MVC arhitektura u izradi mrežnih aplikacija

Ahrendt, David

SVEUČILIŠTE U ZAGREBU
FILOZOFSKI FAKULTET
ODSJEK ZA INFORMACIJSKE I KOMUNIKACIJSKE ZNANOSTI
Ak. god. 2019/2020

David Ahrendt

MVC arhitektura u izradi mrežnih aplikacija

Završni rad

Mentor: dr.sc. Kristina Kocijan, izv.prof.

Zagreb, Veljača, 2020.

Izjava o akademskoj čestitosti

Izjavljujem i svojim potpisom potvrđujem da je ovaj rad rezultat mog vlastitog rada koji se temelji na istraživanjima te objavljenoj i citiranoj literaturi. Izjavljujem da nijedan dio rada nije napisan na nedozvoljen način, odnosno da je prepisan iz necitiranog rada, te da nijedan dio rada ne krši bilo čija autorska prava. Također izjavljujem da nijedan dio rada nije korišten za bilo koji drugi rad u bilo kojoj drugoj visokoškolskoj, znanstvenoj ili obrazovnoj ustanovi.

_____

(potpis)

Zahvala

Ovim putem želio bih zahvaliti svojoj mentorici dr.sc. Kristini Kocijan, koja je svojim trudom i dobrom voljom obogatila moj studiji i olakšala pisanje ovog rada. Nadalje omogućila je moje prvo sudjelovanje u PHP zajednici s čime me navela na put uspješnog PHP programera za što sam također jako zahvalan. Nadam se da će još u mnogim generacijama moći probuditi interes za web i PHP programiranje.

Sadržaj

1. Uvod

PHP je danas i dalje jedan od najpopularnijih (The state of the Octoverse, 2019) jezika za razvijanje mrežnih (engl. *web*) aplikacija, iako je nastao u samim početcima web-a. Mnogi smatraju da je PHP zastarjeli jezik upravo zbog toga što je nastao davne 1994. godine, no i dalje se velika većina (PHP Usage Statistics, *n.d.*) web-a bazira na njemu, prema nekim izvorima čak osamdeset posto (Usage statistics of PHP for websites, 2020). Ključ uspješnosti ovog jezika jest velika zajednica (engl. *community*) programera, koji direktno ili indirektno sudjeluju u razvoju jezika. Većina modernih PHP aplikacija koristi Model-Pogled-Upravljač arhitekturu kôd-a te objektno orijentiranu paradigmu, bilo to u nekom prilagođenom (engl. *custom*) rješenju ili koristeći neki od brojnih programskih okvira kao što su, na primjer, Laravel, Symfony, CodeIgniter, Phalcon, Yii i tako dalje (Sinha, 2017).

U ovom radu bit će objašnjen način funkcioniranja Model-Pogled-Upravljač (engl. Model-View-Controller), skraćeno MVC, arhitekture kôd-a iz perspektive PHP programera. S obzirom na kompleksnost tematike, rad će biti podijeljen u nekoliko cjelina počevši od samog začetka jezika, odnosno njegova povijesnog razvoja u 2. poglavlju. Nakon povijesnog pregleda, u 3. poglavlju predstavljen je koncept modularnog programiranja, razvojne strategije s takvim pristupom te alat Composer koji olakšava primjenu tog pristupa u PHP jeziku. Međutim, ubrzo je shvaćeno kako je za pisanje kvalitetnog modularnog kôda nužna njegova standardizacija. Iz tog razloga u 4. poglavlju prolazi se kroz proces standardizacije PHP ekosistema te koji su stupovi te standardizacije. Pošto je fokus rada na MVC arhitekturi u PHP programskim okvirima kroz 5. poglavlje pobliže će biti objašnjeni dijelovi MVC arhitekture te standardni ciklus njihove interakcije, a zatim u 6. poglavlju najbitniji moduli makro PHP programskih okvira. Na samom kraju rada u 7. poglavlju sve opisano bit će predstavljeno kao smislena cjelina na primjeru projekta eRječnik.

2. PHP kroz povijest

Prvu inačicu ovog programskog jezika razvio je Rasmus Lerdorf 1994. godine kao rješenje za praćenje broja pogleda njegovog digitalnog životopisa (Achour et al., 2020a*)*. Taj jednostavan set komandi ili točnije zajedničko sučelje poveznika (engl. *Common Gateway Interface (CGI)*) pisane u C programskom jeziku, Lerdorf naziva *Personal Home Page Tools*, koji se često nazivaju i *PHP Tools* otkud dolazi i današnje ime jezika. Prva verzija PHP-a mogla se koristiti za kreiranje jednostavnih dinamičkih web stranica. Uvođenjem ove verzije PHP-a dodana je mogućnost korištenja varijabli, te ubacivanje HTML-a u kôd. Sintaksa jezika bila je složena po uzoru na PERL programski jezik.

U rujnu 1995. godine Lerdorf je proširio mogućnosti koje je PHP pružao korisnicima i implementirao bazične funkcionalnosti, koje se koriste i u današnjim verzijama PHP-a. U ovom kratkom razdoblju Lerdorf svoj proizvod naziva FI odnosno *Forms Interpreter*, no već u Listopadu iste godine objavljuje kompletno nanovo napisanu verziju koju ponovno naziva PHP. U ovom razdoblju PHP nije bio zamišljen kao nešto što bi se moglo razviti u samostalan jezik, već kao skup komandi za lakše razvijanje dinamičkih web stranica.

2.1. PHP/FI

U Travnju 1996. godine Lerdorf objavljuje prvu verziju PHP jezika koja se može nazvati pravim jezikom, a ne samo skupom komandi. U jezik su bile uključene funkcionalnosti upravljanja bazama podataka (engl. *Database Management System (DBM)*), kolačići (engl. *cookie*), mogućnosti kreiranja prilagođenih funkcija i još mnogo toga. U lipnju te godine PHP/FI dobio je kôd verzije 2.0 što označava pravi početak PHP-a kao jezika (Achour et al., 2020b.).

Popularnost ovog inovativnog jezika pokazuje i podatak Netcraft ankete iz svibnja 1998. koja pokazuje da oko 60 000 servera ima PHP zaglavlje, odnosno PHP jezik instaliran na poslužitelju. Taj broj je značajan za tadašnji web i mladi jezik kao što je PHP, te označavao oko 1 posto svih poslužitelja na mreži u to vrijeme (Achour et al., 2020b*).*

Na slici 1 vidljiva je sintaksa PHP/FI programskog jezika koja se sastoji od nekoliko blokova. Počevši s blokom za uključivanje vanjskog HTML dokumenta za zaglavlje stranice. Potom slijedi blok provjere kojim web preglednikom je korisnik došao do stranice te u slučaju da se radi o tadašnjem Netscape Navigatoru, korisnik na to bude upozoren. Sljedeći blok zatim, na vrlo primitivan način, obavlja ono za što se PHP i danas koristi, interakciju između stranice i baze podataka pomoću jezika SQL. Jednostavnom logičkom provjerom traži korisnika u bazi podataka te ako ga ne nađe javlja da taj zapis ne postoji, a u protivnom pozdravlja korisnika njegovim imenom i javlja mu neki podataka vezan uz njegov račun. Na kraju je opet napisan blok za uključivanje vanjskog HTML koda, ali u ovom slučaju za podnožje web stranice.

```
<!--include /text/header.html-->

<!--getenv HTTP_USER_AGENT-->
<!--ifsubstr $exec_result Mozilla-->
  Hey, you are using Netscape!<p>
<!--endif-->

<!--sql database select * from table where user='$username'-->
<!--ifless $numentries 1-->
  Sorry, that record does not exist<p>
<!--endif exit-->
  Welcome <!--$user-->!<p>
  You have <!--$index:0--> credits left in your account.<p>

<!--include /text/footer.html-->
```

Slika 1*. Primjer sintakse PHP/FI verzije jezika*[1]

1 Izvor slike <https://en.wikipedia.org/wiki/PHP>

2.2.PHP 3

PHP 3.0 je prva verzija jezika koja usko nalikuje današnjem jeziku. Ovu verziju PHP-a razvili su Andi Gutmans i Zeev Suraski, koji su shvatili da u prijašnjoj verziji jezika nedostaju kritične značajke koje su im bile potrebne za razvoj eKomerc stranice. U kolaboraciji s Lerdofom, kojeg su upoznali preko interneta, bacili su se na kompletno ponovno pisanje PHP rastavljača sintakse. U svibnju 1998. mnogo drugih programera pridružilo se Lerdorfu, Gutmansu i Surasku te je tad započela testna faza ove verzije jezika. Nakon nepunih devet mjeseci testiranja i razvoja, službeno je objavljen PHP 3.0.

