Adrian Smijulj [1]
Ana Meštrović [2]

Stručni rad
UDK 004.415.2

IZGRADNJA MVC MODULARNOG RADNOG OKVIRA[3]

SAŽETAK

Cilj je ovoga stručnoga rada prikazati tehnologije i komponente vezane uz izradu radnog okvira te prikazati implementaciju jednostavnog radnog okvira koji sadrži najnužnije komponente za brzi razvoj web-*aplikacija. Iznesena su teorijska razmatranja vezana uz primjenu radnog okvira u razvoju aplikacije. Predstavljene su i analizirane potencijalne komponente radnog okvira. Nadalje, prikazana je implementacija radnog okvira uz korištenje suvremenih tehnologija kao što su PHP, MySQL, HTML itd. Prikazani radni okvir cjelokupno je izgrađen primjenom objektno-orijentiranog pristupa. Korištena je Model-View-Controller arhitektura i alati poput ORM sučelja za automatizirani rad s entitetima i sustavom za upravljanje događajima. Prednost korištenja izrađenog radnog okvira je prestanak ulaganja programerova vremena na osmišljavanje arhitekture i izradu alata, već će se u potpunosti moći posvetiti zadacima projekta. Iako sadrži samo temeljne komponente, radni okvir će po svojoj modularnoj arhitekturi biti fleksibilan, što će omogućiti programeru da ga proširi prema vlastitim potrebama i prilagodi prema zahtjevima aplikacije koju je potrebno izraditi. Zaključak je rada kako je u razvoju složenih aplikacija poželjno uložiti vrijeme u izradu radnog okvira ili neke slične komponente koja će kasnije omogućiti brži i pouzdaniji razvoj cjelokupne aplikacije.*

Ključne riječi: *radni okvir, Model-View-Controller arhitektura (MVC), programiranje pogonjeno događajima, modularni sustavi , PHP*

1. UVOD

Programiranje i razvoj aplikacija postaju sve zahtjevniji zadaci. Razvojem novih tehnologija, mogućnosti i nastajanjem sve kompleksnijih problema, potrebno je uložiti više vremena za izradu projekta. Programer ponekad mora razvijati svoja rješenja na dvije, pa čak i više programskih platformi. Jedan od mogućih pristupa koji može olakšati i ubrzati razvoj aplikacija je korištenje radnih okvira (engl. frameworks). To su alati koji odrađuju poslove koji mogu biti vrlo često repetitivni, trivijalni, ponekad kritični i općenito poslovi kojima se programeri ne bi trebali baviti prilikom rada na projektu jer uzimaju dodatno vrijeme za razvoj. Programeri na projektu moraju razvijati zadani projekt, a ne probleme arhitekture kôda, nedostatka funkcionalnosti ili sigurnosti[4].

[1] Mag. edu. *E-mail:* adrian.1358@gmail.com
[2] Dr. sc., docent, Odjel za informatiku, Sveučilište u Rijeci, Radmile Matejčić 2, Rijeka, Hrvatska. *E-mail:* amestrovic@inf.uniri.hr
[3] Datum primitka rada: 27. 2. 2014.; datum prihvaćanja rada: 5. 5. 2014.
[4] SensioLabs, (2013)., The Book for Symfony 2.3, preuzeto 10. 8. 2013.

U ovome radu fokus je postavljen na izradu radnog okvira i to u jednom od najpoznatijih skriptnih jezika u domeni web-tehnologija, a to je PHP. s ciljem povećanja učinkovitost programera i samog kôda, integrirano je nekoliko različitih komponenti, odnosno modula, a svaki je odgovoran za odrađivanje jednog segmenta aplikacije koje će programeri-korisnici koristiti po potrebi. Kreiranje ovog radnog okvira nastalo je temeljem različitih iskustava s drugim popularnim radnim okvirima, a istovremeno sa željom da se iz svakoga uzmu dobre ideje i pretvore u jedan novi alat. Korisno je napomenuti kako radni okvir nije produkt koji se kreira jednokratno, stoga je logičan nastavak ovoga rada usavršavanje, održavanje i dodavanje novih mogućnosti, kako bi bio u skladu s novim tehnologijama i nadolazećim potrebama programera.

U radu je posebno analizirana Model-View-Controller (MVC) arhitektura. MVC arhitektura sedamdesetih je godina inicijalno uvedena u jezik Smalltalk-76, a potom implementirana unutar biblioteka verzije Smallatlk-80 (Krasner, Pope, 1988). Nakon toga je MVC arhitektura prihvaćena kao općeniti koncept.

U nastavku rada opisana je ideja radnog okvira i MVC arhitekture te koncepti bitni za njegovu realizaciju: modularno programiranje, Object-Relational Mapping (ORM) koncept, te programiranje pogonjeno događajima. U trećem poglavlju opisan je postupak implementacije modula za kreiranje korisnika i korisničkih grupa. U četvrtom poglavlju iznesena su zaključna razmatranja.

2. IZRADA RADNOG OKVIRA

U ovome poglavlju bit će prikazana izrada radnog okvira, i to tako da će prvo biti teorijski obrađene komponente, a nakon toga će biti prikazana implementacija. Bit će obrađeno nekoliko poznatijih i popularnijih termina iz područja programiranja, i to počevši od osnovne MVC arhitekture koja će biti korištena. Osim toga, bit će razmotrene i karakteristike modularnog programiranja, ORM sučelja i programiranja pogonjenog događajima, kao sloj koji će dodatno podržavati radni okvir.

Radni okvir je alat koji se sastoji od mnoštva različitih komponenti koje programeru mogu olakšati posao i ubrzati proces izrade aplikacije. Okviri su kolekcije različitih pomoćnih funkcija, klasa, knjižnica i dr. koje pomažu nudeći već gotova rješenja za specifične i često korištene probleme. Programeri se mogu odlučiti za korištenje već popularnih gotovih rješenja, no ponekad mogu krenuti i u izradu vlastitog. Za ovaj korak prethodno treba utvrditi potrebno vrijeme izrade i predvidjeti probleme koji se mogu pojaviti. Vlastita izrada može biti i štetna, jer često ne postoji dovoljno znanja i iskustva kako bi se sve prepreke u potpunosti riješile. Ipak, često je onima koji rade s vlastitim radnim okvirom kasnije lakše riješiti određene aplikacijske probleme jer poznaju arhitekturu koju su sami postavili pa lakše mogu ponovno iskoristiti isti kôd na drugim projektima (Fowler, 2002).

Neki od primjera rješenja koja nude različiti radni okviri su sustav rada s korisnicima, MVC arhitektura, klase za rad s kolačićima (engl. cookies) i sesijama (engl. sessions), formulari, ORM sučelje, klase za rad s rutama, datotekama itd. Korištenjem radnog okvira, sve navedene

probleme programer preskače, čime se može odmah baviti stvarnim problemom, odnosno poslovnom logikom. Osim navedenog, postoji još niz prednosti:

- sigurnost,
- nema cijene *(open source)*,
- podrška od strane proizvođača i zajednice.

