Универзитет у Београду
Математички факултет

Андреј Панић

Имплементација и примена архитектурних образаца у развоју веб и мобилних апликација

Мастер теза

Ментор: **др Владимир Филиповић**

Београд,
2017.

Садржај

Глава 1

Увод

Будући да је интернет динамички еволуирао у последњим деценијама, то су интернет технологије, као што је Јава, почеле интензивно да се користе код пословних апликација. Са појавом нових типова апликација, од ниског нивоа презентационих сервиса до високог нивоа система чији је рад критичан за мисију предузећа, граница између оригиналног веба (базираног на статичким садржајима) и конвенционалних апликација почиње да се губи. Међутим, веб апликације развијене на брзину почињу да показују велике проблеме у раду, као што су лоша способност слања повратних информација и лоша скалабилност, што указује да је кључан фактор успеха апликације добро постављена софтверска архитектура.

Интернет технологије постале су веома популарне за информатичку подршку пословања. У првом таласу представљен је интранет. Интранет нуди много више могућности него традиционална локална рачунарска мрежа. Његова снага је базирана на изузетном прожимању стандарда и протокола као што су HTML, HTTP и TCP/IP. Често коришћена документа, мултимедијални објекти и мали програми могу лако постати доступни запосленима преко интерне мреже. Интернет прегледачи скоро на свакој радној станици служе као универзалан front-end. У ситуацијама када треба интегрисати хетерогене оперативне системе (на пример Windows), релевантна стратегија за постизање тог циља је увођење тзв. "мршавог клијента". У таквом сценарију, покушаји скупог GUI-програмирања, које подржава разнородне платформе, могу бити смањени дефинисањем интернет прегледача као универзалног клијента.

Интернет технологије омогућавају организацијама да промене своје постојеће апликације, код којих велика количина пословних података постоји на централизованим

рачунарима (енг. mainframe). Главни ралози зашто се ове апликације не могу поставити на клијент/сервер архитектуру су цена, испорука и правни проблеми. У прошлости је приступ овим подацима био резервисан за кориснике који су имали везу с централним рачунаром. Коришћењем интернет технологија, интерни системи се могу проширити на нове групе корисника, као што су трговински партнери, добављачи и купци. Под маркентишким слоганом "мрежно израчунавање" пропагира се нова парадигма пословног израчунавања, која комбинује предности централизованог складиштења података и децентрализованог процесирања.

Међутим, развој нових типова апликација мора да се позабави са неколико изазова као што су:

- Постојање разнородних платформи;
- Скалабилност;
- Наслеђена интеграција;
- Реализација критичних тачака.

Дакле, главни проблеми са којима програмери морају да се суоче су избор адекватне софтверске архитектуре и стратегија имплементације.

Структура веб апликација обухвата сервисе и комуникационе механизме [1]. Стандардни системски сервиси, који могу бити интегрисани у веб апликације су:

- интернет прегледач, додаци
- веб сервер (систем)
- контролор трансакција (енг. TP monitor)
- систем управљања базом података (енг. DBMS).

Презентацију и пословну логику креирају програмери. Ово се може класификовати на следећи начин:

- Презентациони елементи су HTML стране, мултимедијални објекти (аудио, видео, 2D и 3D графици) и скрипте.

- Програмски кôд се интерпретира од стране интерпретатора или га извршава директно рачунар.
- Компоненте су софтверски блокови који испуњавају услове специфичног компонентног модела. Распрострањени примери су ActiveX, JavaBeans и Enterprice JavaBeans (EJB).

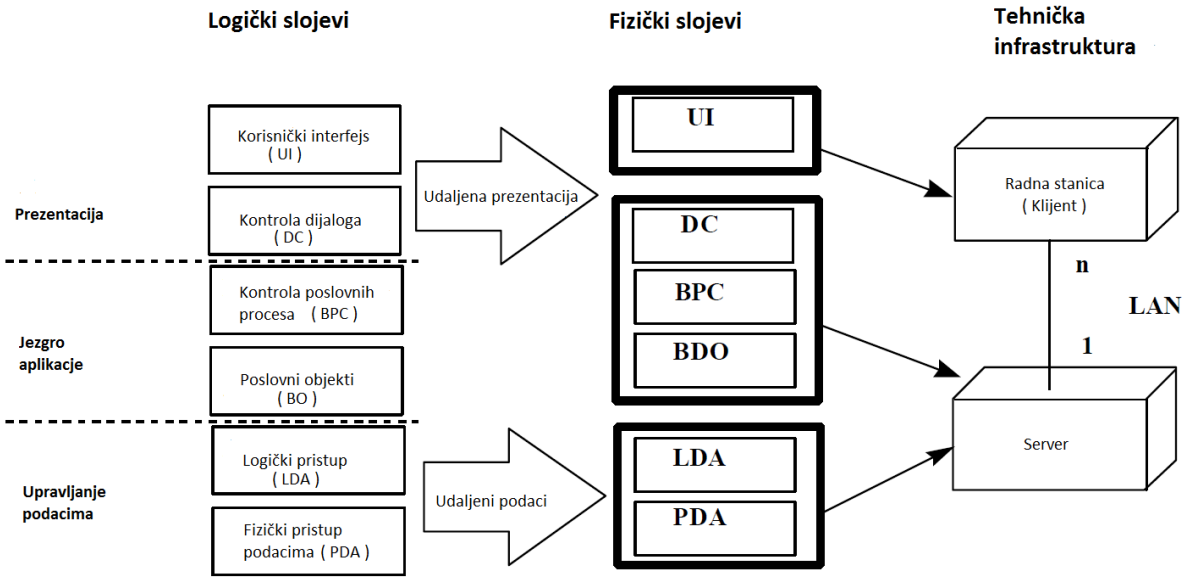
Комуникација између различитих сервиса базирана је на скупу протокола или интерфејса:

- Интернет прегледач/веб сервис: HTTP (и његова сигурна варијанта HTTPS) је стандардан протокол за комуникацију између интернет прегледача и веб сервера. Будући да HTTP протокол нема стања, његова употреба је ограничена.
- На веб серверу се извршава серверски део кода апликације: Комуникација између веб сервера и апликативног програма базирана је на различитим API-јима (енг. Application Programming Interface), као што су: CGI, Servlet-API, ISAPI, NSAPI итд.
- Комуникација између објеката: Дистрибуирани програми, написани у Јави или било ком другом веб програмском језику, могу да комуницирају једни са другима коришћењем различитих протокола, као што су: RMI, IIOP, RPC и веб сервисе.
- Приступ бази података: постоји неколико стандарда за приступ базама података, као што су: SQL, ODBC и JDBC.
- Пролази (енг. gateway): Омогућују везу између централног рачунара и веб окружења. Тако, на пример, "CICS пролаз за Јаву" спаја интернет прегледач са Јавом и CICS програм који се покреће на централном рачунару.

Глава 2

Опис софтверских архитектура и образаца

Рашчлањивање по слојевима је инжењерски приступ који помаже да се смањи комплексност система. Сваки слој је апстракција специфичног проблема домена. Рашчлањивањем по слојевима добија се добро дефинисан интерфејс. Хијерархијско уређење гарантује да слој може бити имплементиран коришћењем интерфејса добијеног од слоја на нижем нивоу. Рашчлањивање по слојевима даје следеће предности: мања зависност између модула, локална обрада проблема, преносивост и аспекти поновног коришћења.



Слика 2.1: Архитектурна пресликавања у клијент/сервер окружењу

Шест слојева на слици 2.1 приказују стандардну трослојну архитектуру, познату под називом "Сихајмов модел" (енг. "Seeheim-model"). Презентациони слој садржи кориснички интерфејс и контролу дијалога. Језгро апликације се састоји од пословних, доменских објеката, који имплементирају потребе пословне логике, као и од процесних објеката, који имплементирају процес коришћења специфичних информација. Слој управљања подацима омогућава функције приступа подацима и бави се питањима перзистентности (енг. persistence) објеката. Да би се компоненте апликације могле испоручити, логичка архитектура мора бити пресликана на физичке слојеве. Гартнер група (енг. Gartner Group) дефинише 5 дистрибуираних образаца за двослојну клијент/сервер парадигму [1]:

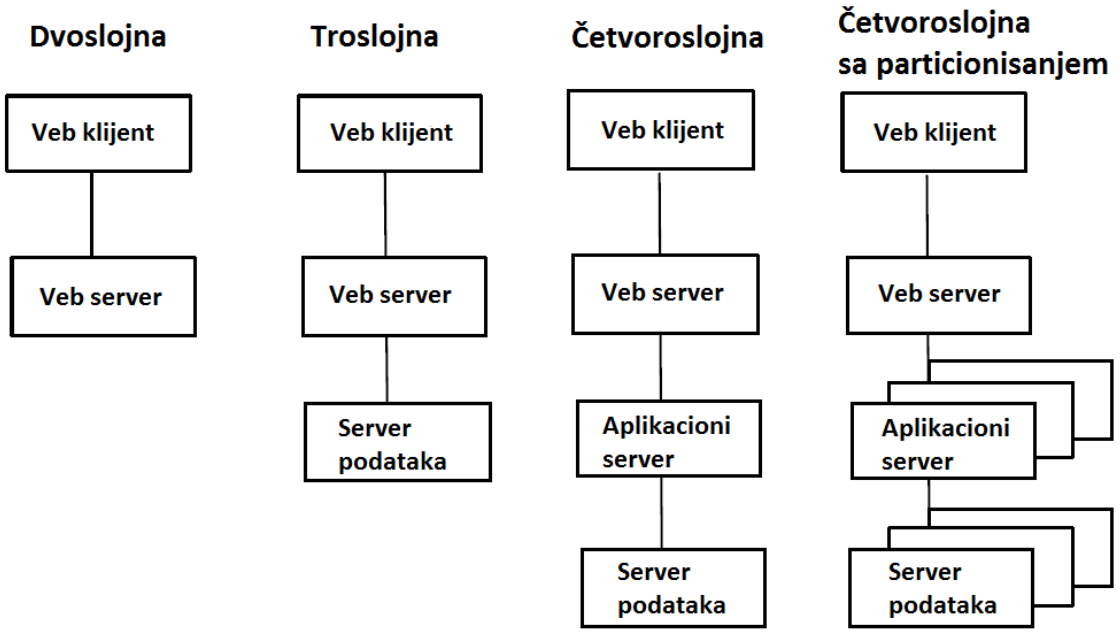
- Дистрибуирана презентација;
- Удаљена презентација;
- Дистрибуирана функција;
- Удаљени подаци;
- Дистрибуирани подаци;

Ови обрасци се могу комбиновати за вишеслојне приступе. Слика 2.1 илуструје трослојну архитектуру наслеђену из апликације са удаљеном презентацијом и дистрибуираним подацима. У следећем кораку, дистрибутивне јединице морају бити алоциране на радним чворовима техничке инфраструктуре. Пошто је кориснички интефејс пресликан на радним станицама, постоје алтернативе за друге слојеве у клијент/сервер окружењу. Обезбеђивање скалабилности апликације је разлог зашто дистрибутивне јединице, које се покрећу на истој радној станици и треба да буду одвојене у различите слојеве. На слици 2.1 подебљана линија представља границу за сваки од три слоја. Кориснички интерфејс је алоциран на радној станици. Друга два слоја се покрећу на серверу.

Описани архитектурни процес може се лако адаптирати за окружења пословног израчунавања. Приликом пресликавања логичких на физичке слојеве, јавља се неколико специфичности. Обично се разликују четири физичка слоја:

- Веб клијент обично представља кориснички интерфејс у интернет прегледачу.

- Веб сервер је одговоран за дистрибуцију HTML-докумената, мултимедијалних објеката или Јава аплета које захтева веб клијент преко HTTP протокола.
- Апликацијски сервер контролише пословне процесе и доменске објекте.
- Сервер података омогућава функције приступа подацима и бави се питањима перзистентности објеката.



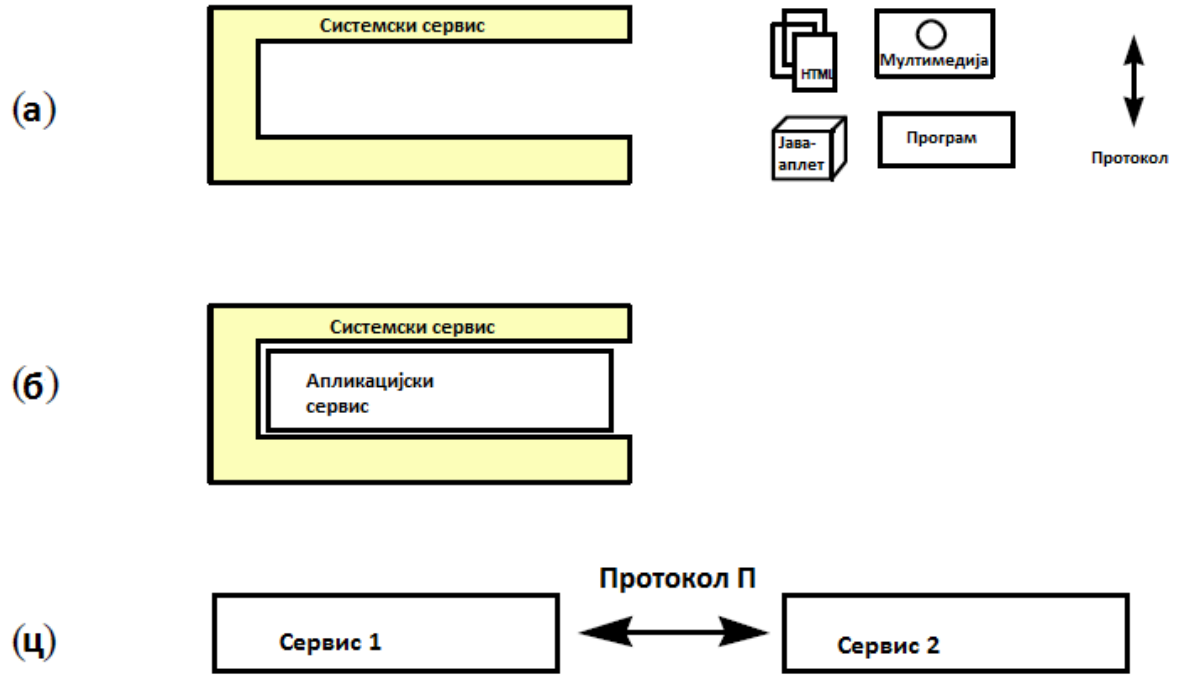
Слика 2.2: Физички слојеви веб апликације

Различити типови веб апликација могу се описати коришћењем релевантне комбинације слојева (слика 2.2). Двослојна архитектура може се користити за једноставну презентацију статичких HTML-докумената, које захтева интернет прегледач, док је високо софистицирани сценарио са вишеструким апликативним серверима и серверима за податке базиран на четворослојној архитектури са партиционисањем.

2.1 Графички описни језик

Архитектурни опис обухвата речник елемената дизајна (компоненте и конектори), ограничења која описују валидне комбинације ових елемената и семантичку интерпретацију за опис система.

Нотација магнетног пола (eng. Pole shoe) (слика 2.3) обезбеђује описни језик за физичке архитектуре. Прагматични приступ узима у обзир ограничења, али изоставља формални аспект семантичке интерпретације.



Слика 2.3: Нотација магнетног пола

Системски и апликацијски сервиси одговарају компонентама и комуникационим механизмима са конекторима (слика 2.3а). Пошто су системски сервиси презентовани како окружују апликацијски сервис, то значи да се апликацијски сервис извршава под контролом системског сервиса (слика 2.3б). Протокол дефинише основни комуникациони механизам између два сервиса (слика 2.3ц). Очигледна ограничења која се могу извести из претходних дискусија су следећа:

- Интернет прегледач је део слоја веб клијента.
- Системски веб сервер је део слоја веб сервера.
- DBMS је део слоја сервера података.
- Контролор трансакција је део слоја апликацијског сервера.

2.2 Архитектурни обрасци

Дизајнирање објектно-оријентисаног софтвера је тешко, али је дизајнирање објектно-оријентисаног софтвера који се може поново користити (енг. reusable) још теже. Морају се издвојити значајни објекти, који ће се потом рашчланити на класе праве величине, а затим дефинисати интерфејси, хијерархија наслеђивања и успоставити везе између њих. Дизајн треба да буде прилагођен тренутном проблему, али тако да буде и довољно генерализован да опише будуће проблеме и захтеве. Треба избећи поновни дизајн или га, у најмању руку, минимизовати. Искусни дизајнери ће рећи да се флексибилан дизајн, који се може поново користити, тешко креира "из прве", ако не и да је немогуће. Обично се неколико пута покуша да се дизајн поново употреби пре него што се заврши његово креирање.