Zanimljiv podatak jest da je PHP 3 na danu objavljivanja već bio instaliran na otprilike 70 000 domena i više nije bio limitiran na samo POSIX operativne sustave, već ga je bilo moguće koristiti i na Macintosh i Windows sustavima. Na vrhu popularnosti ove verzije, PHP 3 bio je instaliran na otprilike deset posto domena tadašnjeg web-a (Achour et al., 2020a).

2.3.PHP 4

Dok je PHP 3.0 još bio u fazi testiranja, Andi Gutmans i Zeev Suraski su započeli s ponovnim pisanjem PHP rastavljača sintakse. Novi rastavljač nazvali su 'Zend Engine', čiji naziv je nastao spajanjem njihovih imena Zeev i Andi. Ovaj rastavljač tvori bazu za novu verziju PHP-a te predstavlja veliki napredak u razvoju jezika.

Cilj za ovu verziju je bio poboljšanje performansi za kompleksne aplikacije te omogućavanje veće modularnosti jezika. Prvi puta se počinje koristiti objektno orijentirana sintaksa, te je dodano mnogo novih značajki poput HTTP sezona, siguran način za dohvaćanje podataka koje unosi korisnik, pa čak i neke nove konstrukcije jezika.

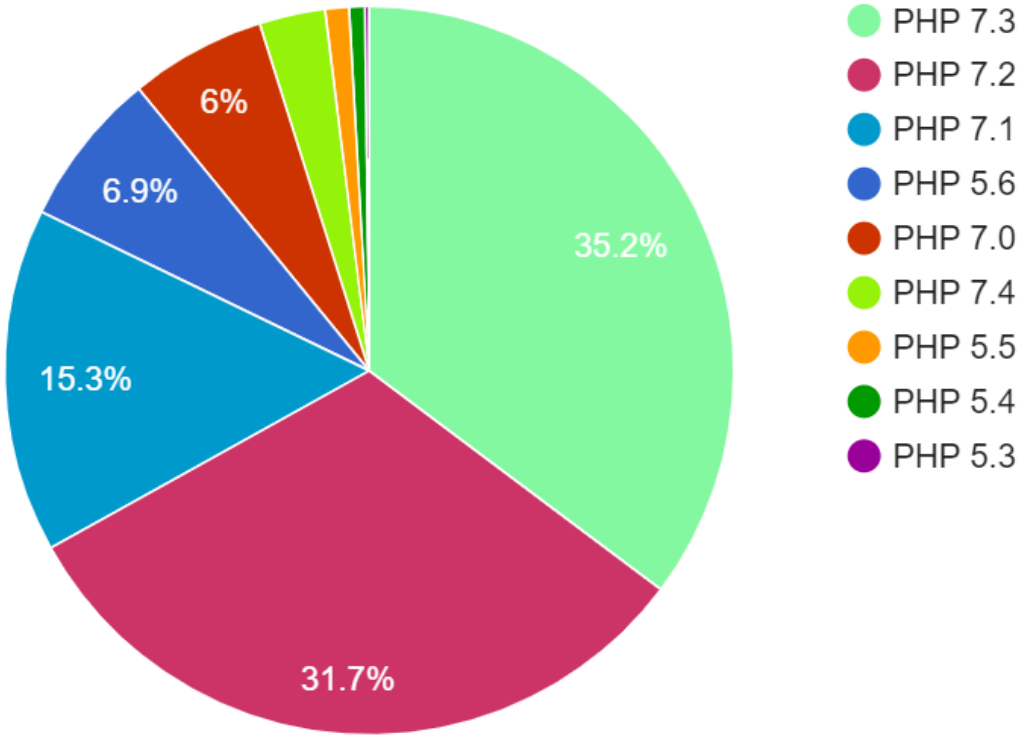
'Zend Engine' i uz njega PHP 4 objavljeni su u lipnju 2000. godine, gotovo dvije godine nakon objavljivanja prijašnje verzije (Achour et al., 2020a).

2.4.PHP 5 i novije verzije

PHP 5 objavljen je 2004. godine, baziran na novom Zend Engine 2.0', s novom strukturom objekata i mnogim novim značajkama koje su uvelike olakšale implementaciju objektno orijentirane sintakse (Zend Engine version 2.0, 2003). Uvedene su konstrukcije poput tvornica (engl. *Factory)* i PHP podatkovni objekti (*engl. PHP Data Objects),* skraćeno PDO. Tvornice omogućuju kreiranje objekata s centraliziranim metodama (Zend Engine version 2.0, 2003), a PHP podatkovni objekti omogućuju jednostavan i konstantan način za upravljanje podacima u bazi podataka (Achour et al., 2020c). PHP 5 bila je jedna od najduže korištenih verzija te možemo pronaći web stranice koje i danas koriste tu verziju PHP-a, iako ista nije preporučljiva za korištenje. Službeni prestanak razvoja ove verzije PHP-a bio je 31. prosinca 2018. (Purkis, 2018).

U 2015. godini izašla je verzija 7.0, koja je i danas najnovija glavna verzija, s tim da se predviđa da će iduća verzija 8.0 biti objavljena u zadnjem kvartalu 2020. (Domogalla, 2020).

Pretpostavlja se da je PHP danas instaliran na stotinama milijuna poslužitelja, te da pokreče čak 80 posto današnjeg web-a kao što je već spomenuto u uvodu rada. Mnoge velike i poznate web stranice, poput Facebook-a, Wikipedije i Wordpress-a (Kamaruzzaman, 2020) danas koriste PHP za razvoj svojih web aplikacija. Teško je pronaći točne podatke oko toga koje se točno verzije danas koriste na web-u, ali jedan od najboljih uvida u stvarno stanje nudi statistički pregled na web stranici *packagist.org*, koja se bavi distribucijom PHP paketa. Na slici 2 možemo razmotriti vizualnu reprezentaciju tih podataka iz jedanaestog mjeseca 2019. godine iz kojeg je jasno vidljivo kako PHP nije arhaični jezik koji prevladava samo u starim projektima na webu, već je velika većina projekata, preko 80%, napisano u najnovijoj inačici PHP-a 7.



***Slika 2**. Statistika rasprostranjenosti pojedinih verzija PHP jezika iz 11. mjeseca 2019. godine* [2]

2 Izvor: https://blog.packagist.com/content/images/2019/12/image.png

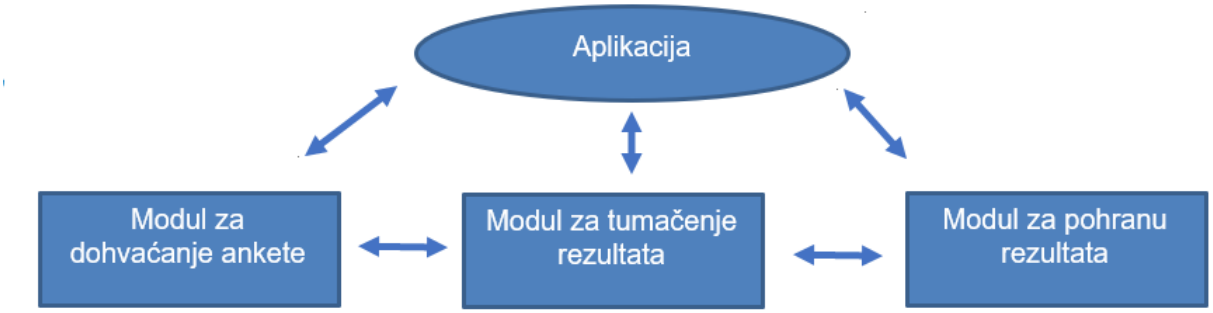
3. Modularno Programiranje

Implementacijom objektno orijentirane paradigme u jezik koja pogoduje decentraliziranom razvoju kôda, PHP zajednica je evoluirala. Iz centraliziranog modela dizajniranja programskih okvira, u model distribuiranog ekosistema s efektivnim, specijaliziranim i interoperabilnim komponentama. Ideja distribuiranog načina razvoja softvera oslanja se na neka temeljna znanja i metode koja će biti objašnjena u ovom poglavlju. Ona uključuju modularno programiranje i pripadajuću razvojnu strategiju koja pogoduje razvoju takvih programskih rješenja, te softver koji je zauvijek promijenio način razvoja modularnih PHP aplikacija.

