**etf+grb**

**ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ**

**УНИВЕРЗИТЕТ ИСТОЧНО САРАЈЕВO**

ИЗРАДА АПЛИКАЦИЈЕ ЗА ЕЛЕКТРОНСКУ

ТРГОВИНУ ПОМОЋУ *ASP.NET* И *REACT* ТЕХНОЛОГИЈА

**ЗАВРШНИ РАД**

Студије: I циклус

Одсјек: Рачунарство и Информатика

Кандидат/Студент:

*Дејан Чанчар, 1661*

Ментор:

*Доц. др Сњежана Милинковић*

**Источно Сарајево, децембар 2020. године**

САДРЖАЈ:

[1 УВОД 7](#_Toc59066566)

[2 МОДЕЛИ И СЕРВИСИ ЕЛЕКТРОНСЕ ТРГОВИНЕ 8](#_Toc59066567)

[2.1 Историја е-трговине 8](#_Toc59066568)

[2.2 Модели електронске трговине 8](#_Toc59066569)

[2.2.1 Компанија према компанији (*B2B*) 8](#_Toc59066570)

[2.2.2 Компанија према клијенту (*B2C*) 9](#_Toc59066571)

[2.2.3 Клијент према клијенту (*C2C*) 10](#_Toc59066572)

[2.2.4 Клијент према компанији (*C2B*) 10](#_Toc59066573)

[3 ПОЈАМ ВЕБ АПЛИКАЦИЈА 11](#_Toc59066574)

[3.1 Технологије клијентске стране (*Front-end)* 12](#_Toc59066575)

[3.1.1 *HTML* 12](#_Toc59066576)

[*3.1.2* *CSS* 13](#_Toc59066577)

[3.1.3 *JavaScript* 14](#_Toc59066578)

[3.2 Асинхрони *JavaScript* 16](#_Toc59066579)

[3.3 Технологије серверске стране (*Back-end*) 17](#_Toc59066580)

[3.4 Архитектура веб апликација 17](#_Toc59066581)

[3.4.1 Архитектура клијентске стране 17](#_Toc59066582)

[3.4.2 Архитектура серверске стране 18](#_Toc59066583)

[3.4.3 Вишеслојна архитектура и *RestFul* веб сервиси 20](#_Toc59066584)

[4 РАД СА БАЗОМ ПОДАТАКА 21](#_Toc59066585)

[4.1 Релационе базе података 21](#_Toc59066586)

[4.2 *NоSQL* базе података 22](#_Toc59066587)

[5 РАДНИ ОКВИРИ 24](#_Toc59066588)

[5.1 *ASP.NET CORE* 24](#_Toc59066589)

[5.1.1 *NuGet* 26](#_Toc59066590)

[5.1.2 Уметање зависности 27](#_Toc59066591)

[*5.2* *Entity* 28](#_Toc59066592)

[5.2.1 Историја *Entity* радног оквира 30](#_Toc59066593)

[5.2.2 Архитектура 31](#_Toc59066594)

[5.2.3 Приступи развоју 32](#_Toc59066595)

[5.2.4 Сценарији развоја 33](#_Toc59066596)

[5.3 *React* 35](#_Toc59066597)

[5.4 *Bootstrap* 36](#_Toc59066598)

[6 ПРАКТИЧНА РЕАЛИЗАЦИЈА ПРОЈЕКТА 36](#_Toc59066599)

[7 ЗАКЉУЧАК 37](#_Toc59066600)

СПИСАК СЛИКА

[Слика 1 Дијаграм B2B модела пословања 9](#_Toc59066601)

[Слика 2 Дијаграм B2C модела пословања 9](#_Toc59066602)

[Слика 3 Дијаграм C2C модела пословања 10](#_Toc59066603)

[Слика 4 Дијаграм C2B модела пословања 10](#_Toc59066604)

[Слика 5 Начин рада веб апликације 11](#_Toc59066605)

[Слика 6 Начин рада АЈАКС-а [6] 16](#_Toc59066606)

[Слика 7 Технологије серверске стране 17](#_Toc59066607)

[Слика 8 Начин функционисања МВВМ шаблона [7] 18](#_Toc59066608)

[Слика 9 Примјена MVC дизајн шаблона на серверској страни [8] 19](#_Toc59066609)

[Слика 10 Дијаграм вишеслојне архитектуре[9] 20](#_Toc59066610)

[Слика 11 Архитектура .NET CORE радног оквира[12] 25](#_Toc59066611)

[Слика 12 Ток NuGet пакета[13] 26](#_Toc59066612)

[Слика 13 Примјер директне зависности објеката. 27](#_Toc59066613)

[Слика 14 Примјер инвертоване контроле тока 28](#_Toc59066614)

[Слика 15 Позиција Entity радног оквира у апликацији 29](#_Toc59066615)

[Слика 16 Архитектура Entity радног оквира 32](#_Toc59066616)

[Слика 17 Дијаграм одлучивања код развоја базе података (EF) [14] 32](#_Toc59066617)

[Слика 18 Приступ „База прво“ [14] 33](#_Toc59066618)

[Слика 19 Приступ „Код прво“ [14] 33](#_Toc59066619)

[Слика 20 Приступ „Модел прво“ [14] 33](#_Toc59066620)

[Слика 21 Повезани сценарио[14] 34](#_Toc59066621)

[Слика 22 Неповезани сценарио[14] 34](#_Toc59066622)

[Слика 23 Карактеристике React библиотеке 35](#_Toc59066623)

[Слика 24 DOM веб странице[15] 36](#_Toc59066624)

СПИСАК КОДОВА

[Кодни листинг 1 Примјер дефиниције HTML документа 12](#_Toc59066625)

[Кодни листинг 2 Примјер HTML етикете без и етикете са атрибутом 13](#_Toc59066626)

[Кодни листинг 3 Примјер минималне структуре HTML документа 13](#_Toc59066627)

[Кодни листинг 4 Примјер позивања вањског CSS документа 14](#_Toc59066628)

[Кодни листинг 5 Примјер JSX екстензије 35](#_Toc59066629)

СПИСАК ТАБЕЛА

[Табела 1 Историја .NET CORE радног оквира 25](#_Toc59066630)

[Табела 2 Историјски развој EF 6 радног оквира 31](#_Toc59066631)

[Табела 3 Историјски развој EF Core радног оквира 31](#_Toc59066632)

СПИСАК СКРАЋЕНИЦА

*ЕНГ. - Енглески*

*ТЈ. – То јесте*

*НПР. – На примјер*

*ИТД. – И тако даље*

*EF – Entity Framework*

*DOM – Document Object Model*

# УВОД

Појавом *IMB*-овог *OLTP* (енг*. Online transaction processing*) система за интернет обраду трансакција први пут се јавља и термин електронске трговине, систем је омогућавао обраду финансијских трансакција у реалном времену. [1][2]

Развојем технологија, експанзијом интернета и порастом потреба корисника апликације за електронску трговину се константно мијењају. Изворно ове апликације су биле презентационе веб странице чија је сврха била промовисање производа одређене компаније, али јако брзо презентационе веб странице су прерасле у потпуно функциноналне платформе за трговину, платформе какве данас познајемо. [3]

Развој *HTTPS* (енг. *Hypertext Transfer Protocol Secure*) протокола је битно утицао на побољшање сигурности електронске трговине. Захваљујући томе трансакције путем интернета су постале доста сигурније него раније, те су веб апликације за електронску трговину придобиле повјерење корисника.

Због све бржег начина живота и повјерења корисника електронска трговина тежи да замјени традиционални начин трговања.

Циљ овог рада јесте реализација веб апликације за електронску трговину за потребе једне компаније, коришћењем тренутно актуелних технологија.

Рад се састоји од пет поглавља:

1. Модели и сервиси електронке трговине
2. Појам веб апликација
3. Рад са базом података
4. Радни оквири
5. Практична реализација веб апликације за електронску трговину

# МОДЕЛИ И СЕРВИСИ ЕЛЕКТРОНСЕ ТРГОВИНЕ

Појам електронске трговине се односи на обављање пословних трансакција путем интернета. Било која куповина односно продаја добара или трансфер новца путем интернета сматра се е-трговином. Термин електронске трговине такође обухвата и друге активности попут *online* аукција, интернет банкарства, система за електронско плаћање (енг. *Payment gateways*) и продаја електронских карата.

## Историја е-трговине

Фил Бранденбергер (Phil Brandenberger) је 1994. године обавио прву електронску трансакцију у историји интернета. Он је користио своју *Mastercard* платну картицу како би купио *Sting’s Ten Summoners’ Tales* музички албум путем интернета. Трансакција је износила свега 12,48 долара али то је био први пут у историји интернета да се техника шифрирања користи у сврху електронског плаћања, тиме је ова трансакција показала свијету да је „интернет спреман“ за сигурну електронску трговину.

Од тог тренутка, па све до данас електронска трговина је наставила да се развија великом брзином. Брз развој електронске трговине битно је доприњео паду пословања многих предузећа.

Постоји велики број компанија које пружају многобројне сервисе за побољшање електронске трговине. Компанија *Google* нуди посебан програм рекламирања за веб продавнице, *PayPal* нуди могућност интеграције њиховог *API*-а за вршење електронских трансакција за лични бизнис профил, Амазон нуди сервисе интернет продаје у „облаку“ (енг. *E-commerce cloud computing*) за мала и велика предузећа и сл.