Кристофер Александер[1] каже: *"Сваки образац описује проблем који се појављује изнова и изнова у нашем окружењу, као и срж решења за тај проблем, на такав начин да се ово решење може користити милион пута, а да се не понови два пута на исти начин"*. Иако је Александер говорио о обрасцима у грађевинама и градовима, овај његов исказ је истинит и у објектно-оријентисаним дизајн обрасцима. Решења која се овде разматрају су изражена у терминима објеката и интерфејса уместо зидова и врата, али у сржи обе врсте образаца су решења проблема у контексту.

Генерално, обрасци имају четири суштинска елемента [2]:

1. **Назив обрасца** се користи да опише проблем дизајна, његово решење и последице у једној или две речи. Именовање обрасца обогаћује речник дизајна. Оно омогућава дизајнирање на вишем нивоу апстракције. Проналажење добрих назива за обрасце често бива један од тежих делова развоја овог речника.

2. **Проблем** се односи на то када се образац може применити. Овде се описује проблем и његов контекст. Овај елемент обрасца би могао описати специфичан проблем дизајна, као што је репрезентација алгоритама као објеката, а могао би описати класу или структуре објеката који су симптоми нефлексибилног дизајна. Понекад ће елеменат обрасца, који се односи на проблем, садржати листу услова коју треба испунити пре него што се образац примени.

[1] Кристофер Александер (енг. Christopher Alexander) - рођен 1936, аустријски архитекта и дизајн теоретичар, професор на универзитету Калифорнија, Барклi. Најпознатији је по својој књизи A Pattern Language: Towns, Buildings, Construction

3. **Решење** садржи опис елемената који чине дизајн, њихове везе, задужења и сарадњу. У решењу се не описује посебан дизајн или имплементација, зато што је образац као шаблон који се може применити у многим различитим ситуацијама. Уместо тога, образац обезбеђује апстрактан опис проблема дизајна и како се општим распоредом елемената (у нашем случају класа и објеката) то решава.

4. **Последице** су резултати и компромиси примене обрасца. Последице се често односе на просторне и временске компромисе. Такође се могу односити на проблеме језика и имплементације. Како је поновно коришћење чест фактор у објектно-оријентисаном дизајну, последице обрасца укључују његов утицај на флексибилност система, надоградњу или преносивост. Ове последице експлицитно помажу разумевање и процену обрасца.

Обрасци помажу програмерима приликом развоја апликација на основу колективног искуства софтверских инжењера. Они користе постојеће искуство, које се добро показало у софтверском развоју и промовишу праксу доброг дизајна апликација. Сваки образац се односи на специфичан проблем (који се често понавља) у дизајну. Обрасци се могу користити за конструисање софтверских архитектура са специфичним особинама.

Више о дизајн обрасцима може се пронаћи у [2], [3].

2.3 Модел-поглед-контролер

Термин модел-поглед-контролер (МПК, енг. MVC, Model-View-Controler) је у употреби још од касних 1970-тих и јавља се у Smalltalk пројекту компаније Xerox PARC, где је замишљен као начин организовања раних GUI апликација. Неки од детаља оригиналног МПК обрасца су везани за специфичне концепте Smalltalk пројекта, као што су екрани и алати, али су шири концепти и даље применљиви у апликацијама. Данас је овај образац стекао велику популарност као образац за веб апликације из следећих разлога:

- Корисничка интеракција са МПК апликацијом прати природни циклус: корисник предузима неку акцију и, као одговор, апликација мења модел података и кориснику приказује ажурирани поглед. Затим се циклус понавља. Ово је

веома погодно за веб апликације које се испоручују као серија HTTP захтева (енг. requests) и одговора.

- Веб апликације намећу комбиновање неколико технологија (база података, HTML и извршног кода, ...) обично подељених на скуп нивоа (енг. tiers) или слојева (енг. layers).
- Поновно коришћење кода.

Даље, МПК присиљава одвајање појединих делова - доменски модел и логика контролера су одвојени од корисничког интерфејса. У веб апликацији, ово значи да је HTML раздвојен од остатка апликације, што чини одржавање и тестирање једноставнијим и лакшим. До додатне популаризације овог обрасца у веб апликацијама довела су два веома популарна развојна окружења - Struts и Ruby on Rails.

Општије, МПК образац дели апликацију на најмање три дела:

- **Моделе** (енг. Models), који садрже или презентују податке са којима корисници раде. Они могу бити једноставни поглед модели (енг. view models), који само репрезентују податке који се преносе између погледа и контролера, или могу бити доменски модели, који садрже податке пословног домена, као и операције, трансформације, и правила за манипулисање тим подацима.
- **Погледе** (енг. Views), који се користе за приказивање делова модела у оквиру корисничког интерфејса.
- **Контролере** (енг. Controllers), који обрађују захтеве, врше операције на моделу и бирају погледе које ће приказати кориснику.

Модели су дефиниције универзума у којем апликација ради. Примера ради, у апликацији за банку, модел представља све што банка подржава. То су налози, главна књига, кредитни лимити за клијенте, као и операције које су могу користити за манипулисање подацима у моделу, као што су депоновање средстава и подизање са рачуна. Модел је такође одговоран за очување укупног стања и конзистенцију података (на пример, пазити да су све трансакције додате у главну књигу и да клијент не повуче више новца него што има, или више новца него што банка има).

Модели се не баве приказивањем корисничког интерфејса или обрадом захтева - то су одговорности погледа и контролера. Поглед садржи логику потребну за приказивање елемената модела кориснику. Погледи немају директан приступ моделу и не комуницирају директно са моделом. Контролери су мост између погледа и модела - захтеви стижу од клијента и услужени су од стране контролера, који бирају одговарајући поглед да прикажу кориснику и, ако је потребно, изводе одговарајућу операцију над моделом.

Кориснички интерфејси су склони честим променама. Развој система са потребном флексибилношћу је скуп и подложан грешкама, ако је кориснички интерфејс уско повезан са функционалним језгром. Ово може довести до потребе да се развије и одржава неколико различитих софтверских система, по један за сваку имплементацију корисничког интерфејса.

Сваки део МПК архитектуре је добро дефинисан и самосталан (ово се односи на раздвајање проблема). Начин манипулисања подацима модела је садржан само у моделу; подаци се приказују само у погледу, а кôд који обрађује корисничке захтеве и улазе се налази само у контролеру. Са јасном поделом између свих делова, апликација је лакша за одржавање и проширивање током трајања, без обзира на то колико велика постаје.

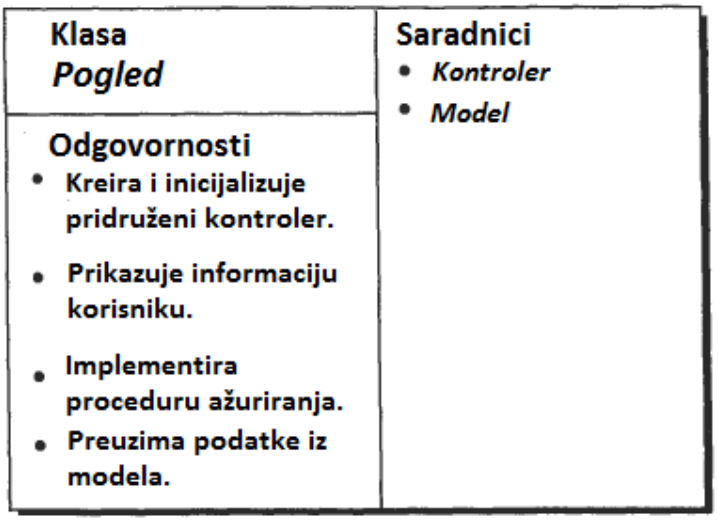
Раздвајање модела од погледа и контролера дозвољава вишеструке погледе за исти модел. Ако корисник промени модел кроз контролер једног погледа, сви други погледи, који зависе од ових података, треба да рефлектују промене. Дакле, модел обавештава све погледе, кад год се њихови подаци промене. Погледи претварају нове податке из модела и ажурирају приказане информације.

Слика 2.4: Модел

Механизам пропагирања промена одржава регистар зависних компонената у оквиру модела. Сви погледи и изабрани контролери региструју своју потребу да буду информисани о променама. Промене у стању модела покрећу механизам пропагирања промена. Овај механизам је једина веза између модела, погледа и контролера.

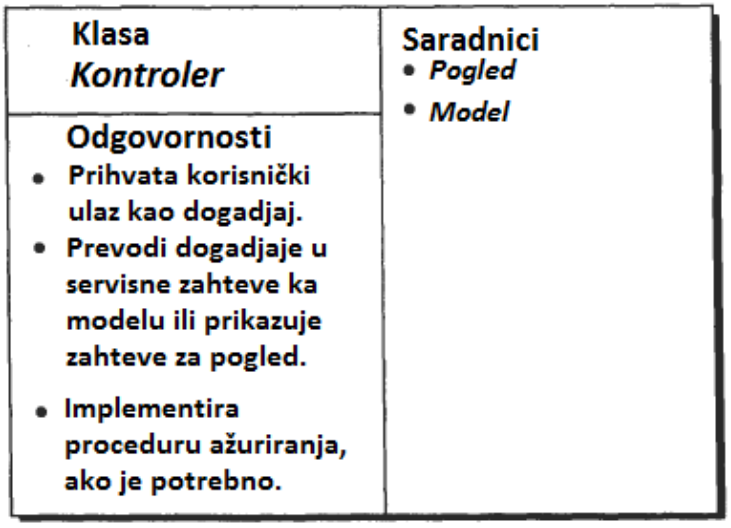
Компоненте за поглед приказују информације кориснику. Различити погледи приказују информације модела на различите начине. Сваки поглед утиче на ажурирање процедуре, која је активирана механизмом пропагирања промена. Када је процедура за ажурирање позвана, ажурирају се вредности података који се приказују.

Током иницијализације, сви погледи су повезани са моделом и регистровани механизмом пропагирања промена. Сваки поглед креира одговарајући контролер. Дакле, оформљен је један-један однос између погледа и контролера. Погледи често нуде функционалност која дозвољава контролерима да манипулишу приказом, што је корисно за корисничке операције које немају утицаја на модел (на пример, скроловање).



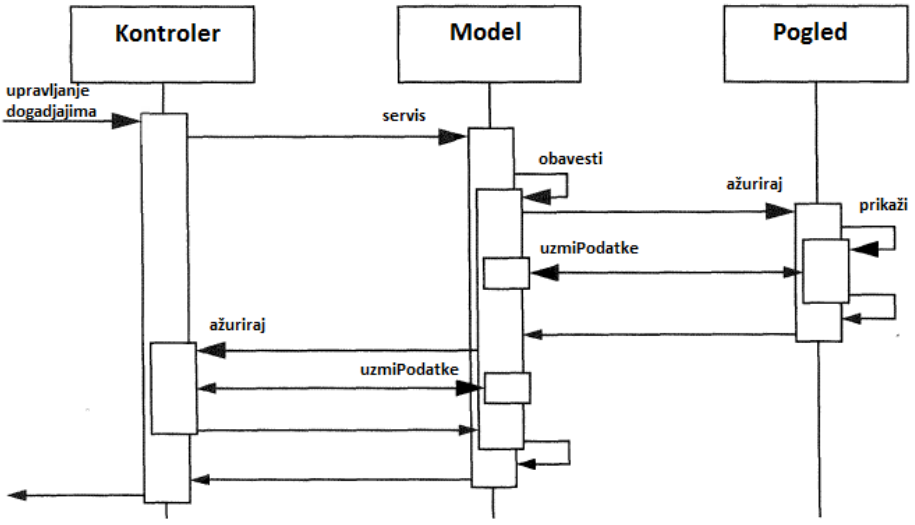
Слика 2.5: Поглед

Контролер компонента прихвата кориснички улаз као догађај. Начин на који су ови догађаји испоручени контролеру зависи од платформе корисничког интерфејса. Ради једноставности, нека сваки контролер имплементира процедуру руковања догађајем и нека се она позива за сваки релевантни догађај. Догађаји су преведени у захтеве за модел или у захтеве за повезани поглед. Ако понашање контролера зависи од стања модела, контролер се региструје на механизам пропагирања промена и имплементира процедуру ажурирања. На пример, ово је неопходно, када се у зависности од промене на моделу, приказује ставка менија.



Слика 2.6: Контролер

Следећа два сценарија одсликавају динамичко понашање МПК обрасца [3]. Због једноставности, на дијаграмима је приказан само један пар погледа и контролера.

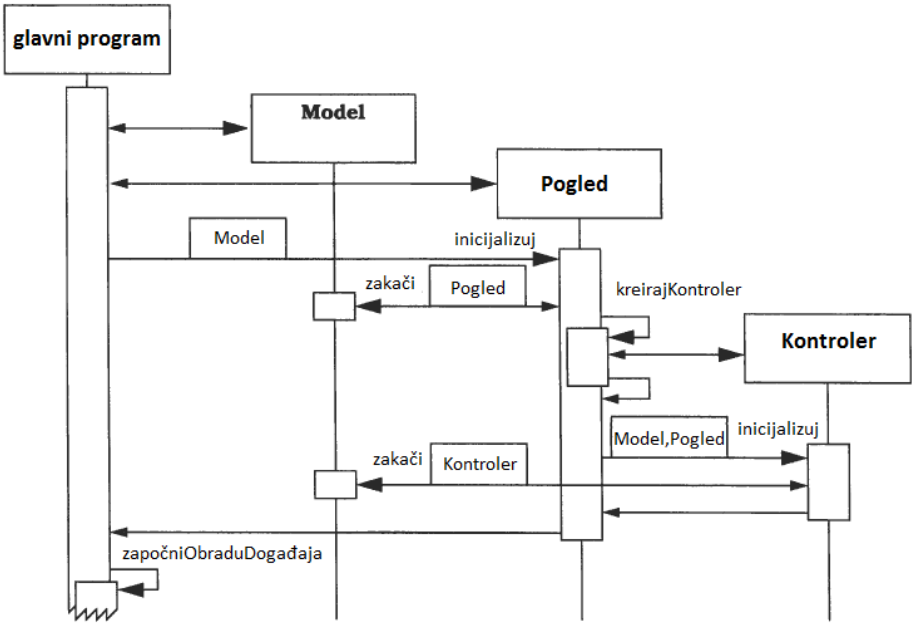


Слика 2.7: Сценарио 1

Први сценарио (описан сликом 2.7) показује како кориснички улаз који је резултат промена у моделу активира механизам пропагирања промена:

- Контролер прихвата кориснички улаз у поступку руковања догађајима, интерпретира догађаје и активира сервисну процедуру модела.
- Модел обавља захтеване сервисе. Ово доводи до промена у интерним подацима.

- Модел обавештава све погледе и све регистроване контролере са механизмом пропагирања промена о променама тако што позива њихове процедуре за ажурирање.
- Сваки поглед захтева измењене податке од модела, а затим се поново приказује.
- Сваки регистровани контролер преузима податке из модела, да омогући или забрани одређену корисничку функцију. На пример, омогућавање уноса у мени за чување података, може бити последица модификација модела података.
- Оригинални контролер поново добија контролу и враћа је из процедуре руковања догађајима.



Слика 2.8: Сценарио 2

Други сценарио (описан сликом 2.8) показује како је МПК образац иницијализован. Овај кôд је обично лоциран изван модела, погледа и контролера, на пример у главном програму.

- Примерак модела се креира, а он затим иницијализује структуру података.
- Креира се објекат који представља поглед. Параметар за иницијализацију погледа је референца на претходно направљени модел.
- Поглед се претплаћује на механизам пропагирања промена модела тако што позива придружену процедуру.

- Поглед наставља иницијализацију тако што креира свој контролер. Он прослеђује референцу и моделу и себи до процедуре за иницијализацију контролера.
- Контролер се такође претплаћује на механизам пропагирања промена тако што позива прикачену процедуру.
- После иницијализације, апликација почиње да обрађује догађаје.

2.3.1 Предности и мане МПК обрасца

Предности:

- Убрзан је процес развоја: МПК подржава брз и паралелан развој. Са МПК обрасцем, један програмер може радити на погледу, док други могу радити на контролеру.
- Могућност вишеструких погледа: исти модел може имати више погледа. Дуплирање кода је ограничено, зато што МПК одваја податке и пословну логику од корисничког интерфејса.
- Подржава асинхроне технике.
- Лако је додати приказ нових података. Да би се нови подаци приказали, потребно је имплементирати само контролер и поглед.
- Промене не утичу на цео модел, зато што модел не зависи од погледа. Било која промена на моделу не утиче на целу архитектуру.
- Апликације се лако проширују и скалабилне су.