Modularno programiranje je tehnika dizajniranja softvera koja naglašava razdvajanje kôda u neovisne cjeline koje se u svakom trenutku mogu uključiti ili isključiti i to bez utjecaja na ostatak sustava (Šribar, Motik, 2010; Matković, 2006). Objektno orijentirana paradigma pogoduje ovakvom dizajnu sustava, jer sve komponente mogu biti međusobno neovisne. Za razliku od takozvanih monolitnih aplikacija, ovaj princip omogućuje da se velike aplikacije razdvoje u manje funkcionalne cjeline.

Ako želimo pokazati primjenu ovog principa na modernoj web aplikaciji, možemo, na primjer, zamisliti sustav za provođenje online anketa. U jednostavnoj aplikaciji takve vrste možemo kôd razdvojiti u tri funkcionalne cjeline ili modula, koji su međusobno neovisni i samim time se mogu upotrijebiti i na drugačiji način. Strukturu takve aplikacije možemo vidjeti na slici 3.

***Slika 3**. Primjer strukture modularne aplikacije*[3]



***Slika 4**. Primjer monolitne strukture aplikacije*[4]

Ako usporedimo sliku 3 koja je primjer modularne strukture i sliku 4 koja je primjer monolitne aplikacije možemo uočiti jednu bitnu razliku. U modularnom dizajnu nema direktne komunikacije između pojedinih modula već se za komunikaciju koristi aplikacijski sloj koji povezuje funkcionalnosti sva tri modula. To osigurava da svaki od modula može raditi zasebno te je tako neovisan od drugih modula. Kod monolitnog dizajna sustava, iako je kôd često također razdvojen u cjeline, kao što vidimo u primjeru sa slike 4, svaka od cjelina se direktno oslanja na rezultate i operacije koje se obavljaju u nekoj drugoj od cjelina. Zbog te zavisnosti nije moguće samostalno funkcioniranje cjelina, i time nije moguće ponovno upotrebljavanje kôd-a. Moderne PHP aplikacije trebale bi težiti modularnom pristupu razvoja te samim time samostalnosti komponenata.

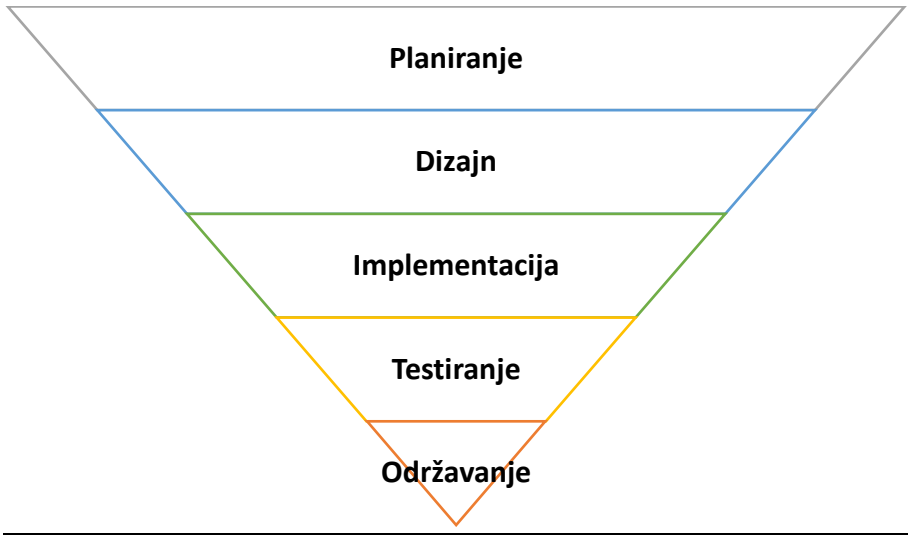
[3] Autorsko djelo
[4] Autorsko djelo

3.1.Razvojna strategija modularnih aplikacija

AGILE je model razvoja softvera, odnosno skup metoda za planiranje, organizaciju i vođenje projekata (Denning, 2019). Značajke ovog pristupa su mali fleksibilni timovi koji rad obavljaju u manjim cjelinama (engl. *Sprint*), što donosi velike prednosti naspram klasične „vodopad" (engl. *waterfall*) metodologije te se izrazito dobro uklapa s modularnim pristupom razvoja programa.

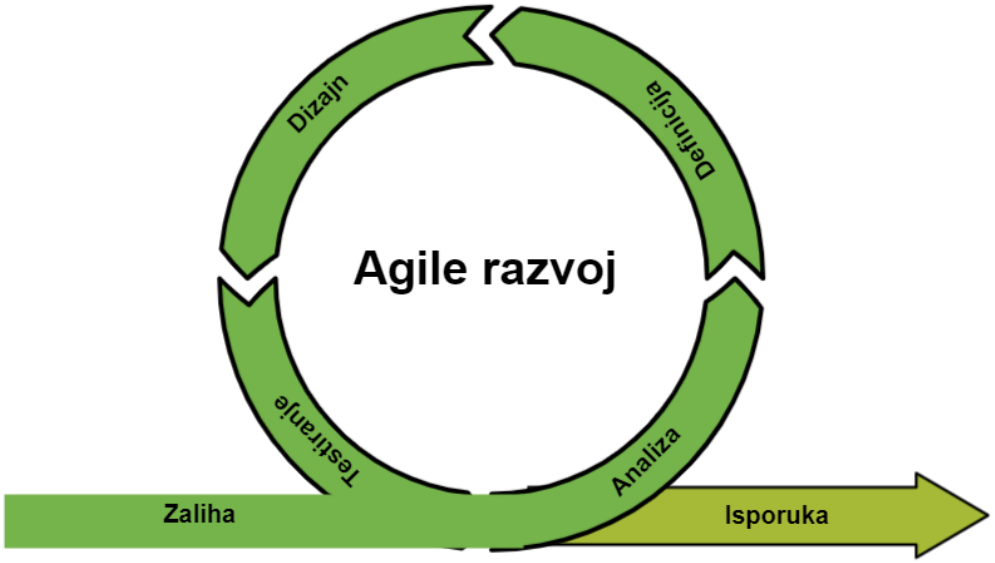
Kod klasične metodologije koraci planiranja i dizajna poprilično su dugi, jer se piše detaljna dokumentacija koja obuhvaća svaki aspekt proizvoda. Proizvod se daje klijentu na korištenje tek nakon završetka svih faza razvoja koje, kod velikih projekata, mogu trajati i nekoliko godina.

Kod AGILE pristupa, faza planiranja je kratka, te se odmah kreće na razvoj proizvoda. Faze razvoja traju od dva do četiri tjedna, nakon kojih se na sastanku s klijentom diskutira o idućoj razvojnoj fazi (Dingsøyrab et al., 2012). Ovaj pristup, kao što samo ime kaže, pruža agilne prilagodbe kôd-a i prioriteta u razvoju, te se u pravilu nakon svakog sprinta isporučuje dio aplikacije. Razlike možda najbolje možemo uočiti na slikama 5 i 6 pomoću kojih možemo usporediti pristupe.



Slika 5. Grafički prikaz „Waterfall" metodologije razvoja[5]

5 Prilagođeno na hrvatski prema izvoru < https://1.bp.blogspot.com/-NnXAeykbJ0I/XaE4aRmzZqI/AAAAAAAAT1s/9nXTY5O5lNIkaM5sDrd6m3J9KYWW7iKUACLcBG



Slika 6. *Grafički prikaz agilne metodologija razvoja*[6]

3.2. Composer

Composer je upravljač ovisnostima (engl. *dependency manager*) nastao prema uzoru na NPM u Node.js-u. Iako *Composer* nije direktno dio samog PHP jezika, to je software koji je uveliko olakšao razvijanje modularnih PHP aplikacija, a prvi put je objavljen 2012. godine (Brown, 2013). Bitno je naglasiti i da *Composer* nije upravljač paketima (engl. *Package manager*) jer se koristi za upravljanje ovisnostima unutar projekta, iako ima mogućnost instaliranja paketa na globalnoj razini sustava (Boggiano, 2012). On omogućava lako dodavanje PHP paketa koje onda možemo koristiti u kôd-u naše aplikacije.