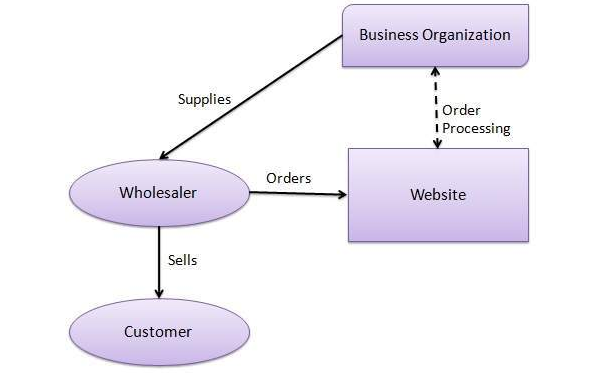
## Модели електронске трговине

Избор и примјена одговарајућег бизнис модела на одређени електронски бизнис (нпр. интернет продавница) представља велики изазов али и битно утиче на пословање самог бизниса.

У е-трговини постоје четири уобичајене врсте пословних модела који дефинишу циљну групу компаније. Свака компанија нуди различите производе, тако да пословни модели представљају неку врсту водича током процеса планирања са циљем побољшања пословања те компаније.

### Компанија према компанији (*B2B*)

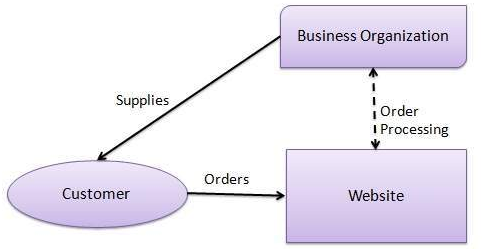
Како и сам назив каже, овај модел се заснива на пословању између двије компаније, у већини случајева компанија која врши куповину није крајњи купац већ представља посредника између произвођача и крајњег купца односно компанија која купује производе од произвођача врши препродају тих производа (Слика број 1).



Слика 1 Дијаграм *B2B* модела пословања

### Компанија према клијенту (*B2C*)

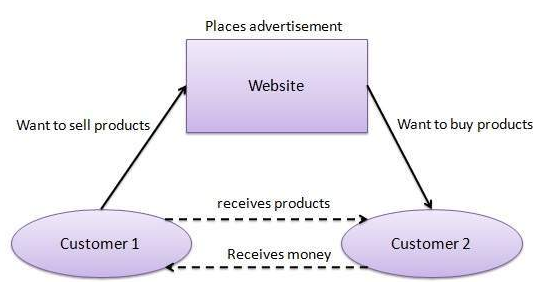
Овај модел пословања се односи на продају производа једне компаније директно крајњем купцу. Купац све доступне производе одређене компаније поручује путем веб странице те компаније. Веб страница мора посједовати систем обавјештавања компаније о поруџбини (нпр. путе електронске поште), те након пријема обавјештења о поруџбини, компанија врши процес достављања производа поручиоцу (Слика број 2).



Слика 2 Дијаграм *B2C* модела пословања

### Клијент према клијенту (*C2C*)

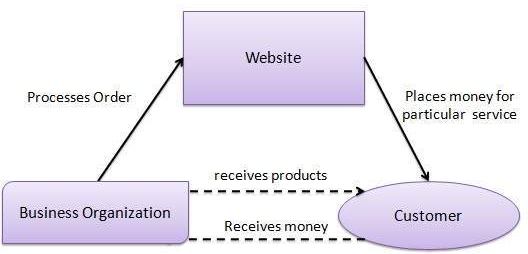
Овај модел пословања данас има велику популарност, а омогућава лакшу размјену добара између клијената (Слика број 3), нпр. продаја аутомобила, изнајмљивање апартмана и слично. Веб странице које пружају услуге клијент према клијенту модела могу, а и не морају да врше наплату услуга, у зависности од политике веб странице.



Слика 3 Дијаграм *C2C* модела пословања

### Клијент према компанији (*C2B*)

За разлику од стандардног модела гдје компанија нуди и наплаћује услуге или производе клијенту, модел, клијент према компанији функционише тако што клијент нуди одређене услуге или производе компанији (Слика број 4.). А компанија клијенту плаћа за услуге али услуге односно производи које клијент пружа утичу позитивно на профит компаније.



Слика 4 Дијаграм *C2B* модела пословања

# ПОЈАМ ВЕБ АПЛИКАЦИЈА

Данас, веб апликације су скоро у потпуности замијениле десктоп апликације. За разлику од десктоп апликација, веб апликације се покрећу у веб прегледнику те су за корисника платформски независне, а такође не зависе ни од спецификација уређаја на коме се покрећу.

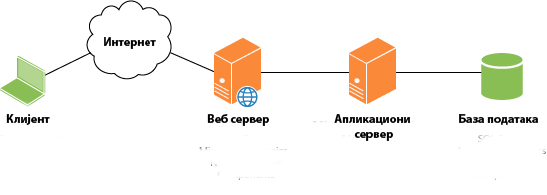
Адекватним организовањем архитектуре веб апликације могуће је направити информациони систем који је могуће, без потребе за инсталирањем, покренути и користити на било ком уређају који има приступ интернету путем веб прегледника. Ова особина игра велику улогу у расту популарности веб апликација. Веб апликације је знатно лакше одржавати од десктоп апликација, све измјене које се направе на хост серверу су одмах видљиве свим корисницима апликације.

Битно је увидјети разлику између веб апликације и динамичке веб странице (презентационе веб странице). Та разлика се најбоље може уочити у самој архитектури веб апликације и према њеној сврси.

Динамичке веб странице се у потпуности извршавају на корисничкој страни, тј. потпуно се извршавају у веб прегледнику. Комуникација између веб странице и корисника је у потпуности једносмјерна, веб страница кориснику шаље податке са сервера.

Веб апликације се само једним дијелом извршавају на корисничкој страни. Комуникација код веб апликација је двосмјерна, корисник од сервера захтјева одређене податке, а сервер шаље одређени одговор.

Начин рада веб апликације је приказан на слици 5.



Слика 5 Начин рада веб апликације

Веб апликацији корисник приступа путем веб прегледника. Веб прегледник путем интернета шаље захтјев за приказ веб апликације веб серверу, а веб сервер шаље одговор на захтјев који је добио од веб прегледника, те корисник добија почетни приказ веб апликације.

Све функционалности које веб апликација пружа се одвијају на исти начин, након одабира неке опције коју апликација пружа веб прегледник шаље захтјев веб серверу, веб сервер обрађује тај захтјев, у колико има потребе веб сервер контактира базу података путем апликационог сервера, обрађује одговор који је добио од апликационог сервера и као посљедњи корак путем интернета шаље одговор веб прегледнику.

## Технологије клијентске стране (*Front-end)*

Читав изглед веб апликације се заснива на три основне технологије *HTML* (енг. *Hypertext markup language*), *CSS* (енг. *Cascade style sheet*) и *JavaScript* скриптног објектног језика.

### *HTML*

*HTML* је описни језик који се користи за креирање хипертекст документа, а веб прегледник је задужен за приказ хипертекста. Основни задатак *HTML* језика јесте усмјерити веб прегледник како да прикаже садржај датог хипертекст документа.

Да би веб прегледник препознао *HTML* документ потребно је да *HTML* документ почиње дефиницијом типа документа и дефиницијом стандарда по коме је тај документ писан.

Примјер дефиниције *HTML* документа:

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">

Кодни листинг 1 Примјер дефиниције *HTML* документа

Синтакса *HTML* језика се заснива на етикетама (енг. *Tags*) и атрибутима.

Разликује двије врсте етикета, просте етикете <X> и сложене етикете (<Х></Х>). Просте етикете служе за означавање, док сложене етикете служе као ограђивачи. Веб прегледник интерпретира сваку етикету посебно примјењујући правила дефинисана том етикетом[4].

Унутар сложених *HTML* тагова се могу налазити атрибути који додатно описују начин на који веб прегледник треба приказати садржај који се налази унутар сложених тагова. Атрибути се, по правилу, пишу између пара наводника, а неправилне атрибуте и елементе веб прегледник игнорише. *HTML* језик не прави разлику између малих и великих слова, а уколико је потребно садржај прелама према ширини блока односно прозора.

Примјер *HTML* етикете без атрибута и етикете са атрибутом:

<**p**>Текст параграфа.</**p**>

<**p** align="right">Текст параграфа</**p**>

Кодни листинг 2 Примјер *HTML* етикете без и етикете са атрибутом

Сложена етикета *<p></p>* означава параграф, а атрибут *align="right"* говори веб прегледнику да текст унутар параграфа поравна десно.

HTML документ разликује два основна дијела, дио који описује особине документа тј. на који начин документ треба бити приказан и дијела који представља садржај документа. Особине документа, као и наслов документа се смјештају између сложене етикете *<head>,</head>*, а садржај документа се смјешта између етикете *<body>,</body>.* Наслов документа се означава етикетом *<title>,</title>.* Ове етикете обједињене етикетом *<html>,</html>* чине минималну структуру HTML документа.