Postoje i mane, npr.:

- ponekad programer pasivno uči rad u radnom okviru, a ne u jeziku,
- ograničenja od strane radnog okvira,
- kôd radnog okvira je otvorenog tipa, što znači da napredniji programeri mogu znati kako aplikacija interno funkcionira i mogu iskoristiti slabosti.

Trenutno postoji mnogo dostupnih gotovih radnih okvira, a neki od poznatijih su zasigurno Symfony, CakePHP, CodeIgniter, ZendFramework, Laravel, Yii (McArthur, 2008). Specifičnost rada s navedenim radnim okvirima jest njihov neidentičan rad. Stoga, promjenom radnog okvira s jednog na drugi, programer mora detaljno proučiti strukturu, konvencije i njegove specifičnosti.

Ukupno gledajući, postoje dobri razlozi za korištenje gotovog radnog okvira. No, izrađujući vlastiti, moguće je u potpunosti prilagoditi kompletnu arhitekturu vlastitim potrebama i željama, odnosno dodavati samo ono potrebno za određeni projekt, te kasnije napraviti izmjene ili nadogradnje sustava. S druge strane, tijekom izrade ovakvih sustava nije potrebno apsolutni svaki dio samostalno izraditi jer postoje već dobra modularna rješenja, pa će tako za sustav upravljanja predlošcima u radnom okviru biti integriran popularan Smarty Templating Engine (Hayder, & Gheorghe, 2006), čime će se uštedjeti vrijeme potrebno za izradu radnog okvira.

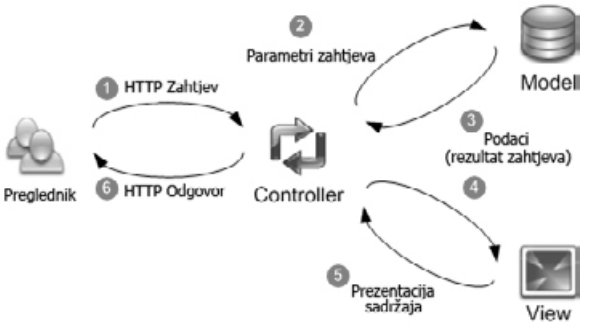
2.1 Model–View–Controller arhitektura

MVC je kôdna arhitektura koja je, iako originalno razvijana za *desktop* platforme, danas popularna na suvremenim *web*-platformama. Glavne ideje ovoga modela su razdvajanje kôda u zasebne cjeline, odnosno dosljedna struktura i mogućnost ponovnog korištenja istog kôda (engl. *code reusability*). Sama ideja je prvi put javno prezentirana već 1988. godine (Krasner, Pope, 1988), a do današnjeg dana doživjela je nekoliko iteracija pa je stvoreno nekoliko inačica originalne ideje (HMVC, MVA, MVP, MVVM, itd.).

MVC arhitektura sastoji se od određenih komponenti u kojima je svaka zadužena za obavljanje specifičnih funkcija. To je puno bolje u odnosu na prethodne inačice u kojima se sve nalazilo na jednom mjestu, a što je pak dovodilo do otežanog čitanja kôda, otežanog pronalaženja i ispravljanja pogrešaka, ali i do znatno manje funkcionalnosti i fleksibilnosti.

Cijela arhitektura se sastoji od triju specifičnih i međusobno zavisnih komponenti: *model*, *view* (pogled, prezentacija) i *controller* (kontroler, upravljač). Na slici 1 prikazan je općenit princip rada MVC arhitekture u *web*-okruženju:

Slika 1. Općenit princip rada MVC arhitekture u *web*-okruženju



Izvor: (Hayder, Gheorghe, 2006)

Cijeli proces započinje korisnikovim zahtjevom (u trenutku kada korisnik unese *web*-adresu u svom pregledniku) kojeg najprije analizira kontroler, a koji sadržava skup prethodno definiranih slijedova operacija, odnosno funkcija koje je potrebno izvršiti. U tom procesu vrši se i komunikacija s preostale dvije komponente – modelom i prezentacijom, tako da potrebne parametre prosljeđuje svakoj komponenti te, ovisno o rezultatu, kontrolira daljnji slijed izvršavanja cjelokupnog procesa. Uloga modela je rad s podacima, odnosno komunikacija s bazom podataka, dodatna obrada dobivenih podataka, ukoliko je potrebno, te u konačnici vraćanje rezultata prema kontroleru. Dobiveni podaci će, po potrebi, biti poslani drugim procesima i biti opet naknadno obrađivani. Ipak, u većini slučajeva kontroler će završiti proces pozivajući prezentacijsku komponentu, koja će pak generirati zadani predložak po kojemu će se dobiveni podaci i prezentirati. Tako dobiveni predložak, odnosno generirani HTML kôd, kontroler će poslati prema korisniku, odnosno korisnikovom pregledniku.

MVC arhitektura se neko vrijeme razvija unapređivanjem prethodnih verzija pa je razvijeno više različitih verzija MVC arhitekture. Arhitekture su uglavnom slične, s manjim razlikama u logici odvijanja procesa i nazivlju komponenata. Jedna verzija MVC arhitekture je MVA, odnosno *Model-View-Adapter* i zanimljivo je da se danas sve više koristi u kontekstu *web*-tehnologija, naročito na klijentskoj strani aplikacije (npr. u skriptnom jeziku JavaScript). U ovom konceptu postoji posebna komponenta adapter koja se nalazi između komponenti model i prezentacija i ona regulira procese tako što prati rad i procese temeljem odgovora koje dobiva od okolnih susjednih komponenti. Npr. prezentacijska komponenta šalje „signal" da je u tekstualno polje u formularu upisano ime, adapter prima tu informaciju i tada šalje zahtjev modelu za ažuriranjem podataka koji će se kasnije koristiti za spremanje, npr. u bazu podataka. Obrnuti proces je također moguć, ukoliko se podaci u modelu promijene, model komponenta će poslati signal adapteru koji će na isti način reagirati prema prezentacijskooj komponenti, odnosno sadržaj formulara će se također ažurirati. Danas se primjena ovakvog način rada može vidjeti npr. u JavaScript radnim okvirima kao što su Backbone.js, Angular.js, Kendo UI (Adams, 2013) itd., a koji omogućuju izvršavanje MVC arhitekture djelomično na klijentskoj strani, za razliku od klasičnog pristupa u kojem se arhitektura primjenjivala samo na poslužitelju.

Sličan princip koristi i *three-tier-pattern* arhitekura (Eckerson, 1995), u kojoj se kao i u MVC arhitekturi, cjelina nastoji razdvojiti u tri razine, a to su prezentacijska, logička i podatkovna razina. Zanimljivo je da se ove razine izvršavaju na različitim platformama, pa se tako prezentacijska razina može nalaziti u *desktop* aplikaciji u obliku korisničkog sučelja, dok se logička i podatkovna razina mogu nalaziti na različitim poslužiteljima u izoliranoj okolini. Iz ovoga nastaje, također interesantna, prednost: razine ne ovise međusobno, i to u smislu da je na različitim razinama moguće kompletno zamijeniti okruženje (npr. u prezentacijskoj razini izmijeniti operativni sustav ili aplikaciju) i sve dok specifične razine „odrađuju" svoju funkciju, cijela arhitektura će ispravno funkcionirati. U odnosu na MVC arhitekturu, osim ove razlike, ovdje također ne postoji mogućnost komunikacije prve i zadnje razine, već cijeli proces uvijek započinje prezentacijskom, a završava podatkovnom razinom, što u određenim situacijama može biti i mana.