Prvi korak za korištenje *Composer*-a je instalacije *Composer* softvera na naše računalo ili skidanja samostalne *Composer* datoteke koju pokrećemo uz pomoć PHP-a. Dostupne pakete možemo potražiti na mrežnoj stranici packagist.org ili Github-u[7] koji je mrežna stranica za otvoreno dijeljenje kôd-a. Prilikom biranja paketa važno je

AsYHQ/s1600/Waterfall-vs-Agile-2.jpg >
6 Prilagođeno na hrvatski prema izvoru < https://1.bp.blogspot.com/-NnXAeykbJ0I/XaE4aRmzZqI/AAAAAAAAT1s/9nXTY5O5lNIkaM5sDrd6m3J9KYWW7iKUACLcBG AsYHQ/s1600/Waterfall-vs-Agile-2.jpg >
7 GitHub <https://github.com/>

obratiti pozornost na to je li paket trenutno održavan, postoji li stabilna verzija paketa, te moramo procijeniti hoće li paket biti održavan u budućnosti.

U nastavku teksta kratko ću opisati kako inicijalizirati *Composer* projekt te dodati paket za korištenje u kod-u aplikacije. Pošto se radi o novom projektu na kojem nismo koristili *Composer* prije, prvi korak je pokretanje komande za inicijalizaciju projekta koju možemo vidjeti na slici 7.

```
$ composer init
```

Slika 7. *Komanda za inicijalizaciju Composer projekta*

Nakon pokretanja i upisivanja nekih osnovnih podataka[8], komanda će generirati novu *composer.json* datoteku, koja služi kao konfiguracijska datoteka za naš *Composer* projekt.

Kada je projekt inicijaliziran te je generirana *Composer* konfiguracijska datoteka možemo krenuti na dodavanje paketa. Postoje dva načina za dodavanje paketa, prvi je pokretanje *composer requiere* komande, kao što možemo vidjeti na slici 8.

```
$ composer require prozivođać/ime_paketa:~verzija
```

Slika 8. *Comsposer komanda za dodavanje paketa*

Po završetku izvršavanja komande, paket će biti dostupan za korištenje u našem kôd-u te će biti dodan u našu konfiguracijsku datoteku (*composer.json*). Ono što možemo uočiti nakon izvršavanja komande, jest da se uz konfiguracijsku datoteku generirala još jedna datoteka s imenom *composer.lock*. U toj datoteci zabilježene su točne verzije sa svim dopunama paketa koje se koriste u našem projektu. Možda se pitate zašto je to bitno, ako već imamo skoro pa istu definiciju u konfiguracijskoj datoteci. Bitno je zbog toga što u konfiguracijskoj datoteci ne moramo nužno definirati točnu verziju paketa, već možemo definirati da želimo, na primjer, najnoviju verziju paketa. U tom slučaju *Composer*, prilikom instalacije paketa, nalazi najnoviju

[8] Osnovni podatci - ime paketa odnosno projekta, opis projekta, ime i email autora, tip composer projekta i licenca pod kojom se paket distribuita

verziju, te ju u tom trenutku zapisuje u *lock* datoteku. Zapisivanje točne verzije paketa bitno je zbog postizanja konzistentnosti verzija paketa na svim serverima na kojima je instaliran naš projekt uz pomoć *Composer*-a. Bitno je još nadodati da se *lock* datoteka ne mijenja prilikom dodavanja novih paketa ili prve instalacije projekta uz pomoć *composer install* komande, već samo ako eksplicitno želimo ažurirati verzije paketa pokretanjem *composer update* komande.

Drugi način za dodavanje paketa u naš projekt, jest direktno uređivanje konfiguracijske datoteke, odnosno dodavanje paketa pod *requierd* ključem u composer.json datoteku. Ako odlučimo tako dodavati pakete, obavezno je pokretanje *composer update* ili *composer install komande*, ovisno o tome želimo li ažurirati stare pakete ili ne.

4. PHP standardi i komponente

U današnje vrijeme postoji nevjerojatan broj PHP programskih okvira. Obično ih dijelimo na makro i mikro programske okvire. Podjela se zasniva na veličini baze kôda (engl. *Code base*). Ako je baza kôda programskog okvira manja od 5000 linija onda takav programski okvir nazivamo **mikro** programskim okvirom, a ako je baza kôda veća, onda ga nazivamo **makro** programskim okvirom (Habib, 2015.). Kako bi osigurali interoperabilnost pojedinih programskih okvira, pogotovo mikro programskih okvira koji se često koriste i kao komponenta u većim projektima, potrebna je standardizacija kôda.

Upravo da bi se postigao cilj takvog standardiziranog PHP ekosistema, osnovana je PHP-FIG[9] (engl. *Framework Interop Group*) grupacija. Članovi grupacije su predstavnici svih velikih PHP programskih okvira, ali član može biti bilo tko, tko želi nešto pridodati PHP zajednici. Prema pravilima koja je postavila ova grupacija interoperabilnost komponenti temelji se na tri stupa koja ću opisati u nastavku teksta:

1. ***Sučelje (engl. Interface)***

   Sučelje možemo zamisliti kao ugovor između dva PHP objekta koji dozvoljava da jedan objekt može ovisiti o onome što objekt može napraviti, a ne o onome što on je, to jest kako je implementirana ta funkcionalnost. Ako ovu logiku prenesemo u kontekst paketa i programskih okvira, otvaramo velike mogućnosti za fleksibilnost. Uzmemo li za primjer bilježenje podataka (engl. *Logging*), s programskim okvirom možemo koristiti bilo koji paket gdje klasa koja bilježi podatke implementira metode poput *error(), warning(), info()* itd. Svaki programski okvir samo zanima implementira li *logger* klasa te metode, a isto garantiramo implementacijom sučelja.

9 PHP-FIG mrežna stranica <https://www.php-fig.org/>

2. *Automatsko učitavanje (engl. Autoloading)*

PHP programski okviri rade zajedno pomoću automatskog učitavanja. Automatsko učitavanje je proces pomoću kojeg PHP interpretator locira i dohvaća PHP klase tijekom pokretanja skripte. Prije nego što je uveden *autoloading* u PHP verziji 5, svaka se skripta morala ručno dohvatiti u kôdu pomoću *include* ili *requier* izraza. Prije uvođenja standardnog načina automatskog dohvaćanja, svaka PHP komponenta i programski okvir imao je svoju implementaciju pomoću magične metode *__autoload()* ili u kasnijim verzijama *spl_autoload_register().* Taj način implementacije zahtijevao je od programera da zna kako pojedina komponenta implementira automatsko učitavanje, no taj se problem riješio uvođenjem standarda koji se danas koristi u svim programskim okvirima i komponentama. To omogućuje korištenje raznih komponenata sa samo jednom metodom za automatsko učitavanje.

3. *Stil (engl. Style)*

Stil kôda odlučuje o tome gdje i kako će se koristiti razmaci, koja slova u nazivljima će biti velika, gdje i kako će biti postavljene zagrade, i tako dalje. Možda se ova značajka standardizacije kôd-a čini trivijalna, ali izrazito je bitna kada koristimo komponente iz različitih programskih okvira jer kôd automatski postaje čitljiviji. To znači da možemo više vremena iskoristiti za pronalaženje i rješavanje problema, a manje vremena trošiti na snalaženje i učenje nama nepoznatog stila. Svaki programer ima svoj stil i posebnosti koje koristi, što brzo može postati problem ako nekoliko programera surađuje na istom projektu. Standardni stil kôda pomaže svim programerima odmah razumjeti kôd bez obzira tko je njegov autor. Kôd stil kojeg danas koristi većina programera i programskih okvira je PSR standard. Ako ste ikad čitali o PHP-u sigurno ste vidjeli pojmove kao što su PSR-1, PSR-2, PSR-4 i tako dalje. PHP-FIG je do danas objavio šest preporuka vezanih uz kôd stil, a to su:

1. PSR-1: *Basic code style*
2. PSR-2: *Strict code style*
3. PSR-3: *Logger interface*

4. PSR-4: *Autoloading standard*
5. PSR-6: *Caching interface*
6. PSR-7: HTTP *Message interface*

Kao što možemo primijetiti po nazivlju preporuka, sve se odnose na tri već prije spomenute domene na kojima se temelje PHP-FIG preporuke. Postoji još i sedma preporuka vezana uz stil kôda, PSR-0, ali se ona više ne koristi i zamijenjena je sa PSR-4 preporukom (Lockhart, 2015).