Мане:

- Повећана је комплексност.
- Неефикасност приступа подацима у погледу.
- Обично је потребно познавање различитих технологија.
- Потребно је више програмерског рада.
- Постоје тешкоће у заједничком коришћењу МПК обрасца са модерним технологијама корисничког интерфејса.

2.3.2 Други обрасци који се користе у МПК

Веза између погледа и контролера је пример Strategy дизајн обрасца. Strategy дизајн образац дефинише фамилију алгоритама, енкапсулира сваки алгоритам и омогућава да се сваки алгоритам може заменити у оквиру те фамилије. Strategy дозвољава да алгоритам варира у зависности од клијената који га користе.

МПК користи и друге дизајн обрасце, као што су *Factory* који служи да одреди класу контролера за поглед и *Decorator* за скроловање у оквиру погледа. Везе између главних компонената МПК обрасца се остварују коришћењем *Observer*, *Composite* и *Strategy* дизајн образаца.

Дизајн образац Observer дефинише једну или више зависности између објеката. Када један објекат промени стање, сви који од њега зависе су обавештени и аутоматски ажурирани. Главни објекти у овом обрасцу су субјекат и посматрач. Субјекат може имати више посматрача. Сви посматрачи су обавештени кад год субјекат пролази кроз промену стања. Као одговор, сваки посматрач ће питати субјекта да синхронизује своје стање са субјектовим стањем.

Ова врста интеракције је такође позната као објава-претплата (енг. Publish-Subscribe). Субјекат је издавач обавештења. Било који број посматрача може да се претплати да добија обавештења. Дизајн образац Observer се користи у следећим ситуацијама:

- када апстракција садржи два аспекта, тако да један зависи од другог. Енкапсулирање ових аспеката у одвојене објекте дозвољава варијације и њихово поновно независно коришћење.
- када промена на једном објекту захтева промену других објеката, а није познато колико објеката треба променити.
- када неки објекат треба да обавести друге објекте, без претпоставки о томе ко су ови објекти. Другим речима, ови објекти не треба да буду уско повезани.

Дизајн образац Composite смешта објекте у структуре дрвета тако да представљају део хијерархија. Composite дозвољава клијентима да на униформан начин третирају индивидуалне објекте и композиције објеката. Кључ овог дизајн обрасца је апстрактна класа која представља примитивне објекте и њихове контејнере. Composite

дизајн образац такође декларише операције које сви композитни објекти деле, попут операција за приступ и управљање својом децом. Дизајн образац Composite се користи у следећим ситуацијама:

- када се представља део хијерархије објеката.
- када клијенти могу да игноришу разлику између композиције објеката и индивидуалних објеката. Клијенти ће третирати све објекте у композитној структури униформно.

Образац Factory дефинише интерфејс за креирање објеката, али дозвољава потклasама да одлучују које ће класе инстанцирати. Овај образац омогућава класи да одложи инстанцирање потклаcа.

Радни оквири (енг. frameworks) користе апстрактне класе да дефинишу и одржавају везе између објеката. Радни оквир је такође често одговоран за креирање ових објеката. Factory се користи у следећим ситуацијама:

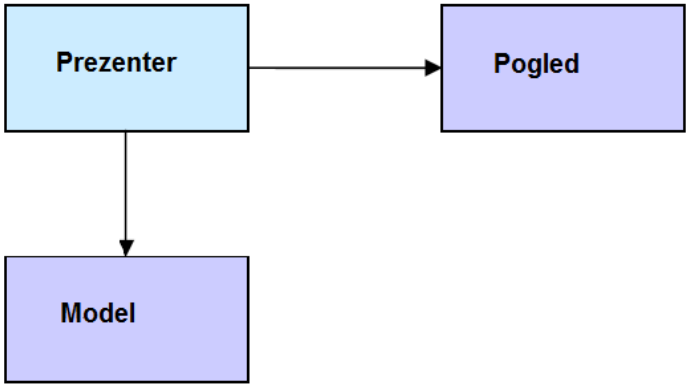
- када класа не може да предвиди класу објеката коју мора да креира
- када класе преносе одговорност на једну од неколико помоћних потклаcа

2.4 Модел-поглед-презентер

Модел-поглед-презентер (МПП, енг. MVP, Model-View-Presenter) образац ставља нагласак на поглед, да контролише ток логике кроз презентациони слој.

Три различита дела чине МПП образац:

- **Модел** садржи пословне податке који ће бити приказани или промењени од стране погледа.
- **Поглед** приказује податке добијене из модела преко презентера и поверава му кориснички улаз.
- **Презентер** се позива из погледа и служи да прикаже податке повучене из модела и да управља корисничким улазом.



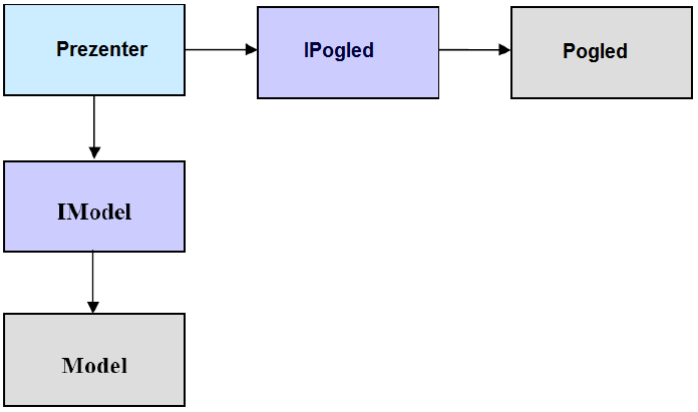
Слика 2.9: Модел-поглед-презентер

Основна улога МПП обрасца (слика 2.9) је раздвајање доменске и апликативне логике (модела) од слоја корисничког интерфејса (погледа) [4]. Ту улогу обавља презентер тако што креира и приказује кориснички интерфејс. Он податке добија од модела и прослеђује их корисничком интерфејсу. Када корисник промени податке на корисничком интерфејсу, презнтер те податке прослеђује моделу. Презентер делује као посматрач (енг. Observer) догађаја, који се јављају у моделу или корисничком интерфејсу. Предности овог одвајања су следеће:

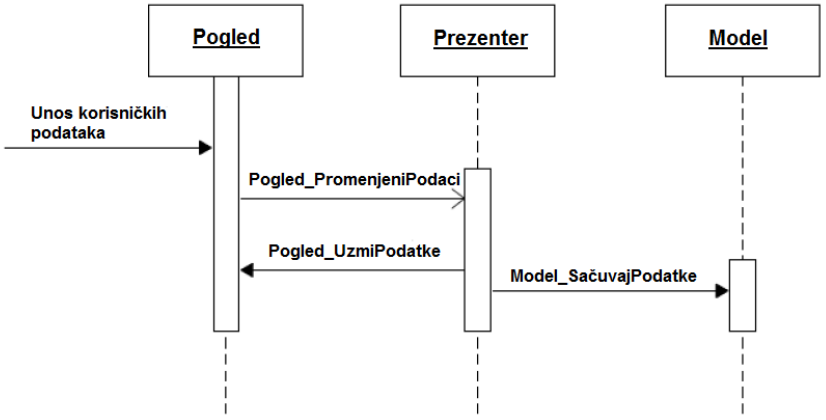
- Логика домена није распршена по коду корисничког интерфејса. Када је логика домена распршена по коду корисничког интерфејса, постоји велика вероватноћа да ће доћи до дуплирања кода. Мање дуплираног кода олакшава отклањање грешака, одржавање, побољшава и промовише поновно коришћење кода.
- У многим случајевима кориснички интерфејс се може променити независно од модела.
- Одвајање логике домена од корисничког интерфејса олакшава тестирање кода.

Презентер сарађује са моделом и корисничким интерфејсом путем интерфејса, као што је приказано на слици 2.10.

Презентер објекат или узима поглед и модел објекте као аргументе свог конструктора или инстанцира ове објекте у конструктору. *Presenter.Initialize()* креира поглед и даје му податке које је преузео из модела. Презентер тада приказује поглед кориснику. Презентер је претплаћен на догађаје које генеришу поглед и модел. По пријему ових догађаја, презентер предузима одговарајуће акције, као што су ажурирање делова погледа или ажурирање модела.



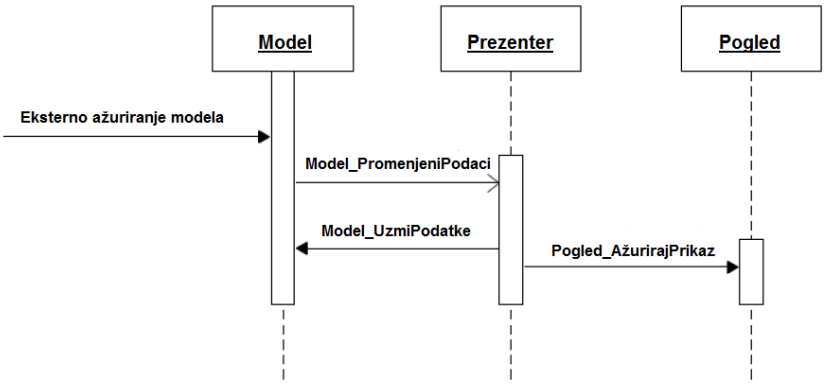
Слика 2.10: Коришћење интерфејса у апстракцији имплементације МПП обрасца



Слика 2.11: Секвенцијални дијаграм приказа у коме корисник мења поглед

На слици 2.11, приказана је секвенца догађаја у којој корисник уноси податке помоћу корисничког интерфејса. Будући да је презентер претплаћен на догађаје корисничког интерфејса и модела, након што примети догађај, презентер предузима одређену акцију, односно узима податке са корисничког интерфејса и прослеђује их моделу који те податке чува.

Слика 2.12 приказује нешто другачији сценарио, у коме модел, као резултат спољне операције (као што је промена у бази или промена модела иницијализована неком другом компонентом у систему), обавештава презентера да се десила промена. По примљеном обавештењу, презентер узима податке од модела и прослеђује их корисничком интерфејсу (поглед).



Слика 2.12: Секвенцијални дијаграм приказа измена погледа када је модел промењен екстерним утицајем

2.4.1 Предности и мане МПП обрасца

Предности:

- Разграничен је кôд корисничког интерфејса од кода пословне логике. Подаци су извучени у модел, а комплексна логика корисничког интерфејса налази се у презентеру.
- У погледу се налази само кôд за исцртавање корисничког интерфејса. Делови корисничког интерфејса (или цео кориснички интерфејс) се могу лако заменити.
- Олакшано је тестирање јер је довољно тестирати само презентер у којем се налази цела логика програма.

Мане:

- Дизајн није погодан за мале апликације.
- Повећана је комплексност.
- Отклањање грешака код модела са догађајима може да буде тешко.

Очигледан недостатак коришћења МПП обрасца је додатно кодирање укључено у одвајање модела, погледа и презентера. Међутим, за већину апликација, које укључују интеракцију корисника, време утрошено у имплементацију МПП обрасца није узалуд утрошено јер омогућава одвајање домена и апликативне логике од корисничког интерфејса [5].

2.5 Модел-поглед-поглед модел

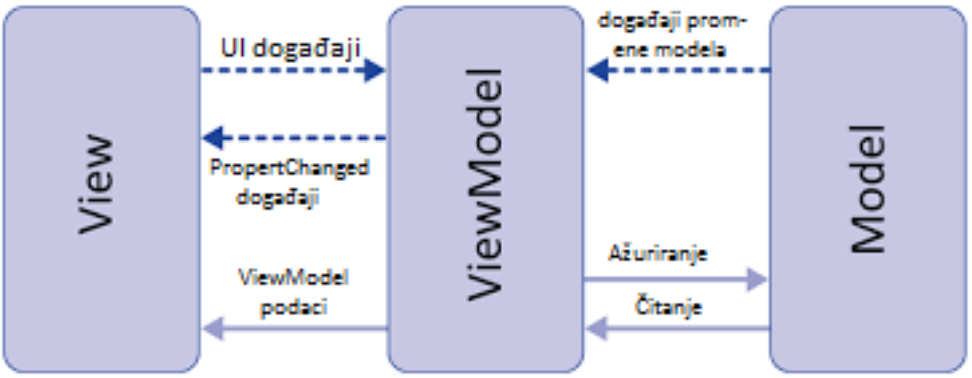
Модел-поглед-поглед модел (МППМ, енг. MVVM, Model-View-ViewModel) образац је најновија варијација МПК обрасца. Користи се у Microsoft технологијама. То је образац за раздвајање проблема у технологијама које користе везивање података (енг. Data Binding). Омогућена је независна имплементација корисничког интерфејса од пословне логике апликације. На тај начин, сваки модул развијеног софтвера може да има посебне верзије, независно од остатка софтвера.

Главне мотивације за имплементацију МППМ обрасца су следеће [6]:

- Обезбеђује одвајање проблема. Кôд који је уско везан, отпоран је на промене и изазива све врсте дугорочних проблема приликом одржавања, који на крају производе незадовољство купаца. Раздвајање између логике апликације и корисничког интерфејса чини апликацију лакшом за тестирање, одржавање и развијање. Ово даје могућност поновне употребе кода и могућност да програмер и дизајнер паралелно раде.
- То је природни образац за XAML платформе. Главне карактеристике МППМ обрасца су везивања података Silverlight платформе и зависна својства. Њихова комбинација обезбеђује средства за повезивање корисничког интерфејса са поглед моделом.
- Омогућава паралелан рад програмера и дизајнера. Када кориснички интерфејс није уско повезан с кодом, лакше је да дизајнери остваре слободу која им је потребна, да буду креативни и да направе добар производ.
- Олакшава тестирања апликације. Премештање логике корисничког интерфејса у посебну класу, која се може инстанцирати независно од технологије корисничког интерфејса, чини Unit тестирања много лакшим.

МППМ може бити коришћен на свим XAML платформама. МППМ омогућава чисто раздвајање проблема између контрола корисничких интерфејса и њихове логике. Састоји се од три елементарна дела: модела, погледа и поглед модела (енг. ViewModel). Слика 2.13 илуструје везе између ове три компоненте.

Компоненте су одвојене једне од других, што омогућава да се компоненте могу заменити - интерна имплементација се може заменити без утицаја на друге компоненте,



Слика 2.13: Модел-поглед-поглед модел

компоненте раде независно једна од друге, јединице тестирања су изоловане. Важно је разумети како компоненте комуницирају међу собом. На највишем нивоу, поглед "зна" о поглед моделу, и поглед модел "зна" о моделу, али модел "не зна" за поглед модел и поглед модел "не зна" за поглед. Поглед модел изолује поглед од класа модела и омогућава моделу да се развија независно од погледа.

Поглед је одговоран за дефинисање структуре, распоред и изглед онога што корисник види на екрану. У идеалном случају, поглед је дефинисан кодом који не садржи пословну логику.

У Windows мобилној апликацији, поглед је страна апликације. Као додатак, поглед може бити поткомпонента родитељског погледа, или *DataTemplate* за неки објекат у *ItemsControl-u*. Поглед може имати сопствени поглед модел, или може да наследи поглед модел свог родитеља. Поглед добија податке од свог поглед модела кроз везивање података, или позивајући методе поглед модела.

Модел у МППМ обрасцу је имплементација апликативног доменског модела који укључује модел података заједно с пословном и валидационом логиком. Примери објеката модела укључују: складишта, пословне објекте, објекте преноса података (енг. DTOs), *Plain Old CLR Objects (POCOs)*, генерисане ентитете и прокси објекте.

Поглед модел делује као посредник између погледа и модела и одговоран је за руковање логиком погледа. Типично, поглед модел комуницира са моделом позивајући методе класе модела. Он затим даје податке погледу, у облику у којем поглед може да их користи. Поглед модел такође садржи имплементације команди које корисник апликације покреће у погледу. На пример, када корисник кликне на дугме у оквиру корисничког интерфејса, та акција може покренути команду у поглед моделу.

Да би поглед модел учествовао у двоструком везивању података са погледом, промене својстава морају активирати *PropertyChanged* догађај. Поглед модел испуњава овај захтев тако што имплементира *INotifyPropertyChanged* интерфејс и активира *PropertyChanged* догађај када је својство промењено. Ослушкивачи могу адекватно одговорити на промене својстава, када се она појаве.