Примјер минималне структуре:

<!DOCTYPE HTML>

<**html**>

<**head**>

<**meta** charset="UTF-8">

<**title**>Наслов документа</**title**>

</**head**>

<**body**>

<**h1**>Примјер документа</**h1**>

<**p**>Ово је примјер једног простог HTML документа.</**p**>

</**body**>

</**html**>

Кодни листинг 3 Примјер минималне структуре *HTML* документа

### *CSS*

За разлику од *HTML* -а који се користи за организацију приказа садржаја веб странице, CSS је задужен за дефинисање стилова елемената веб странице.

Првобитно *HTML* је одређивао и организацију садржаја и стилове појединих елемената веб странице. Временом, како су веб сајтови постајали све комплекснији наилазило се на проблем писања и читања *HTML* докумената, те је од верзије 4.0 *HTML* -а уведен *CSS*, језик форматирања.

Осим докумената за израду веб страница, *CSS* може користити било која врста *XML* (енг. *EXtensible Markup Language* ) докумената.

Синтакса *CSS* -а се огледа у дефинисању описа елемената документа. Опис елемента се састоји од дефиниције циљних елемената, атрибута и вриједности тих атрибута. Опис елемента може да се примјени на више елемената, а такође и више описа се може примијенити на један елемент, одакле и потиче назив Cascading. *CSS* стилове је могуће писати у одвојеном документу који мора имати екстензију .*CSS*, директно унутар *HTML* документа унутар *<style></style>* етикете или унутар самог *HTML* елемента кориштењем *style*=” ” атрибута.

Уколико се *CSS* стил пише у одвојеном документу, да би се примијенио на *HTML* документ, потребно је у заглављу *HTML* документа укључити *CSS* документ кориштењем *<link>* етикете.

Примјер позивања вањског *CSS* документа:

 <head>  
        <link rel="stylesheet" type="text/css"href="mystyle.css">  
    </head>

Кодни листинг 4 Примјер позивања вањског *CSS* документа

Циљне елементе је могуће дефинисати наводећи *HTML* етикету, класу или идентификациону вриједност елемента.

У првом случају, наводећи *HTML* етикету елемента, дефинисани опис се односи на све елементе унутар документа који имају наведену етикету.

Елементи *CSS* документа чији назив почиње тачком (нпр. .стил-табеле) означава класу. Класа унутар *HTML* етикете се позива помоћу атрибута *class=“ “,* а један *HTML* елемент може имати више различитих класа које дифинишу тај елемент, раздвојених бијелим простором унутар class атрибута.

Идентификациона вриједност елемента је вриједност која се налази унутар *ID* атрибута *HTML* етикете и јединствена је за сваки појединачни елемент *HTML* документа. Унутар *CSS* документа елементи чиј назив почиње тарабом (#) означавају идентификациони елемент, те стил написан унутар тог елемента се односи само на *HTML* елемент са идентификационом вриједношћу која одговара називу тог *CSS* елемента.

Како су веб прегледници задужени за приказ веб странице, неријетко се дешава да се исти *CSS* стил другачије приказује на различитом прегледнику што представља основни проблем код развоја веб апликација.

### *JavaScript*

*JavaScript* интерпретациони програмски језик који се у потпуности извршава на клијентској страни, унутар веб прегледника. У комбинацији са *HTML* -ом језиком и *CSS* -ом, JavaScript чини *DHTML* (Динамички *HTML*).

Осим за развој веб апликација, *JavaScript* се користи и у изради десктоп виџета, веб читача за специфичне веб сајтове као и у *PDF* документима, а нове технологије као што је *Node.js* користе *JavaScript* програмски језик на серверској страни.

Неке од главних предности *JavaScript* језика, кориштеног на клијентској страни, су:

* Смањење промета ка серверу, могуће је провјерити валидност података на клијентској страни прије слања серверу чиме се смањује и потреба за комуникацијом са сервером.
* Одзив у реалном времену, није потребно поновно учитавање странице како би се промијениио приказ странице.
* Могуће је препознати акције корисника (нпр. Клик миша) те креирати „реакције“ веб странице, тиме се уводи интерактивност странице са корисником.
* *JavaScript* омогућава рад са колачићима, често захтјевани подаци се могу сачувати на корисничком рачунару у виду колачића. Нпр. уколико веб апликација посједује могућност промјене језика, одабрани језик се може сачувати на корисничком рачунару у виду колачића, те при сљедећем покретању веб апликације садржај се приказује на претходно одабраном језику.

*JavaScript* посједује и одређена ограничења по чему се разликује од осталих програмских језика. Нека од основних ограничења су:

* Из сигурносних разлога, *JavaScript* нема могућност писања и читања датотека на клијентском рачунару.
* Не постоји подршка за мрежне апликације
* Вишенитно и више-процесорско извршавање није могуће вршити кориштењем *JavaScript* програмског језика

Структуру података *JavaScript* програмског језика чине примитивни типови и објекти.

Примитивни типови:

* Бројеви
  + Све бројне вриједности су представљене као реални бројеви (64 бита).
  + +/- *Infinity* број се генерише као резултат прекорачења
  + 0/ негативна нула (енг. *Negative zero*) се генерише као резултат поткорачења
  + *NaN* се генерише као резултат дијељења нуле нулом *0/0*, сва остала дијељења нулом не резултују грешком
* Стрингови
  + Промјењиве секвенце 16-битних вриједности.
  + Означавају се једноструким или двоструким наводницима
  + Индексирање стрингова креће од нуле
  + Стринг може садржати и *HTML* кодове, као и специјалне ознаке (нпр. \t –табуларатор, \n – нови ред, итд.)
* Логички (булов) тип
* Специјалне вриједности
  + *Null* представља недостајућу вриједност
  + Недефинисана вриједност (енг. *Undefined*), добија се као повратна врједност непостојећег објекта или функције која нема повратну вриједност, за разилику од *Null* који је објекат, недефинисана вриједност представља тип.

За разлику од примитивних типова, објекти су динамичке природе и преносе се по референци, а не по вриједности. При сваком покретању интерпретатора креира се глобални објекат, на глобалном ниву кориштењем резервисане ријечи *this* може се позвати објекат.

Неки од битнијих објеката су:

* Функције
* Низови (енг. *Array*)
* Датум (енг. *Date*)
* Објекти за рад са регуларним изразима (енг. *RegEx*)
* Грешка (енг. *Error*)

## Асинхрони *JavaScript*

Асинхрони *JavaScript* у комбинацији са *XML*-ом односно *JSON*-ом чини *AJAX*, групу за динамичко креирање страница. Кориштењем *AJAX* -а побољшава се корисничко искуство, његова главна особина јесте динамичко освјежавање (динамичка промјена) одређеног дијела веб странице без потребе за поновним учитавањем исте.

Начин рада *AJAX* -а је приказан на слици 5.



Слика 6 Начин рада АЈАКС-а [6]

Асинхроно понашање подразумијева да након интеракције корисника са интерфејсом, *JavaScript* генерише *XMLHttpRequest* који се просљеђује серверу приказујући резултате када они буду расположиви (Након одговора сервера), а у међувремену корисник може несметано да настави кориштење веб апликације.

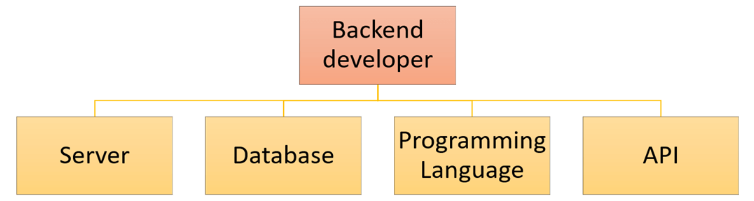
## Технологије серверске стране (*Back-end*)

Уградњом логике управљања веб страницом на серверској страни, веб страница постаје веб апликација. За разлику од тхенологија на клијентској страни, *Back-end* технологије се извршавају на серверској машини и потпуно су невидљиве крајњем кориснику.

Сва пословна логика и рад са подацима се дешавају на серверској страни, али такође серверска страна мора обавјештавати крајњег корисника о резултатима обраде тих података али никако о начину функционисања пословне логике.

Да би се успоставила функционална веза корисничке апликације, серверске апликације и базе података могуће је користити велики број различитих технологија као што су *PHP*, *Ruby*, *Python*, *Java*, .*NET* за развој апликације на серверској страни и алате као што су *MySQL*, *MSSQL*, *Oracle* и многе друге за чување, мијењање и проналажење података потребних за приказ на корисничкој апликацији.

Шта се све подразумјева *back-end* развојем је приказано на слици 7.



Слика 7 Технологије серверске стране

## Архитектура веб апликација

Архитектура веб апликација се првенствено огледа у раздвајању развоја клијентске стране (енг. *Front-end*) и развоја серверске стране (енг. *Back-end*).

У зависности од технологија које се користе за развој, и клијентска и серверска страна могу имати своју архитектуру.

### Архитектура клијентске стране

Један од најзаступљенијих дизајн шаблона који се користе у развоју клијентске стране јесте *MVVM* шаблон (енг. *Modev View ViewModel*).

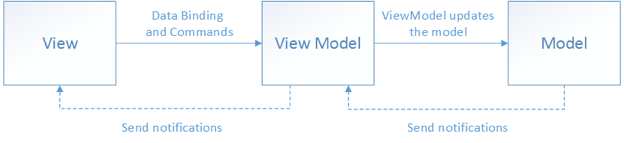
*MVVM* је техника програмирања која прикупља захтјеве извора и корисника и синхронизује их. Знатно смањује количину кода који је потребан за класичну синхронизацију код Моде-Поглед технике.

