo upit jezikom koji *Doctrine* nudi za pisanje upita (eng. *Doctrine Query Language*). Taj je jezik sličan jeziku SQL, ali nam omogućuje pisanje upite pozivajući se na PHP objekte.

Uspostavljanje relacija među entitetima

U aplikacijama su entiteti nerijetko u nekom odnosu. Taj odnos između njih zovemo relacijom. Razlikujemo dvije vrste relacija: prvu čine najčešće korištene relacije „mnogo naprema jedan" (eng. *many-to-one*) i „jedan naprema mnogo" (eng. *one-to-many*), a drugu „mnogo naprema mnogo" (eng. *many-to-many*).

Pokažimo na primjeru kako bismo pomoću biblioteke *Doctrine* uspostavili relaciju „mnogo naprema jedan" između dva entiteta. Pretpostavimo da u bazi čuvamo tablicu s člancima koja je mapirana s entitetom *Clanak* i da želimo svakom članku dodijeliti kategoriju kojoj pripada. Uzmimo da je tablica s kategorijama mapirana s entitetom *Category*. Pretpostavimo da svakoj kategoriji može pripadati mnogo članaka, ali svaki članak može pripada točno jednoj kategoriji. Iz perspektive entiteta *Clanak*, ovo je relacija „mnogo naprema jedan".

Da bismo uspostavili relaciju, potrebno je u klasi *Clanak* definirati svojstvo *category* s oznakom imena *ManyToOne*. To je moguće napraviti ručno, ali je lakše korištenjem naredbe *make:entity* kojoj na pitanje o vrsti svojstva odgovorimo s *relation*. Nakon toga, naredba postavlja pitanja o odnosu čime se generira sav potreban kod za uspostavljanje relacije. Naredbom se rade promjene u oba entiteta. U entitetu *Clanak* dodaje se već spomenuto novo svojstvo *category* te metode dohvaćanja i postavljanja. Definicija svojstva i pripadna oznaka izgledaju ovako:

```
/**
 * @ORM\ManyToOne(targetEntity="App\Entity\Category",
 * inversedBy="clanci")
 */
private $category;
```

Oznaka *ManyToOne* govori *Doctrine* da koristi stupac *category_id* u tablici clanaka kako bi mapirala svaki redak u toj tablici s retkom u tablici kategorija.

U entitetu *Category* naredba je dodala svojstvo *clanci* i uz standardne metode dohvaćanja i postavljanja, u konstruktoru je definirala niz koji će čuvati sve članke koji su dio određene kategorije:

```
namespace App\Entity;

// ...
use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\Common\Collections\Collection;

class Category
{
    // ...

    /**
     * @ORM\OneToMany(targetEntity="App\Entity\Clanak",
     mappedBy="category")
     */
    private $clanci;

    public function __construct()
    {
        $this->clanci = new ArrayCollection();
    }

    /**
     * @return Collection|Clanak[]
     */
    public function getClanci(): Collection
    {
        return $this->clanci;
    }
```

```
    // ...
}
```

Oznaka *ManyToOne* unutar entiteta *Clanak* je obavezna, ali ova *OneToMany* unutar entiteta *Category* nije. Međutim, nju dodajemo jer želimo imati pristup svim člancima koji su povezani s određenom kategorijom. U ovom primjeru to možemo ostvariti pozivanjem *$category->getClanci()*.

Migrations

Kako bismo izgrađenu klasa mapirali s tablicom u bazi podataka, potrebno je koristiti biblioteku *DoctrineMigrationsBundle* pokretanjem naredbe:

```
$ php bin/console make:migration
```

Naredba stvara datoteku unutar direktorija *migrations* koja sadrži kod u jeziku *SQL* potreban za ažuriranje baze podataka. Čini to tako što uspoređuje trenutnu bazu podataka sa svim klasama entiteta i zatim generira *SQL* naredbe potrebne za njihovu sinkronizaciju. Međutim, da bi se taj *SQL* kod izvršio, potrebno ga je još pokrenuti naredbom:

```
$ php bin/console doctrine:migrations:migrate
```

Ova naredba izvršava kod u datotekama migracije koji još nije pokrenut. Preporuča se redovno pokretanje migracijskih komandi kako bi baza podataka bila ažurna.

Za vođenje evidencije migracija, prethodna naredba stvara tablicu u bazi podataka naziva *doctrine_migration_versions*. Svaki put kada sustav izvrši neku migracijsku datoteku, dodaje se novi red u tablicu koji bilježi da je ta migracija izvršena. To spriječava da se ista migracija izvrši dvaput jer gledajući u tablicu sustav vidi da je već pokrenuta te ju preskače.

Unošenje i dohvaćanje podataka iz baze

Jedna od ključnih filozofija biblioteke *Doctrine* je da se objekti koji se unose ili dohvaćaju iz baze promatraju kao klase i svojstva, a ne tablice i stupci, a da se pojedinosti oko spremanja podataka i upita prema bazi obavljaju „iza kulisa". Pokažimo to sada na primjeru stvaranja objekta *Clanak* unutar metode *createClanak()*. Definirat ćemo novi objekt tipa *Clanak*, popuniti ga podacima i zatim zatražiti od biblioteke *Doctrine* da ga spremi.

```
/**
 * @Route("/clanak/new", name="create_clanak")
 */
```

```
public function createClanak(ManagerRegistry $doctrine): Response
{
    $entityManager = $doctrine->getManager();

    $clanak = new Clanak();
    $clanak->setNaslov('Prvi članak');

    $entityManager->persist($clanak);
    $entityManager->flush();

    return new Response('Saved new Clanak with id '.$clanak->getId());
}
```

Argument tipa *ManagerRegistry* kojeg metoda prima govori Symfonyu da u metodi kontrolera omogući usluge biblioteke *Doctrine*. Metoda *$doctrine−>getManager()* dohvaća objekt tipa *ObjectManager* koji je najvažniji element biblioteke *Doctrine*. On je odgovoran za spremanje objekata i dohvaćanje objekata iz baze podataka. Nakon njegovog poziva instancira se novi objekt *$clanak* kao i svaki drugi objekt u PHP-u. Poziv *persist($clanak)* govori *Doctrine* da „upravlja" objektom *$clanak*, ali to još uvijek ne uzrokuje postavljanje upita bazi podataka. Kada se pozove metoda *flush()*, *Doctrine* pregledava sve objekte kojima „upravlja" kako bi za svaki provjerio treba li ga promijeniti u bazi podataka. U ovom primjeru, objekt tipa *$clanak* ne postoji u bazi pa se izvršava SQL naredba *INSERT* koji stvara novi redak u tablici *Clanak*.

Bez obzira stvaraju li se ili ažuriraju objekti, postupak je uvijek isti. Bibilioteka *Doctrine* sama zna treba li dodati novi ili ažurirati postojeći entitet.