2.2 Modularno programiranje

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). Ovo se znatno razlikuje od tzv. „monolitičnih aplikacija" u kojima je i najmanji dio kôda zapravo neizostavni dio cjeline, što je gotovo uvijek nepoželjan faktor u procesu izrade softvera. Uobičajeni primjeri modularnih sustava su integrirani informacijski sustavi za podršku poslovanju (ERP sustavi, engl. *Enterprise Resource Planning Sastems*) ili sustavi za elektroničko poslovanje koji se mogu sastojati također od više modula. Primjer modularnog sustava za elektroničko poslovanje prikazan je na slici 2.

Slika 2. Primjer modularnog aplikacijskog sustava za elektroničko poslovanje

Izvor: Obrada autora

U navedenim primjerima vidimo kako se kompleksni sustavi mogu sastojati od svojih manjih zasebnih cjelina. Tako npr. ERP sustav može sadržavati modul za upravljanje odnosima s korisnicima (engl. *Customer Relationship Management*, CRM), ali i ne mora, ukoliko za to ne postoji zahtjev, dok će sustav i dalje funkcionirati uredno. S druge strane, sustav za elektroničko poslovanje može, ali i ne mora, sadržavati modul za izradu popusta na cijenu proizvoda.

Bitno je napomenuti kako moduli kao takvi ne komuniciraju međusobno u sustavu, osim u smislu korištenja drugog modula kako bi prvi mogao ispuniti svoju namjenu (npr. modul koji bi mogao služiti za ispisivanje izvješća može funkcionirati jedino ako prethodno u sustavu postoji i modul koji kreira izvješća).

Osim što omogućuje bolju strukturiranost i neovisnost kôda, ova tehnika ima još neke prednosti:

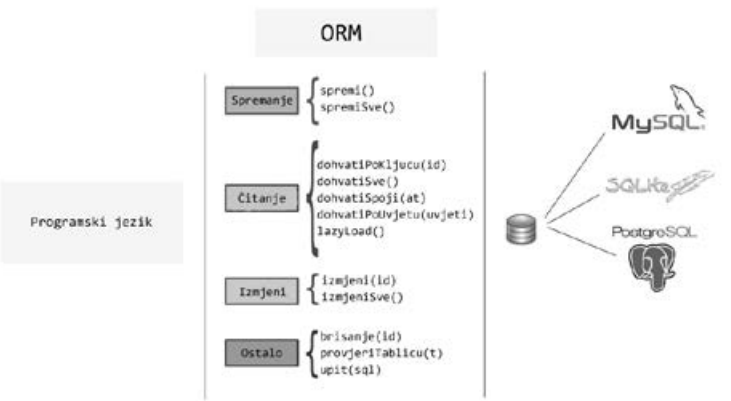
- kôd je smješten u različitim datotekama koje se kasnije uključuju u cjelinu,
- u isto vrijeme nekoliko programera može raditi na različitim dijelovima sustava,
- osnovu sustava je lakše održavati (popravci i unaprijeđivanje kôda) – ušteda na vremenu a time i novcu,
- kod isporuke proizvoda, moguće je definirati ukupnu cijenu na temelju modula koji su uključeni u paketu.

2.3 ORM sučelje

Općenito, pri kreiranju programskih aplikacija, programer će se gotovo uvijek susresti s različitim postupcima manipuliranja podacima, odnosno bazom podataka. Kod takvih operacija, programer piše različite upite nad bazom podataka (SQL upiti) i time kreira, čita, mijenja ili briše postojeće podatke. No, upiti često mogu biti slični ili čak isti za različite dijelove aplikacije. U praksi kôd se kopira, a u konačnici ga je i vrlo teško održavati jer se promjene ne izvode samo na jednom mjestu, nego na cijelom sustavu (automatski postoji mogućnost pogreške u kôdu na svakoj „kopiji").

Object relational mapping je način organizacije kôda odgovornog za izvršavanje upita prema bazi podataka. To je tehnika kojom je omogućena komunikacija i konverzija podataka između dva zapravo nekompatibilna sustava – objektno orijentiranog jezika i baze podataka. Konkretnije, ova tehnika omogućuje programerima da u kôdu gotovo u cijelosti isključe pisanje SQL upita jer to za njih odrađuje ORM sučelje, dok se njihov kôd svodi na rad s objektima. Na slici 3 je generalno prikazan način rada ORM sučelja.

Slika 3. Uloga i način rada ORM sučelja



Izvor: https://code.google.com/p/quickdb/

U praksi, korištenjem ORM-a, programer radi s objektima različitih klasa koje predstavljaju neke tipove entiteta koji imaju implementirane različite metode za rad s bazom podataka (npr. različite metode za čitanje ili spremanje podataka). Sama logika je enkapsulirana i ona se brine o

točnom izvođenju SQL upita umjesto programera. Možemo reći da ORM zapravo stvara virtualnu reprezentaciju objektne baze podataka, jer u kôdu zapravo virtualno spremamo objekt u bazu, odnosno čitamo ili dohvaćamo objekt iz baze. No, to u stvarnosti nije tako (za razliku od objektnih baza podataka), jer se u konačnici podaci zapisuju u obliku retka u tablici u bazi podataka. Danas postoje već nekoliko gotovih ORM rješenja (gotovo da se mogu i nazvati radni okviri zbog svoje kompleksnosti i mogućnosti), kao npr. Doctrine Project[5], Active Record[6] itd., a nerijetko se pišu i razna manja rješenja zbog bolje prilagodbe postojećoj strukturi podataka. Neka od rješenja nude i mogućnost rada s različitim platformama kao što su MySQL, PostgreSQL, SQLite, itd. U sljedećem primjeru kôda prikazan je primjer rada s poznatim ORM sučeljem Doctrine Project.

```
1  // Primjer pretraživanje članaka
2  $article = $entityManager->find('CMS\Article', 1234);
3  $article->setHeadline('Hello World!');
4  $article2 = $entityManager->find('CMS\Article', 1234);
5  echo $article2->getHeadline();
6  // Svi korisnici koji imaju 20 godina i prezivaju se Miller
7  $users = $em->getRepository('MyProject\Domain\User')->findBy(array('age' => 20, 'surname' => 'Miller'));
```

Velika prednost korištenja ove tehnike očituje se u tome što programeri ne moraju pisati SQL upite, a osim toga, kôd izgleda preglednije jer se sve svodi na rad s objektima te pozivanje njihovih metoda. Međutim, vrlo je teško kreirati ORM sučelje koje može pokriti sve naše zahtjeve. Drugim riječima, kompleksniji SQL upiti se moraju i dalje ručno pisati. Uz ovo, pokazalo se da programeri ponekad nesvjesno kreiraju loše dizajnirane tablice jer ih automatski prilagođavaju ORM sučelju koje koriste.

U izgradnji radnog okvira bit će implementirano i ORM sučelje. Pri razvoju ove komponente, nastojalo se što više pojednostaviti upotrebu, čime će se povećati produktivnost programera i smanjiti ukupna količina potrebnog kôda.