5. Model-pogled-upravljač arhitektura

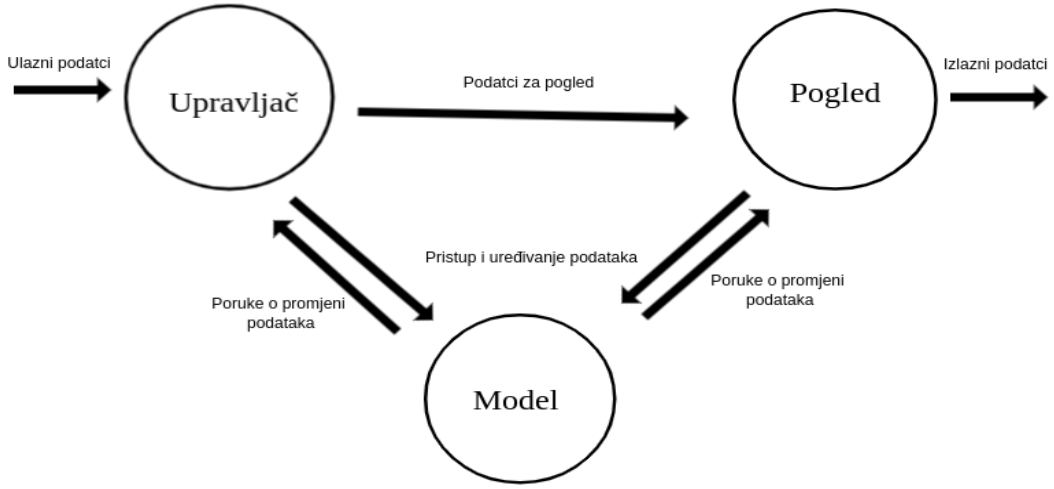
Model-pogled-upravljač arhitektura, skraćeno MVC, originalno je razvijena za desktop platforme, no danas je popularna i u web okruženju. Ideja ove arhitekture prvi puta se pojavljuje 1970. godine kada ju Trygve Reenskaug implementira u jezik *smalltalk*-79 (Vera, 2016).

Arhitektura se s vremenom razvijala i adaptirala za različite primjene, tako su nastale i različite arhitekture bazirane na MVC arhitekturi kao, na primjer, *hierarchical model–view–controller* (HMVC), *model–view–adapter* (MVA), *model–view–presenter* (MVP), *model–view–viewmodel* (MVVM) i tako dalje. U nastavku ću objasniti osnovne dijelove MVC arhitekture po kojima ona dobiva i ime.

- Model (engl. *Model*)
    - Model ili model podataka je klasa koja opisuje izgled tablice u bazi podataka gdje svako polje u klasi označava jedan stupac u tablici. Uz ta polja najčešće sadrži i funkcije za manipulaciju podataka koje model sadrži.
- Pogled (engl. *View*)
    - Pogled se bavi svim grafičkim dijelovima aplikacije, dohvaća podatke iz svojih modela te ih prikazuje. Ne sadrži samo komponente za prikaz podataka već može sadržavati i pod poglede ili biti dio nekog drugog pogleda.
- Upravljač (engl. *Controller*)
    - Upravljač dohvaća, kreira, manipulira i briše podatke iz modela na zahtjev korisnika. Nadalje, zadužen je za zakazivanje interakcija s drugim pogled-upravljač parovima.

Kako bi arhitektura funkcionirala moramo definirati način interakcije između pojedinih dijelova što je prikazano na slici 9 koja prikazuje shemu arhitekture iz 1988. po Krasneru i Popeu. U toj shemi pogledi i upravljači imaju točno jedan model

povezan s njima, ali model može imati nekoliko pogleda i upravljača povezanih s njim.



***Slika 9**. Prikaz komunikacije između komponenata MVC arhitekture*[10]

Za maksimalnu enkapsulaciju kôda, odnosno maksimalizaciju ponovne upotrebljivosti kôd-a, upravljač i pogled eksplicitno definiraju koji model koriste, za razliku od modela koji ne znaju kojem upravljaču ili pogledu služe. Prilikom promjene u modelu, koju najčešće inicijalizira upravljač, promjene će biti vidljive u svim pogledima koje koristi dati model, a ne samo u pogledu koji je povezan s upravljačem koji inicijalizira promjenu.

Ako pratimo taj proces možemo reći da standardni ciklus interakcije u MVC arhitekturi izgleda ovako: korisnik daje ulazne podatke upravljaču koji onda prosljeđuje te podatke modelu koji se mijenja prema datim podatcima. Nakon što su obavljene operacije u modelu, on obavještava sve svoje ovisnosti (poglede i upravljače) o svom novom stanju. Tako možemo garantirati da ovisnosti u svakom trenutku znaju stanje modela, te da se mogu prilagođavati datom stanju (Krasner, Pope, 1988).

10 Prilagođeno prema: Krasner, Pope, 1988.

6. Prednosti i dijelovi MVC PHP programskih okvira

U prijašnjem poglavlju prikazana je generalna shema MVC arhitekture. No, pošto se ovaj rad specifično bavi PHP programskim okvirima koji implementiraju tu arhitekturu, u nastavku teksta bit će prikazane prednosti koje oni pružaju prilikom razvoja mrežnih aplikacija te bitne komponente koje dolaze s makro programskim okvirima.

Osim što programski okvir nudi dobru osnovu za rješavanje klasičnih zadataka kao što su, na primjer, registracija i prijava korisnika, izrada formi, formatiranje podataka zahtjeva koji dolaze na server (Pitt, 2012) i tako dalje, postoji još jedan razlog zašto nije preporučljivo pisati 'čisti' PHP kôd. Taj razlog je sigurnost. Osim što programski okviri implementiraju obrambene mehanizme protiv klasičnih hakerskih napada kao što su, na primjer, umetanje SQL-a (engl. *SQL injection*), programski okviri, ako se pravilno koriste, pružaju i zaštitu protiv vrsta napada od kojih se manji timovi programera nikad ne bi mogli obraniti bez dodjeljivanja velikog dijela resursa tom zadatku. Pošto pružaju rješenje za takve napade, uvelike povećavaju sigurnost aplikacije bez da to troši vrijeme programera (Mindfire Solutions, 2018). Kako bi se prikazale još neke prednosti korištenja programskih okvira, u nastavku teksta bit će predstavljeni neki od ključnih paketa koji dolaze sa svakim modernim PHP MVC programskim okvirom.

6.1. Usmjerivač

Komponenta za usmjeravanje (engl. *Router*) služi kako bi se zahtjevi korisnika preusmjerili na predviđenu metodu unutar upravljačke klase. Usmjerivači modernih programskih okvira vrlo su kompleksni i težište stavljaju na sigurnost. Kako bi bolje shvatili što je zadatak usmjerivača, proći ćemo kroz vrlo jednostavan primjer takve klase. Treba imati na umu da je kôd na slici 10 vrlo nesiguran te da nije za korištenje

u bilo kojem produkcijskom okruženju, međutim vrlo je dobar za shvaćanje koncepta rada ove komponente.

```php
/**
 * Primjer najjednostavnijeg rutera
 */
$parametri=explode($ruta, string: '/');
//Ulazni parametri
$controller = $parametri[0];
$method = $parametri[1];
//standardiziranje imena,
// sva imena kontrollera imaju sufix Controller
//sva imena metoda(funkcija unutar klase) imaju sufix Action
$controllerName = ucfirst(strtolower($controller)) . 'Controller';
$method = strtolower($method).'Action';

//Ako ne postoji trazeni kontroller vracamo gresku
if (!file_exists( filename: '../src/controller/' . $controllerName . '.php')) {
    die('404');
}
require_once '../src/controller/' . $controllerName . '.php';

if (class_exists($controllerName)) {
    // inicijalizacija controller klase
    $controllerClass = new $controllerName;
    //provjera dali postoji metoda
    if ($controllerClass instanceof ControllerInterface && !method_exists($controllerClass, $method)) {
        die('404');
    }
    //pozivanje metode u controller klasi
    $controllerClass->$method();

}
```

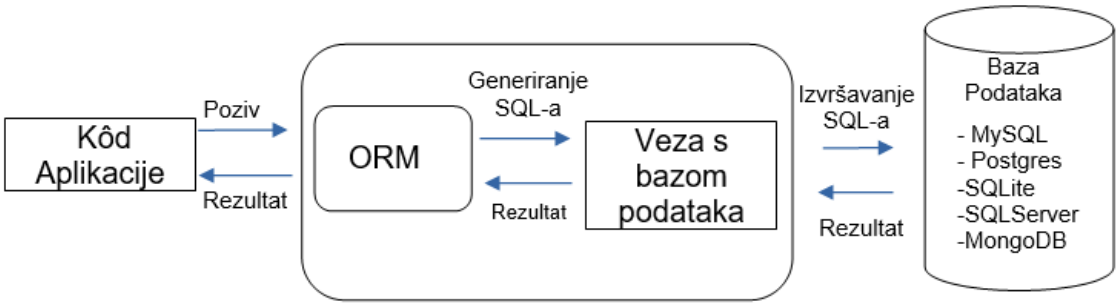
***Slika 10**. Primjer usmjerivačke klase*