Поглед (енг. *View*) представља структуру изгледа корисничке апликације и извршава логику корисничког интерфејса.

*ViewModel* је задужен за обавјештавање погледа о промјенама, а поглед одлучује да ли ће те промјене приказати или не.

Модел (енг. *Model*) није видљив кориснику али садржи податке који се путем погледа приказују кориснику.

Начин функционисања *MVVM* шаблона је дат на слици 8.



Слика 8 Начин функционисања *МВВМ* шаблона [7]

Са слике се може закључити да *View* зна за постојање *ViewModel-a*, а *ViewModel* зна за постојање *Model-a*, обратно не важи.

Уколико дође до неке промјене у моделу, *ViewModel* добија обавјештење о тој промјени, а затим *View* добија обавјештење да се *ViewModel* промијенио. Као крајњи корак *View* одлучује да ли ће приказати промјене које су се десиле у моделу.

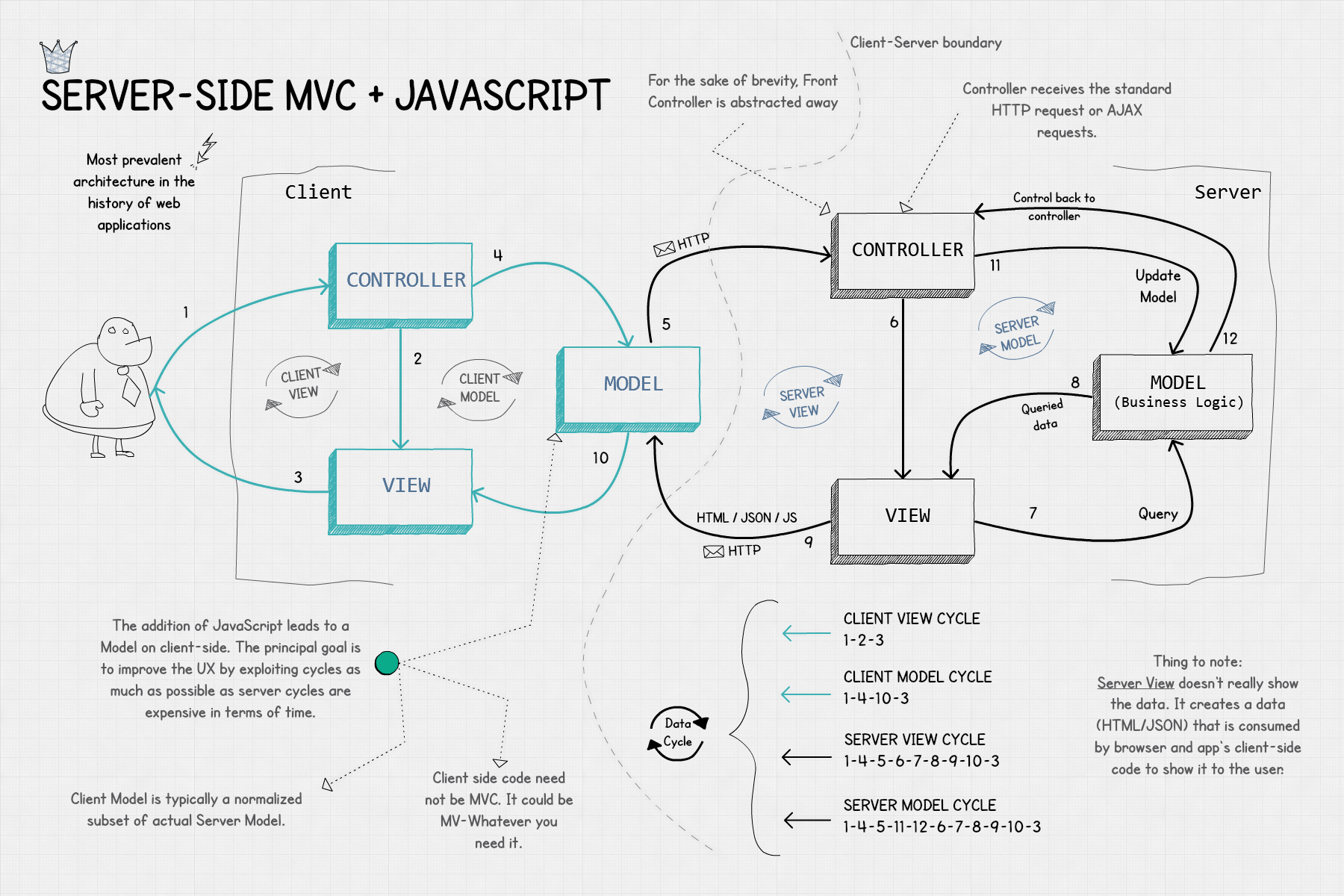
### Архитектура серверске стране

*MVC* дизајн шаблон је један од најпознатијих дизајн шаблона, али неријетко долази до забуне када је ријеч о десктоп *MVC* и веб *MVC* дизајн шаблону.

Када је ријеч о *MVC* дизајн шаблону приликом развоја веб апликација постоје три различите имплементације *MVC* шаблона, а то су:

* *MVC* на корисничкој страни
* *MVC* веб прегледника
  + Програмер не мора водити рачуна о *MVC* дизајн шаблону веб прегледника
* *MVC* на серверској страни
  + Уколико се користи *MVC* дизајн шаблон на серверској страни није неопходно користити *MVC* шаблон и на корисничкој страни, већ је могуће користити било који други дизајн шаблон за развој корисничке апликације (нпр. може се користити *MVVM* дизајн шаблон).

Примјена *MVC* дизајн шаблона на серверској страни је приказанa на слици 9.



Слика 9 Примјена *MVC* дизајн шаблона на серверској страни [8]

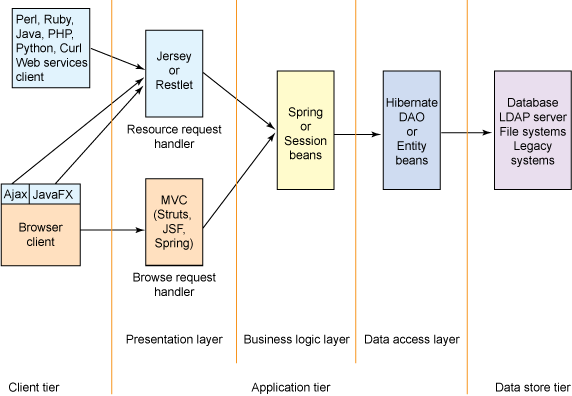
*MVC* шаблон који се примјењује приликом развоја веб апликација је јако сличан класичном *MVC* шаблону за десктоп апликације, али десктоп *MVC* шаблон посједује два циклуса података док веб *MVC* шаблон посједује три, односно четири уколико и корисничка апликација за приказ посједује модел.

Контролер серверске стране служи за пријем *HTTP/HTTPS* захтјева од клијента и о томе обавјештава модел и приказ на серверској страни. Сва логика обраде захтјева као и обраћање бази, у колико је то потребно, врши се у моделу, а поглед (енг. *View*) не служи за стварни приказ података на серверској страни, већ за генерисање података (Обично *HTML* или *JSON* формат података) који се прослијеђују корисничкој веб апликацији која те податке приказује на адекватан начин.

### Вишеслојна архитектура и *RestFul* веб сервиси

Приликом развоја веб апликација, осим логичког раздвајања корисничке апликације (енг. *Front-end*) од серверске апликације (енг. *Back-end*), могуће је извршити и физичко раздвајање. Такође одвајање пословне логике од приступа подацима укида везу зависности базе података од саме пословне логике, што резултује добијањем могућности повезивања различитих база података на исту серверску апликацију.

Дијаграм вишеслојне архитектуре је дат на слици 10.



Слика 10 Дијаграм вишеслојне архитектуре[9]

Приликом обраћања апликационом слоју, клијенски слој шаље *stateless* захтјев са свим потребним инфромацијама. Комуникацја између клијентског и апликационог слоја се одвија тако да је сваки захтјев неовисан од претходног захтјева, тиме се губи зависност између захтјева и одговора.

*HTML* за приказ корисничког интерфејса се добија као одговор презентационог слоја. Унутар презентационог слоја *Browser Requester Handler* је задужен за прихватање свих захтјева веб прегледника (Приликом интеракције корисника), након пријема правилног захтјева презентациони слој се обраћа бизнис слоју, након што добије одговор од бизнис слоја, презентациони слој обрађује тај одговор те га просљеђује клијентском слоју.

Задатак бизнис слоја јесте размјена података између презентационог слоја и слоја за приступ подацима. Сва пословна логика се налази унутар бизнис слоја. Прикупљени подаци из слоја приступа подацима се презентационом слоју шаљу у облику домен објеката или у облику објеката вриједности. Раздвајање презентационог слоја од бизнис слоја доводи до флексибилне и прошириве архитектуре. Уколико се јави потреба за промјеном презентационог слоја то се може урадити без мјењања бизнис слоја и обрнуто.

Разлог раздвајања бизнис слоја од слоја приступа подацима је сличан разлогу раздвајања бизнис слоја од презентационог слоја. Слој приступа подацима повећава укупну флексибилност система, отвара могућност интеграције различитих система и различитих база података у зависности од потреба корисника.