2.4 Izrada MVC arhitekture

U ovome poglavlju će biti prikazana izrada MVC arhitekture radnog okvira u skriptnom jeziku PHP, koja čini osnovu modularnog radnog okvira. Na slici 4 dan je prikaz UML dijagram[7] s kojim će se moći zornije prikazati cjelokupna struktura radnog okvira.

Radni okvir sastojat će se od nekoliko temeljnih klasa koje će sadržavati sve što je potrebno kako bi se sustav pravilno inicijalizirao, što će se izvoditi na svakom zahtjevu klijenta prema određenoj aplikaciji na poslužitelju. Sve započinje s klasom *Application*, izvršavanjem metoda kao što *init*() ili *getModuleConfig*(), nakon čega će se na kraju instancirati jedan objekt klase *Controller*. Ovdje će se vrlo često instancirati različiti objekti klase *Entity*, koja zbog dodatnih potrebnih metoda proširuje klasu *Model*, koja pak, u konačnici, proširuje klasu *DB_Resource* i koristi klasu *DB*, što omogućuje rad s bazom podataka. Jedino je u ponekim slučajevima moguće izravno korištenje klase *DB_Resource*, i to kada se želi zaobići klasa *Entity*, odnosno

[5] http://www.doctrine-project.org/, preuzeto 10. 8. 2013.

[6] http://www.phpactiverecord.org/, preuzeto 10. 8. 2013.

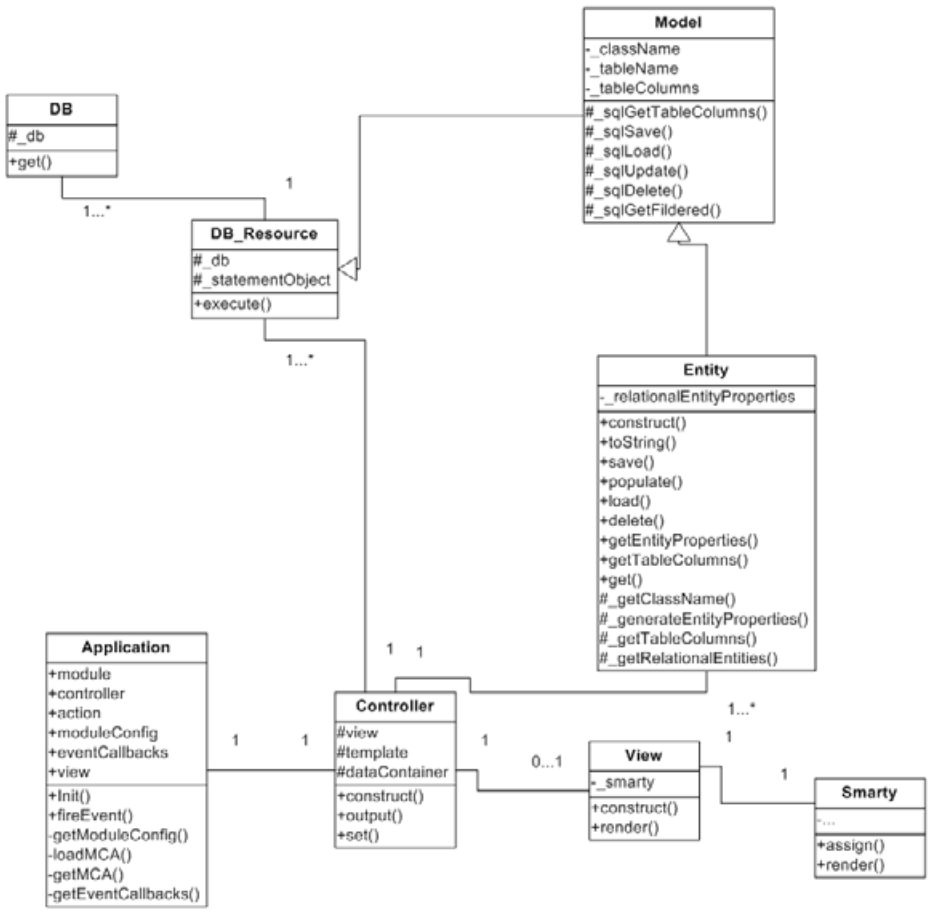
[7] UML dijagram - Ujedinjeni jezik za modeliranje (eng. Unified Modelling Language, kraće: UML) je normirani jezik opće namjene koji se koristi za modeliranje računalnih sustava temeljenih na objektno-orijentiranoj paradigmi.

kada ona, zbog okolnih faktora, programeru jednostavno nije potrebna. Iza ovog dijela nalazi se klasa *View* koja će također instancirati objekt klase *Smarty*. On će nam koristiti za generiranje HTML sadržaja, a koji će se u konačnici prikazati krajnjem korisniku.

Klase, atributi i metode su razvrstane logički prema svojoj namjeni. Struktura nije strogo definirana, stoga ju je moguće, prema eventualnim potreba klase, dodatno proširiti vlastitim metodama ili umetnuti dodatnu klasu između nekih postojećih, ukoliko je to potrebno.

Radni okvir će biti modularan, što znači da će podržavati proizvoljno uključivanje i isključivanje modula iz sustava. Nadalje, svaki modul koji će se pisati radit će prema MVC ahitekturi - drugim riječima, svaki modul će imati svoje kontrolere (može ih biti i više, ovisno o potrebi) te će sustav, prema definiranom algoritmu, prepoznati u kojem se modulu, kontroleru i akciji radi i to iz HTTP zahtjeva korisnika, odnosno pomoću upisane adrese. Za ovo će biti odgovoran usmjerivač, tzv. „ruter" i njegovo procesiranje će se u pravilu uvijek izvršavati prije bilo kojeg kontrolera (ponekad se ovaj kontroler naziva i prednji kontroler).

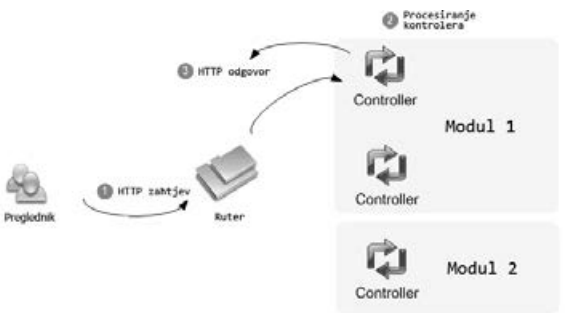
Slika 4. UML dijagram radnog okvira



Izvor: Obrada autora

Nakon prepoznavanja, nastavak procesa je jednostavan - sustav će instancirati objekt zadanog kontrolera, izvršiti metodu na njemu i ispisati rezultat korisniku. Ovaj proces je prikazan na slici 5.