Na primjeru sa slike 10 u varijabli „ruta“ dobivamo usklađeni lokator resursa (URL), odnosno ono što je korisnik upisao u svoj web preglednik. URL dijelimo po kosim crtama kako bi dobili parametre za nastavak izvršavanja kôd-a. Ovaj **usmjerivač** radi na premisi da je prvi dio URL-a ime upravljačke klase, a drugi dio URL-a ime metode u upravljačkoj klasi. Sve upravljačke klase imaju sufiks 'Controller' kao što i sve metode u upravljačkim klasama imaju sufiks 'Action'. Nakon provjere postoji li klasa, odnosno tražena metoda u toj klasi, izvršava se pozvana metoda. Ovo je osnovni zadatak usmjerivača, izvršavanje kôd-a u upravljačima prema parametrima koje je korisnik unio u svoj web preglednik. Međutim, kao što je bilo napomenuto na početku

ovog poglavlja, usmjerivači programskih okvira puno su sofisticiraniji i nude još mnoge druge mogućnosti poput identifikacije korisnika.

6.2.Komponenta za upravljanje bazom podataka (ORM)

Pošto se programer u različitim okruženjima može susresti s različitim načinima manipuliranja podataka, odnosno različitim vrstama baza podataka (mySql, Postgres, Mongo), ORM pruža dodatni sloj apstrakcije (O'Neil, 2008). Ta apstrakcija omogućava da se programer prema svim bazama podataka odnosi na isti način. Umjesto da programer piše direktne upite za podatke, primjerice u SQL jeziku, to se kroz ORM obavlja manipulirajući objektima. ORM u tom slučaju preuzima ulogu "prevoditelja", te upite pretvara u odgovarajući jezik, koji trenutna baza podataka podržava, u našem primjeru SQL.



Slika 11. *Grafički prikaz rada ORM komponente*[11]

Na primjeru sa slike 11 vidimo način rada ORM-a. U ovakvom slučaju u kôd-u pišemo apstraktan upit uz pomoć programskog okvira, te programski okvir, odnosno ORM taj upit "prevodi" i prosljeđuje bazi podataka. Nakon izvršavanja SQL kod-a ORM vrača rezultate u naš aplikacijski kôd gdje možemo nastaviti s njihovom manipulacijom. Ako bi ručno izvršili upit, za rezultat bi dobili asocijativni skup, no pošto koristimo ORM odmah ćemo dobiti podatke pospremljene u model podataka koji imamo definiran za *Flight* tablicu.

11 Slika prilagođena prema izvoru <https://ai2-s2-public.s3.amazonaws.com/figures/2017-08-08/1448dae25733d1ebdd5c03c25b9a5230f99acb69/11-Figure2:2-1.png>

Uz ovakav model ne moramo brinuti oko SQL kôd-a za interakciju s bazom, te se ne moramo brinuti oko promjena vrste i verzije baze podataka.

Na slikama 12 i 13 vidimo isti upit, prvo izvršen uz pomoć ORM sučelja, a zatim isti taj upit preveden u SQL kôd koji se izvršava u bazi podataka.

```
$flight = App\Flight::where('number', 'FR 900')->first();
```

Slika 12. *Primjer upita na bazu uz pomoć Eloquent ORM sučelja*

```
SELECT * FROM flight WHERE number="FR 900" LIMIT 1
```

Slika 13. *Primjer upita na mySql bazu podataka uz pomoć SQL kod-a*

6.3. Ubrizgavanje ovisnosti

Ubrizgavanje ovisnosti (engl. *Dependency Injection*), ili skraćeno DI, je koncept u razvoju programa kojem je cilj uklanjanje čvrstih zavisnosti između klasa (Dhanji, 2009). Najjednostavnije ćemo ga objasniti na primjeru. Ako za primjer uzmemo klasu koja komunicira s bazom podataka, *Database*, njoj je za komunikaciju s bazom podataka potrebna adapter klasa. Adapter klasa je zavisnost naše *Database* klase. Na primjeru sa slike 14 prikazat ću kako bi ta klasa s čvrstom zavisnošću izgledala u kôdu.

```php
<?php

namespace Database;

class Database
{

    /** @var DatabaseAdapter  */
    protected $adapter;

    /**
     * Database constructor.
     */
    public function __construct()
    {
        $this->adapter = new MySqlAdapter();
    }


}
```

Slika 14. *Primjer klase s čvrstom zavisnošću*

Ovo nazivamo čvrstom zavisnošću, jer se pri svakoj inicijalizaciji klase *Database* kreira nova instanca klase *MySqlAdapter* i uvijek će se koristiti *MySqlAdapter*, iako klasa može koristiti bilo koju klasu koja implementira *DatabaseAdapter* sučelje. Ovu klasu možemo lako konfigurirati tako da koristi DI i s time razbijemo čvrstu zavisnost između klasa. Kako ta promjena izgleda u PHP kôdu, možemo vidjeti na primjeru sa slike 15.

```php
<?php

namespace Database;

class Database
{

    /** @var DatabaseAdapter  */
    protected $adapter;

    /**
     * Database constructor.
     * @param DatabaseAdapter $adapter
     */
    public function __construct(DatabaseAdapter $adapter)
    {
        $this->adapter = $adapter;
    }


}
```

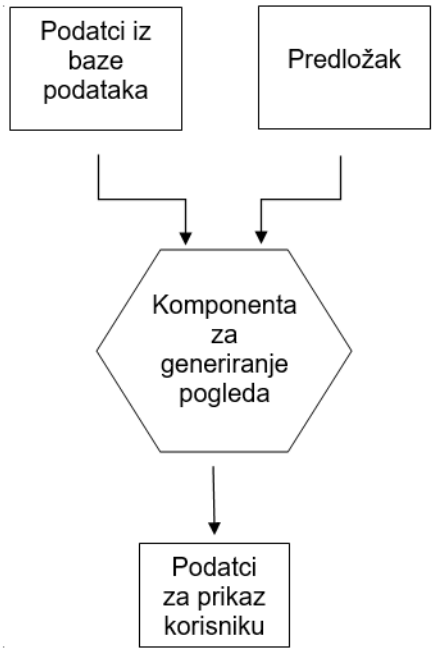
***Slika 15**. Primjer klase bez čvrste zavisnosti*

Sada umjesto da svaki put inicijaliziramo klasu *MySqlAdapter* u konstruktoru klase *Database*, ubrizgavamo instancu klase koja implementira *DatabaseAdapter* sučelje. Kako bi bilo lakše ubrizgavali servise, programski okviri nude takozvane *Dependency Injection* kontejnere (engl. *Container*). Kontejner pruža mogućnost automatskog pronalaska i inicijalizacije servisa koji su potrebni za inicijalizaciju neke klase (po definicijama u konstruktoru), takozvani *autowire* (Dhanji, 2009). Ovo uvelike olakšava pisanje razdvojenih komponenti, testiranje, te prelazak iz testnih u produkcijska okruženja.

6.4.Komponenta za generiranje pogleda

Komponenta za generiranje pogleda (engl. *Templating engine*), je paket koji iz predloška koji je pisan posebnom sintaksom stvara HTML kôd, koji se prikazuje korisniku. Takav pristup kreiranja HTML kôda nam omogućuje da jednom napišemo predložak i koristimo ga za prikaz bezbroj različitih podataka, odnosno da dobijemo

dinamički HTML kôd (Gheorghe L., Hayder H., Maia P.J., 2006). Vrijednost pojedinih varijabli postavljamo kroz PHP kôd. Te vrijednosti najčešće dolaze iz baze podataka, a taj proces možemo vidjeti grafički prikazan na slici 16.



***Slika 16**. Shema funkcioniranja komponente za generiranje pogleda*[12]

Postoji puno paketa koji pružaju takvu funkcionalnost i skoro svaki PHP programski okvir dolazi s jednim od njih. Na primjer *Symfony* s *Twig*-om, *Laravel* s *Blade*-om i tako dalje. Kada pričamo o takvim paketima mnogi programeri će vam reći da nema potrebe za njima, odnosno da i sam PHP može pružiti takvu funkcionalnost. No, PHP je opširan i predlošci brzo postaju nečitljivi kada ga koristimo za njihovo kreiranje. Uzmimo na primjer ispisivanje vrijednosti varijable i „steriliziranje"[13] izlaznog teksta.