Ukoliko bismo željeli dohvatiti podatke iz baze, metodi bismo pored objekta tipa *ManagerRegistry* prosljeđivali primarni ključ objekta kojeg želimo dohvatiti. Promotrimo na primjeru kako bismo iz baze dohvatili članak po njegovom primarnom ključu *id*-u:

```
$clanak = $doctrine->getRepository(Clanak::class)->find($id);
```

U Poglavlju 1.3 spomenut je i automatski način dohvaćanja podataka korištenjem ekstenzije *SensioFrameworkExtraBundle*. U tom slučaju metoda prima objekt tipa kojeg tražimo, a u metodi ga direktno koristimo. To je moguće jer se u pozadini odvija pretraživanje po parametru iz rute. Ovo uvelike olakšava programiranje jer nam argumenti metode ne moraju odgovarati parametrima rute nego samim entitetima kojima pripadaju, a koji su nam najčešće i korišteni u metodama.

1.6 Form

U web-aplikacijama za unošenje podataka od strane korisnika koriste se HTML obrasci (forme). Stvaranje i obrada obrazaca sastoji se od izrade HTML polja za unos podataka, provjeru valjanosti i mapiranje unesenih podataka obrasca u objekte. Symfony nudi moćan alat za olakšavanje tog postupka koji se zove *Form*. Prije korištenja potrebno ga je instalirati pokretanjem naredbe:

```
$ composer require symfony/form
```

U pravilu se u Symfonyu postupak rada s obrascima sastoji od izrade obrasca u kontroleru ili pomoću posebne klase namijenjene samo obrascu, zatim pozivanja obrasca s predloškom kako bi ga korisnik mogao koristiti i na kraju provjere valjanosti unesenih podataka, njezino transformiranje u podatke u PHP-u i ukoliko je potrebno spremanje u bazu.

Klasa namijenjena obrascu

Obrazac je moguće definirati u kontroleru, međutim, preporučuje se da se definira posebna klasa za njega kako bi obrazac bio dostupan u slučaju ponovnog korištenja, ali i da bi kod u kontroleru bio pregledniji.

Klase namijenjene za obrasce nalaze se u poddirektoriju *Form* unutar direktorija *src*. Ako je vezana za entitete, klasa se obično naziva prema imenu tog entiteta i sufiksa *FormType*. Jedino pravilo takvih klasa je da moraju proširiti klasu naziva *AbstractType*. Promotrimo jednostavni primjer definiranja obrasca za unos podataka novog članka:

```
namespace App\Form\Type;

use App\Entity\Clanak;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;

class ClanakFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder,
    array $options): void
    {
        $builder
            ->add('naslov')
            ->add('sadržaj')
```

```
        ;
    }
}
```

Unutar metode *buildForm()* koja prima objekt tipa *FormBuilderInterface* poziva se funkciju *add()* koja dodaje elemente entiteta *Clanak*: naslov i sadržaj.

Interakcija obrasca i predloška

Promotrimo na jednostavnom primjeru na koji se način u kontroleru šalje obrazac predlošku i procesiraju uneseni podaci:

```
// ...
class ClanakController extends AbstractController
{
    public function new(Request $request): Response
    {
        $clanak = new Clanak();

        $form = $this->createForm(ClanakFormType::class, $clanak);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            $clanak = $form->getData();
            // ...
            return $this->redirectToRoute('clanak_success');
        }

        return $this->renderForm('clanak/new.html.twig', [
            'form' => $form,
        ]);
    }
}
```

Prilikom početnog učitavanja stranice u pregledniku, forma je stvorena i prikazana, ali kako obrazac još nije poslan, metoda *$form->isSubmitted()* vraća false. Kada korisnik pošalje obrazac, metoda *handleRequest()* to prepoznaje. U gornjem kodu, podaci se upisuju u varijablu *$clanak* tek nakon što je provjereno da su podaci u formi ispravni (inicijalno se stvara samo prazna varijabla *$clanak*). Ukoliko su podaci ispravni, pokreću se

radnje koristeći objekt *$clanak* kao na primjer, podaci se spremaju u bazu podataka. Ukoliko podaci u formi nisu ispravni, *isValid()* vraća false i obrazac se ponovno prikazuje, ali sada s porukom o pogreški u unosu.

Pogledajmo sada kako bismo upravo definirani i prosljeđeni obrazac pozvali u narednom predlošku *new.html.twig*:

```
{% extends 'base.html.twig' %}
{% block content_body %}
    <h1>Želiš unijeti novi članak?</h1>
    {{ form_start(form) }}
    <div class="form-label-naslov">
        {{ form_label(form.naslov) }}
    </div>
    <div class="form-widget-naslov">
        {{ form_widget(form.naslov) }}
    </div>
    {{ form_rest(form) }}
    <button type="submit" class="btn btn-primary">
        Unesi novi članak
    </button>
    {{ form_end(form) }}
{% endblock %}
```

U pravilu, dovoljno je samo pozvati funkciju *form()* koja prima naziv forme koja se želi prikazati imena koje joj je dodijeljeno pri prosljeđivanju u kontroleru. U tom slučaju, elementi obrasca prikazani su redom kojim su dodani u funkciju *buildForm()* gdje je definiran i popratni tekst uz komponente obrasca. Međutim, u ovom primjeru prikazane su neke od pomoćnih funkcija za prikazivanje dijelova obrasca koje omogućavaju veću fleksibilnost u prikazivanju.

Pomoćna funkcija *form_start()* prikazuje početnu oznaku obrasca, *form_widget()* prikazuje sva potrebna polja za unos, *form_label()* ispisuje tekst uz polja za unos, *form_rest()* generira sva polja koja nisu unesena ručno, a *form_end()* generira oznaku za zatvaranje obrasca. Gumb za slanje se može ugraditi u klasu obrasca, ali ga je moguće i ručno postaviti kao u primjeru. Pomoćne funkcije u *Twig*-u daju nam potpunu kontrolu nad načinom na koji se svako polje obrasca prikazuju. U primjeru je elementu *naslov* omogućeno specifično uređivanje dok je ostalim elementima obrasca uređivanje zadano.

Obrasci kreirani komponentom *Form* prema zadanim postavkama uključuju tokene CSRF (eng. *Cross-site request forgery*) te smo automatski zaštićeni od CSRF napada.

Provjera valjanosti unosa

Provjera valjanosti unesenih podataka u Symfonyu se odvija na način da se poslani podaci iz obrasca prvo spreme u pripadajuće elemente objekta *$form* te se na njih pozove funkcija *$form−>isValid()* koja provjerava jesu li svi elementi objekta valjani. Prije korištenja provjere valjanosti, potrebno je pokretanjem sljedeće naredbe dodati podršku za nju:

```
$ composer require symfony/validator
```

Provjera valjanosti se vrši dodavanjem restrikcija uz definicije elemenata u klasi entiteta ili u klasi posvećenoj obrascu.