Slika 5. Usmjerivač i ostali procesi radnog okvira

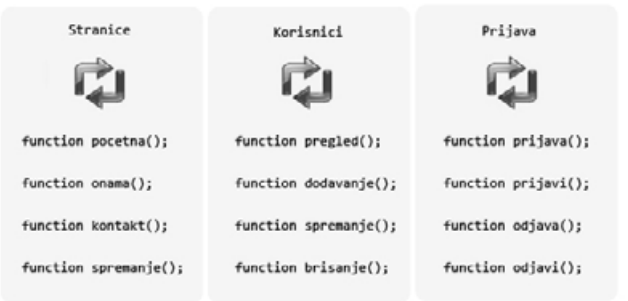


Izvor: Obrada autora

Iako su u dosadašnjim primjerima akcije vraćale rezultat, odnosno generirani HTML (engl. *output*), to ne mora biti uvijek tako. Npr., akcije ponekad mogu samo odraditi spremanje ili brisanje podataka u bazi, vršiti preusmjeravanje itd.

Kontroleri i njihove akcije se zapravo logički grupiraju u cjeline, pa tako može postojati sljedeća podjela – tri modula i klase kontrolera koje sadrže tipične akcije (slika 6):

Slika 6. Jednostavni primjeri kontrolera



Izvor: Obrada autora

U prvom slučaju definiran je kontroler *Stranice* istoimenog modula, koji je odgovoran za prikaz statičkih stranica. Prve tri akcije *pocetna*(), *onama*() i *kontakt*() mogu poslužiti za dohvaćanje predloška u kojima se nalaze potrebne informacije, dok četvrta metoda može spremati podatke iz formulara koji je korisnik dobio prethodno ispunjavajući formular dobiven pomoću akcije *kontakt*(). Organizacija je slična i u druga dva kontrolera, definirane su akcije koje će vraćati formulare za unos podataka (npr. *dodavanje*() i *prijava*()) i akcije koje će procesuirati podatke (npr. *spremanje*(), *odjava*()).

U ovom poglavlju analiziran je način rada spomenutog usmjerivača i elementi potrebni za njegovo pravilno funkcioniranje. Preciznije, analiziran je način na koji sustav bira potrebnu klasu i njezinu metodu, odnosno akciju kojom korisnik šalje zahtjev za određenom stranicom na poslužitelj. Da bi se ovo postiglo, potrebno je definirati sustav koji će za dani zahtjev znati koji se objekt mora instancirati i koja metoda se mora izvršiti. Ovo se postiže tako da se najprije u postavkama poslužitelja svi zahtjevi preusmjere na jednu datoteku u kojoj će cijeli sustav, i u konačnici usmjerivač, biti definiran. Navedena datoteka će u imati naziv *index.php* a preusmjeravanje će se ostvariti pomoću datoteke *.htaccess*.

2.5 Programiranje pogonjeno događajima

Kao što je poznato, pri pokretanju klasičnih programa, kompletan kôd se izvršava u određenom redoslijedu kojega je definirao programer. Dakle, nakon izvršene naredbe, izvršit će se sljedeća i tako sve do posljednje, odnosno do kraja programa. Iako postoji mogućnost da određeni dijelovi kôda nikada ni ne budu izvršeni (npr. kod grananja), nikada se neće dogoditi da se kasnija naredba izvrši prije nego što treba. Ovakva struktura može biti pogodna u okolinama gdje program korisniku ispisuje poruku, zatim čeka njegov odgovor te na temelju toga postupa dalje prema svojem kôdu. S druge strane, razvojem softvera i korisničkih sučelja, nastala je potreba za osmišljavanjem drugačije kodne arhitekture. Npr. u Windows operativnom sustavu, u prozoru koji nudi više od samo jedne mogućnosti, takav program ne bi imao mogućnost otkriti koji gumb (naredbu) je korisnik kliknuo. Drugim riječima, nastaje potreba za osmišljavanjem sustava koji će moći pratiti akcije koje je korisnik napravio i temeljem njih reagirati na potreban način.

Programiranje pogonjeno događajima" (engl. *„event-driven programming"*) je pristup koji se znatno razlikuje od klasičnog proceduralnog programiranja, jer se dijelovi koda odvijaju pri aktivaciji događaja. To može biti npr., klik na gumb, fokusiranje elementa, pritisak tipke na tipkovnici, prijelaz preko određene površine na sučelju, itd. Sve ovo su različite vrste događaja koje je moguće pratiti, a što programeru omogućuje da odrađuje baš onu funkcionalnost koja je potrebna. To mogu biti npr., prikaz, skrivanje ili pomicanje elemenata na sučelju, prikaz ili spremanje podataka u bazu, zatvaranje programa, itd. Ovakav način rada je zapravo najprisutniji u današnjem programiranju jer su aplikacije i sučelja jednostavno dizajnirana tako da je njihovo funkcioniranje moguće ostvariti jedino putem ovog koncepta.

No, na koji način programsko okruženje „doznaje" za neki događaj? Radi se, zapravo, o konstantom ispitivanju okruženja u kojemu se korisnik nalazi. Iako to korisniku nije vidljivo, u pozadinskom dijelu svakog programa pogonjenog događajima nalazi se petlja (engl. *event-loop*, naziv koji se također koristi je i *event-listener*) koja provjerava sve elemente na kojima se može ostvariti događaj (poput klika, fokusiranja i sl.). Ukoliko se stanje promijenilo, takva petlja će odmah izvršiti funkciju na istom elementu koja će odraditi ono što je programer specificirao. Funkcije koje se izvršavaju nakon određenog događaja nazivaju se *callback* ili *handler* funkcijama. Ova generalna ideja je prikazana na sljedećoj slici (slika 7):

Slika 7. Koncept detektiranja događaja

Izvor: Obrada autora

Osim velike fleksibilnosti koju ovaj kocenpt pruža, postoji i još nekoliko prednosti kao što su:

- strukturiranost i kontrola (razdvojen kôd),
- omogućava izradu boljih sučelja,
- kompleksnije kôdne strukture i veće mogućnosti (neovisan kôd).

Pored opisane primjene, sličan koncept se primjenjuje i u objektno-orijentiranom kontekstu, a koji će detaljnije biti analizirati u sljedećem odlomku.

2.5.1 Klase za praćenje stanja objekta

U uvodnom dijelu ovog poglavlja, u kojem su prikazani ključni elementi programiranja pogonjenog događajima, govorilo se uglavnom o primjeni koncepta pri izradi sučelja i definiranju interakcije s korisnikom. U nastavku slijedi malo drugačija implementacija pristupa koji ima elemente programiranja pogonjenog događajima te neke vlastite specifične zahtjeve i karakteristike, a služit će za praćenje stanja objekata te izvođenju adekvatnih *callback* funkcija.

Cilj izrade klasa za praćenje stanja objekta (klasa *Observer*) je također postojanje veza između elemenata u zajedničkom okruženju koje će omogućiti praćenje njihova stanja, no, konkretno, ovog puta navedeni elementi su zapravo objekti. Npr., za promjenu stanja jednog objekta, ostali ovisni objekti će moći „saznati" za tu promjenu te reagirati po potrebi. Fleksibilnost, definirana na ovaj način, može omogućiti bolju kontrolu objekata te olakšati dodavanje novih funkcionalnosti u postojeći kôd.