# РАД СА БАЗОМ ПОДАТАКА

База података је свако прикупљање података или информација посебно организованих за брзу претрагу и проналажење кориштењем рачунара. Базе података су структуриране тако да олакшавају складиштење, проналажење, измјену и брисање података, а омогућавају и различите операције над подацима. Систем управљања базом података (енг. *DBMS, Database managment system*) добија информације из базе података као одговор на упите.

База података је датотека или скуп датотека, а подаци унутар датотека се могу рашчланити на записе, а сваки запис се може састојати од једног или више поља за складиштење информација. Записи су организовани у табеле које имају информације о односима између различитих поља.

Постоји више техника за организацију података унутар базе података, а за коју технику се треба одлучити зависи од самих потреба система. На примјер, уколико је потребно смањити појаве сувишних или дуплих података користи се техника нормализације података. Међутим када постоји велики број информација којима се треба лако и брзо приступати користе се технике мултидимензионалних табела.

Дакле, прије креирања шеме базе података потребно је препознати приоритете и на основу тога одабрати адекватну технику која ће резултовати најбољим перформансама.

## Релационе базе података

Релациона база података је скуп података са унапријед дефинисаним односима између њих. Подаци су организовани у скуп табела, а табеле се састоје од редова и колона. Свака колона у табели садржи одређену врсту података, а поље чува стварну вредност. Редови у табели представљају колекцију повезаних вриједности једног објекта или цјелине. Сваки ред у табели може бити означен јединственим идентификатором који се назива примарни кључ, а редови у више табела могу се међусобно повезати помоћу страних кључева. Овим подацима се може приступити на много различитих начина без потребе за реорганизациом самих табела базе података.

Релационе базе имају широку употребу, користе их организације свих врста и величина за најразличитије потребе за информацијама. Користе се складиштење података, обраду трансакција е-трговине, управљање великим количинама осјетљивих информација о купцима итд.

Конзистентност података је једна од основних особина релационих база података. Релационе базе омогућавају да више истанци базе података у сваком тренутку имају исте податке. Друге врсте база података тешко могу да одрже овај ниво правовремене усклађености са великим количинама података.

Често постоји потреба за вишеструким приступом истим подацима, што доводи до проблема вишеструког понављања истог дијела кода у различитим тачкама система у којима су потребне те информације. Релационе базе овај проблем рјешавају увођењем ускладиштених процедура (енг. *Stored procedures*), најједноставније речено то су функције над базом које се могу једноставно позивати из апликације. Осим смањења понављања кода унутар апликација, ускладиштене процедуре уводе и одређени ниво енкапсулације и тиме повећавају ниво заштите података.

Један од проблема код база података јесте паралелни приступ подацима, често се дешава да више корисника покуша измјенити исте податке. У случају паралелног приступа подацима, да би се избјегли конфликти и одржала конзистентност података, уводи се техника закључавања (енг. *DBMS Concurrency Control*). Ове технике смањују могућност појаве конфликата, те одржавају интегритет података. Закључавање спрјечава друге кориснике и апликације да приступе подацима док се они ажурирају. У неким базама података закључавање се односи на цијелу табелу, што ствара негативан утицај на перформансе апликације. Друге базе података, као што су *Oracle* релационе базе података, примјењују закључавање на нивоу записа, остављајући остале записе у табели на располагању, тиме пружају боље перформансе апликације.

## *NоSQL* базе података

За разлику од релационих база података, нералационе базе (познатије као *NoSQL*) податке не чувају у табелама. Нерелационе базе података, на основу модела података, разликују више типова.

Основни типови нерелационих база података су:

* Кључ-вриједност (енг. *Key-value*)
  + База података кључ-вриједност, повезује вриједност (која може бити било која, од броја или једноставног низа, до сложеног објекта) са кључем који се користи за праћење објекта.
* Документ (енг. *Document*)
  + Документна база података је модеран начин за складиштење података у *JSON* формату, за разлику од чувања у редовима и колонама. Омогућава чување података у њиховом природном облику.
* График (енг. *Graph*)
  + Граф орјентисана документна база података омогућава руковање великим скупом структурисаних, полуструктурисаних и неструтурисаних података. Има особину да интегрише хетерогене податке из различитих извора и успостави везе између скупова података. Помаже организацијама да приступе, интегришу и анализирају податке из различитих извора, помажући им тако у анализирању велике количине података.

Особине које нерелационих база података су:

* Флексибилност
  + Нерелационе базе пружају флексибилне шеме које омогућавају бржи и итеративнији развој, флексибилан модел података чини нерелационе базе идеалним за полуструктурисане или неструктурисане податке. За разлику од релационих база података гдје подаци за упис морају бити јасно дефинисани, морају одговарати особинама табеле у коју се уписију, нерелационе базе података не воде рачуна о томе.
* Прилагодљивост
* Добре перформансе
  + Прилагођеност специфичним типовима модела података резултује бољим перформансама него код релационих база података за исте моделе.

Већина нерелационих база података нуди концепт „Релативне досљедности“, гдје се подаци у чворовима базе мијењају у милисекундама те упити за подацима можда неће одмах вратити ажурне податке[10].

Иако имају широку примјену и у посљедње вријеме добијају на популарности, посебно захваљујући развоју *IoT* технологија, нерелационе базе података не гарантују интегритет података, нерјетко долази и до губитка истих.

# РАДНИ ОКВИРИ

Радни оквир је платформа за развој апликација. Представља основу на којој програмери могу развијати апликације. Радни оквир може садржати унапријед дефинисане класе и функције које се могу користити за обраду уноса, управљање хардверским уређајима и интеракцију са системским софтвером, што поједностављује и убрзава развојни процес.

Већином системи имају интегрисану подрушку за радни оквир, без потребе за инсталирањем додатних софтвера односно пакета. Међутим, неке апликације захтјевају одређени радни оквир да би се могле покренути. На примјер, поједине апликације прављене за *Windows* оперативни систем захтјевају *.NET* радни оквир, те је исти потребно инсталирати да би се апликација могла покренути.

## *ASP.NET CORE*

*ASP.NET CORE* је мултиплатформски веб радни оквир који је креирао *Microsoft* за израду веб апликација, *API-a* и микро-сервиса. Користи познате шаблоне као што су *MVC*, *Dependency injeciont* и *request pipeline*. Радни је оквир отвореног кода (енг. *Open-source*) под лиценцом *Apache* 2.0, што значи да је бесплатан за кориштење.

Развој *ASP.NET* апликација је могућ на више различитих језика као што су *C#, Visual Basic, F#*, а покреће се на најновијој *Microsoft* *runtime* библиотеци која је слична *Java* виртуелној машини или *Ruby* интерпретеру[11].

Иако већ постоји велики број различитих радних оквира особине које *ASP.NET CORE* радни оквир издвајају од осталих су:

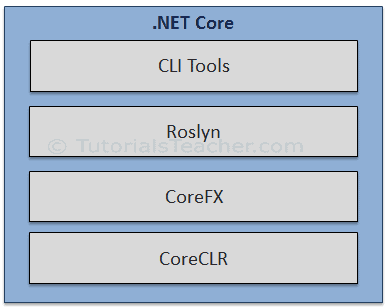
* Брзина рада
  + За разлику од интерпретерских језика, као што су нпр. *JavaScript* или *Ruby*, код .*NET*-а се компајлира, што знатно утиче на брзину рада. Асинхроно извршавање задатака такође битно утиче на брзину рада (Може бити 5 до 10 пута брже него код *Node.js*).
* Екосистем
  + Иако је *CORE* радни оквир прилично нов, .*NET* постоји већ дуже вријеме, тако да постоји велики број *NuGet* пакета као што су пакети за *JSON* десеријализацију, генерисање *PDF* докумената, конектори за базу података и многи други.
* Сигурност
  + На сигурности *ASP.NET CORE* је рађено од самог почетка. Посједује велики број алата и библиотека за побољшање сигурности апликација. На једноставан начин рукује „тајнама“ унутар апликације, чува осјетљиве податке без излагања коду.

Историја *ASP.NET CORE* је приказана у табели 1.

| Version | Latest Version | Visual Studio | Release Date | End of Support |
| --- | --- | --- | --- | --- |
| .NET 5 | Preview 1 | VS 2019 | 16th March, 2020 |  |
| .NET Core 3.x - latest | 3.1.3 | VS 209 | 24th March, 2020 | 12th March, 2022 |
| .NET Core 2.x | 2.1.17 | VS 2017, 2019 | 24th March, 2020 | 21st August, 2021 |
| .NET Core 1.x | 1.1.13 | VS 2017 | 14th May, 2019 | 27th May, 2019 |

Табела 1 Историја .*NET* CORE радног оквира

Састоји се од четири основне компоненте дате на слици 11.



Слика 11 Архитектура .*NET* *CORE* радног оквира[12]

*CLI Tools* представља скуп алата који служе за развој софтвера.

*Roslyn* је програмски преводилац за *C#* и *Visual Basic* програмксе језике.

*CoreFX* представља скуп билбиотека радног оквира.

*CoreCLR* је извршни механизам (енг. *execution engine*).