12 Slika prilagođena prema izvoru <https://upload.wikimedia.org/wikipedia/en/thumb/a/a2/TempEngWeb016.svg/220px-TempEngWeb016.svg.png>

13 „steriliziranje" – uklanjanje kôd-a koji je unesen od strane korisnika i nije dozvoljen u tekstu

- PHP

```
<?php echo $var ?>
<?php echo htmlspecialchars($var, flags: ENT_QUOTES, encoding: 'UTF-8') ?>
```

***Slika 17**. Primjer sintakse predloška u PHP-u*

- *Twig* paket za predloške

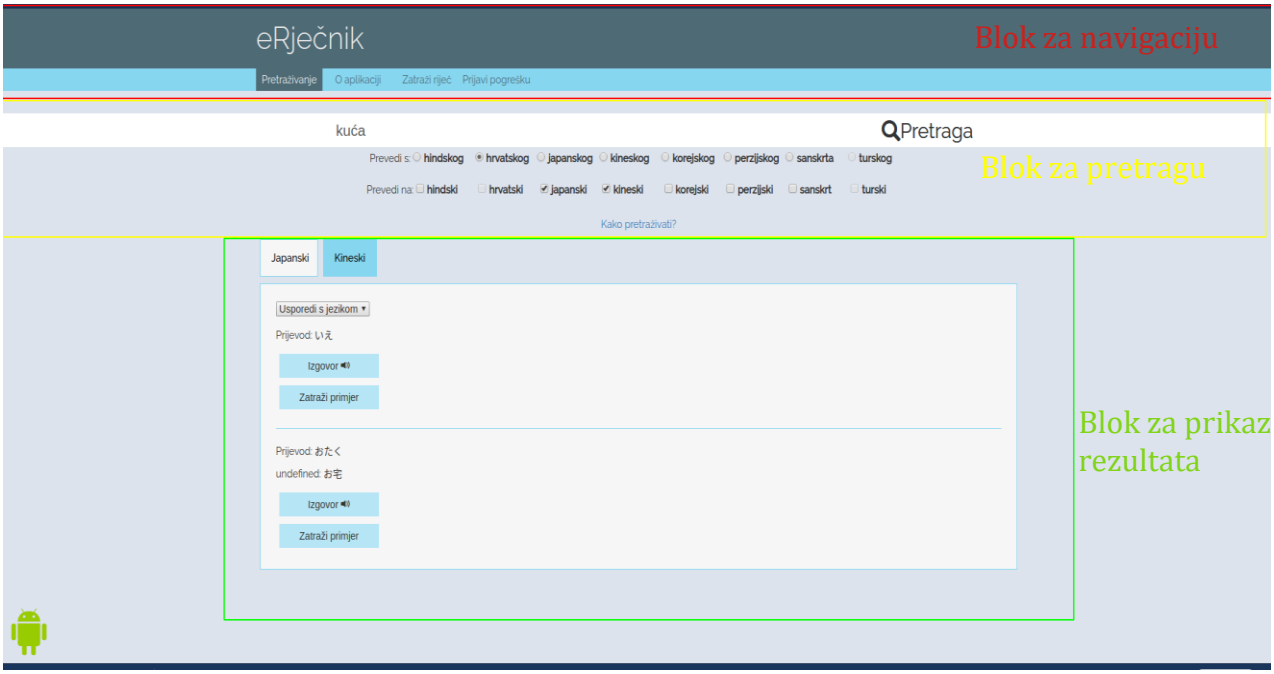
```
{{ var }}
{{ var|escape }}
{{ var|e }}
```

***Slika 18.** Primjer sintakse predloška u twig paketu za predloške*

Na prvi pogled, kada razmotrimo primjere kôd-a na slikama 17 i 18, možemo uočiti da je sintaksa u *twig*-u puno kraća i jasnija, pogotovo ako zamislimo da je ovo dio veće datoteke u kojoj su ti dijelovi pomiješani s HTML i JavaScript kôd-om.

Još jedna od bitnih prednosti je to da se predlošci mogu razdvajati u različite datoteke, odnosno takozvane blokove. Blokovi su iznimno korisni ako se neki element pojavljuje na nekoliko stranica kao, na primjer, navigacija.

Ako ne koristimo blokove, odnosno nemamo modularni kôd, morali bi ga duplicirati na svakoj od stranica. S tako dupliciranim kôd-om promjene postaju teške i mukotrpne. Na slici 19 možemo vidjeti kako vizualno izgleda implementacija takvog pristupa na primjeru stranice eRječnik.



***Slika 19.**. Vizualni prikaz blokova na primjeru eRječnik stranice za pretragu*

7. Projekt eRječnik

Projekt eRječnik nastao je kao dio projekta MemAzija, koji istražuje prednosti korištenja tehnologije za lakše učenje azijskih jezika. Razlog zašto se krenulo na realizaciju ovog projekta je taj što do tada nije postojao niti jedan online rječnik azijskih jezika fokusiran na hrvatske studente, odnosno s direktnim prijevodima s hrvatskog jezika (Janjić et al., 2019). Prvi korak prije početka svakog projekta je definiranje osnovnih ciljeva i funkcionalnosti projekta. Na projektu je korišten modularni pristup i AGILE metodologija razvoja, a dokumentacija nije bila opširna što je i samu fazu planiranja svelo na minimum.

Nakon što je definirana dokumentacija projekta, krenuli smo na dizajniranje baze podataka i protoka podataka (engl. *data flow*) s MVC arhitekturom kôda na umu. S obzirom na to da je fokus rječnika na studente iz Hrvatske, odlučeno je da će se kao meta jezik koristiti hrvatski. Kako je jedna od značajki sustava definirana dokumentacijom, mogućnost pretraživanja na nekoliko jezika te uspoređivanje rezultata, kreirane su pivot tablice s hrvatskim riječima kao meta riječi, što je omogućilo laku korelaciju riječi različitih jezika. Takva implementacija omogućuje i puno lakše dodavanje novih riječi, ali i potpuno novih jezika.

Cilj faze programiranja je bio što brže objaviti radnu verziju rječnika kako bi se u što ranijoj fazi dobila povratna informacija od budućih korisnika platforme, model koji se podudara i s AGILE filozofijom razvoja kôda. Zbog toga je odlučeno da se programski kôd razvija uz pomoć PHP MVC programskog okvira Laravel. Modularni pristup omogućio je razdvajanje programskog kôda u logične jedinice, što je uvelike ubrzalo proces razvoja cijele platforme. Kôd je podijeljen u 3 velike cjeline, tako da je svaki od programera mogao raditi zasebno na svom dijelu. Te komponente su:

1. Komponenta za uvoz podataka

   Podaci su pripremljeni od strane MemAzija tima u Excel formatu koji se zatim pretvara u CSV datoteke, koje se pomoću ove komponente uvoze u

bazu podataka. Uz ovo, komponenta omogućava i uvoz novih riječi kroz administratorsko sučelje.

2. Komponenta za pretraživanje

   Ova komponenta, kao što samo ime sugerira, bavi se pretraživanjem baze podataka i pronalazi prijevode unesene riječi. Ako se ne pronađe točno tražena riječ prikazuje se lista predloženih riječi koja se generira na osnovi Levenstinove udaljenosti[14], koja ocjenjuje sličnost riječi.

3. Komponenta za prikaz

   Komponenta za prikaz, odnosno *frontend*, je skup predložaka koji koriste B*lade*[15] paket za generiranje pogleda, koji dolazi ugrađen s Laravel programskim okvirom. Uz predloške i sve pogodnosti koje oni donose, koristimo i AJAX tehnologiju kako bi radili asinkrone pozive na server, tako da stranica bude što fluidnija te da ne zahtijeva ponovno učitavanje prilikom svake pretrage. Za samu strukturu i stil web stranica koristimo HTML i CSS (*Cascading Style Sheets*) odnosno SASS (*Syntactically Awesome Stylesheet*) kompajliranu verziju CSS-a.