1.7 Sigurnost web-aplikacije

Pojam sigurnost web-aplikacija podrazumijeva zaštitu podataka koje aplikacija koristi. Sigurnost se postiže razvojem sigurnosnih značajki koje imaju za cilj pronaći, popraviti i po mogućnosti spriječiti sigurnosne probleme unutar aplikacija kao što su neovlašteni pristup i modifikacija. Sigurnost web-aplikacije se grubo može podijeliti na autentifikaciju i autorizaciju. Jednostavno rečeno, autentifikacija je proces provjere tko je osoba koja pokušava pristupiti aplikaciji, dok je autorizacija proces provjere kojim specifičnim aplikacijama, datotekama i podacima korisnik ima pristup. Sigurnosnu komponentu u aplikacijama izrađenim pomoću Symfonya integrira paket *SecurityBundle*. Sigurnosni sustav jedan je od najmoćnijih dijelova Symfonya, a njime se u velikoj mjeri može kontrolirati putem njegove konfiguracije. Kako bismo instalirali paket *SecurityBundle* potrebno je pokrenuti naredbu:

```
$ composer require symfony/security-bundle
```

U nastavku navodimo naredbe za generiranje gotovih rješenja za većinu potrebnih sigurnosnih značajki.

Dozvole u Symfonyu uvijek su povezane s korisničkim objektom. Ako je potrebno osigurati dijelove aplikacije, mora se stvoriti korisnička klasa koja implementira korisničko sučelje. Najlakši način za generiranje korisničke klase je korištenjem naredbe *make:user* iz paketa *MakerBundle*. Odgovarajući na upite nakon naredbe stvaraju se datoteke *User.php* i *UserRepository.php* s izabranim svojstvima, ali i svim potrebnim metodama. Paket *SecurityBundle* nudi više načina autentifikacije. Većina web-aplikacija koristi autentifikaciju putem obrasca za prijavu korisnika u kojem se korisnici provjeravaju pomoću identifikatora poput e-mail adrese ili korisničkog imena i lozinke. Ova se funkcionalnost u aplikacijama izrađenim pomoću Symfonya ostvaruje pokretanjem naredbe:

```
$ symfony console make:auth
```

Naredba ažurira konfiguraciju datoteku *security.yaml* i stvara datoteku *LoginFormAuthenticator.php* unutar direktorija *Security* koja obrađuje prijavu, kontroler *SecurityController.php* koji definira rute prijave i odjave te predložak *login.html.twig* koji prikazuje obrazac za prijavu.

Za stvaranje obrasca za registraciju Symfony nudi naredbu:

```
$ symfony console make:registration-form
```

Naredba stvara tri elementa: obrazac *RegistrationFormType*, kontroler *RegistrationController* i predložak *register.html.twig*. Uz registraciju je korisno dodati značajku provjere adrese e-maila pomoću paketa *SymfonyCastsVerifyEmailBundle* pokretanjem:

```
$ composer require symfonycasts/verify-email-bundle
```

Koristeći pakete *MakerBundle* i *SymfonyCastsResetPasswordBundle*, moguće je stvoriti sigurno rješenje za rukovanje zaboravljenim lozinkama. Prvo je potrebno instalirati *SymfonyCastsResetPasswordBundle*:

```
$ composer require symfonycasts/reset-password-bundle
```

Zatim upotrijebiti naredbu:

```
$ symfony console make:reset-password
```

Naredba postavlja nekoliko pitanja o aplikaciji te generira sve potrebne datoteke. Kao i u primjerima iznad, paket će se pobrinuti za kreiranja konfiguracije te generiranja kontrolera, predložaka i entiteta.

Nakon što su pokazane osnovne značajke autentifikacije, pogledajmo kako se u aplikacijama izrađenim pomoću Symfonya ostvaruje autorizacija. Prvi dio procesa autorizacije je dodjeljivanje određene uloge korisniku prilikom prijave, dok je drugi dio dodavanje zahtjeva za određenu ulogu u nekom od resursa pri pristupu nekom dijelu aplikacije.

Nakon autentifikacije, trenutno prijavljenom korisniku spremljenom u objektu *User* moguće je pristupiti pomoću metode *getUser()* u baznoj klasi kontrolera *AbstractController*:

```
$user = $this->getUser();
```

Prijavljenom korisniku moguće je pristupiti na isti način iz bilo kojeg kontrolera ukoliko on nasljeđuje baznu klasu kontrolera.

Kada se korisnik prijavi, Symfony poziva metodu *getRoles()* nad upravo kreiranim objektom tipa *User* kako bi odredio koje uloge taj korisnik ima. U klasi *User* koja je

ranije generirana, uloge su spremljene u niz *$roles* koji je pohranjen u bazi podataka i svakom korisniku je uvijek dodijeljena barem jedna uloga naziva *ROLE_USER*. Ostale uloge se kreiraju po potrebi, a jedino pravilo je da naziv uloge ima prefiks *ROLE_*. Umjesto da se svakom korisniku daje mnogo uloga, moguće je definirati pravila nasljeđivanja stvaranjem hijerarhije uloga varijablom *role_hierarchy* unutar datoteke *security.yaml*.

Zabranjivanje pristupa nekom dijelu aplikacije može se ostvariti na tri načina. Prvi je varijablom *access_control* u datoteci *security.yaml* omogućiti zaštitu nekoj ruti. Na primjer, ako se svim rutama koje počinju s */admin* može pristupiti jedino kao korisnik s ulogom *ROLE_ADMIN*, dio s kontrolom pristupa unutar datotece *security.yaml* bi izgledao:

```
access_control:
    - { path: '^/admin', roles: ROLE_ADMIN }
```

Drugi, jednostavniji, ali manje fleksibilan način je da se u kontroleru pozove funkcija *denyAccessUnlessGranted()* koja prima ulogu koju je potrebno imati da bi se izvršila funkcija u kojoj je pozvan. Treći način je pomoću paketa *SensioFrameworkExtraBundle* koji omogućava osiguravanje cijelog kontrolera ili određene funkcije pomoću oznake *@IsGranted()*.

1.8 Korisničko sučelje web-aplikacije

U aplikacijama izrađenim pomoću Symfonya za dizajniranje korisničkog sučelja (eng. *frontend*) moguće je koristiti CSS i JavaScript datoteke smještene izravno u direktoriju *public* te ih uključiti u predloške. Međutim, Symfony nudi biblioteku *Webpack Encore* koja olakšava rad s CSS-om i JavaScript-om. Biblioteka *Webpack Encore* stvara moćno programsko sučelje za aplikacije (eng. *application programming interface*) koje omogućuje organiziranje i povezivanje JavaScript modula i CSS datoteka.