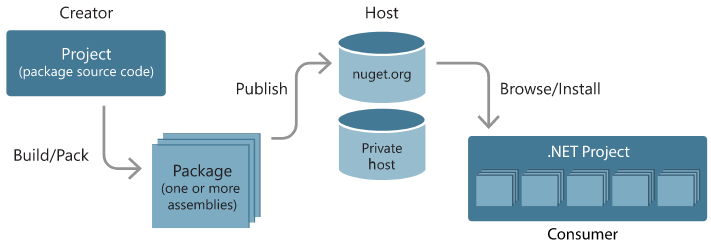
### *NuGet*

У модерном процесу развоја софтвера као есенцијална потреба јавља се потреба за механизмом који омогућава развој, дијељење и кориштење кода. *Microsoft* је ово реализовао креирањем *NuGet* механизма.

У основи *NuGet* пакет је *ZIP* датотека са екстензијом .*nupkg* и у себи садржи компајлиран код, датотеке повезане са тим кодом и описни манифест фајл који посједује информације попут верзије пакета. Кодови се могу објављивати на привати или јавни хост, а корисници пакете могу преузети са одговарајућег хоста, а затим користити функционалности преузетог пакета у свом коду. Процес постављања и преузимања кодова контролише *NuGet*.

С обзиром да *NuGet* подржава приватне хостове заједно са јавним *nuget.org* хостом, он пружа могућност кориштења *NuGet* пакета у сврху дијљења кода унутар неке оргранизације или радне групе.[13]

Ток пакета од хоста ка серверу, па до корисника дат је на слици 12.



Слика 12 Ток *NuGet* пакета[13]

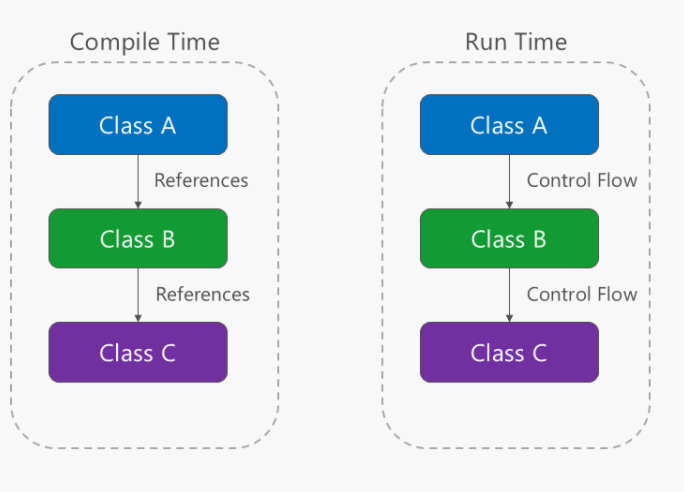
Хост сервер, јавни или приватни, служи као повезна тачка креатора и корисника. Креатор прави *NuGet* пакете и поставља их на хост сервер, а клијент контактира сервер како би преузео пакете. Када је у питању приватни хост сервер, креатор има потпуну контролу над пакетима, те може тачно одредити који пакети су доступни одређеној групи корисника. Након инсталирања *NuGet* пакета у пројекту, сви *API-*ји тог пакета постају доступни у читавом пројекту.

### Уметање зависности

Уметање зависности (енг. *Dependency injection*) је дизајн шаблон у коме се класа односно објекат убацује у своје зависне класе. Ова техника се користи у циљу постизања инвертовања контроле (енг. *Inversion of Control*) између класа.

У пракси апликације се развијају по принципу да класе виших нивоа зависе од класа нижих нивоа, нпр. класа задужена за бизнис логику зависи од класе која приступа бази података. Због те зависности често долази до проблема када се врше измјене у класи нижег нивоа, неријетко је потребно измјенити класу вишег нивоа која зависи од измјењене класе, а понекад то утиче на низ класа вишег нивоа.

Примјер директне зависности објеката је дат на слици 13.

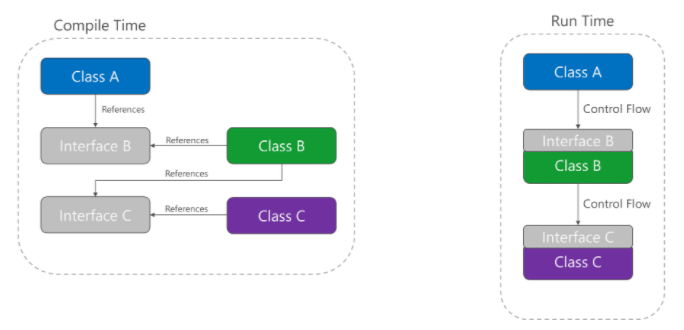


Слика 13 Примјер директне зависности објеката.

Инвертовање контроле у суштини представља технику гдје објекти не креирају објекте од којих зависе, већ те објекте добијају екстерно (обично путем конструтора као параметар).

Дакле, уметање зависноти значи да креирање зависних објеката врши радни оквир обично као параметар у конструктору или помоћу *get/set* метода.

Примјер инвертоване контроле тока је дат на слици 14.



Слика 14 Примјер инвертоване контроле тока

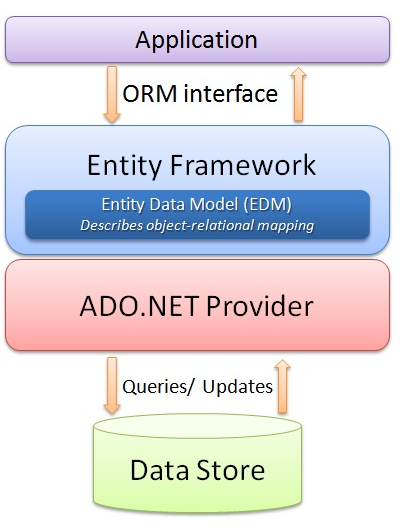
Као што се види на слици, класе односно објекти вишег нивоа не зависе директно од класа нижег нивоа, оба модула зависе од апстракције, такође апстракција не зависи од детаља, већ детаљи зависе од апстракције.

## *Entity*

У ранијим верзијама (прије верзије 3.5) *.NET* радног оквира приступ бази података и рад над базом података био је компликованији и подложнији грешкама. Морала су се водити рачуна о повезивању на базу података и креирању модела за читање и упис података. Такође након читања података из базе било је потребно те податке адекватно пресликати у одговарајуће моделе за обраду података у бизнис слоју и обрнуто при успису базу података.

Овај проблем рјешава *Entity* радни оквир. Он представља ОРМ (Објектно-релационо мапирање) радни оквир који нуди аутоматизовани механизам прикупљања података и чувања података у базу података. Омогућава рад са табелама базе података у облику .*NET* објеката, те елиминише потребу за писањем кода за приступ бази података.

*Entity* радни оквир се налази између слоја базе података и апликационог слоја односно бизнис слоја (Слика број 15).



Слика 15 Позиција *Entity* радног оквира у апликацији

Са слике се види да је *Entity* посредник у процесу размјене података између апликације и базе података.

Осим што је мултиплатформски радни оквир, карактеристике *Entity* радног оквира су:

* Моделовање
  + Креира *EDM* (*Entity Data Model*), *POCO* (*Plain Old CLR Object*) базиране, ентитете са *get/set* методама за различите типове података. Користи овај модел у комуникацији са базом података.
* Упити
  + Користи *LINQ* упите код прибављања података из базе података, а провајдер базе података преводи *LINQ* упите у одговарајуће *SQL* упите.
  + Оставља могућност директног писања *SQL* упита за рад са базом података.
* Праћење измјена
  + Води рачуна о свим промјенама, унутар ентитета, које се касније морају сачувати у бази података.
* Штедња
  + *INSERT, UPDATE,* и *DELETE* команде се извршавају тек након позива *SaveChanges()* методе.
  + Посједује и асинхрону *SaveChangesAsync()* методу.
* Конкурентност
  + Користи „оптимистичну“ конкурентност, спрјечава пребрисавање измјенњених података све док корисних не заврши са радом над тим подацима.
* Трансакције
  + Води рачуна о трансакцијама приликом рада са подацима.
  + Нуди и могћност измјене трансакционог процеса уколико је то потребно.
* Кеширање
  + Упити који се понављају више пута враћају резултате из кеш меморије умјесто из базе података директно.
* Уграђене конвенције
  + Прати уграђене конвенције програмирања и аутоматски подешава *EF* модел у складу са тим конвенцијама.
* Подешавања
  + Омогућава измјену *EF* модела (Модела по уграђеним конвенцијама) кориштењем анотационих атрибута или *Fluent API-а*.
* Миграције
  + Нуди скуп команди за креирање и или измјену базе података.