Nakon što su bile završene osnovne funkcionalnosti svake komponente, u opticaj je puštena prva testna verzija rječnika. Nakon još nekoliko iteracija AGILE procesa, projekt je završen te objavljen na domeni http://erjecnik.ffzg.hr/

[14] Levinstainova udaljenost je algoritam za određivanje sličnosti između dva niza znakova. Rezultat je broj akcija potrebnih da se iz jednog znakovnog niza dobije drugi, pri čemu je dozvoljeno dodavanje, brisanje ili izmjena znakova na bilo kojoj lokaciji unutar niza.
[15] Dokumentacija Blade paketa za generiranje pogleda: https://laravel.com/docs/5.8/blade

8. Zaključak

Iako mnogi kažu kako je PHP zastarjeli jezik i da polako opada u popularnosti, mislim da ipak možemo zaključiti da se PHP dobro prilagodio zahtjevima modernih web aplikacija. Implementacijom objektno orijentirane paradigme, MVC arhitekture i standardizacije kôd-a, otvorile su se mnoge mogućnosti i poboljšanja u razvoju web aplikacija uz pomoć PHP-a. Modularni pristup razvoja te mentalitet otvorenih izvora koji stoje iza jezika i njegovih programskih okvira, svakodnevno donose brojna rješenja za postojeće i nove probleme s kojima se svakodnevno susrećemo prilikom razvoja web aplikacija.

Ni sam Rasmus Lerdorf nije predvidio da će iz malog projekta kojeg je osmislio za vlastite potrebe nastati projekt ovolikih razmjera, što je potvrdio i sam kada sam ga upoznao na PHP konferenciji u Veroni prošle godine. O proporcijama koje je poprimio svjedoči njegova i dalje jako velika zastupljenost na web-u, ogromna zajednica koja se stvorila iza njega, kao i iza mnogih popularnih PHP MVC programskih okvira. Moje mišljenje je da upravo otvoreni i modularni duh koji je prisutan u PHP ekosistemu najviše pridonosi njegovoj popularnosti. Nadam se da sam ovim radom dao polaznu točku čitateljima da istraže više o PHP-u i mogućnostima koje pružaju brojni programski okviri.

9. Literatura

1. Achour M., Betz F., Dovgal A., Lopes N., Magnusson H., Richter G., Seguy D., Vrana J. (9.1.2020a.) PHP: History of PHP – Manual, dostupno na: https://www.php.net/manual/en/history.php.php#history.phpfi, pristupljeno: 17.1.2020.
2. Achour M., Betz F., Dovgal A., Lopes N., Magnusson H., Richter G., Seguy D., Vrana J. (9.1.2020b.) PHP: PHP/FI Version 2.0 Documentation, dostupno na: https://www.php.net/manual/phpfi2.php, pristupljeno: 17.1.2020.
3. Achour M., Betz F., Dovgal A., Lopes N., Magnusson H., Richter G., Seguy D., Vrana J. (9.1.2020c.) PHP Data Objects, dostupno na: https://www.php.net/manual/en/book.pdo.php, pristupljeno: 18.1.2020.
4. Brown, P. (7. 1. 2013.), What is PHP Composer?, dostupno na: https://culttt.com/2013/01/07/what-is-php-composer/, pristupljeno: 1.2.2020
5. Denning, S. (25.8.2019) Why the future of agile is bright, dostupno na: https://www.forbes.com/sites/stevedenning/2019/08/25/why-the-future-of-agile-is-bright/, pristupljeno: 1.2.2020.
6. Dhanji, P.R., (2009.) Dependency Injection, Greenwich: MANNING
7. Dingsøyrab T., Nerurc S., Balijepallyd V.G., Moe N.B. (2012) A decade of agile methodologies: Towards explaining agile software development, Journal of Systems and Software Volume 85, Issue 6, June 2012, Pages 1213-1221
8. Domogalla, M. (23.1.2020) PHP 8 interview: „JIT will bring the language to a whole new level“, dostupno na: https://jaxenter.com/php-8-interview-da-silva-167335.html, pristupljeno: 1.2.2020.
9. Gheorghe L., Hayder H., Maia P.J, (2006) Smarty: PHP Template Programming and Applications, Birmingham, UK: Packt Publishing
10. Habib, O. (7.11.2015.) PHP Microframework vs. Full Stack Framework dostupno na: https://www.appdynamics.com/blog/engineering/php-microframework-vs-full-stack-framework/, pristupljeno: 1.2.2020

11. Janjić M., Poljak D., Kocijan K. (2019) eDictionary: the Good, the Bad and the Ugly, Zagreb (Croatia): Department of Information and Communication Sciences, Faculty of Humanities and Social Sciences, University of Zagreb
12. Kamaruzzaman, M. (4.2.2020), Top 10 In-Demand programming languages to learn in 2020, dostupno na: https://towardsdatascience.com/top-10-in-demand-programming-languages-to-learn-in-2020-4462eb7d8d3e, pristupljeno: 9.2.2020.
13. Krasner, G.E., Pope, S.T. (1988). A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System, Mountain View, CA: ParcPlace Systems, Inc.
14. Lockhart, J. (2015.), Modern PHP: new features and good pracices, Sebastopol, CA: O'reilly media Inc.
15. Matković S., Đurišić M., Zoranović D. (2006) Osnove programiranja u okruženju grafičkih operativnih sistema: Programski jezik C#, Beograd: RG i CET
16. Mindfire Solutions (30.5.2018). Advantages and Disadvantages of PHP Frameworks, dostupno na: https://medium.com/@mindfiresolutions.usa/advantages-and-disadvantages-of-php-frameworks-c046d50754e5, pristuplejno: 16.1.2020.
17. O'Neil, E.J. (2008). Object/relational mapping 2008: hibernate and the entity data model (edm) <https://dl.acm.org/doi/abs/10.1145/1376616.1376773
18. PHP Usage Statistics (n.d) dostupno na: https://trends.builtwith.com/framework/PHP, pristupljeno: 16.1.2020
19. Pitt, C. (2012). Pro PHP MVC, New York, NY: Apress Media LLC
20. Purkis, M. (13.11.2018) PHP 5 is Going EOL, It is Time to Migrate to PHP 7 and Here is How to Do It, dostupno na: https://www.liquidweb.com/blog/php-5-going-eol/, pristupljeno: 19.1.2020.
21. Sinha, S. (2017) PHP7: object oriented study of design patterns, Victoria, Canada: Leanpub
22. Šribar J., Motik B. (2010) Demistificirani C++, Zagreb: Element

23. The State of the Octoverse (6.11.2019) dostupno na: https://octoverse.github.com/, pristupljeno: 18.1.2020.
24. Usage statistics of PHP for websites(16.1.2020), dostupno na: https://w3techs.com/technologies/details/pl-php, pristupljeno : 16.1.2020
25. Vera, D. (21.12.2016). Software Architecture: MVC Design Pattern dostupno na: https://medium.com/@dennisvera.z/software-architecture-mvc-designpattern-ceae5d5083d7, pristupljeno: 26.1.2020
26. Zend Engine version 2.0 Feature Overview and Design (9.6.2003.) dostupno na: https://web.archive.org/web/20030609072908/http://www.zend.com/engine2/ZendEngine-2.0.pdf, pristupljeno: 16.1.2020.

MVC arhitektura u izradi mrežnih aplikacija

Sažetak

U ovom radu detaljno ću razraditi korisnost i osnovne koncepte MVC (engl. *Model View Controller*) softverske arhitekture za izradu korisničkih sučelja mrežnih (engl. *web*) aplikacija. U radu ću se koncentrirati na PHP programski jezik te Laravel razvojni okvir kao jednu od modernijih instanci ove arhitekture. Nadalje ću opisati razvoj objektno orijentirane paradigme u programskim jezicima te važnost tog koncepta u dizajnu modernih programskih rješenja otvorenog izvora (engl. *open source*) za web. Implementaciju takve arhitekture programskog rješenja, pokazat ću na primjeru projekta eRijecnik kojeg smo izradili za potrebe studenata Filozofskog fakulteta u Zagrebu na odsjeku za Informacijske i komunikacijske znanosti.

Ključne riječi: *PHP, MVC, Modularno programiranje, AGILE, PHP-FIG, PSR*

Model View Controller architecture in web development

Summary

In this paper, I will discuss the basic concepts of Model-View-Controller code architecture in web development, with particular emphasis on PHP language and framework implementations of the architecture. I will also discuss the importance of object-oriented syntax in PHP and its importance for developing open source software for the modern environment. In addition, I will show how all the concepts mentioned above come together by sharing the development story of the eDictonary project, which developed for students Faculty of Humanities and Social Sciences

Key words: *PHP, MVC, Modular programing, AGILE, PHP-FIG, PSR*