МППМ усклађује могућност повезивања података у *Silverlight-y*, за управљање везом између погледа и поглед модела, заједно са понашањима и окидачима догађаја. Ове могућности ограничавају потребу да се пословна логика постави у кôд погледа. Постоје многи приступи повезивања поглед модела са погледом, укључујући директне везе и приступе базиране на контејнерима.

2.5.1 Предности и мане МППМ обрасца

Предности:

- Током процеса развоја, програмери и дизајнери могу радити независно и конкурентно на својим компонентама. Дизајнери се могу сконцентрисати на поглед, а програмери на поглед модел и модел компоненте.
- Смањује количину кода који је везан за поглед (енг. code behind).
- Елементи корисничког интерфејса могу бити написани у XAML.
- Добро везивање података, што доводи до смањивања количине кода. Нема потребе за мануелним освежавањем погледа.

Мане:

- Везивање података је декларативно и теже за отклањање грешака.
- МППМ образац је претежак за једноставне операције корисничког интерфејса.
- Дизајн није погодан за мале апликације.

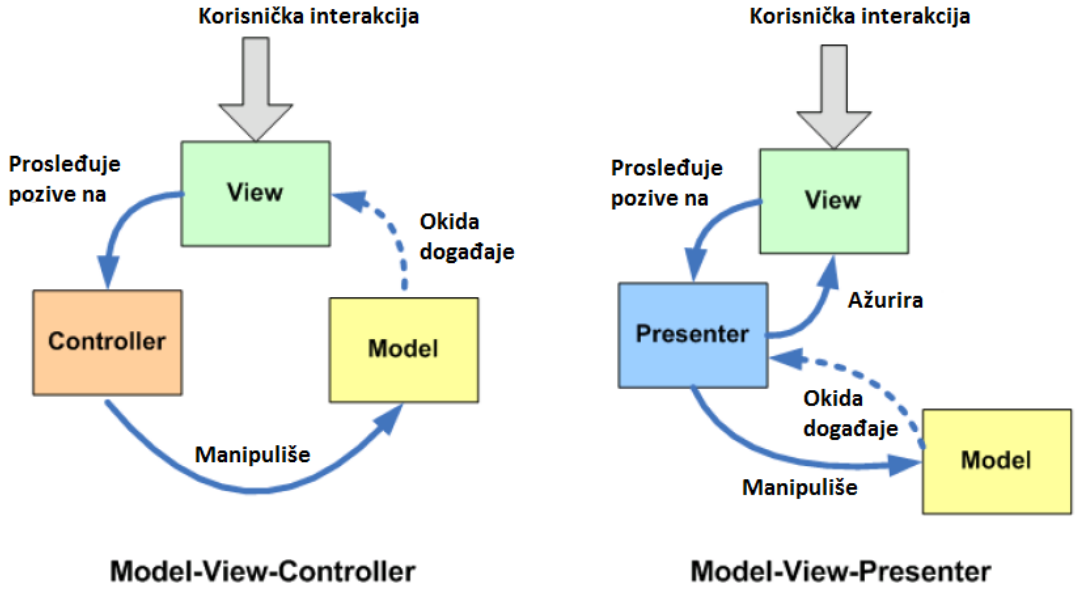
2.6 Сличности и разлике МПК, МПП и МППМ образаца

Понашања софтвера, која су заједничка за ове обрасце, су следећа:

- Слој података (Модел) садржи податке и пословну логику апликације, која се на њима примењује.
- Кориснички интерфејс (Поглед) је одговоран за визуелну презентацију апликације.
- Контролер, презентер и поглед модел садрже логику која имплеметира интеракцију између модела и погледа.

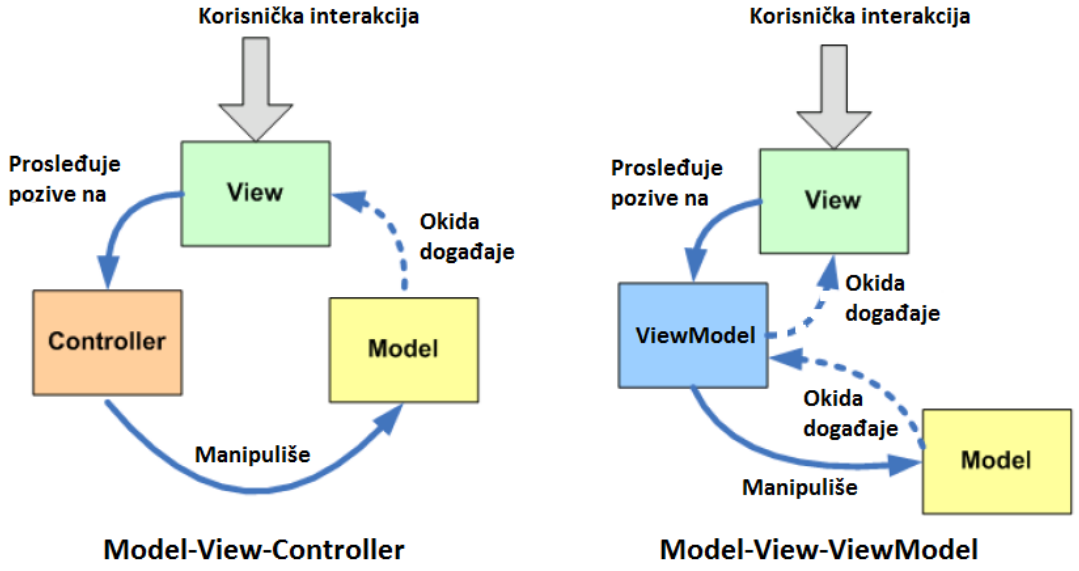
МПП образац представља измењену верзију МПК обрасца. Основна разлика између ова два обрасца је приказана на слици 2.14.

Основна разлика се огледа у томе, што је, у случају МПП обрасца, поглед потпуно контролисан помоћу презентера, док у случају МПК обрасца и контролер и модел могу променити поглед.



Слика 2.14: Сличности и разлике МПК и МПП обрасца

Код МПП обрасца поглед је комплетно пасиван. Све интеракције са моделом иду кроз презентер. Ово није случај код МПК обрасца, где поглед може директно да



Слика 2.15: Сличности и разлике МПК и МППМ обрасца

врши упите над моделом. МПК образац има лимитиране могућности за Unit тестирање, за разлику од МПП обрасца.

Основна разлика између МПК и МППМ образаца је у поглед моделу. Он је направљен да служи погледу и да му даје податке и команде које су му потребне.

Код МППМ обрасца постоји један поглед модел по погледу. Поглед модел је апстрактна репрезентација погледа и "не зна" за конкретан кориснички интерфејс, али комуницира са моделом.

Глава 3

ASP.NET MVC

ASP.NET (енг. Active Server Pages .NET) је веб технологија компаније Microsoft која омогућава креирање динамичких веб сајтова, веб апликација и веб сервиса. Први пут је објављена у јануару 2002. године са верзијом 1.0 .NET радног оквира, па је самим тим ASP.NET наследник ASP технологије. ASP.NET је изграђен на CLR-у (енг. Common Language Runtime), па тиме омогућава писање ASP.NET кода у било ком језику који подржава технологије .NET радног оквира. ASP.NET SOAP проширење омогућава ASP.NET компонентама да користе SOAP поруке.

ASP.NET MVC је радни оквир веб апликација који је развио Microsoft, а који имплементира МПК (енг. MVC) архитектурни шаблон. То је софтвер отвореног кода. Овде је МПК образац реализован на следећи начин:

- **Модели** су класе које представљају домен апликације. Ови доменски објекти често енкапсулирају податке сачуване у бази података као и кôд који се користи за манипулацију подацима и спроводи доменски специфичну пословну логику. Код ASP.NET MVC, модел је слој приступа подацима (енг. Data Access Layer). Овде се често користе алати, као што су Entity Framework или NHibernate, у комбинацији са кодом који садржи доменску логику.
- **Поглед** је шаблон за динамичко генерисање HTML-а.
- **Контролер** је специјална класа која управља везама између погледа и модела. Контролер одговара на кориснички улаз, комуницира са моделом и одлучује који ће поглед приказати. У ASP.NET MVC класа контролера садржи суфикс Controller.

3.1 Доменски модел

Најважнији део ASP.NET MVC апликације је доменски модел. Домен представљају ентитети из реалног света, као и операције и правила које апликација треба да подржава. Први корак, приликом развоја апликације, је идентификовање њеног домена.

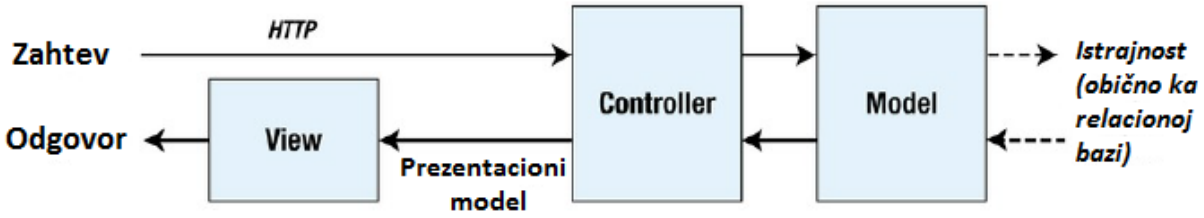
Даље се креира софтверска репрезентација домена - доменски модел. Доменски модел је скуп C# типова (класа, структура, итд.) колективно познатих као доменски типови. Операције из домена су представљене преко метода дефинисаних у доменским типовима и доменске улоге су изражене у логици унутар ових метода или применом C# атрибута на методе. Када се креира примерак доменског типа за приказ посебног дела података, тада се креира доменски објекат. Доменски модели су обично дуготрајно перзистентни - постоји много различитих начина да се ово постигне, али релационе базе података остају најчешћи избор. Кратко, доменски модел је једна ауторативна дефиниција пословних података и процеса у апликацији.

Приступ доменским моделима омогућава да се реше многи проблеми који настају у паметним обрасцима корисничког интерфејса. Пословна логика је садржана на једном месту. Ако је потребно да се манипулише подацима у моделу, или да се дода нови процес или правило, доменски модел је једини део апликације који мора бити промењен.

Уобичајени начин да се доменски модел одвоји од остатка ASP.NET MVC апликације је постављање модела у одвојен C# склоп (енг. assembly). На овај начин, могу се креирати референце на доменски модел из других делова апликације, али се обезбеђује да референце у другом правцу не постоје.

3.2 ASP.NET имплементација МПК обрасца

У ASP.NET MVC-у контролери су C# класе, обично наслеђене из System.Web.Mvc. Controller класе. Сваки јавни метод у класи, изведеној из класе Controller, је назван *action method*, који је повезан са конфигурабилним URL-ом кроз ASP.NET систем рутирања. Када је захтев послат URL-у повезаним са неким методом акције, наредбе у класи контролера се реализују таквим редом да изврше неке операције над доменским моделом и онда изаберу поглед за приказ клијенту.



Слика 3.1: Интеракције у ASP.NET MVC апликацији

ASP.NET MVC радни оквир пружа подршку избора механизма погледа [7]. Раније верзије су користиле ASP.NET механизам погледа који обрађује ASPX странице. ASP.NET MVC 3 је увео Разор[1] механизам погледа који је предефинисан у ASP.NET MVC 4 тако да од те верзије користи потпуно различиту синтаксу. Visual Studio обезбеђује IntelliSense подршку за оба механизма погледа.

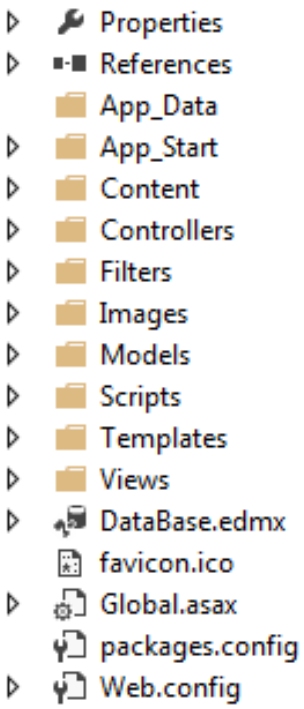
У ASP.NET MVC не постоје ограничења за имплементацију доменског модела. Могуће је креирати модел коришћењем регуларних C# објеката и имплементирати перзистентност с било којом базом података, релационим радним оквирима за пресликавање објеката или других алата за податке које подржава .NET. Visual Studio креира /Model фолдер као део МПК шаблона. Ово је погодно за једноставне пројекте, али комплексније апликације чешће дефинишу доменске моделе у одвојеном пројекту у оквиру Visual Studio-a.

Приликом креирања ASP.NET MVC апликације помоћу Visual Studio-a, аутоматски се креира неколико фајлова и директоријума у пројекту [8]. Ово је приказано на слици 3.2.

- /Controllers садржи Controller класе које управљају URL захтевима
- /Models садржи класе које представљају и управљају подацима и пословним објектима
- /Views садржи фајлове који представљају шаблоне корисничког интерфејса који су одговорни за приказивање излаза, као што је HTML
- /Scripts садржи фајлове JavaScript библиотека и скриптове (.js)
- /Images садржи слике које се користе у апликацији

[1]Разор (енг. Razor) је ASP.NET механизам који се користи за креирање динамичких веб страница помоћу C# или Visual Basic програмским језиком. Разор омогућава интегрисање C# или Visual Basica са HTMЛ-ом. Разор је развијен у јуну 2010. године уз верзију Microsoft Visual Studio 2010, а појавио се 2011.

- /Content садржи CSS и друге садржаје који нису скрипте и слике
- /Filters садржи кôд за филтере.
- /App_Data служи за чување фајлова са подацима који се читају или уписују
- /App_Start садржи конфигурациони кôд за функционалности као што су рутирање, увезивање (енг. Bundling), и Web API



Слика 3.2: Структура ASP.NET MVC апликације

ASP.NET MVC апликације се подразумевано ослањају на конвенције. Ово омогућава да се умањи или чак потпуно избегне конфигурисање и да се одреде ствари које се могу закључити на основу конвенције.

На пример, MVC користи именовање директоријума на основу конвенције, која омогућава изостављање локације из путање која референцира поглед из класе контролера. Подразумевано, ASP.NET MVC тражи поглед у оквиру \Views\ControllerName директоријума апликације.

Концепт "*конвенција пре конфигурације*" је постао популаран са Ruby on Rails и значи: пожељно је искористити то искуство у радном оквиру, да би се избегла конфигурација.

Концепт "*конвенција пре конфигурације*" функционише на примеру три основна директоријума који чине да апликација ради: Controllers, Models и Views. У ASP.NET MVC конвенције су поприлично једноставне:

- Сваки назив класе контролера се завршава са Controller: ProductController, HomeController, итд. и оне се налазе у Controllers директоријуму.
- Постоји један Views директоријум за све погледе у апликацији.
- Погледи, које контролери користе, налазе се у поддиректоријуму главног директоријума Views и именовани су према називу контролера (минус Controller суфикс). На пример, погледи за ProductController се налазе у /Views/Product.

Сви заједнички елементи корисничког интерфејса налазе се у Shared директоријуму у оквиру Views директоријума.

Са пројетком креираним у Visual Studio-у добија се и прилагодљив дизајн веб страница.

3.3 Прилагодљив дизајн веб стране

Прилагодљив дизајн веб стране (енг. Responsive web design, RWD) представља веб дизајн који омогућава оптимизовано приказивање, једноставно коришћење, читање и кретање кроз веб сајт који ће бити прегледан на различитим уређајима. Заправо, то је техника која омогућава да се једна веб страница може прегледати на уређајима различитих резолуција. Прегледање веб странице, која нема прилагодљиве карактеристике, може бити неугодно на уређајима са мањом резолуцијом. Да би корисник дошао до тражених информација, део времена изгубиће крећући се кроз саму страну (скроловање, зумирање, померање). При креирању прилагодљивих веб страна, садржај се прилагођава различитим ширинама уређаја коришћењем течних (енг. fluid) мрежа, флексибилних слика као и CSS3 упитима о медији [2] (енг. media queries) и на тај начин пружа корисницима једноставније коришћење.

Етан Маркоте (енг. Ethan Marcotte) сковао је термин "прилагодљив веб дизајн", маја 2010. године, у чланку "A List Apart". У својој краткој књизи, "Responsive Web

[2] Технологија која омогућава да се на различитим ширинама уређаја користе различита CSS својства

Design", описао је како теоретску основу, тако и практичну примену ове технике у креирању веб страна (2011. године) [9].