### Историја *Entity* радног оквира

Прва верзија *Entity 6* радног оквира појавила се 2008. године са *.NET* 3.5 радним оквиром (Табела број 2), а у јуну 2016. године објављена је *Entity Core* верзија која се користи у развоју *.NET Core* апликација (Табела број 3).

| EF Version | Release Year | .NET Framework |
| --- | --- | --- |
| EF 6 | 2013 | .NET 4.0 & .NET 4.5, VS 2012 |
| EF 5 | 2012 | .NET 4.0, VS 2012 |
| EF 4.3 | 2011 | .NET 4.0, VS 2012 |
| EF 4.0 | 2010 | .NET 4.0, VS 2010 |
| EF 1.0 (or 3.5) | 2008 | .NET 3.5 SP1, VS 2008 |

Табела 2 Историјски развој *EF 6* радног оквира

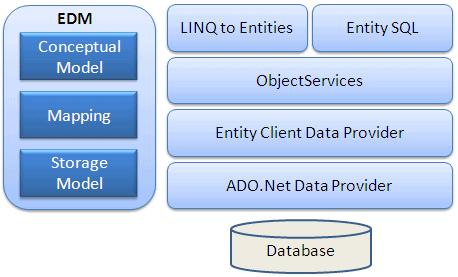
| EF Core Version | Release Date | .NET Framework |
| --- | --- | --- |
| EF Core 2.0 | August 2017 | .NET Core 2.0, VS 2017 |
| EF Core 1.1 | November 2016 | .NET Core 1.1 |
| EF Core 1.0 | June 2016 | .NET Core 1.0 |

Табела 3 Историјски развој *EF Core* радног оквира

### Архитектура

Компоненте *Entity* радног оквира су (Слика број 16):

* *Entity* модел података (енг. *Entity Data Model*) који садржи:
  + Концептуални модел
    - Садржи класе модела и везе између њих независно од дизајна базе података.
  + Модел за складиштење је дизајн базе података и садржи:
    - Табеле
    - Погледе
    - Ускладиштене процедуре
    - Везе и кључеве
  + Пресликавања
    - Садрже инфромације на који начин концептуални модел, треба да се преслика у модел за складиштење.
* *LINQ to Entities*
  + Користи се за писање упита према објектном моделу. Враћа ентитете дефинисане у концептуалном моделу.
* *Entity SQL*
  + Користи се за писање упита према објектном моделу али само за *EF 6*.
* Објект сервис
  + Главна тачка размјене података између базе података и апликације.
* *Entity Client Data Provider*
  + Главни задатак ове компоненте јесте превођење *LINQ-to-Entities* или *Entity SQL* упита у упите *SQL* језика. Комуницира са *ADO.Net data provider-ом* који шаље или прикупља податке из базе података.
* *ADO.Net Data Provider*
  + Слој за комуникацију са базом података.

**

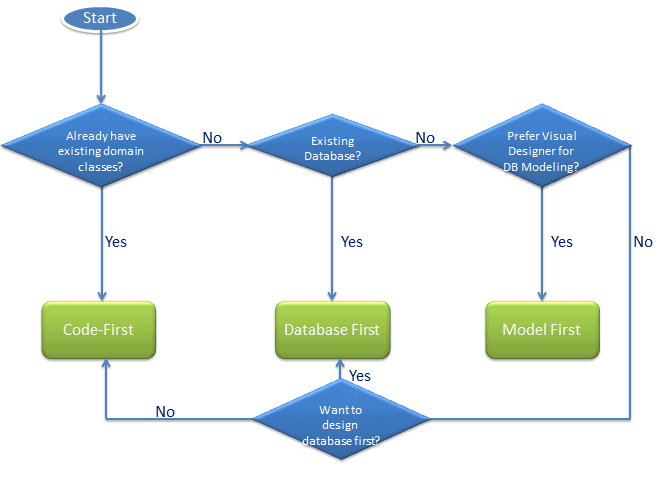
Слика 16 Архитектура *Entity* радног оквира

### Приступи развоју

Постоје три различита приступа развоју:

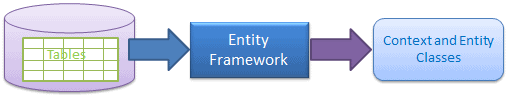
* База прво (енг. *Database-First*)
* Код прво (енг. *Code-First*)
* Модел прво (енг. *Model-First*)

За који од приступа се одлучити зависи од архитектуре читаве апликације, приоритета и од ресурса. Дијаграм који може помоћи при одлучивању је дат на слици број 17.[14]



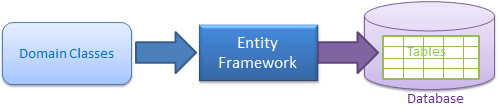
Слика 17 Дијаграм одлучивања код развоја базе података (*EF*) [14]

„База прво“: контекст и ентитети се генеришу на основу већ постојеће базе података (Слика број 18).



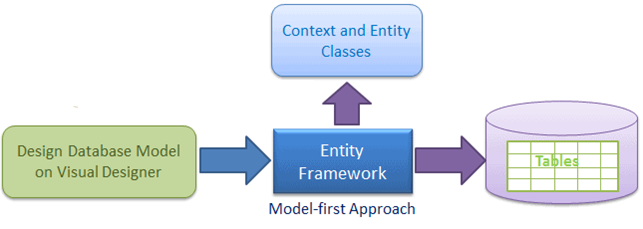
Слика 18 Приступ „База прво“ [14]

„Код прво“: база података се генерише на основу већ постојећих доменских класа и контекстне класе (Слика број 19).



Слика 19 Приступ „Код прво“ [14]

„Модел прво“: контекст, ентитети и база података се генеришу на основу модела базе података (Слика број 20).

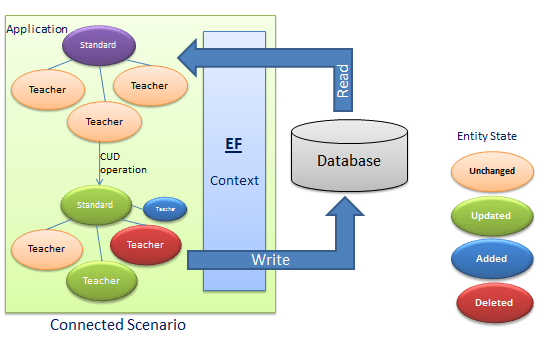


Слика 20 Приступ „Модел прво“ [14]

### Сценарији развоја

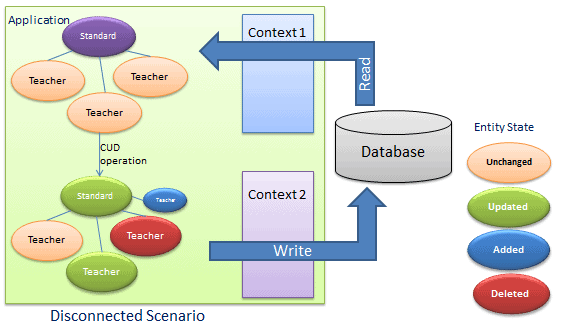
У зависности од надлежности контекстне класе постоје повезани сценарио (енг. *Connected Scenario*) и неповезани сценарио (енг. *Disconnected Scenario*).

Код повезаног сценарија контекстна класа служи и за читање и за уписивање (чување) података. Прати измјене свих ентитета током читавог њиховог животног циклуса (Слика број 21). Овај сценарио је користан ако се користи локална база података или база података која је на истој мрежи.



Слика 21 Повезани сценарио[14]

Неповезани сценарио користи двије различите истанце контекстне класе за читање и за уписивање података. Након читања података, контекстна класа се уништава, а креира се нова истанца контекстне класе за уписивање података (Слика број 21). Пошто истанца контекстне класе не води рачуна о ентитетима потребно је поставити одговарајуће стање ентитета, кориштењем методе *SaveChanges()*, прије уписа у базу података.



Слика 22 Неповезани сценарио[14]

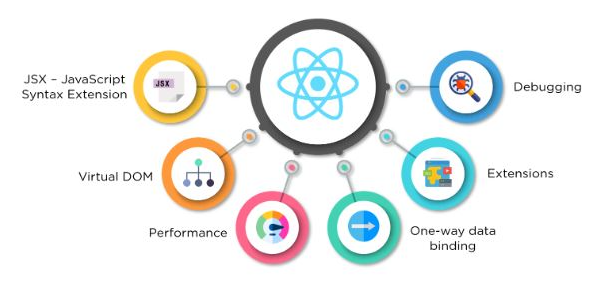
## *React*

*React* или *React.js* је на компонентама базирана *JavaScript* библиотека отвореног кода. Користи се за израду брзог и интерактивног корисничког интерфејса за веб и мобилне апликације.

Неки од разлога зашто ова библиотека има већу популарност него већина других радних оквира за израду корисничког интерфејса су:

* Лака израда динамичких апликација
* Побољшане перформансе
* Поновно искористиве компоненте
* Ток података у једном смјеру
* Лако за учење
* Може се користити за развој и веб и мобилних апликација
* Постоје алатке намјењене за проналажење грешака

Карактеристике *React* библиотеке су приказане на слици 23.



Слика 23 Карактеристике *React* библиотеке

*JSX* је синтаксно проширење на *JavaScript* које служи за опис изгледа корисничког интерфејса. *JSX* омогућава писање *HTML* структуре у истој датотеци као и *JavaScript* код и тиме олакшава процес отклањања грешака и смањује комплексност кода.

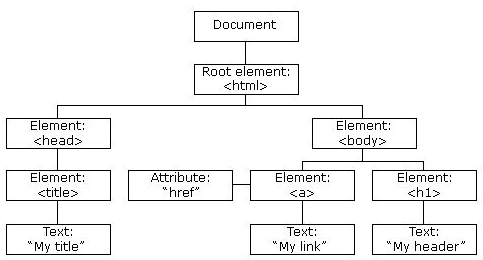
Примјер *JSX-a* је приказан у кодном листингу 5.