Tok ovog koncepta je vrlo jednostavan, a sastoji se od četiri elementa (ovoga puta koristit ćemo englesku terminologiju radi lakšeg razumijevanja):

1. Klasa *Observable* („pratljiva" klasa) je zapravo apstraktna klasa ili sučelje, koje će svaki konkretni objekt morati implementirati, a sadržavat će potrebne metode za rad s centralnom klasom *Observer* (priključenje ili isključenje *Observera*, praćenje stanja ostalih objekata itd.).
2. Klasa *ConcreteObservable* (konkretna klasa) koja implementira navedeno sučelje, a predstavlja objekt koji može mijenjati stanja, pri kojima će doći do obavještavanja ostalih objekata klase *ConcreteObservable*.
3. Klasa *Observer* (promatrač), također apstraktna klasa ili sučelje, koje mora implementirati konkretna klasa *Observer*.
4. Klasa *ConcreteObserver* – konkretna *Observer* klasa po kojoj će se instancirati svi *Observer* objekti, prati i obavještava objekte o promjenama stanja.

Prvo što je potrebno napraviti jest instanciranje konkretnih „pratljivih" objekata klase *ConcreteObservable*, koji će, naravno, implementrati sučelje *Observable*, čime će biti obavezni definirati potrebne metode za rad s klasom *ConcreteObserver*. Nakon toga, prethodno instancirani navedeni objekti klase *ConcreteObserver,* moraju se povezati s „pratljivim" objektima. Pri prijelazu u nova stanja, *ConcreteObserver* će „obavještavati" ostale povezane objekte koji će reagirati na određeni način, odnosno izvršavati određeni kôd, sami mijenjati stanja itd.

Za primjer, moguće je pretpostaviti da postoji objekt klase *Osoba*, koja, recimo, predstavlja oca. Uz navedenu, pretpostavimo prisutnost još dva objekta, ovoga puta klase *Dijete*, a koji predstavljaju djecu prve osobe. Ovim konceptom moguće je izvesti da se, automatski, pri brisanju prvog objekta (oca), brišu i objekti djece. Za ovo bi, naravno, bio odgovoran spomenuti *ConcreteObserver*, koji bi o promjeni stanja obavijestio preostala dva objekta, a koji bi onda reagirali na isti način. Navedena logika za određene situacije u kojima djeca nisu od velike važnosti sasvim je korektna, a i pritom automatizirana.

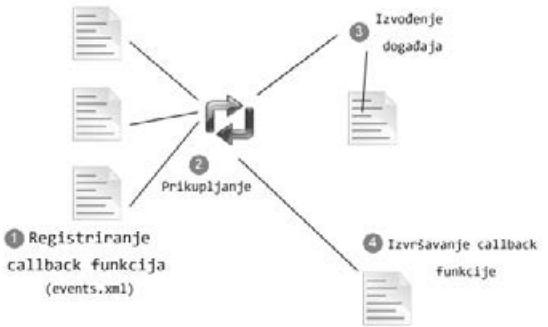
Moguće je uvidjeti da i ovdje postoje, na neki način, događaji koji mogu izazvati promjene u drugim dijelovima koda, odnosno u drugim objektima. Ovaj koncept je poznat već duže vrijeme i primjenjuje se upravo zbog fleksibilnosti i automatizacije koju je moguće postići. Osim toga, također smanjuje mogućnost pogrešaka, odnosno mogućnost da se objekti ne nađu u nekom od nelogičnih stanja.

2.5.2 Programiranje pogonjeno događajima u PHP-u

U ovom poglavlju bit će prikazana primjena programiranja pogonjenog događajima u PHP-u, tako što će se pokušati emulirati originalna ideja. Ono što će zapravo biti učinjeno jest interakcija između određenih dijelova kôda, gdje će jedan dio pozivati jedan ili više drugih dijelova i to putem „emitiranja" događaja. Nakon toga, funkcije koje su se prethodno „registirale" kao *callback* funkcija za navedeni događaj će se izvršiti (funkcija, dakako, može biti više od samo jedne). Potrebno je uzeti obzir da će i ti pozvani dijelovi također imati mogućnost emitiranja događaja, odnosno poziva drugih dijelova kôda, pa čak i onog iz kojega je poziv izvorno stigao, stoga je ponekad potrebno pripaziti da program ne uđe u beskonačnu petlju (moguće je, dakako, i implementirati provjeru koja će osigurati da ne dođe do ovakvih pogrešaka).

Implementacija programiranja pogonjenog događajima u radni okvir zapravo nije složen korak. Prije analize kôda, slijedi okvirni prikaz na koji način će se ostvariti željena funkcionalnost (slika 8). Prije svega, prvo što je potrebno napraviti jest registrirati *callback* funkcije za određene događaje (točka 1). Drugim riječima, upisat će se imena metoda koje se moraju izvršiti pri određenom događaju. Nakon toga, pri svakoj inicijalizaciji sustava, sve metode će se prikupiti (točka 2) i pohraniti u internu varijablu kako bi bile stalno dostupne ostatku sustava. I sada se dolazi, zapravo, do konkretnog slučaja (točka 3) u kojemu se nalazi emitiranje događaja (engl. *firing an event*) gdje će sustav potražiti ima li registirirane *callback* funkcije za dobiveni događaj. Ako one postoje, sustav će ih jednostavno izvršiti (točka 4)., a u protivnom se ništa neće dogoditi.

Slika 8. Način implementacije načina rada pogonjenog događajima



Izvor: Obrada autora

Registriranje *callback* funkcija je vrlo jednostavno, sve potrebne parametre potrebno je unijeti u XML datoteku events.xml. Radi bolje preglednosti, programer može za svaki modul kreirati zasebnu datoteku i u njoj upisivati *callback* funkcije. Sadržaj jedne datoteke *events.xml* prikazan je u sljedećem isječku kôda:

```
<events>
 <event>
  <type>beforeController</type>
  <callback>Pages|updatePageStats</callback>
 </event>
 <event>
  <type>afterController</type>
  <callback>Pages|test</callback>
 </event>
</events>
```

U primjeru se zapravno nalaze dvije registrirane *callback* funkcije koje će se pozivati na događajima *beforeController* i *afterController*. Moguće je uvidjeti da su ovo zapravo događaji koje izvršava sam sustav pri svojoj inicijalizaciji. Ovo je ugrađeno s ciljem da se programeru pružaju dodatne mogućnosti izvršavanja dodatnih operacija prije učitavanja kontrolera i poslije. Naravno, ovdje su mogli biti događaji koje je definirao sam programer, a moguće je i dodati dodatne događaje sustava, kao npr., prije same inicijalizacije i na kraju. Poslije „type" parametra slijedi definiranje callback funkcije koja će se pozvati. Sustav podrazumijeva da će se definicije tih funkcija nalaziti u poznatim datotekama kontrolera, stoga je u parametru potrebno upisati samo ime kontrolera i njegovu metodu, što se odvaja znakom „|".

Nakon toga, sve što preostaje je izvršavanje događaja u kontroleru, a to će biti učinjeno pomoću posebne metode koja je prethodno ugrađena u radni okvir. Ova metoda će proći kroz internu varijablu u kojoj su spremljene sve *callback* funkcije i provjeriti postoje li one koje su definirane za dani događaj. Što se tiče poretka izvršavanja, odnosno prioriteta, ovo je također jednostavno izvedeno, i to tako da se u *events.xml* callback funkcije upisuju u željenom poretku. Ako je potrebno promijeniti prioritet

izvršavanja, dovoljno je samo zamijeniti mjesta registriranja funkcija u datoteci i sustav će izvršavati funkcije po novom poretku.