Mashable је 2013. годину, назвао годином прилагодљивог веб дизајна. Треба истаћи, да многи извори препоручују прилагодљив дизајн веб страна, као исплативу алтернативу за креирање мобилних апликација.

С обзиром на све већу употребу мобилних уређаја и таблет рачунара широм света, техника креирања прилагодљивих веб страна је неизбежна уколико је садржај стране доступан свима.

Технике које се користе приликом креирања прилагодљивих веб страна су [10] :

- Течне мреже - концепт течних мрежа заправо говори да би требало користити релативне јединице (нпр. ширина параграфа је изражена у процентима) приликом креирања променљиве веб стране.
- Флексибилне слике користе релативне јединице (висина и ширина слике задају се у процентима) да би се избегло приказивање слике ван садржаја стране.
- CSS упити о медији - дозвољавају страни да користи различите CSS стилове на различитим димензијама уређаја, на којима се садржај приказује. На тај начин, креатор сајта може прилагодити величину слова, маргине итд. разним уређајима, како би страна била оптимизована за коришћење.

Глава 4

JavaScript библиотеке клијентске стране

Клијентска страна се односи на операције које врши клијент у оквиру рачунарске мреже. Клијент је у овом случају прегледач преко ког корисник шаље захтеве ка серверској страни. Прегледач се покреће на локалном рачунару или радној станици клијента и по потреби се повезује на сервер. Операције се могу извршавати на клијентској страни, јер захтевају приступ информацијама или функционалностима које су доступне клијенту. Корисник може само да посматра информације или да унесе податке, док серверска страна није у могућности да одговори на све клијентске захтеве благовремено. Осим тога, ако се операције могу изводити на клијентској страни, без слања података преко мреже, могу одузети мање времена, користити мање протока и сносити мањи безбедносни ризик.

4.1 AngularJS

AngularJS је радни оквир за развој веб апликација отвореног кода, који одржава Google и заједница појединаца и корпорација који решавају многе изазове у развоју SPA[1] апликација [11]. Његов циљ је поједностављивање развоја и тестирање таквих апликација, пружањем оквира за клијентску страну модел-поглед-контролера (МПК) и модел-поглед-поглед-модел (МППМ) архитектура.

[1]SPA (Single Page Applications) - је веб апликација која сву навигацију врши у једној страни.

AngularJS библиотека ради тако што прво чита HTML страницу у коју су уграђени додатни атрибути. Angular интерпретира ове атрибуте као директиве за везивање делова улаза или излаза странице за модел, који је представљен стандардним JavaScript променљивама. Вредности ових променљивих могу се подешавати унутар кода, или преузети од статичких или динамичких JSON[2] ресурса.

AngularJS је првобитно развио 2009. године Misko Hevery-a у Brat Tech LLC-у као софтвер који стоји иза JSON сервиса за складиштење, ради лакшег креирања апликација за пословање. Овај подухват је лоциран на веб домену "GetAngular.com" и имао је неколико претплатника, пре него што је одлучено да се пословна идеја напусти и Angular изда као библиотека отвореног кода.

Веб апликације са вишеструким HTML страницама добијају податке на серверу и те податке придружују HTML-у, а затим такве HTML странице испоручују претраживачу. Већина SPA апликација ради ово исто у извесној мери. Angular се разликује у томе што се шаблон и подаци шаљу претраживачу, па се у оквиру претраживача подаци придружују HTML шаблонима. У овом случају улога сервера је да служи као статички ресурс за шаблоне и да на одговарајући начин ове шаблоне услужи подацима.

Пример „Здраво свете” у AngularJS-у.

```
<html ng-app>
<head>
    <script src="angular.js"></script>
    <script src="controllers.js"></script>
</head>
<body>
    <div ng-controller='ZdravoController'>
        <p>{{greeting.text}} svete</p>
    </div>
</body>
</html>
```

Скрипта 4.1: hello.html

[2]JSON (Java Script Object Notation) - отворени стандардни формат који користи лако читљив текст за слање објеката који се састоје од парова кључ-вредност.

```
function ZdravoController($scope) {
    $scope.greeting = { text: 'Zdravo' };
}
```

Скрипта 4.2: Контролер

У примеру се види како изгледа састављање података и шаблона на претраживачу. За то је креиран шаблон у hello.html. Очитавањем hello.html-а у било ком претраживачу даје испис Zdravo svete. Постоји неколико занимљивих ствари, које се овде могу приметити, у поређењу са методама које су данас у широкој употреби:

- Не постоје класе или идентификатори у HTML-у, да идентификује где треба поставити ослушкиваче догађаја.
- Када *ZdravoController* постави *greeting.text* на Zdravo, није потребно регистровати ниједан ослушкивач догађаја, нити повратну методу.
- Класа *ZdravoController* је обична JavaScript класа која није наслеђена.
- *ZdravoController* узима $scope[3] објекат, који је потребан, а који пре тога није потребно креирати.
- Није потребан позив конструктора *ZdravoController-a*.

У Angular апликацијама МПК образац представљају:

- модел - подаци смештени као својства објеката;
- поглед - DOM (Document Object Model) дрво[4];
- контролер[5] - JavaScript класе.

МПК образац је овде прикладан из неколико разлога. Прво, МПК образац одређује како структура апликације треба да изгледа. Други људи, који раде на пројекту, могу одмах да разумеју шта је написано, када знају да се користи МПК структура за организацију кода. Можда и најважније, доноси велике користи приликом креирања апликације која је лакша за надоградњу, одржавање и тестирање.

[3] Angular 2 не садржи $scope објекат, већ користи zone.js за праћење промена. Angular 2 користи Typescript

[4] у Angular 2 се уместо DOM дрвета могу користити и друга дрвета приказа

[5] контролер не постоји у Angular 2, уместо контролера се користе "компоненте" (компонента = директива + контролер)

4.1.1 Везивање података (енг. Data binding)

Пре него што су SPA апликације постале уобичајне, платформе као што су Rails, PHP или JSP помогле су креирање корисничког интерфејса, спајањем HTML ниски са подацима, пре него што се подаци пошаљу кориснику на приказ. Библиотеке, као што су JQuery, надограђују овај модел клијенту и допуштају праћење сличног стила, али са могућношћу ажурирања делова DOM-а, уместо ажурирања целе странице. Овде се спаја HTML шаблон са подацима, а затим се добијени резултат убацује у DOM, тако што се поставља особина innerHtml датог елемента.

Све ово ради веома добро, али када се убацују нови подаци у кориснички интерфејс или се подаци мењају у зависности од корисничког улаза, потребно је урадити доста нетривијалног посла и у корисничком интерфејсу и у JavaScript својствима, да би подаци били коректни. Везивање података је процес којим се остварује веза између елемената корисничког интерфејса и JavaScript својстава, тј. модела. Везивање података, код МПК обрасца, елиминише кôд при писању погледа и модела. Кретање података од погледа до модела врши се аутоматски [12].

Модификујмо први пример да буде динамичнији. До сада је *ZdravoController* постављао модел *greeting.text* и никада га није мењао. Пример је модификован тако што је додат текстуални улаз, који може да мења вредност *greeting.text-a*, у зависности од корисниковог уноса.

```
<html ng-app>
<head>
    <script src="angular.js"></script>
    <script src="controllers.js"></script>
</head>
<body>
    <div ng-controller='ZdravoController'>
        <input ng-model='greeting.text'>
        <p>{{greeting.text}} svete</p>
    </div>
</body>
</html>
```

Скрипта 4.3: hello.html са тексуалним улазом

Ако се Zdravo замени са Pozdrav, добија се порука Pozdrav svete. Без регистровања ослушкивача догађаја на улазном пољу, кориснички интерфејс се аутоматски ажурира. Исто би важило и за промене које долазе од и до сервера. У контролеру може постојати захтев ка серверу, који враћа одговор и сетује *greeting.text* на ту вредност. Angular тада аутоматски ажурира улаз и текст у витичастим заградама.

4.1.2 Убризгавање зависности (енг. Dependency Injection)

Објекат $scope, који обавља везивање података, прослеђен је аутоматски; није потребно креирање позивом функције. Позван је тако што се налази у конструктору *ZdravoController-a*. Ако треба везати податке за URL локацију у корисничком прегледачу, може се тражити објекат који овим управља, тако што се постави $location у конструктору:

```
function ZdravoController($scope, $location) {
    $scope.greeting = { text: 'Zdravo' };
}
```

Скрипта 4.4: Убризгавање зависности конструктора

Убризгавање зависности[6] омогућава праћење стила развоја у којем уместо креирања зависности, класе само траже шта им треба. Ово прати дизајн образац Law of Demeter, који је још познат као принцип најмањег знања. Како је посао *ZdravoController-a* да подеси иницијално стање за greeting модел, контролер не треба да брине како се $scope објекат креира или како да га пронађе. Ова одлика се не односи само на објекте креиране Angular радним оквиром, већ и на остале објекте.

4.1.3 Директиве (енг. Directives)

Један од најбољих делова Angular-a, је то, да се могу написати HTML шаблони. Ово се може урадити, јер је у језгро оквира укључен моћан DOM трансформацијски механизам (енг. engine), који дозвољава надоградњу синтаксе HTML-a. У шаблонима из примера, постоји неколико нових атрибута који нису део HTML спецификације. Примери укључују двоструко-витичасту нотацију за везивање података, *ng-controller*

[6] Angular 2 користи хијерархијски систем убризгавање зависности

за одређивање који контролер надгледа који део погледа, и *ng-model*, који везује улаз за део модела. Ове HTML екстензије се зову директиве.

Angular долази са многим директивама које су од користи приликом креирања погледа апликације [13]:

- *ng-app*: указује Angular-у о ком делу DOM дрвета треба да води рачуна; када се страна учита, Angular обилази DOM дрво почевши од чвора који садржи ову директиву и надаље придружује функционалности свим чворовима за које пронађе **ng** директиве унутар њега
- *ng-controller*: JavaScript функције задужене за поједине делове DOM дрвета
- *ng-model*[7]: омогућава везивање између шаблона и података
- *ng-repeat*[8]: омогућава понављање делова DOM дрвета
- *ng-init*: омогућава иницијализацију
- *ng-class, ng-class-even, ng-class-odd, ng-show*: неке од директива за контролу CSS својства
- *ng-click, ng-mouseup, ng-change, ng-focus, ng-blur, ng-keyup*: својства JavaScript догађаја
- *ng-required, ng-change, ng-maxlength, ng-form, ng-submit*: неке од директива које се користе у раду са формама

Ове директиве могу да дефинишу шаблоне. Такође, директиве могу да конфигуришу апликацију или се могу користи за креирање компоненти које ће се поновно користити. Могу се писати сопствене директиве како би се унапредиле могућности шаблона HTML-а.

4.1.4 Фабрика (енг. Factory)

У скрипти 4.5 је приказана фабрика. Њена улога у овом примеру је извршавање GET захтева ка серверу.

[7] у Angular 2 *ng-model* је замењен са [(*ngModel*)]. Локалне промењљиве су дефинисане коришћењем # префикса

[8] у Angular 2 *ng-repeat* је замењен са **ngFor* директивом

```
var zdravoSveteServis = angular.module('zdravoSveteServis', ['ngResource
    ']);

zdravoSveteServis.factory('uzmiZdravoSveteFabriku', ['$resource',
    function ($resource) {
        return $resource('http://localhost:8090/zdravosvete', {}, {
            result: {method: 'GET', isArray: true}
        });
    }]);
```

Скрипта 4.5: Фабрика

У овом примеру контролер се везује за фабрику која је описана у 4.6. У телу контролера може се наћи логика која се користи за извршавање серверских захтева, преко $http-а. Такође, врши се уписивање резултата у *$scope.results* који се у HTML-у везују и приказују.

```
var zdravoSveteKontroler = angular.module('zdravoSveteKontroler', []);

zdravoSveteAplikacija.controller('zdravoSveteKontroler', ['$scope', '
    uzmiZdravoSveteFabriku',
    function ($scope, uzmiZdravoSveteFabriku) {
        $scope.results = uzmiZdravoSveteFabriku.result();
    }]);
```

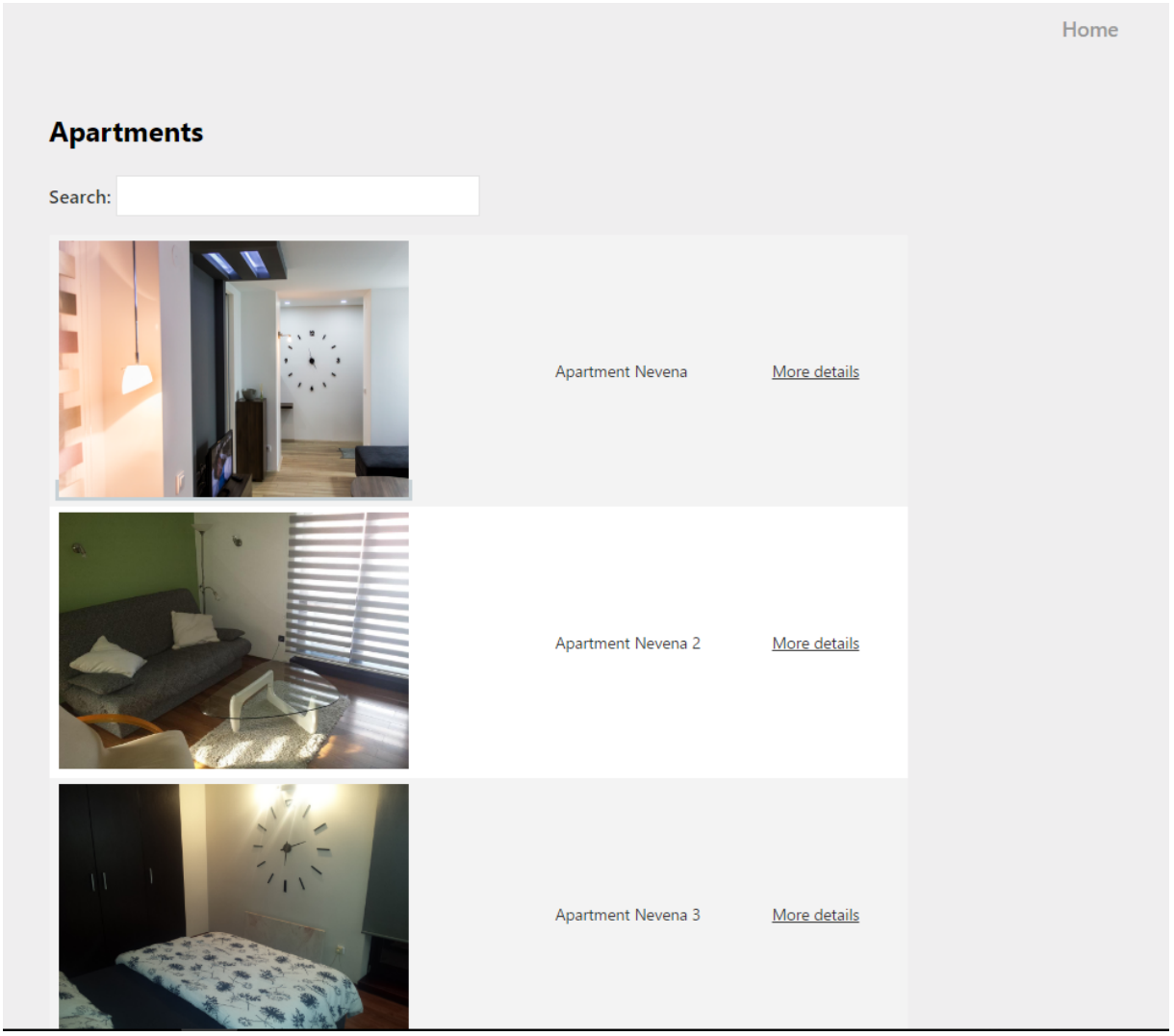
Скрипта 4.6: Захтев ка серверу у контролеру

4.1.5 Пример 1

Уз рад је развијена апликација за преглед апартмана за издавање на дан. Њен примарни циљ је представљање апартмана са описима кориснику. По покретању апликације, приказују се сви апартмани (слика 4.1). Кориснику је омогућена њихова претрага по називу. Кликом на детаље, приказује се детаљан опис апартмана. У опису су приказани: адреса, град, краћи опис и цена. Поред наведених информација, приказује се и галерија слика.