Prije instaliranja paketa *Webpack Encore* potrebno je imati razvojnu platformu za pokretanje JavaScript-a naziva *Node.js*. Korisno je instalirati jedan od upravitelja paketima za JavaScript *yarn* ili *npm*. Ukoliko bismo se odlučili za upravitelja *yarn*, instaliranje paketa i upravitelja pokrenuli bismo naredbama:

```
$ composer require symfony/webpack-encore-bundle
$ yarn install
```

Naredba će instalirati i omogućiti paket *WebpackEncoreBundle*, potom stvoriti direktorij *assets*, dodati datoteku *webpack.config.js* te dodati direktorij *node_modules* u datoteku *.gitignore*. Unutar direktorija *assets*, nalaze se sljedeće datoteke: *app.js*, *bootstrap.js*, *controllers.json*, te direktoriji *styles* s datotekom *app.css* i *controllers* s datotekom

hello_controller.js. U radu s paketom *Webpack Encore* datoteka *app.js* se ponaša kao samostalna JavaScript aplikacija jer se u nju stavljaju sve naredbe za uključivanje CSS datoteka koje se koriste te biblioteke poput jQuery-a ili React-a. Sve vezano za paket *Webpack Encore* konfigurira se putem datoteke *webpack.config.js* koja na početku sadrži osnovnu konfiguraciju. Ključni dio konfiguracije je metoda *addEntry()* koja dodaje JavaScript datoteku koja treba biti zapakirana (eng. *webpacked*), a čiji poziv u osnovnoj konfiguraciji izgleda ovako:

```
.addEntry('app', './assets/app.js')
```

Funkcija prima dva argumenta. Prvi je string *name* koji će se koristiti kao naziv izlazne datoteke, a drugi je string *src* koji je put do izvorne datoteke. Funkcija govori paketu da učita datoteku *assets/app.js* i slijedi sve naredbe *require()* definirane na početku konfiguracijske datoteke. Zatim se sve zajedno zapakira i, zahvaljujući prvom argumentu aplikacije, definiraju se datoteke *app.js* i *app.css* u direktoriju *public/build*. Zatim je potrebno pokrenuti sljedeću naredbu:

```
$ yarn watch
```

Naredba dodaje nove datoteke u direktorij *public/build* koje čuvaju sve JavaScript i CSS datoteke potrebne aplikaciji. Pogledajmo kako bismo u predlošku *base.html.twig* uključili nove JavaScript i CSS datoteke koristeći funkcije iz paketa *WebpackEncoreBundlea*:

```
<!DOCTYPE html>
<html>
    <head>
        {% block stylesheets %}
            {{ encore_entry_link_tags('app') }}
        {% endblock %}

        {% block javascripts %}
            {{ encore_entry_script_tags('app') }}
        {% endblock %}
    </head>
</html>
```

Funkcije *encore_entry_link_tags()* i *encore_entry_script_tags()* čitaju imena datoteka za uključenje u predložak iz datoteke *entrypoints.json* unutar direktorija *build*. Argument unutar obje funkcije treba odgovarati prvom argumentu prije spomenute funkcije *addEntry()*. Nakon poziva ovih funkcija, izvršit će se JavaScript kod iz datoteke *assets/app.js* kao i sve druge JavaScript datoteke koje su uključene te će se prikazati sve potrebne CSS datoteke.

Za izgradnju korisničkog sučelja u Symfonyu je moguće koristiti i JavaScript biblioteku *React.js*. Ukoliko koristimo upravitelj paketima *yarn* biblioteku bismo omogućili pokretanjem naredbe:

```
$ yarn add react react-dom prop-types
```

i pozivanjem metode *enableReactPreset()* unutar datoteke *webpack.config.js*. Ukoliko bismo koristili razvojni okvir *Vue.js* pozvali bismo metodu *enableVueLoader()*.

U Symfonyu je moguće koristiti i *Bootstrap*, najpopularniji HTML, CSS i JavaScript razvojni okvir za razvoj responzivnih web-aplikacija [6]. Instalirati i integrirati razvojni okvir *Bootstrap* moguće je također koristeći paket *Webpack Encore*. Za instaliranje je potrebno pokrenuti naredbu:

```
$ yarn add bootstrap --dev
```

Bootstrap se nalazi u direktoriju *node_modules* i moguće ga je dodati u bilo koju datoteku pisanu u Sass-u ili JavaScript-u. Na primjer, ako bismo željeli dodati *Bootstrap* u datoteku *global.scss*, napisali bismo:

```
@import "~bootstrap/scss/bootstrap";
```

Ukoliko bismo željeli koristiti JavaScript kod iz *Bootstrap*-a, prvo je potrebno instalirati JavaScript ovisnosti koje zahtijeva verzija *Bootstrapa* koja se koristi u aplikaciji:

```
$ yarn add jquery @popperjs/core --dev
```

Nakon instalacije moguće je zahtijevati *Bootstrap* iz bilo koje JavaScript datoteke pozivom:

```
require('bootstrap');
```

Ovime završava opisivanje osnovnih dijelova razvojnog okvira Symfony koji su potrebni za izradu jednostavne web-aplikacije. Filozofija rasta broja značajki koje nudi Symfonya jest da one rastu s potrebama programera koji razvijaju aplikaciju. Symfony tako pored opisanih komponenti nudi razne druge napredne značajke koje ga čine moćnim i fleksibilnim razvojnim okvirom. Neke od njih su komponenta *Messenger* koja pruža mogućnost slanja i primanja poruka, komponenta *Notifier* koja Symfonyu pruža dinamičan način upravljanja slanja poruka putem SMS-a, e-maila i drugih sličnih servisa, zatim komponenta *Serializer* koja pruža serijalizaciju i deserijalizaciju objekata različitih formata te sesije (eng. *session*) koje služe za pohranjivanje informacija o korisniku između zahtjeva.

Poglavlje 2

Web-aplikacija u Symfonyu

Nakon opisa osnovnih komponenti razvojnog okvira Symfony, demonstrirat ćemo njegove mogućnosti opisom složenije web-aplikacije u kojoj će se pored osnovnih, već spomenutih, komponenti koristiti i neke druge naprednije mogućnosti.

2.1 Aplikacija *Gažer*

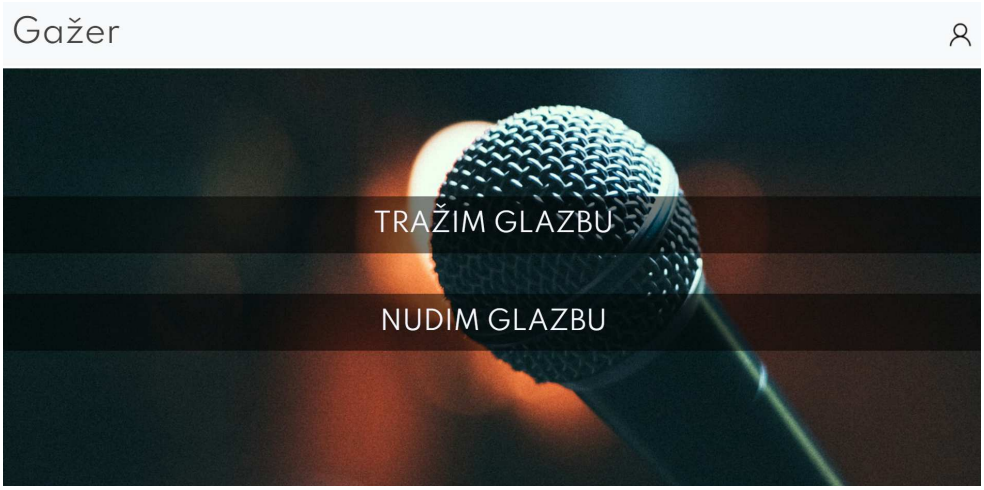
Vlasnici ugostiteljskih objekata nerijetko su u potrazi za glazbenim izvođačima koji bi kod njih pjevali na glazbenim nastupima, ali isto tako glazbeni izvođači tragaju za mjestima gdje se mogu promovirati. Na tržištu zato postoji velika potreba za aplikacijom koja bi ih umrežavala.