3. IMPLEMENTACIJA MODULA ZA UPRAVLJANJE KORISNICIMA I KORISNIČKIM GRUPAMA

U ovom poglavlju bit će prikazan postupak izrade pojednostavljenog modula za kreiranje korisnika te korisničkih grupa. Navedenim primjerom će se pokazati koliko je jednostavno i brzo moguće kreirati dijelove aplikacija ovim radnim okvirom, odnosno njegovim ugrađenim komponentama. Za početak, potrebno je definirati konačni cilj ostvariv izradom ovog modula. Korisnik će imati sljedeće mogućnosti:

1. pregledavati trenutno unesene korisnike iz tablice,
2. dodavati, brisati, mijenjati korisnike spremljene u bazi podataka,
3. pregledavati trenutno unesene korisničke grupe iz tablice,
4. dodavati, brisati, mijenjati korisničke grupe spremljene u bazi podataka.

3.1 Tipovi entiteta i baza podataka

Kako će u sustavu biti potrebni korisnici i korisničke grupe, kreirat će se dva tipa entiteta i to User i Usergroup. Potrebno je napraviti dvije klase i kreirati tablicu u bazi podataka, a sustav će znati kako stupce tretirati i na koji način ih spremati. Za svaki entitet je potrebno napraviti nekoliko tablica u bazi, i to prema sljedećm SQL kôdu:

```
1  CREATE TABLE `user` (
2  `id` BIGINT(10) NOT NULL AUTO_INCREMENT,
3  `username` VARCHAR(20) NULL DEFAULT '' COLLATE 'utf8_bin',
4  `usergroup_id` BIGINT(20) NULL DEFAULT '0',
5  `password` VARCHAR(20) NULL DEFAULT '' COLLATE 'utf8_bin',
6  `first_name` VARCHAR(30) NULL DEFAULT '' COLLATE 'utf8_bin',
7  `last_name` VARCHAR(30) NULL DEFAULT '' COLLATE 'utf8_bin',
8  `age` INT(10) NULL DEFAULT NULL,
9  PRIMARY KEY (`id`)
10 )
11 COLLATE='utf8_bin'
12 ENGINE=InnoDB
```

Navedenim SQL kôdom kreirat će se tablica u bazi po imenu *user* i to s gore navedenim stupcima, od kojih će svaki sadržavati konkretan podatak osim *usergroup_id*, koji predstavlja vanjski ključ korisničke grupe za koju će korisnik biti vezan. Vanjski ključevi, kao što je i navedeno u prethodnim poglavljima, moraju sadržavati sufiks „_id" zbog ORM sučelja koje će pri učitavanju podataka prepoznati da se ovdje radi o drugom entitetu, odnosno o korisničkoj grupi. Nakon ovoga, potrebna je i, naravno, tablica korisničkih grupa koja je prikazana u narednom odsječku kôda. Ovdje postoji prostor za dodavanje još nekih podataka, no zbog ovog primjera nastojalo se maksimalno pojednostaviti strukturu. Nakon pripreme entiteta i baze podataka, slijedi izrada kontrolera koji će upravljati aplikacijom.

```
CREATETABLE`usergroup`(
 `id`INT(10)NOTNULLAUTO_INCREMENT,
 `name`VARCHAR(50)NULLDEFAULT''COLLATE'utf8_bin',
 PRIMARYKEY(`id`)
 )
COLLATE='utf8_bin'
ENGINE=InnoDB
```

3.2 Izrada kontrolera i predložaka

Kako postoje dvije sekcije aplikacije, odnosno korisnici i korisničke grupe, tako će biti potrebno kreirati i dva kontrolera, te na taj način odvojiti logiku. U aplikaciji je definiran dio kôda odgovoran za upravljanje korisnicima. Koristi se metoda *index*() koja je odgovorna je za prikazivanje početne stranice, odnosno popisa svih korisnika zbog čega je i korištena metoda *get*(), čime se dohvaćaju svi korisnici. Osim toga, specijalno za ovu aplikaciju razvijena je i Grid klasa koja će automatski generirati potreban HTML kôd za kreiranje tablice koju će ujedno i popuniti s podacima. Klikom na gumb „Add user" korisnik dolazi na drugu stranicu, čiji tok kontrolira metoda *action*() (iz prethodno prikazanog kôda). Potrebno je napomenuti da će ova ista metoda biti odgovorna za kreiranja novog korisnika, ali i za ažuriranje postojećih. Ovo se moglo izvesti pisanjem dvije različite metode, no ponekad je kôd zgodnije smanjiti, pogotovo kada je moguće postići više različitih funkcionalnosti. Kôd metode *action*() zapravo na samom početku provjeva jesu li poslani podaci iz formulara korištenjem metode *$_POST*. Ukoliko jesu, to znači da je korisnik popunio obrazac, te je upisane podatke potrebno spremiti. U suprotnome, učitava korisnika prema dobivenom ID-ju iz URL-a. Ako je validan, entitet User će se uspješno popuniti potrebnim podacima koji će se zatim prenijeti i u predložak koji će se koristiti za prikaz formulara. U suprotnome, entitet se neće popuniti podacima, te će se korisniku prikazati prazan formular. Kako je pri kreiranju korisnika potrebno odrediti kojoj će korisničkoj grupi pripasti, tako su i učitane sve korisničke grupe.

```
class User extends Controller{
  public function index() {
    $e = new Entity();
    $users = $e->get('User');
      $columns = array('username' => 'Username', 'first_name' => 'First name', 'last_name' => 'Last name',
'usergroup|name' => 'Usergroup');
    $grid = new Grid('datatable', 'users');
    $grid->setColumns($columns)->setData($users);
    $this->grid = $grid->render(); }
  public function action(){
    $user = new UserEntity();
    if (_getParam('id')) {
      $user->load(_getParam('id'));
    }
    if (isset($_POST) && count($_POST) > 0) {
      $user->populate($_POST)->save();}
    $e = new Entity();
    $this->usergroups = $e->get('Usergroup');
    $this->user = $user->load(_getParam('id'));}
```

3.3 Ažuriranje statistike stranica

U poglavlju koje je bilo posvećeno implementaciji kôda pogonjenog događajima, spomenuto je da se prije i poslije instanciranja svakog kontrolera emitiraju sustavni događaji *beforeController* i *afterController*. U ovom primjeru će se ovi događaji iskoristiti kako bi se zapisao broj posjeta određenoj stranici. Ukratko, registrirat će se funkcija na *beforeController* događaju, a koja će dobivenoj stranici povećati broj posjeta.