Подаци су смештени у бази података, при чему је коришћен Microsoft SQL Server [9]. База података садржи две табеле: Apartment и ApartmentImages. Табела Apartment

[9] Microsoft SQL Server је релациони систем за управљање базама података који је развио Microsoft



Слика 4.1: Изглед апликације по покретању

садржи описе апартмана, док се у табели ApartmentImages налазе само подаци о сликама које су за њега везане. За комуникацију између апликације и базе података коришћен је ASP.NET Web API 2, који је идеалан за креирање RESTful [10](енг. REST, Representational State Transfer) сервиса у .NET радном оквиру.

```
var app = angular.module('app', ['ngRoute']).config(function (
$routeProvider, $locationProvider) {

    $routeProvider
        .when('/', {
            redirectTo: function () {
```

[10]Преношење репрезентације ресурса (енг. REST) је софтверски архитектурални стил који се састоји од смерница и најбољих пракси за израду скалабилних веб сервиса. REST је координисан скуп ограничења примењен на дизајн компоненти система за дистрибуирање хипермедија који може да доведе до бољих перформанси и одрживе архитектуре.

```
                return '/Pocetna';
            }
        })
        .when('/Pocetna', {
            templateUrl: '/Templates/Pocetna.html',
            controller: 'PocetniController'
        })
        .when('/Detalji/:id', {
            templateUrl: '/Templates/Detalji.html',
            controller: 'DetaljiController'
        })
        .otherwise({
            templateUrl: '/Template/Greska.html',
            controller: 'GreskaController'
        })

    $locationProvider.html5Mode(false).hashPrefix('!');
});
```

Скрипта 4.7: app.js

Апликација комуницира са сервисом помоћу AJAX (енг. Asynchronous JavaScript And XML) позива. Састоји се од главног модула, три радна модула и помоћних сервиса (скрипта 4.7). Први модул приказује почетну страницу апликације, на којој су приказани сви апартмани. Кликом на детаље апартмана, апликација води корисника на други радни модул који приказује више инфорамција о апартману. Трећи радни модул приказује поруку о грешци, уколико до ње дође.

Неопходно је навести да главни модул зависи од ngRoute модула, који је дефинисан у Scripts/angular-route.js библиотеци и пружа сервисе за управљање рутама. Један од њих је *$routeProvider* сервис, који служи за конфигурисање рута. Помоћу њега, у config блоку су регистроване три руте:

- /Pocetna рута одговара #/Pocetna делу URL-а, поглед се налази у Templates/Pocetna.html директоријуму, а контролер који се користи је *PocetniController*
- друга рута садржи id и одговара #/Detalji/2 делу URL-а: одговарајући поглед се налази у Templates/Detalji.html, а контролер је *DetaljiController*
- *otherwise* дефинише како се апликација понаша ако се у URL-у појави рута која није регистрована.

```
<div ng-controller="PocetniController">

    <label>Search: <input ng-model="searchText"></label>

    <table class="tableData" width="80%" border="0" cellpadding="0"
   cellspacing="0">
        <tr ng-repeat="e in Apartments | filter:searchText" ng-class-odd
   ="'odd'" ng-class-even="'even'">

            <td ng-if="e.ImageName !== null">
                <a ng-href="{{'#!/Detalji/' + e.ID}}">
                    <img ng-src="{{'../Images/' + e.Name + '/' + e.
   ImageName}}" width="300" height="220" />
                </a>
            </td>
            <td ng-if="e.ImageName === null">-</td>

            <td>{{e.Name}}</td>
            <td>
                <a ng-href="{{'#!/Detalji/' + e.ID}}">More details</a>
            </td>
        </tr>
    </table>
</div>
```

Скрипта 4.8: Pocetna.html

Како је неопходан за достављање података, у склопу контролера је дефинисан сервис *ContactService* који представља модел података.

```
angular.module('app').controller('PocetniController', function ($scope,
   ContactService) {
    $scope.Apartments = null;
    ContactService.GetApartments().then(function (d) {
        $scope.Apartments = d.data;
    }, function () {
        alert('Failed');
    })
})
.factory('ContactService', function ($http) {
    var fac = {};
    fac.GetApartments = function () {
```

```
        return $http.get('http://localhost:8090/api/values');
    }
    return fac;
})
```

Скрипта 4.9: AngularController/PocetniController.js

4.2 KnockoutJS

KnockoutJS је JavaScript библиотека отвореног кода, која помаже развој динамичних и богатих веб апликација. Направљена је по модел-поглед-поглед модел (МППМ) обрасцу. Knockout чини веома једноставним имплементирање комплексног корисничког интерфејса, који одговара интеракцијама корисника.

KnockoutJS је, као пројекат отвореног кода, развио и одржавао Стив Сандерсон (енг. Steve Sanderson), запослен у Microsoft-у, почев од петог јула 2010. године [14].

Имплементација Knockout-а укључује три различите ствари:

- креирање погледа са HTML и CSS елементима који везују податке за њега,
- креирање поглед модела са подацима који се везују за поглед,
- обавештавање Knockout-а да обави везивање података погледа са поглед моделом.

Главне карактеристике KnockoutJS су [15]:

- Елегантно праћење зависности - аутоматски ажурира делове корисничког интерфејса када се модел података промени.
- Декларативно везивање података - једноставан и очигледан начин повезивања делова корисничког интерфејса са моделом података.
- Једноставно проширив - имплементира функционалности (помоћу везивања података), које се лако поново употребљавају.

Додатне бенефиције:

- Чиста JavaScript библиотека - ради са било којим сервером или технологијом на страни клијента.
- Може се додати на врх постојеће веб апликације без већих измена у архитектури.
- Ради на било којем од најчешће коришћених интернет прегледача (IE 6+, Firefox 2+, Chrome, Safari, Edge, и други).

4.2.1 Везивање података

Везивање података се постиже додавањем HTML атрибута, који се зове data-bind, у било који HTML елемент, који Knockout треба да замени информацијом из поглед модела.

```
<!DOCTYPE html>
<html>
<head>
    <title>Vezivanje podataka u KnockoutJS-u</title>
</head>
<body>
    <h1>Zdravo, <span data-bind="text: name"></span></h1>
    <script type='text/javascript' src='js/knockout-3.2.0.js'></script>
    <script>
        var viewModel = function() {
            this.name = 'Andrej Panic';
        };
        ko.applyBindings(viewModel);
    </script>
</body>
</html>
```

Скрипта 4.10: Везивање података у KnockoutJS-у

Када се овај програм изврши, у оквиру интернет прегледача се прикаже Zdravo, Andrej Panic унутар h1 елемента. Span елемент је везан податком name који представља својство поглед модела.

Као што је већ истакнуто, имплементирање KnockoutJS захтева три одвојене ствари. Прва ствар је поглед и у овом примеру то је HTML који садржи h1 и span елементе за идентификовање вазивања података. Друга ствар је поглед модел. У примеру је то JavaScript променљива/функција која се зове *viewModel* и која садржи променљиву name. Трећа ствар је позив функције ko.applyBindings у поглед моделу. Она обавештава Knockout да обави везивање података погледа са поглед моделом. Када се ова функција изврши, Knockout обрађује поглед и поглед модел. Сва везивања података у погледу су извршена и динамички замењена подацима из поглед модела.

МППМ је дизајниран да имплементира везивање података између поглед модела и погледа. То KnockoutJS ради веома добро. Ово је постигнуто коришћењем HTML атрибута (који се лако имплементирају) и JavaScript поглед модела.

```
var mySecondViewModel = function() {
    var self = this;
    self.name = 'Andrej Panic';

    self.getName = function() {
        return self.name;
    };
};
```

Скрипта 4.11: Објектно оријентисани поглед модел

У скрипти 4.11, поглед модел је JavaScript функција која се понаша као класа у објектно-оријентисаном програмирању. Променљива *name* је енкапсулирана у класи и понаша се као њено својство (енг. property). Функција *getName* даје приступ *name* својству, без директног позива својства класе.

```
function ViewModel(name) {
    var self = this;
    self.name = name;
    self.getName = function() {
        return self.name;
    };
};
var myThirdViewModel = new ViewModel('Andrej Panic');
```

Скрипта 4.12: Поглед модел са параметром

Пример 4.12 је другачији, јер је функција (понаша се као класа) са називом *ViewModel* креирана да прихвати *name* својство. Ова класа се онда инстацира прослеђивањем

имена конструктору. Knockout може везати скоро сваки HTML атрибут, CSS класу или CSS стил [16].

```
<p data-bind="
style: { marginBottom: 0, paddingBottom: '1em' },
css: 'myClass'">
This text has custom styles and a CSS class.</p>
```

Скрипта 4.13: Пример CSS и стил везивање података

4.2.2 Контексти везивања података

Knockout одржава родитељ/дете хијерархију контекста. Када се приступа својству, да би се податак везао, све је у вези са контекстом у којем се налази. Корен (енг. root) контекст се односи на поглед модел који је прослеђен функцији *ko.applyBindings*.

Knockout нуди неколико корисних променљивих које омогућавају промену контекста, у родитељском или чак корен контексту:

- *$root* - приступа корен контексту из контекста детета. Ово је згодно када се не зна колико родитељ/дете контекста постоји изнад тренутног.
- *$parent* - приступа контексту директног родитеља, када је тренутни контекст детета
- *$parents* - сличан *$parent* променљивој. Садржи низ родитељских контекста до тренутног контекста (*$parents[0]* је исто што *$parent*).
- *$data* - омогућава приступ текућем објекту у којем је тренутни контекст. Ово је веома корисно када је тренутни контекст у којем је променљива и не постоје никаква својства.
- *$index* - на располагању је само у foreach везивању података и садржи цео број који представља тренутну позицију у петљи, са почетком у нули и крајем у length-1.

4.2.3 Динамички промењива својства

Knockout назива ова својства посматраним (енг. observables). Када се дефинише промењива или својство као посматрано, Knockout прати њене промене. Постоје три различите врсте посматраних елемената које се најчешће користе. Прва врста посматраних елемената, приказана у скрипти 4.14, је посматрана промењива.

```
var myObservable = ko.observable();
myObservable('Zdravo');
alert(myObservable());
```

Скрипта 4.14: Посматрана промењива

Да би се направила посматраном, потребно је доделити *ko.observable* функцију промењивој. Knockout претвара промењиву у посматрану промењиву и прати када се вредност промени, како би обавестио елемент корисничког интерфејса који је с њом повезан.

Друга врста је посматрани низ. У скрипти 4.15, инстациран је празан низ, прослеђивањем две угласте заграде конструктору. Као и код промењивих, када су елементи додати или обрисани из низа, Knockout обавештава елементе који су претплаћени за обавештења. Посматрани низ је добар за коришћење с табелама, где се елементи динамички додају и бришу.

```
var myObservableArray = ko.observableArray([]);
myObservableArray.push('Zdravo');
```

Скрипта 4.15: Посматрани низ

Последњи тип је израчуната посматрана (енг. computed observable). Овај тип се користи за комбиновање једне или више посматраних у један објекат.

```
self.firstName = ko.observable('Andrej');
self.lastName = ko.observable('Panic');
self.fullName = ko.computed(function() {
    return 'Zdravo ' + self.firstName() + ' ' + self.lastName();
});
```

Скрипта 4.16: Израчуната посматрана

Када се поглед модел веже за Knockout, израчуната функција се извршава. За сваку посматрану, која се користи у тој функцији, поглед модел претплати било који догађај на те променљиве. Када се промени, Knockout зна да израчуната променљива треба да се ажурира.

4.2.4 Пример 2

У овом примеру, описана је употреба Knockout библиотеке у развоју апликације за преглед апартмана, која је раније детаљно описана.

```
function Apartment(id, Name, ImageName) {
    var self = this;
    self.Id = ko.observable(id);
    self.Name = ko.observable(Name);
    self.ImageName = ko.observable(ImageName);
    self.src = ko.observable("../Images/" + Name + "/" + ImageName);
    self.url = ko.observable("#/" + id),
    self.addApartment = function () {
        var dataObject = ko.toJSON(this);
        $.ajax({
            url: 'http://localhost:8090/api/values',
            type: 'post',
            data: dataObject,
            contentType: 'application/json',
            success: function (data) {
                apartmentRegisterViewModel.apartmentListViewModel.
   apartments.push(new Apartment(data.ID, data.Name));
                self.Id(null);
                self.Name('');
            }
        });
    };
}
```

Скрипта 4.17: Поглед модел апартмана

За рутирање је коришћена Sammy.js библиотека, јер Knockout нема свој систем за рутирање. У скрипти 4.17 приказан је поглед модел апартмана са дефинисаним променљивама, које се приказују у листи апартмана на почетној страници. Дефинисане су посматране променљиве које чувају податке о називу и идентификатору апартмана, као и назив главне слике апартмана и путање до директоријума у којем се

она налази. Посматране промењиве добијају вредности, након што сервис врати податке у оквиру AJAX позива.

```
<div data-bind="with: apartmentListViewModel">
    <div data-bind="visible: showSearch">
        <h2>Apartments</h2>
        <br />
        <label>Search: <input type="text" id="search" data-bind="value:
   Query, valueUpdate: 'keyup' "></label>
    </div>
    <br />
    <table class="tableData apartments" border="0" cellpadding="0"
   cellspacing="0">
        <tbody data-bind="foreach: searchResults, visible:
   showApartments">
            <tr data-bind="click: $parent.goToApartment">
                <td data-bind="if: ImageName">
                    <a data-bind="attr: { href: url}, click: $parent.
   goToApartment">
                        <img data-bind="attr: { src: src}" width="300"
   height="220" />
                    </a>
                </td>
                <td data-bind="ifnot: ImageName">-</td>
                <td data-bind="text: Name"></td>
                <td>
                    <a data-bind="attr: { href: url }">
                        More details
                    </a>
                </td>
            </tr>
        </tbody>
    </table>
</div>
```

Скрипта 4.18: Поглед за апартмане

Поглед садржи table HTML-а, за приказивање листе апартмана. У оквиру table елемента, налази се tr елемент, којим се исцртава сваки апартман из листе.

4.3 BackboneJS

BackboneJS је лака JavaScript библиотека, која даје структуру за кôд клијентске стране. BackboneJS олакшава управљање и раздвајање проблема у апликацији, остављајући кôд лакшим за одржавање на дуже стазе. Уобичајено се користи за креирање SPA апликација. Фокусира се на помоћне методе за претраживање и манипулисање подацима. Backbone даје минимални скуп структурирања података (моделе, колекције) и типова корисничког интерфејса (погледа, URL), који су корисни приликом изградње динамичких апликација. Због мале величине, BackboneJS дозвољава корисницима преузимање на мобилне уређаје или слабије конекције. BackboneJS је креирао Jeremy Ashkenas у октобру 2010. године [17].

BackboneJS зависи од Underscore.js библиотеке, која укључује решење за шаблоне (енг. template).

Базиран је на модел-поглед-презентер (МПП) обрасцу:

- модел - одговара Backbone.Model;
- поглед - HTML/DOM дрво;
- презентер - одговара Backbone.View.

4.3.1 Модели

Backbone модели садрже податке апликације и начин оперисања њима. Примера ради, може се користити модел за представљање концепта ставке са којом треба оперисати (енг. todo item), укључујући атрибуте попут наслова (садржаја) и тренутног стања ставке (завршена или није). Модели се креирају наслеђивањем *Backbone.Model-a*:

```
var Stavka = Backbone.Model.extend({});
var stavka1 = new Stavka();
var stavka2 = new Stavka({
    naslov: 'Stavka 2',
    kompletirana: true
});
```

Скрипта 4.19: Креирање модела

Метод *initialize()* се позива када се креира нови примерак модела. Његова употреба није обавезна.

```
var Stavka = Backbone.Model.extend({
    initialize: function(){
        console.log('Model je inicijalizovan.');
    }
});
var stavka = new Stavka();
```

Скрипта 4.20: Иницијализација модела

Модел може имати подразумеване вредности. Оне се могу поставити коришћењем својства *defaults* у моделу.

```
var Stavka = Backbone.Model.extend({
    defaults: {
        naslov: '',
        kompletirana: false
    }
});

var stavka1 = new Stavka();
console.log(stavka1.get('naslov')); // prazan string
console.log(stavka1.get('kompletirana')); // false
```

Скрипта 4.21: Подразумеване вредности модела

Model.get() даје лак приступ атрибутима модела.