Za potrebe ovog rada, izrađena je web-aplikacija pomoću Symfonya prigodnog imena *Gažer* kojoj je cilj vlasnicima ugostiteljskih objekata olakšati pronalazak glazbenih izvođača i komunikacija s njima, ali i glazbenim izvođačima ponuditi sustav putem kojeg mogu organizirati svoje glazbene nastupe. U daljnjem tekstu pod pojmom izvođač smatrat ćemo glazbeni izvođač te pod pojmom nastup glazbeni nastup.

Korisnički načini rada

Aplikacija prepoznaje tri vrste korisnika: one koji traže izvođače, one koji nude nastupe i administratora.

Za pristup aplikaciji, potrebno je imati korisnički račun. Na Slici 2.1 prikazana je naslovna stranica web-aplikacije. Ako nismo prijavljeni, a pokušavamo pristupiti jednoj od ponuđenih opcija, aplikacija nas preusmjerava na stranicu za prijavu gdje nam se pored obrasca za prijavu nudi opcija da se registriramo ukoliko nemamo korisnički račun. U obrascu za registraciju na Slici 2.2, pored unosa osobnih podataka, nudi nam se izbor



Slika 2.1: Početna stranica aplikacije

Slika 2.2: Obrazac za registraciju

između opcija tražimo li izvođače ili nudimo nastupe. U daljnjem tekstu pokazat ćemo razlike između ova dva korisnička načina rada.

Korisnici koji traže izvođače mogu pregledavati korisničke profile onih koji nude nastupe, slati upite za nastupe, vidjeti povijest i stanje svojih poslanih upita te izvođaču ostaviti komentar i ocjenu za odrađeni nastup. Korisnici koji nude nastupe mogu uređivati svoj korisnički profil, vidjeti druge korisničke profile izvođača, prihvatiti ili odbiti upite za nastupe, vidjeti povijest svojih nastupa i dobivene ocjene i komentare.

U gornjem desnom kutu aplikacije nalazi se ikona za prijavu koja je popunjena crnom

bojom ukoliko je korisnik prijavljen, u protivnom ima samo obrub. Ovisno o vrsti prijavljenog korisnika koji klikne na ikonu, prikazuje se odgovarajući popis nastupa. U nastavku opisujemo razliku popisa za one koji traže izvođače i one koji nude nastupe.

Slika 2.3: Obrazac za slanje upita za nastup

Profili izvođača

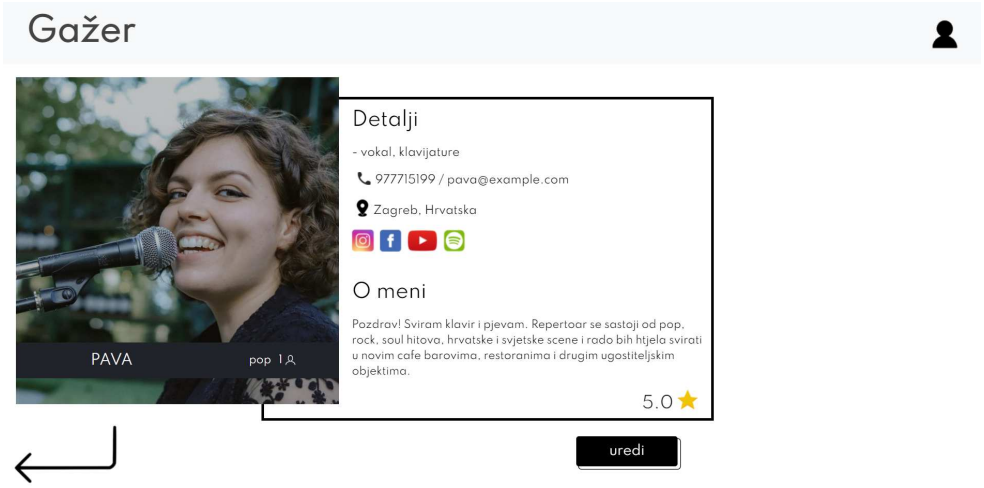
Obje vrste korisnika mogu pregledavati korisničke profile izvođača. Međutim, razlika je u tome koja im se opcija nudi na korisničkom profilu. Oni koji traže izvođače imaju opciju slanja upita izvođaču čiji prikaz vidimo na Slici 2.3, dok oni koji nude nastupe, ukoliko posjećuju svoj korisnički profil, imaju opciju njegovog uređivanja.

Korisnik koji nudi nastup u obrascu za uređivanje korisničkog profila ima mogućnost unijeti linkove na svoje korisničke profile na društvenim mrežama ili glazbenim aplikacijama. Na profilu će biti prikazane ikone onih društvenih mreža odnosno glazbenih aplikacije čije linkove unese u obrazac. Ispod obrasca za uređivanje nudi se i opcija brisanja korisničkog profila. Na Slici 2.4 vidimo primjer korisničkog profila izvođača ukoliko ga posjećuje sam korisnik.

Popis nastupa onih koji traže izvođače

Korisnik koji traži izvođače ima podijeljen popis nastupa na one koje čekaju odgovor na upit, dogovorene i odrađene nastupe.

Svaki nastup određen je datumom, vremenom, trajanjem i izvođačem. Pritiskom na gumb *odrađeno* uz dogovoreni nastup iskače prozor s obrascom u kojeg je moguće unijeti



Slika 2.4: Profil izvođača

ocjenu i komentar. Nakon unosa, nastup prelazi u popis odrađenih. Pored spomenutih podataka, uz svaki nastup nalaze se i unesene ocjene i komentari.

Popis nastupa onih koji nude nastupe

Korisnik koji nudi nastup u svom popisu nastupa ima popis novih upita, popis dogovorenih i odrađenih nastupa. U popisu novih upita uz svaki nastup ima opciju prihvatiti ili odbiti ga. U popisu odrađenih nastupa može vidjeti dobivenu ocjenu i komentar od korisnika koji ga je angažirao za taj nastup.

2.2 Entiteti

Web-aplikacija *Gažer* koristi nekoliko entiteta koji imaju razna svojstva bitna za rad aplikacije i za koja je potrebno da budu pohranjena u bazi. Navedimo sve entitete, objasnimo njihove međusobne relacije i istaknimo neke njihove bitne metode.