Za početak, pogledajmo datoteku *events.xml* pomoću koje će se odraditi registracija događaja i *callback* funkcije:

```
1    <event>
2      <type>beforeController</type>
3      <callback>Pages|updatePageStats</callback>
4    </event>
```

Ovih nekoliko poprilično jednostavnih linija kôda omogućit će, prije svake inicijalizacije kontrolera (na *beforeController* događaju), izvršenje funkcije *updatePageStats*(). Navedena funkcija je prikazana u sljedećem isječku kôda:

```
1  public function updatePageStats() {
2     $model = new Page_Stats();
3     $model->updatePageStats(Application::$action);
4     Application::fireEvent('korisnik_dod');
5  }
```

Navedena funkcija će samo instancirati objekt klase *Page_Stats* te na njemu pozvati metodu *updatePageStats*(), a kao parametar proslijediti naziv trenutne akcije. Navedena metoda zapravo samo izvršava SQL upit koji će ažurirati statistiku tako što će za trenutnu stranicu, ukoliko nije upisana u tablici, u tablici *page_stats* kreirati redak s početnom vrijednosti statistike koja iznosi 1. Ukoliko dođe do ponovnog učitavanja iste stranice, u tablici se neće ponovno kreirati isti redak za trenutnu stranicu, već će se isti povećati za 1, što je postignuto s ON DUPLICATE KEY UPDATE komandom (pri upitu na MySQL).

Ovo je bio jednostavan primjer na kojem možemo iskoristiti ugrađeni mehanizam pogonjen događajima u našem radnom okviru. Osim što imamo veliku fleksibilnost, ovom metodom omogućili smo suradnju različitih dijelova sustava te istovremeno njihovu potpunu neovisnost. Gore navedena funkcija se, naravno, može proširiti, a programer ima i mogućnosti dodavati funkcije za svaku situaciju u kojoj je potrebno koristiti *callback* funkciju. Neki od primjera događaja na koje možemo registrirati funkcije su uspješna autorizacija korisnika, kreiranje stranice, kreiranje ili brisanje korisnika, upload datoteke, itd.

4. ZAKLJUČAK

Zbog sve složenijih i zahtjevnih aplikacija, programeri moraju tražiti sve bolja sredstva rada. Repetitivne radnje se moraju svesti na minimum. U prvom planu mora biti razvoj same aplikacije, a ne rješavanje problema programskog jezika (ili ponekad čak radnog okvira) koji je u uporabi.

Ovim radom nastojalo se barem djelomično prikazati što se može postići pisanjem vlastitog radnog okvira. Na temelju vlastitog iskustva s drugim radnim okvirima i tehnologijama, nastojalo se učiniti programiranje što prirodnijim i jednostavnijim procesom, što je vrlo često problem u drugim razvojnim okruženjima. Često korištenje klasa neintuitivnih imena, ponekad nejasne logike i strukture, te sama težina radnog okvira, odnosno njegova dodatna procesiranja, mogu učiniti cijeli radni okvir manje efikasnim i nezgodnim za korištenje.

Iako je za potpunu funkcionalnost potrebno osmisliti veći broj komponenata, najvažniji je, zapravo, pravilan rad cijeloga sustava kao cjeline te omogućavanje lakog pristupa programeru komponentama sustava. Osim što programer ovime dobiva skup alata koji će mu omogućiti brži i efikasniji rad, radni okvir svojim ugrađenim konvencijama će također forsirati skladnu i dosljednu strukturu, što će pasivno omogućiti organiziranije pisanje kôda te dodavanje dodatnih komponenti po potrebi.

Radni okvir koji je ovdje izrađen ipak predstavlja samo temeljne i najnužnije alate, zbog čega bi bilo zgodno uvesti još neke dodatne komponente, kao što su sustav za autorizaciju i autentifikaciju korisnika, integracija dodatnih „cache" mehanizama, mogućnost korištenja kratkih ruta, izrada CMS rješenja, komponenti za višejezične aplikacije, slanje elektroničke pošte, itd. No, s jasnom strukturom i pravilima, ovaj posao bi se trebao olakšati, a s podržanom modularnom strukturom, pisanje posebnih modula specifičnih za pojedini projekt moguće je ponovno iskoristiti i, po potrebi, prilagoditi na drugim aplikacijama.

Nakon izrade navedenih komponenti, slijedi održavanje radnog okvira, s obzirom da nove tehnologije konstantno donose nove mogućnosti. Zbog velike količina posla u ovome segmentu, ipak bi bilo poželjno okupiti tim programera koji bi zajedno mogli surađivati i doprinositi radnom okviru. Na ovaj način, budući posao razvoja aplikacija bit će još brži, a samim time i produktivniji.

LITERATURA

Adams, J. (2013) *Learning Kendo UI Web Development*, Packt Publishing Ltd.

Ahsanul B., Anupom S., (2008) *Cake PHP Application Development*, Birmingham: Packg Publishing Ltd.

Eckerson, W. W. (1995) „Three tier client/server architectures: achieving scalability, performance, and efficiency in client/server applications", *Open Information Systems*, 3(20), p. 46-50

Fowler P. et al. (2002) *Patterns of Enterprise Application Architecture*, Addison Wesley

Hayder, H., Gheorghe, L. (2006) *Smarty PHP Template Programming And Applications*. Packt Publishing Ltd.

Krasner, G. E.; Pope, S. T. (1988) „A cookbook for using the model–view controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming", 1(3) p. 26-49

Matković S. et al. (2006) *Osnove programiranja u okruženju grafičkih operativnih sistema: Programski jezik C#*, Beograd: RG i CET

McArthur, K. (2008) *Pro PHP: Patterns, Frameworks, Testing and More*. Apress.

Sarode, P. (2001) Writing a reusable implementation of the MVC design pattern, JAVA REPORT, Volume: 6, Issue: 7

SensioLabs, (2013) *The Book for Symfony 2.3* (10. 8. 2013.)

Šribar J., Motik B. (2010) *Demistificirani C++*, Zagreb: Element

Adrian Smijulj [1]
Ana Meštrović [2]

Professional paper
UDC 004.415.2

MVC MODULAR FRAMEWORK BUILDING[3]

ABSTRACT

The objective of this paper was to provide an overview of the existing technologies and components related to framework creation and to show implementation of a simple framework that contains the most essential components for rapid development of web applications. In the first part of the paper, the theoretical considerations related to the implementation of the framework in developing applications are presented. Potential components of the framework are presented and analyzed on the theoretical level. The second part shows the implementation of the framework using modern technologies like PHP, MySQL, HTML etc. The presented framework is built using the object-oriented approach. The MVC architecture and ORM tools such as automated interface are used to work with the entities and for event management system. The advantage of the presented approach is that the programmer will not have to invest time in architecture design and tool development any more, but will be fully able to focus on project tasks. Although it contains only the core components, the framework will be flexible thanks to its modular architecture, allowing the developer to extend it to suit one's specific needs and to adapt it to the requirements of the application to be developed. To conclude, while developing complex applications it is advisable to invest time in making a framework or similar component that will later enable faster and more reliable development of the application.

Key words: *model-view-controller architecture (MVC), framework, event driven programming, modular systems, PHP*

[1] B. Ed. E-mail: adrian.1358@gmail.com
[2] PhD, Assistant professor, Department of informatics University of Rijeka, Radmile Matejčić 2, Rijeka, Croatia. E-mail: amestrovic@inf.uniri.hr
[3] Received: 27. 2. 2014.; accepted: 5. 5. 2014.