Model.set() поставља хеш који садржи један или више атрибута модела. Када било који од ових атрибута промени стање модела, догађај промене стања се активира. Такође се активирају и догађаји промена стања осталих атрибута (на пример, *change:name, change:age*).

```
var stavka = new Stavka({
    naslov: "Postavljen tokom instanciranja."
});
console.log('Stavka naslov: ' + stavka.get('naslov'));
// Stavka naslov: Postavljen tokom instanciranja.
console.log('Kompletirana: ' + stavka.get('kompletirana'));
// Kompletirana: false
stavka.set("naslov", "Naslov postavljen u Model.set().");
console.log('Stavka naslov: ' + stavka.get('naslov'));
```

```
// Stavka naslov: Naslov postavljen u  Model.set().
stavka.set({
    naslov: "Oba atributa postavljena u Model.set().",
    kompletirana: true
});
```

Скрипта 4.22: Постављање вредности модела

За добијање обавештења о промени модела, потребно је поставити ослушкивач на модел (догађај промене стања). Погодно место за додавање ослушкивача је *initialize()* функција.

```
var Stavka = Backbone.Model.extend({
    defaults: {
        naslov: '',
        kompletirana: false
    },
    initialize: function(){
        console.log('Model je inicijalizovan.');
        this.on('change', function(){
            console.log('- Vrednosti ovog modela su se promenile.');
        });
    }
});
```

Скрипта 4.23: Догађај промене стања модела

4.3.2 Погледи

Погледи у Backbone-у не садрже HTML апликације, већ логику која стоји иза презентације модела података. Ово се постиже коришћењем JavaScript шаблона (на пример, Underscore microtemplates, Mustache, jQuerytmpl). *Render()* метод, који се налази у погледу, може се везати за догађај промене стања модела, што омогућује погледу да одмах рефлектује промене на моделу, без потребе да се цела страница освежи.

Креирање нових погледа је слично креирању модела. Потребно је наследити *Backbone.View* [18].

```
var TodoView = Backbone.View.extend({
    tagName: 'li',
    todoTpl: _.template( "Primer sablona" ),
```

```
    events: {
        'dblclick label': 'edit',
        'keypress .edit': 'updateOnEnter',
        'blur .edit': 'close'
    },
    render: function() {
        this.$el.html( this.todoTpl( this.model.toJSON() ) );
        this.input = this.$('.edit');
        return this;
    },
    edit: function() {
        // izvrsava se kada se dva puta klikne na labelu stavke
    },
    close: function() {
        // izvrsava se kada stavka nije vise fokusirana
    },
    updateOnEnter: function( e ) {
        // izvrsava se na svaki pritisnuti taster kada je stavka u modu
   azuriranja
    }
});
var todoView = new TodoView();
```

Скрипта 4.24: Креирање погледа

Централно својство погледа је *el* својство, где је *el* референца на DOM елемент. Сваки поглед мора имати бар по једну такву референцу. Погледи могу користити *el* својство да повежу садржаје својих елемената са DOM-ом, што доприноси бржем представљању, јер интернет прегледач предузима минимални захтевани број прераспоређивања и исцртавања.

Постоје два начина за придруживање DOM елемента погледу: нови елемент може бити направљен за поглед и накнадно додат у DOM, или се референца може креирати на елемент који већ постоји на страници.

Ако се креира нови елемент за поглед, треба поставити било коју комбинацију следећих својстава на погледу: tagName, id и className. Радни оквир ће креирати нови елемент и његова референца ће бити доступна у *el* својству. Уколико се не наведе, подразумевана вредност за tagName је div.

У претходном примеру, tagName је постављен на li, што производи креирање новог li елемента. Следећи пример креира ul елемент са id-јем и class атрибутима:

```
var TodosView = Backbone.View.extend({
    tagName: 'ul', // potreban, ali je podrazumevani 'div' ukoliko nije
   postavljen
    className: 'list', // opcioni, moguce je postaviti vise klasa
    id: 'todos', // opcioni
});
```

Скрипта 4.25: Креирање погледа

Уколико елемент већ постоји на страници, може се поставити *el* својство, као CSS селектор који одговара елементу. Алтернативно, *el* својство се може поставити на постојећи елемент када се поглед креира:

```
el: '#zaglavlje'
// drugi pristup
var todosView = new TodosView({el: $('#zaglavlje')});
```

Скрипта 4.26: el као CSS селектор

```
<div id="stavka">
</div>
<script type="text/template" id="item-template">
    <div>
        <input id="stavka_kompletirana" type="checkbox" <%= kompletirana
    ?
        'checked="checked"' : '' %>>
        <%= naslov %>
    </div>
</script>
```

Скрипта 4.27: Underscore шаблон

Функција *render()* је опциона функција која дефинише логику приказивања шаблона.

Метод *_.template* из Underscore библиотеке компајлира JavaScript шаблоне у функције које се могу користити за приказивање. У TodoView се прослеђује означавање (енг. markup) из шаблона са id-јем ставке шаблона до *_.template()* да би се компајлирала и чувала у *todoTpl* својству када се поглед креира.

Према конвенцији *render()* метода има повратну вредност this. Ово је корисно из следећих разлога:

- Чини погледе лако поновно употребљивим у другим родитељским погледима
- Креира листу елемената без приказивања и исцртавања сваког од њих посебно, већ се они исцртавају само једном када се листа попуни.

4.3.3 Пример 3

Имплементирана је апликација која је представљена у (4.1.5). Најпре се укључују следеће библиотеке: jQuery, Underscore и Backbone.js. У скриптама model.js, view.js и routes.js у коду 4.28 редом се налазе модели, погледи и дефиниција рутера, који се прво инстанцира.

```
<script src="~/Scripts/model.js"></script>
<script src="~/Scripts/view.js"></script>
<script src="~/Scripts/routes.js"></script>

<script type="text/javascript">
    var router = null;

    $(document).ready(function () {
        router = new myRouter();
        Backbone.history.start();
    })
</script>

<div class="main">
    <div id="rAppContainer">
        <h2>Apartments</h2>
        <br />
        Search: <input type="text" id="search">
        <br/><br/>
    </div>
</div>
```

Скрипта 4.28: Иницијализација рутера

Модел за колекцију апартмана је дефинисан на следећи начин:

```
var ApartmentCollection = Backbone.Collection.extend({
    model: Apartment,
    url: 'http://localhost:8090/api/values/',
    search: function (letters) {
        if (letters == "") return this;
        return this.filter(function (data) {
            var res = false;
            if (data.get("Name").toLowerCase().indexOf(letters.
   toLowerCase()) !== -1)
                res = true;
            return res;
        });
    }
});
```

Скрипта 4.29: Дефинисање колекције са методом за претрагу апартмана

```
var appListView = Backbone.View.extend({
    events: {
        "keyup #search": "search"
    },
    search: function (e) {
        var letters = $("#search").val();
        if (letters == '')
        {
            this.render(this.collection);
            return;
        }
        var apartments = new ApartmentCollection(this.collection.search(
   letters));
        this.render(apartments);
    },
    render: function (apartments) {
        this.$el.html();
        $("#rAppContainer tr").remove();
        for (var i = 0; i < apartments.length; ++i) {
            var m_appView = new appView({ model: apartments.at(i) });
            this.$el.append(m_appView.$el);
            m_appView.render();
        }
        return this;
```

```
    },
});

var appView = Backbone.View.extend({
    model: Apartment,
    tagName: 'tr',
    template: '',
    initialize: function () {
        this.template = _.template($('#rAppContainer').html());
    },
    render: function () {
        if (this.model.get("ImageName") !== null)
            this.$el.html('<td><a href="/#Details/' +
            this.model.get("ID") + '"><img src="../Images/' +
            this.model.get("Name") + '/'+ this.model.get("ImageName") +
            '" width="300" height="220" /> </td><td>' + this.model.get("
   Name") +
            '</td><td><a href="/#Details/' +
            this.model.get("ID") + '">More details</a></td>');
        else
            this.$el.html('<td>-</td><td>' + this.model.get("Name") +
            '</td><td><a href="/#Details/' + this.model.get("ID") +
            '">More details</a></td>');

        return this;
    }
});
```

Скрипта 4.30: Погледи за апартмане

Како је за tagName у appView постављен tr елемент (скрипта 4.30), нови tr елемент се креира за сваки апартман. Функција *render* служи за исцртавање и позвана је из render функције appListView погледа.

Шаблони се користе за динамичко креирање HTML-а, убризгавањем модела података на држаче места (енг. placeholders). Један од коришћених шаблона у апликацији приказан је у скрипти 4.31. Интернет прегледач интерпретира ове script елементе као обичан текст. Underscore приступа шаблонима и исцртава фрагменте HTML-а.

```
<script id="details" type="text/html">
    <h2 style="margin-left: 30%;"><%= Name %></h2>
    <div style="display: inline; margin-left: 30%; text-align: center;">
   <%= Address %>, <%= City %></div>
```

```
    <br /><br />
    <div><%= Description %></div>
    <p style="font-weight: bold;">Price: <span><%= Price %></span>&euro;
   </p>
</script>
```

Скрипта 4.31: Шаблон за приказивање детаља апартмана

4.4 EmberJS

EmberJS је JavaScript радни оквир отвореног кода, који омогућава развој скалабилних SPA апликација и реализацију комплексних функционалности на вебу. Базиран је на MPK дизајн обрасцу. Настао је као друга верзија SproutCore [11] радног оквира, у мају 2011. године [19].

EmberJS доноси стандардизацију, номенклатуру и садржи механизме који, уместо програмера, брину о многим стварима, као на пример: цурењу меморије (енг. memory leak) итд. Заснива се на следећим концептима: богат објектни модел, рутирање, двосмерно везивање података, својства која се рачунају и аутоматско ажурирање шаблона помоћу Handlebars.js. Омогућава структурирање JavaScript кода у конзистентном и поузданом дизајн обрасцу, задржавајући HTML и CSS прегледним. EmberJS ово постиже структурирањем апликације у логички апстрактне слојеве, присиљавајући развојни модел да буде објектно-оријентисани, што је више могуће.

У суштини, EmberJS има уграђену подршку за следеће функционалности:

- Везивање података - омогућава да се промене једне промењиве пренесу на другу промењиву и обрнуто.
- Израчуната својства - омогућава означавање функције као својство, које се аутоматски ажурира са својствима на које се ослања.
- Шаблони који се аутоматски ажурирају - осигурава да GUI остаје ажуриран, кад год се промена догоди у означеним подацима.

[11] JavaScript радни оквир отвореног кода, иницијално креиран 2007. године од стране Sproutit, као основа за њихову Mailroom апликацију.

EmberJS укључује комплетну МПК имплементацију, која обогаћује слојеве контролера и погледа:

- Контролер слој - изграђен комбинацијом рутера (енг. routes) и контролера.
- Поглед слој - изграђен комбинацијом шаблона и погледа.
- Модел слој - изграђен са *Ember Data*.

4.4.1 Модели и Ember Data

EmberJS користи *Ember Data* да поједностави апликацију, чиме обезбеђује богат модел података са функционалностима које су потребне за развој богатих веб апликација. *Ember Data* представља једну могућу имплементацију која се може користити за комуникацију са сервером. Налази се у слоју модела и користи за дефинисање објеката модела и клијент-сервер API, као и транспортних протокола између EmberJS апликације и сервера (jQuery, XHR, WebSockets, и други).

Слој модела је одговоран за било коју комуникацију са серверском страном, као и за задатке које се односе на модел, као што је формат података. Поглед везује GUI компоненте са својствима објеката модела преко контролера.

4.4.2 Контролери и Ember рутери

Контролер је веза између модела и погледа. EmberJS садржи неколико уобичајених контролера, пре свега *Ember.ObjectController* и *Ember.ArrayController*. Генерално, *ObjectController* се користи ако контролер представња један објекат, а *ArrayController* ако контролер представља низ ставки.

EmberJS користи *Ember Router* да раздвоји апликацију на јасно дефинисана логичка стања. Свака рута (енг. route) може имати бројне подруте, и рутер се користи за навигацију између стања апликације.

Ember Router је такође механизам који EmberJS користи да ажурира URL апликације и да ослушкује URL промене. Када се користи *Ember Router*, моделују се стања апликације у хијерархијској структури која подсећа на државни графикон.

4.4.3 Погледи и Handlebars.js

Поглед је одговоран за цртање својих елемената на екрану. Погледи не чувају перманентно стање. Подразумевано, сваки поглед у EmberJS има један контролер као свој контекст. Он користи контролер да добије податке и за било каје акције корисника које се на њему дешавају.

Исто тако, EmberJS користи Handlebars.js за шаблоне. Већина Ember апликација дефинише своје корисничке интерфејсе преко Handlebars.js шаблона. Сваки поглед користи један шаблон за приказ.

Handlebars.js се заснива на Mustache, шаблон библиотеци без логике која постоји за низ програмских језика, укључујући и JavaScript. Handlebars.js додаје логичке изразе (if, if-else, each, итд) Mustache библиотеци. Ово, заједно са могућношћу да веже шаблоне са својствима на погледима и контролерима, омогућава изградњу шаблона који су добро структурирани и специфични за EmberJS апликацију.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Ember</title>
  <link rel="stylesheet" href="css/normalize.css">
  <link rel="stylesheet" href="css/style.css">
</head>
<body>
  <script type="text/x-handlebars">
    <h2>Dobro dosli u Ember.js</h2>

    {{outlet}}
  </script>

  <script type="text/x-handlebars" id="index">
    <ul>
    {{#each item in model}}
      <li>{{item}}</li>
    {{/each}}
    </ul>
  </script>

  <script src="js/libs/jquery-1.10.2.js"></script>
  <script src="js/libs/handlebars-1.1.2.js"></script>
```

```
  <script src="js/libs/ember-1.5.1.js"></script>
  <script src="js/app.js"></script>
</body>
</html>
```

Скрипта 4.32: Ember

У скрипти 4.32, body елемент садржи Handlebars шаблоне. Шаблони су садржани у оквиру скрипт елемената са типом text/x-handlebars. Овај тип говори интернет прегледачу да садржај не интерпретира као JavaScript; садржај ових скрипт елемената ће бити на страници и моћи ће да му се присутпи из JavaScript-а, али неће бити извршен док то не уради Ember и Handlebars.

```
App = Ember.Application.create();

App.Router.map(function() {
    // ovde se postavljaju rute
});
App.IndexRoute = Ember.Route.extend({
    model: function() {
        return ['crvena', 'zuta', 'plava'];
    }
});
```

Скрипта 4.33: app.js

Када се инстанцира објекат *Ember Application*, Ember ће одмах тражити у коду имплементације *ApplicationRoute*, *ApplicationController*, или шаблон са атрибутом *data-template-name="application"*, или први шаблон на страници без наведеног *data-template-name* [20]. Уколико не нађе прве две ствари, Ember ће креирати подразумеване инстанце.

Подразумевано је да се користи хеш у URL рути. Уколико у URL рути не постоји хеш, онда се тражи специфични фајл у директоријуму који може бити понуђен традиционалним веб сервером. Овде постоји само један фајл, тако да URL не може да показује на други фајл или фолдер. Када се унесе хеш у URL, веб сервер више не покушава да интерпретира URL као фајл/фолдер структуру од тачке хеша. Може се ставити било шта након хеша и сервер неће пробати да лоцира фајл по том имену. Страница ће узети цео URL и Ember "зна" како да претвори остатак тих ствари у URL-у након хеша у руту.

Када се апликација покрене у интернет прегледачу, редом се извршавају следећи кораци:

1. HTML страница се учита, тј. учитава се и извршава app.js фајл.

2. app.js инстанцира *Ember Application*.

3. Када се *Application* покрене, Ember тражи *ApplicationRoute* и учитава је, ако рута постоји. У сваком случају Ember учитава подразумевану руту (у случају да *ApplicationRoute* класа постоји, онда она наслеђује подразумевану руту).

4. *Application* ради исту рутину са *ApplicationController*.

5. Ако постоји дефинисана *ApplicationRoute* и догађај унутар ње, тада се овај догађај активира.

6. Ember тражи шаблон апликације у HTML фајлу и поставља га са *ApplicationController* као његовим контролером, одакле се подаци везују за шаблоне, мењајући држаче места са стварним подацима.

7. Пошто се прво учитава главни URL апликације, Ember ће лоцирати и инстацирати *IndexRoute* која је дефинисана у app.js.

8. Ember ће сада идентификовати да не постоји *IndexController* и инстанцираће један. Како је у *IndexRoute* дефинисан модел као низ, Ember ће инстацирати *ArrayController* као *IndexController*.

9. Ember проналази шаблон са називом index у HTML документу и приказује га у outlet-у који се налази у шаблону апликације.

4.4.4 Пример 4

У овом одељку биће описана Ember имплементација апликације која је представљена у (4.1.5). У скрипти 4.34, дати су шаблони за приказ детаља апартмана са галеријом слика. Коришћени су Handlebars шаблони.