User

Entitet *User* sadrži osnovne podatke o svakom korisniku aplikacije. Pomoću njega je ostvarena autentifikacija. Njegovi su elementi:

- *id* predstavlja primarni ključ i služi za identifikaciju
- *email* sadrži jedinstvenu e-mail adresu koja služi za prijavu korisnika

- *password* sadrži hashiranu lozinku korisnika
- *isVerified* označava je li korisnik potvrdio svoju e-mail adresu, odnosno je li pristupio linku na e-mailu koji mu je poslan prilikom registracije
- *roles* predstavlja tip korisnika, odnosno je li on administrator, onaj koji traži ili onaj koji nudi glazbeni nastup

Entitet je u relaciji „jedan naprema jedan" s entitetima *Performer* i *Seeker*. Metodom *getRoles()* dohvaćamo, a sa *setRoles()* postavljamo vrstu korisnika aplikacije. Razlikujemo dvije vrste korisnika: one koji nude nastupe i one koji traže izvođače. Korištenjem metode *getPerformer()* moguće je dohvatiti podatake o onom koji nudi glazbeni nastup ukoliko je on prijavljeni korisnik, dok metoda *getSeeker()* omogućava dohvaćanje podatka o onome koji nastup ukoliko je on prijavljeni korisnik.

Seeker

Entitet *Seeker* čuva podatke o korisniku aplikacije koji traži izvođače. Njegovi su elementi:

- *id* predstavlja primarni ključ i služi za identifikaciju
- *name* sadrži ime ugostiteljskog objekta
- *username* sadrži korisničko ime ugostiteljskog objekta
- *location* sadrži lokaciju ugostiteljskog objekta

Entitet je u relaciji s entitetom *User* preko svojstva *user_id*, a s entitetom *Gaze* je u relaciji „jedan naprema mnogo". Metodom *getGaze()* moguće je dohvatiti sve nastupe koje je određeni *Seeker* tražio, dok je koristeći metodu *addGaze()* moguće dodati novi nastup, a s *removeGaze()* obrisati isti iz skupa.

Performer

Entitet *Performer* čuva podatke o korisniku aplikacije koji nudi nastup. Budući da većinu tih podataka prikazujemo na profilu izvođača, on sadrži najviše elemenata. To su:

- *id* predstavlja primarni ključ i služi za identifikaciju
- *name* sadrži ime izvođača
- *username* sadrži korisničko ime izvođača

- *category* označava kategoriju kojoj izvođač pripada (na primjer, vokal, gitara)
- *number_of_members* sadrži broj članova koji čine izvođača
- *location* sadrži lokaciju izvođača
- *genre* sadrži žanr koji izvodi izvođač
- *phone_number* predstavlja broj telefona izvođača
- *description* predstavlja opis izvođača
- *instagram* sadrži link na profil izvođača na *Instagramu*
- *facebook* sadrži link na profil izvođača na *Facebooku*
- *youtube* sadrži link na profil izvođača na *Youtube-u*
- *spotify* sadrži link na profil izvođača na *Spotifyju*
- *image_filename* sadrži naziv slike profila izvođača
- *average_rating* sadrži prosječnu ocjenu koju je izvođač dobio od vlasnika ugostiteljskih objekata gdje je imao nastup

Entitet je u relaciji s entitetom *User* preko svojstva *user_id*, a s entitetom *Gaze* je u relaciji „mnogo naprema mnogo". Kao i kod entiteta *Seeker*, izdvojimo metodu *getGaze()* pomoću koje je moguće dohvatiti sve prijašnje nastupe izvođača, metodu *addGaze()* pomoću koje je moguće dodati novi nastup te *removeGaze()* pomoću koje je moguće obrisati nastup.

Gaze

Entitet *Gaze* čuva podatke o nastupima. Njegovi su elementi:

- *id* predstavlja primarni ključ i služi za identifikaciju
- *state* sadrži stanje nastupa
- *date* sadrži datum nastupa
- *duration* sadrži trajanje nastupa
- *rating* predstavlja ocjenu koju je izvođač dobio za određeni nastup od strane vlasnika ugostiteljskog objekta gdje je nastup odrađen

- *comment* sadrži komentar koji je vlasnik ugostiteljskog objekta ostavio izvođaču za određeni nastup

Entitet je u relaciji s entitetom *Seeker* preko svojstva *seeker_id*, a s entitetom *Performer* preko svojstva *performer_id*. Metodom *getPerformer()* moguće je dohvatiti izvođače određenog nastupa, s metodom *addPerformer()* nastupu dodati izvođača, a s *removePerformer()* obrisati izvođača nekog nastupa. Varijabla *state* poprima tri stanja nastupa: „*u čekanju*", „*potvrđen*" ili „*odrađen*".

2.3 Rute, kontroleri i odgovori

Kako bi korisnik pristupio određenoj stranici web-aplikacije *Gažer*, koristi se ruta i pripadna metoda kontrolera koja određuje odgovor na rutu. Zadaću koju obavlja metoda kontrolera ovisi o tipu korisnika koji joj pristupa. Opišimo sve rute, metode kontrolera i njihove odgovore koji omogućuju da određeni korisnik dođe do željenog sadržaja.

Početna stranica

Početnoj stranici aplikacije pristupa se rutom / koja odgovara na HTTP zahtjeve GET i HEAD. Ruta poziva metodu *homepage()* kontrolera *DefaultController*. Odgovor se ostvaruje korištenjem predloška *homepage.html.twig* koji prikazuje početnu stranicu na kojoj se nudi izbor između: *tražim glazbu* i *nudim glazbu*.

Prijava korisnika

Pomoću rute *login* pristupamo stranici za prijavu korisnika. Ruta poziva metodu *login()* kontrolera *AuthController* koji odgovara predloškom *security/login.html.twig* i prosljeđuje mu varijable *$lastUsername* i *$error*.

Budući da se autentifikacija u Symfonyu ostvaruje putem konfiguracije u datoteci *security.yaml*, prilikom pristupa ruti *login*, provjerava se postoji li u sesiji spremljen neki prijavljeni korisnik te ukoliko postoji, prikazuje stranicu na kojoj piše da smo prijavljeni kao korisnik imena spremljenog u prosljeđenu varijablu *$lastUsername*. Ako u sesiji nema spremljenog korisnika, prikazuje se obrazac za prijavu. Kada se šalje ispunjen obrazac, ruta se poziva s HTTP zahtjevom POST koja pak poziva metodu *authenticate()* kontrolera *LoginFormAuthenticator()* koja obavlja autentifikaciju. Ako je autentifikacija uspješna, korisnik se sprema u sesiju i poziva se metoda *onAuthenticationSuccess()* koja preusmjerava korisnika na početnu stranicu, a ako je neuspješna, vraća na obrazac za prijavu i ispisuje pripadajuću grešku.

Ruta *logout* poziva metodu *logout()* kontrolera *AuthController* koja također ne sadrži nikakvu logiku nego je autentifikacija konfigurirana pomoću vatrozida u datoteci *security.yaml*. Ukoliko postoji spremljen korisnik u sesiji, on se briše iz sesije i preusmjerava na početnu stranicu, a ukoliko ne postoji, preusmjerava se na rutu *login*.

Registracija korisnika

Ruta */register* poziva metodu *register()* kontrolera *RegistrationController* koja odgovara predloškom *registration/register.html.twig* koji prikazuje obrazac za registraciju ukoliko je ruta pozvana s HTTP zahtjevom GET. Ukoliko je ruta pozvana s HTTP zahtjevom POST, validiraju se uneseni podaci i ako je validacija uspješna, stvara se novi korisnik i šalje mu se e-mail s linkom za potvrdu. U slučaju da neki od podataka nije ispravno unesen, korisnik se vraća na obrazac i ispisuje se poruka o grešci.