```
<script type="text/x-handlebars" id="details">
    <h2 style="margin-left: 30%;">{{Name}}</h2>
    <span style="display: inline; margin-left: 30%; text-align: center;"
   >{{Address}}</span>,
    <span>{{City}}</span>
```

```
    <br /><br />
    <div>{{Description}}</div>
    <p style="font-weight: bold;">
    Price: <span>{{Price}}</span>&euro;</p>

    <div id="gallery" class="content">
        <div id="controls" class="controls"></div>
        <div class="slideshow-container">
            <div id="loading" class="loader"></div>
            <div id="slideshow" class="slideshow" style="width:304px;
   height:228px;"></div>
        </div>
        <div id="caption" class="caption-container"></div>
    </div>
    <div id="thumbs" class="navigation">
        <ul class="thumbs noscript" >
            {{#each e in images}}
                <li class="imgdata">
                    <a class="thumb" {{bind-attr href=e.ImageSrc title="
   e.ImageName"}}>
                        <img {{bind-attr src=e.ImageSrc title="e.
   ImageName" alt="e.ImageName"}} width="300" height="220" />
                    </a>
                </li>
            {{/each}}
        </ul>
    </div>
    <p>{{#link-to "index"}}Back{{/link-to}}</p>
</script>
```

Скрипта 4.34: Детаљи апартмана са галеријом слика

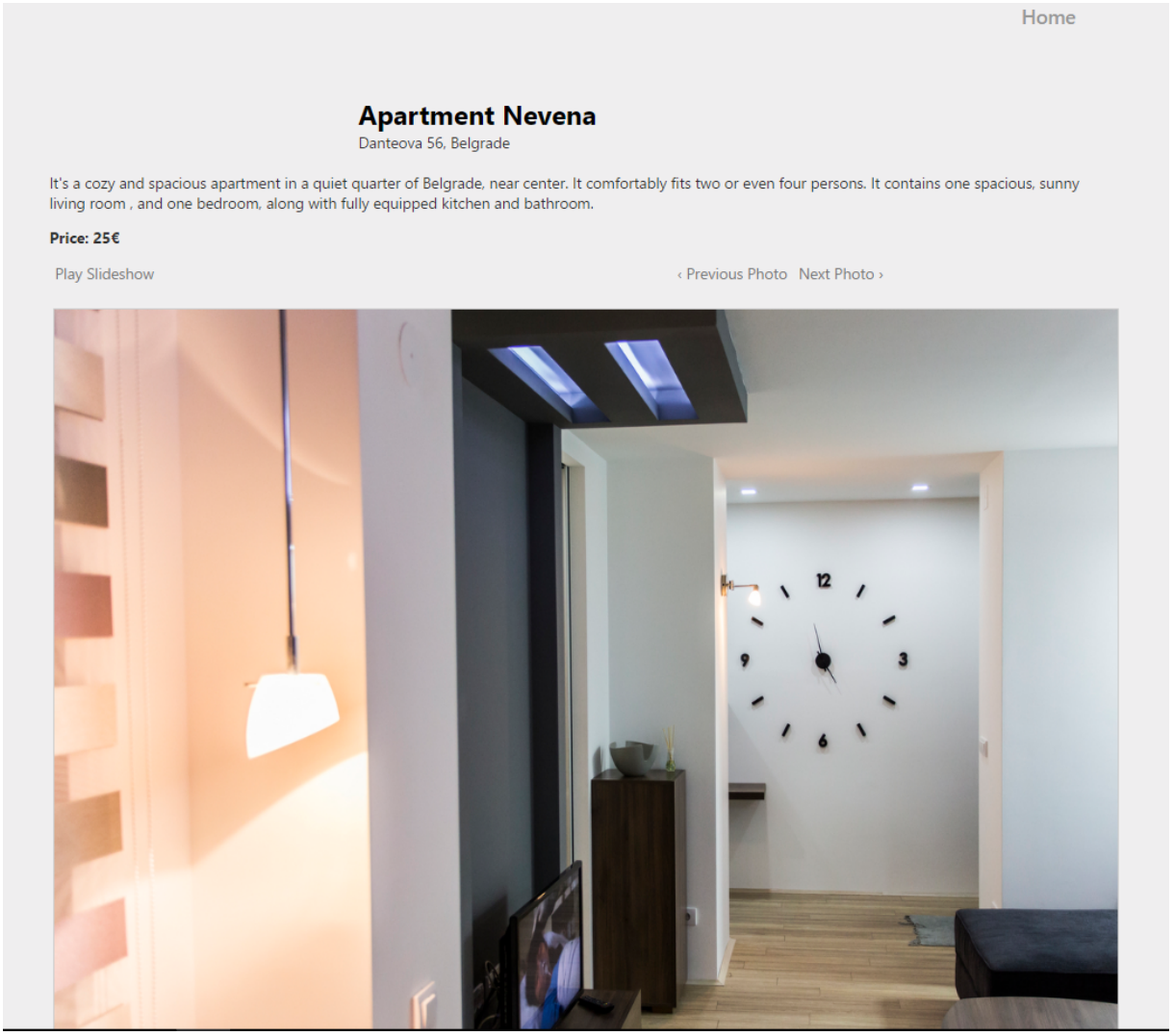
Модели података дефинисани су у рутама. Позива се сервис, који враћа податке о апартаману и слике које су за њега везане.

```
Apartments.Router.map(function () {
    this.resource("details", { path: "/details/:id" });
});
Apartments.IndexRoute = Ember.Route.extend({
        model: function() {
        return Ember.$.getJSON("http://localhost:8090/api/values").then(
   function(json) {
            return json.map(function(item) {
```

```
            return {
            ImageUrl: "#/details/" + item.ID,
            ImageSrc: "../Images/" + item.Name + "/" + item.ImageName,
            ID: item.ID, Name: item.Name, ImageName: item.ImageName
            };
            });
        });
    }
});
Apartments.DetailsRoute = Ember.Route.extend({
    model: function (params) {
        return Ember.$.getJSON("http://localhost:8090/api/values/" +
   params.id);
    },
    afterModel: function (params) {
        var self = this;
        return Ember.$.getJSON("http://localhost:8090/api/images/"
        + params.ID).then(function (result) {
            self.set('images', result);
        });
    },
    setupController: function (controller, model) {
        this._super(controller, model);
        controller.set('images', this.get('images'));
    }
});
```

Скрипта 4.35: routes.js

Након што су дефинисни модел података и поглед кроз Handlebars шаблоне, приказује се опис апартмана са сликама (слика 4.2). Пошто није дефинисан *DetailsController*, Ember ће га сам инстацирати као *ObjectController* због модела који је дефинисан као објекат у *DetailsRoute*.



Слика 4.2: Изглед описа апартмана

За преглед и претрагу апартмана на почетној страници, дефинисан је *IndexController* као *ArrayController*. Његова дефиниција је приказана у скрипти 4.36.

```
Apartments.IndexController = Ember.ArrayController.extend({
searchTerm: '',
actions: {
    filteredList: function () {
        var q = this.get('searchTerm');
        var self = this;
        Ember.$.getJSON("http://localhost:8090/api/values",
        function (data) {
            var list = data.map(function(item) {
                return {
                ImageUrl: "#/details/" + item.ID,
```

```
                ImageSrc: "../Images/" + item.Name + "/" + item.
    ImageName,
                ID: item.ID,
                Name: item.Name,
                ImageName: item.ImageName
                };
            });
             list = list.filter(function (item) {
                return item.Name.indexOf(q) !== -1;
            });
            self.set("model", list);
        });
    }
}
});
```

Скрипта 4.36: controllers.js

4.5 Разлике између описаних JavaScript библиотека

У овом одељку биће упоређене описане JavaScript библиотеке: AngularJS, KnockoutJS, BackboneJS и EmberJS.

4.5.1 Величина радног оквира

Време учитавања странице је пресудно за успех веб сајта. Корисници не показују много стрпљења када је у питању брзина прегледања - тако да је у многим случајевима пожељно да се уради све што је могуће како би рад апликације био што бржи. Постоје два фактора која треба погледати када се разматра утицај радног оквира на време учитавања апликације: величина радног оквира и време које је потребно оквиру за учитавање.

Библиотека	Нето величина	Величина са потребним зависностима
AngularJS	122kb	122kb
KnockoutJS	54kb	54kb
BackboneJS	17kb	61kb (jQuery + Underscore)
EmberJS	90kb	136.2kb (jQuery + Handlebars)

BackboneJS је по величини најмања библиотека, али зависи од других библиотека. AngularJS и KnockoutJS немају зависности, док их поред BackboneJS има и EmberJS.

4.5.2 Шаблони

AngularJS и KnockoutJS укључују механизме за шаблоне. Код њих су ови механизми имплементирани везивањем података. BackboneJS може бити интегрисан са многим механизмима за шаблоне. Подразумевани механизам је Underscore. Како је Underscore библиотека потребна за рад BackboneJS-а и већ постоји на страници, предност је искористити овај механизам за шаблоне без додавања додатних зависности апликацији. EmberJS користи Handlebars за шаблоне.

AngularJS и EmberJS су базирани на МПК дизајн обрасцу, KnockoutJS на МППМ, а BackboneJS на МПП обрасцу.

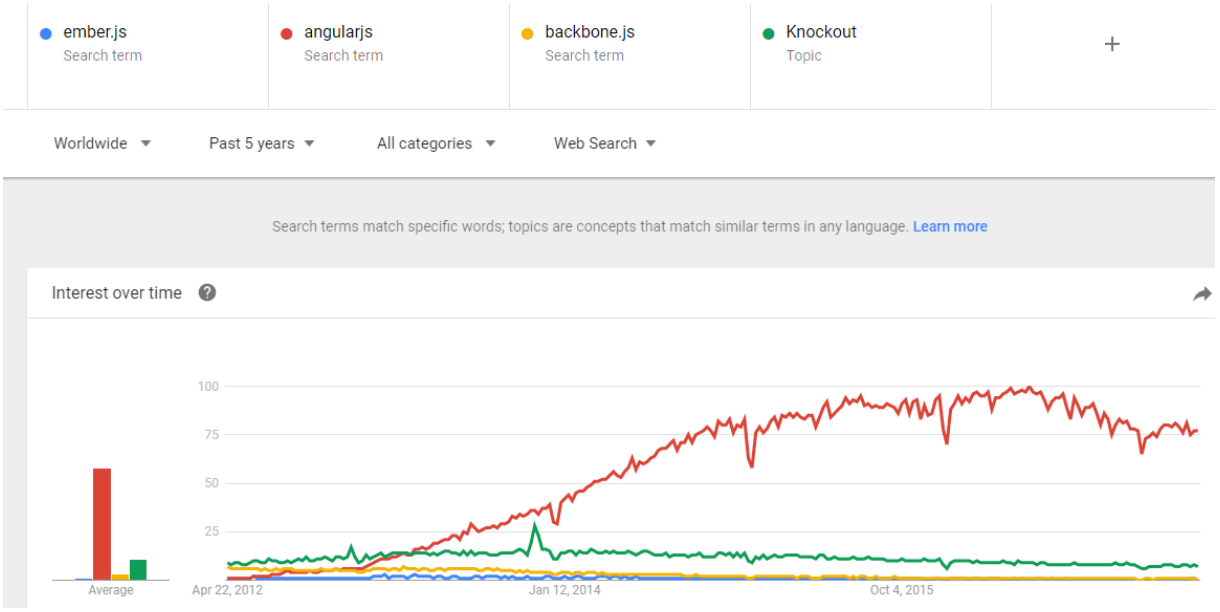
Битне ствари код тестирања кода су: модуларност (мањи делови који могу бити тестирани у изолацији) и убризгавање зависности (могућност промене зависности у тесту). Ове две функционалности су основне код AngularJS-а и због тога он има предност у овој области.

Knockout нема рутер, остале библиотеке га имају. У Knockout-у се као рутер може користити Sammy.js. Sammy.js је мали JavaScript радни оквир, развијен да обезбеди основну структуру за развој JavaScript апликација. Његово језгро садржи једноставан API за дефинисање апликације, које су направљене првенствено од рута и догађаја.

4.5.3 Заступљеност на тржишту

Очекиван је наставак еволуције постојећих радних оквира и библиотека, као и креирање нових који унапређују неке нове идеје или концепте, чинећи креирање веб аплицакија лакшим. Већи радни оквири ће покупити најбоље идеје и сумирати их. Ово се већ могло видети са Angular 2 који прихвата супериорне форме интерног приказивања из прве верзије. Најзаступљенији на тржишту су: Angular, затим Knockout, Backbone и Ember према Google Trends (слика 4.3). У зависности од пројекта (на пример за компанијске пројекте и сл.), избегава се коришћење "великих" радних оквира

као што је Angular, јер се тиме индиректно ствара зависност од компаније Google, која диктира развој. Остали мањи оквири су с тог аспекта бољи.



Слика 4.3: Заступљеност на тржишту

На основу описаних примера, најлакше је радити са Angular библиотеком. Не само да Angular има највећу заједницу и много садржаја на интернету, већ га и Google такође подржава и промовише. Angular је много лакши за разумевање, осим у случају када се креирају директиве. Angular подржава везивање података у оба смера, веома је брз и нема додатних зависности.

Добра ствар код Ember-а је наклоност конвенцији пре него конфигурацији. Приликом дефинисања рутера, Ember дефинише и подразумевану руту и подразумевани контролер. Уколико ниједан контролер није дефинисан, Ember користи подразумевани контролер.

Са BackboneJS-ом се не добија структура, већ се креирају делови HTML-а са Javascript-ом, који је везан за њих. Knockout је веома брз, али нема сопствени рутер. Погодан је за апликације које су вођене подацима, а не за тешке DOM апликације.

Глава 5

Закључак

Примарни циљ овог рада је представљање архитектурних образаца, њихових сличности и разлика, као и њихове примене у развоју веб и мобилних апликација.

Описано је неколико веома важних архитектурних образаца. Уз рад је имплементирана ASP.NET MVC апликација којом је представљен овај оквир и његове предности. Такође су развијене четири мање апликације које користе описане JavaScript библиотеке. Свака од четири развијене апликације, комуницира са RESTful сервисом, који их снадбева подацима. Развијене имплементације омогућиле су доношење адекватних закључака о њиховим карактеристикама, сличностима и разликама.

Литература

[1] Homayoun Mehmaneche Andreas Zendler Jörg Noack, Hamarz Mehmaneche. Architectural patterns for web applications. 2000.

[2] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[3] Hans Rohnert Peter Sommerland Frank Buschman, Regine Meunier. *Pattern Oriented Software Architecture - A system of patterns*. John Willey and Sons, New York, 2000.

[4] Sal Razzaq. Model-view-presenter (mvp), Januar 19 2011. URL https://sal-razzaq.blogspot.rs/2011/01/model-view-presenter-mvp.html.

[5] Addy Osmani. *Learning JavaScript Design Patterns*. O'Reilly, Sebastopol, California, 2015.

[6] The mvvm pattern, Februar 10 2012. URL www.msdn.microsoft.com/en-us/library/hh848246.aspx.

[7] Adam Freeman. *Pro ASP.NET MVC 4*. Apress, 2012.

[8] Brad Wilson K. Scott Allen Jon Galloway, Phil Haack. *ASP.NET MVC 4*. John Wiley Sons, 2012.

[9] Ethan Marcote. *Responsive Web Design*. A Book Apart, New York, 2011.

[10] Responsive web design. URL https://en.wikipedia.org/wiki/Responsive_web_design.

[11] Wiki: Angularjs, Март 04 2015. URL https://en.wikipedia.org/wiki/AngularJS.

[12] Shyam Seshadri Brad Green. *AngularJS*. O'Reilly, Sebastopol, California, 2013.

[13] Анђелка Зечевић. Програмирање за Веб: Angularjs, Март 04 2015. URL http://poincare.matf.bg.ac.rs/~andjelkaz/pzv/cas_angularJS/angularJS.pdf.

[14] Knockout (web framework), 2012. URL https://en.wikipedia.org/wiki/Knockout_(web_framework).

[15] Knockout, 2010. URL http://knockoutjs.com/documentation.

[16] Jamie Munro. *Knockout.js*. O'Reilly, Sebastopol, California, 2015.

[17] Backbone.js, 2013. URL https://en.wikipedia.org/wiki/Backbone.js.

[18] Addy Osmani. *Developing Backbone.js Applications*. O'Reilly, Sebastopol, California, 2013.

[19] Joachim Haagen Skeie. *Ember.js in Action*. Manning, 2014.

[20] Jesse Cravens and Thomas Q Brady. *Building Web Apps with Ember.js*. O'Reilly, Sebastopol, California, 2014.