Zaboravljena lozinka

U obrascu prijave korisniku se nudi opcija zaboravljene lozinke. To je omogućeno putem rute *reset-password* koja pripada kontroleru *ResetPasswordController*.

Ruta poziva metodu *request()* koja, ukoliko je ruta pozvana s HTTP zahtjevom GET, pomoću predloška *reset_password/request.html.twig* prikazuje obrazac za unos e-maila, a ukoliko je pozvana s HTTP zahtjevom PUT obavlja validaciju unosa te poziva metodu *processSendingPasswordResetEmail()* koja na e-mail adresu šalje zahtjev za promjenom lozinke te preusmjerava na rutu *reset-password/check-email*. Ova ruta zatim poziva metodu *checkEmail()* koja odgovara pomoću predloška *reset_password/check_email.html.twig*. Ukoliko je unesen valjan e-mail, predložak ispisuje poruku da je na njega poslan link pomoću kojeg je moguće promijeniti lozinku u određenom vremenu koji se prati putem tokena.

Ruta koju korisnik dobiva na e-mail je *reset-password/reset/token* i poziva se s HTTP zahtjevom GET ili HEAD. Ruta poziva metodu *reset()* koja, ako joj je prosljeđen token, odgovara predloškom *reset_password/reset.html.twig* koji prikazuje obrazac za promjenu lozinke. U protivnom, ispisuje poruku o greški.

Popis svih korisnika koji nude nastupe

Do popisa svih korisnika koji nude nastup dolazi se rutom *performer/overview* koja poziva metodu *index()* kontrolera *PerformerController*. Ruta odgovara na HTTP zahtjeve GET i HEAD. Metoda odgovara predloškom *performer/index.html.twig* kojem prosljeđuje niz svih korisnika koji nude nastup kojeg potom predložak prikazuje.

Profil korisnika koji nudi nastup

Ruta *performer/{username}* poziva metodu *show()* kontrolera *PerformerController* koji odgovara predloškom *performer/show.html.twig*. Ruta odgovara na HTTP zahtjeve GET i POST. Ukoliko je prijavljen korisnik koji nudi nastup, posjetom ruti može pristupiti nekom od profila izvođača. Ukoliko posjećuje svoj profil, nudi mu se mogućnost posjeta stranici za uređivanja profila. Ako je prijavljen korisnik koji traži izvođače, kako bi posjetio profil nekog izvođača, pristupit će istoj ruti, ali uz prikaz profila neće imati mogućnost uređivanja nego slanja upita za nastup.

Raspored glazbenih nastupa

Ruta *schedule/{username}/{action}/{id_gaze}* poziva metodu *showSchedule* kontrolera *ScheduleController*. Ruta sadrži parametre *action* i *id_gaze* koji mogu biti *null*. Njih koristimo kako bismo identificirali s kojim nastupom i koju radnju korisnik želi poduzeti.

Ukoliko ruti pristupi korisnik koji traži izvođača, metoda filtrira nastupe u tri kategorije: one koji čekaju odgovor na upit, dogovorene i odrađene nastupe. Metoda odgovara predloškom *seeker_schedule_gaze.html.twig* kojem prosljeđuje filtrirane nastupe koje predložak prikazuje. Korisniku se nudi i mogućnost da dogovoreni nastup premjesti u odrađenu pri čemu se preusmjerava na rutu s definiranim parametrom *id_gaze*.

Ukoliko rutu posjeti korisnik koji nudi glazbu, metoda na isti način filtrira nastupe te poziva predložak *performer_schedule_gaze.html.twig*. Korisniku se nudi mogućnost odbijanja i prihvaćanja nastupa koji čekaju odgovor. To se ostvaruje pomoću parametra *action* koji govori je li se radi o odbijanju ili prihvaćanju te parametra *id_gaze* koji govori kojem nastupu se želi promijeniti stanje.

Uređivanje profila izvođača

Korisnik koji nudi nastup jedini može pristupiti uređivanju svog profila. To čini putem rute *performer/username/edit*. Ruta poziva metodu *edit* kontrolera *PerformerController* koji definira obrazac kojeg prosljeđuje predlošku *performer/edit.html.twig*. Ukoliko je validacija podnesenog obrasca uspješna, izmjene se spremaju u bazu, a korisnik se preusmjerava na svoj profil.

2.4 Notifikacije

Kako bi se korisnici koji nude nastupe obavijestili da imaju novi upit za nastup ili da su korisnici koji traže izvođače dobili odgovor na poslani upit, aplikacija *Gažer* koristi

naprednu komponentu u Symfonyu imena *Notifier*. U programiranju, takve poruke o obavijestima obično nazivamo notifikacijama (eng. *notification*). Symfony koristi dinamičan način upravljanja notifikacijama i pruža različite kanale za njihovo slanje. Web-aplikacija *Gažer* koristi komponentu *Notifier* za slanje notifikacija korisnicima putem e-maila. *Notifier* se koristi pomoću komponente *Symfony Mailer* i klase *NotificationEmail*.

U aplikaciji *Gažer* korisnik koji traži izvođača šalje upit za nastup korisniku koji nudi nastup putem obrasca na njegovom profilu. Validacija je obrasca uspješna ukoliko su ispravno uneseni svi podaci te ukoliko korisnik nema već dogovoren nastup na traženi datum. U protivnom, validaciju smatramo neuspješnom i ispisujemo poruku o nemogućnosti slanja upita za taj datum. Ukoliko je validacija uspješna, unutar metode *show()* kontrolera *PerformerController* šalje se mail pomoću metode *sendEmail()* koja pripada klasi *EmailSender*. Koristeći klasu *TemplatedEmail* metoda definira sadržaj e-maila pomoću predloška *performer/email_request_Gaze.html.twig* kojem su proslijeđene informacije o tome tko je poslao upit i za koji termin.

Ukoliko korisnik koji nudi nastup ima upit za njega, ima ga mogućnost prihvatiti ili odbiti. Notifikacija putem e-maila o njegovom odabiru šalje se korisniku koji čeka odgovor unutar metode *showSchedule()* kontrolera *ScheduleController*. Sadržaj e-maila uređuje predložak *performer/email_response_Gaze.html.twig* kojem su proslijeđene informacije o korisniku koji je odgovorio na upit, samom odgovoru i datumu nastupa na koji se odnosi odgovor.

Notifikacija putem e-maila također se pošalje kada korisnik koji traži izvođača ocijeni i ostavi komentar za odrađen nastup. E-mail se šalje unutar metode *showSchedule()* kontrolera *ScheduleController*, a sadržaj e-maila uređuje se pomoću predloškom *performer/email_evaluation_Gaze.html.twig* koji u e-mailu prikazuje korisnika, ocjenu, komentar i nastup na koji se evaluacija odnosi.

2.5 Sigurnost

Sigurnost u aplikacijama izrađenim pomoću Symfonya ostvarena je konfiguracijom u datoteci *security.yaml* i definiranjem uloga koje se dodjeljuju korisnicima. U aplikaciji *Gažer* definirane su četiri korisničke uloge. Svi korisnici imaju ulogu *ROLE_USER*, korisnici koji nude nastup imaju i ulogu *ROLE_PERFORMER*, a korisnici koji traže izvođače imaju ulogu *ROLE_SEEKER*. Postoji samo jedan korisnik koji ima ulogu administratora imena *ROLE_ADMIN*.

Kako bi se prepoznala uloga koju prijavljeni korisnik ima, aplikacija koristi metodu *is_granted()* unutar predloška te *getRoles()* unutar metode kontrolera.

Symfony nudi jednostavan način ostvarivanja uloge administratora. Ruti koja počinje s *admin* isključivo je omogućen pristup administratoru pomoću varijable *access_control* koja

se nalazi unutar konfiguracijske datoteke. Administrator jedini ima mogućnost uređivanja i brisanja korisničkih profila.

2.6 Korisničko sučelje

U web-aplikaciji *Gažer* za izradu korisničkog sučelja korištena je biblioteka *Bootstrap*.

Bootstrap je jedna od najpopularnijih biblioteka za HTML, CSS i JavaScript. Služi za razvoj responzivnih web-aplikacija kojima je cilj automatsko prilagođavanje različitim uređajima, odnosno da korisničko sučelje izgleda dobro bez obzira na dimenzije zaslona. Koristi se za lakši i brži razvoj web-aplikacija i uključuje HTML i CSS predloške dizajna za obrasce, gumbe, tablice, slike i mnoge druge komponente aplikacije, a može koristiti i JavaScript dodatke.

U radu su korištene ugrađene teme za uređivanje obrazaca koje dolaze sa Symfonyem (eng. *Symfony Built-In Form Themes*). Zbog različitih vrsta obrazaca korištena je opcija umetanja teme na pojedinačne obrasce putem varijable *form_theme* prije definiranja samog obrasca u predlošku.

Ovime završavamo opis bitnih dijelova web-aplikacije izrađene pomoću Symfonya. Sa svojim brojnim ugrađenim komponentama, ali i velikim mogućnostima za proširenje istih, možemo zaključiti da je Symfony izrazito fleksibilan razvojni okvir koji olakšava i ubrzava izradu web-aplikacija. Podržavajući strukturiranje koda po obrascu MVC, Symfony omogućava izgradnju slojevitog koda jednostavnog za čitanje. Rad sa Symfonyem uvelike ubrzava razvoj web-aplikacije jer preko naredbi u komandnoj liniji Symfony omogućava generiranje koda za najčešće tražene komponente aplikacije. Također, Symfony je poznat po tome da nastoji pratiti najnovije trendove, te da neprestano traži načine kako se poboljšati pa je tako tijekom stvaranja ovog rada nastala nova verzija Symfony 6. Mnoge poznate web-aplikacije koriste Symfony i vjeruju mu zbog njegove stabilnosti i održivosti.

Bibliografija

[1] Sohail Salehi. *Mastering symfony*. Packt Publishing Ltd, 2016.

[2] *Symfony Documentation*. `https://symfony.com/doc/current/index.html`. posjećeno u siječnju 2022.

[3] *Symfony Casts*. `https://symfonycasts.com/screencast/symfony`. posjećeno u siječnju 2022.

[4] Fabien Potencier. *Symfony 5: The Fast Track*. Symfony SAS, 2020.

[5] *Twig. The flexible, fast, and secure template engine for PHP*. `https://twig.symfony.com/`. posjećeno u siječnju 2022.

[6] *Get started with BootStrap*. `https://getbootstrap.com/docs/5.1/getting-started/introduction/`. posjećeno u siječnju 2022.

Sažetak

U ovom radu predstavili smo razvojni okvir Symfony za programski jezik PHP. Symfony se sastoji od skupa PHP komponenti koje nude rješenja za bržu i lakšu izradu web-aplikacije.

U prvom dijelu rada naveli smo zahtjeve za instalaciju i sam način instalacije Symfonya. Prateći pojmove vezane za softverski obrazac MVC, objašnjavali smo ulogu osnovnih komponenti rute i kontrolera, način na koji se spomenute komponente dodaju u projekt te njihovo korištenje u aplikaciji. Pokazali smo rješenje koje Symfony nudi za preslikavanje podataka iz baze u objekte pomoću biblioteke *Doctrine* te sve prednosti vlastitog predloška *Twig* koji služi za izgradnju korisničkog sučelja. Na kraju smo opisali sigurnosnu komponentu koja nudi jednostavnu integraciju gotovih rješenja za procese autentifikacije i autorizacije.

U drugom dijelu rada opisali smo pristup izradi web-aplikacije *Gažer* koja demonstrira sve bitne komponente Symfonya, ali i daje uvid u neke napredne značajke. Najprije smo opisali motivaciju za izradu aplikacije, a potom rješenja za probleme koje je izrada nosila. Zatim smo opisali rute i pripadne kontrolere pomoću kojih su ostvarene značajke aplikacije te smo dali uvid u izradu notifikacija. Na kraju rada, objasnili smo tehniku postizanja sigurnosti u aplikaciji te jednostavan način uređenja korisničkog sučelja ugrađenom temom biblioteke *Bootstrap*.

Summary

In this thesis, we have presented the Symfony framework for the PHP programming language. Symfony consists of a set of PHP components that offer solutions for quick and easy web application development.

In the first chapter, we have listed the installation requirements and the method of installing Symfony. Following the terms related to the MVC software architectural pattern, we explain the role of the basic route and controller components, the way these components are added to the project and their use in applications. We have demonstrated the solution that Symfony offers for mapping database data to objects using the *Doctrine* library and all the advantages of its own *Twig* template engine for building user interfaces. Finally, we have described a security component that offers easy integration of ready-made solutions for the authentication and authorization process.

In the second chapter, we described the approach to creating the web application *Gažer*, which demonstrates important components of Symfony, but also provides insight into some advanced features. Firstly, we described the motivation for creating the application, followed by the presentation of the solutions to the problems that the development brought. We then described the routes and associated controllers used to achieve the application's features and provided insight into the creation of notifications. In the end of the chapter, we explained how the security aspects of the application are implemented and we showed an easy way to edit the user interface using the built-in theme of the *Bootstrap* library.

Životopis

Rođena sam 30. kolovoza 1994. godine u Mostaru. Nakon završene Osnovne škole Petra Bakule, upisujem opći smjer u Gimnaziji fra Grge Martića u Mostaru koju završavam 2013. godine. Iste godine upisujem preddiplomski studij matematike na Prirodoslovno-matematičkom fakultetu u Zagrebu. Preddiplomski studij završavam 2018. godine kada i upisujem diplomski studij Računarstvo i matematika na istom fakultetu. Za vrijeme studija aktivno sam sudjelovala u promoviranju matematike putem manifestacije *Dan i noć na PMF-u* te podučavajući matematiku kroz individualne sate.

Od srpnja 2020. godine dio sam IVI (eng. *in-vehicle infotainment*) tima u tvrtki *Rimac Automobili*.