<h5 id='productTittle'> {product.tittle}</h5>

                            <img className='CartProdImg' src={product.photo}></img>

Кодни листинг 5 Примјер *JSX* екстензије

*React* чува приказ „стварног“ *DOM-a* у меморији, то је познатије као *VDOM* (*Virtual DOM*). Када се стање једног објекта промијени *VDOM* мијења само тај објекат унутар „стварног“ *DOM-a* умјесто да ажурира све објекте.



Слика 24 *DOM* веб странице[15]

*DOM* (*Document Object Model*) посматра *HTML* или *XML* документ као стабло, а чворови стабла су елементи документа.

Када се стање неког објекта у *React* апликацији промијени, *VDOM* се ажурира, а затим врши поређење тренутног стања са претходним и у „стварном“ *DOM-*у ажурира само измјењене елементе. Ова техника знатно убрзава приказ веб апликације јер ажурира само објекте чије се стање промијенило.

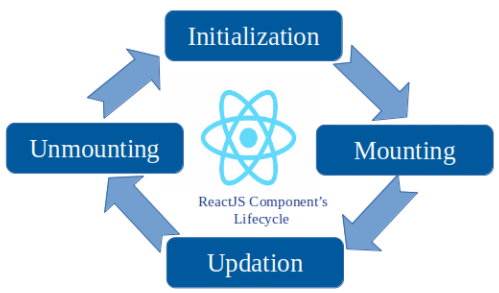
*React* посједује мноштво проширења. Омогућава приказивање на страни сервера (енг. *Server-side rendering* ), страница се обрађује на серверској страни и шаље се кориснику као потпуно приказана страница.

### Животни циклуси

Постоје различите методе животног циклуса које *React* пружа у различитим фазама живота компоненте. *React* аутоматски позива одговарајућу методу на основу фазе у којој се налази компонента. Ове методе пружају бољу контролу компоненте.

Животни циклуси компоненте (Слика 25) су:

* Иницијализација (енг. *Initialization*)
* Уградња (енг. *Mounting*)
* Ажурирање (енг. *Updating*)
* Уклањање (енг. *Unmounting*)



Слика 25 Животни циклуси компоненте

Иницијализација компоненте подразумјева постављање стања и својстава компоненте (Обично путем конструктора).

У фази уградње компонента се уграђује у *DOM* и приказује по први пут. У овој фази су доступне двије методе:

* *componentWillMount()* након позива ове методе компонента се уграђује у *DOM.*
* *componentDidMount()* метода се позива након што је компонента уграђена у *DOM.*

Након што је компонента уграђена у *DOM* почиње трећа фаза, фаза ажурирања. У овој фази се компоненте, као одговор на интеракцију корисника, ажурирају и поновно приказују у складу са промјеном стања њиховог стања.

Методе које су доступне у овој фази су:

* *shouldComponentUpdate()* метода одлучујуе да ли се нека компонента треба ажурирати или не.
* *componentWillUpdate()* метода се позива након *shouldComponentUpdate()* методе и извршава се одмах прије поновног приказа компоненте.
* *ComponentDidUpdate()* метода се позива након што је завршен поновни приказ компоненте.

Посљедња фаза у животном циклусу је уклањање компоненте из *DOM*-a и посједује само једну методу *componentWillUnmount()* која се позива прије уклањања компоненте и тиме се завршава животни циклус једне компоненте.

### Flux и Redux

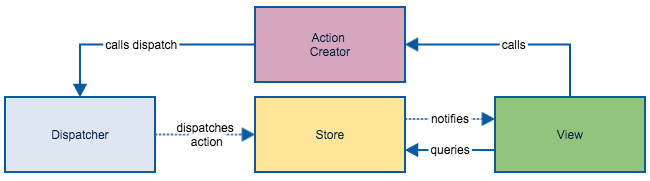
*Flux*  је архитектонски дизајн шаблон који је *Facebook* направио за развој *React*  апликација. Представља малу измјену *observer* дизајн шаблона, није ни библиотека ни радни оквир, и заснива се на концепту једносмјерног тока података.

*Redux* је библиотека заснована на *Flux* шаблону, може се сматрати примјеном *Flux* шаблона, олакшава контролу стања апликације и рукује приказом података при интеракцији корисника са апликацијом.

Сам по себи *React* није веома скалабилан те приликом развоја великих апликација тешко је водити рачуна о стањима свих компоненти. Главни проблем је уска повезаност корисничких акција и промјена стања компоненти. Код сложених апликација често једна корисничка акција може утицати на низ промјена стања. Без *Flux*-a или *Redux*-а ово постаје веома сложен процес.

*Flux* дизајн шаблон се састоји од четири сегмента:

* *Action*
* *Dispatcher*
* *Store*
* *View*

****

Слика 26 Сегменти *Flux* дизајн шаблона

Треба да постоји само један *Dispatcher* сегмент, а може постојати више *Store* и *Action* сегмената. Акције се заснивају на функционалностима апликације, а на основу акција се мијења приказ апликације према слици 26.

Код *Redux*-а *Dispatcher* је замијењен са редукторима (енг. *Reducers*), а такође не постоји више *Store* сегмента, већ након сваке акције старо складиште се копира, примјењују се измјене и поставља се ново складиште. Сегменти *Redux* шаблона су дати на слици 27.



Слика 27 Сегменти *Redux* шаблона

### Историја

Прву верзију *React* библиотеке*,* под називом *FaxJS,* направио је *Jordan Walke*, инжењер у *Facebook* компанији, мотивисан *XHP* компонентном библиотеком за *PHP.* Први пут *React* библиотека је кориштена на *Facebook* платформи 2011. године, а касније и на *Instagram* платформи 2012. године. На *JSConf US* конференцији 2013. године је званично постала библиотека отвореног кода.

*React Native* радни оквир, за развој мобилних апликација, први пут је објављен на *Facebook*-овој *React Conf* конференцији у фебруару 2015. године, а званично је постао радни оквир отвореног кода у марту те исте године.

Од 2013. године *Facebook* ради у континуитету на развоју *React* библиотеке.

## *Bootstrap*

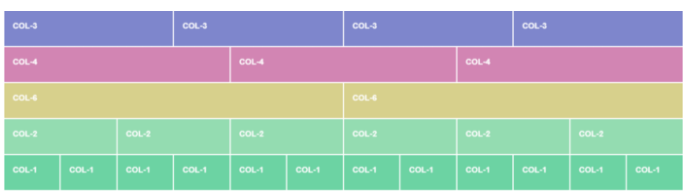
*Bootstrap* најпопуларнији *HTML*, *CSS*, и *JavaScript* радни оквир намјењен за развој респонзивних веб апликација. Радни оквир је отвореног кода који је креирао *Twitter*, а хостован је на *GitHub* платформи.

Структура *Bootstrap*-а је прилично једноставна, садржи прекомпајлиране кодове који се лако користе у било ком веб пројекту.

Мрежни систем *Bootstrap*-а је врло честа техника која се користи за креирање веб корисничког интерфејса. Омогућава да се кориснички интерфејс адекватно прилагоди величини екрана уређаја који приступа апликацији.

Мрежни систем верзије *Boorstrap-*а 4 омогућава подјелу реда на дванаест колона исте ширине. Ред се може подијелити на колоне различитих ширина, све док је упупни збир колона дванаест.

На примјер можемо подијелити један ред на четири колоне, три колоне, двије колоне итд. што је приказано на слици 28.



Слика 28 Мрежни систем *Bootstrap-a*

Постоји пет класа којима се може дефинисати респонзивност елемента у зависности од величине екрана.

* .*col-\** користи се за екране уже од 576 пилсела
* .*col-sm-\** користи се за екране шире од 576 пиксела
* .*col-md-\** користи се за екране шире од 768 пиксела
* .*col-lg-\** користи се за екране шире од 992 пиксела
* .*col-xl-\** користи се за екране шире од 1200 пиксела

# ПРАКТИЧНА РЕАЛИЗАЦИЈА ПРОЈЕКТА

# ЗАКЉУЧАК

У закључку треба истаћи постигнуте резултате и правце будућег рада.

ЛИТЕРАТУРА

[1] Lambert, Laura (2005). *The Internet: A Historical Encyclopedia : Chronology. Vol. 3, Volume 3*. Santa Barbara, CA: ABC-CLIO. p. 100. [ISBN](https://en.wikipedia.org/wiki/ISBN_(identifier)) [978-1851096596](https://en.wikipedia.org/wiki/Special:BookSources/978-1851096596)

[2] Botha, J.; Bothma, C.; Geldenhuys, P. (2008). *Managing E-commerce in Business*. Cape Town: Juta and Company Ltd. p. 3. [ISBN](https://en.wikipedia.org/wiki/ISBN_(identifier)) [9780702173042](https://en.wikipedia.org/wiki/Special:BookSources/9780702173042).

[3] Zinkan, George (2011). *Advertising Research: The Internet, Consumer Behavior, and Strategy*. Chicago: American Marketing Association. p. 33. [ISBN](https://en.wikipedia.org/wiki/ISBN_(identifier)) [9781613112717](https://en.wikipedia.org/wiki/Special:BookSources/9781613112717).

[4] „ПМФ, Ниш - Увод у веб програмирање: HTML“, др Млан Б. Тасић

[5] http://poincare.matf.bg.ac.rs/~andjelkaz/pzv/cas\_js/js.pdf

[6] <https://www.w3schools.com/js/js_ajax_intro.asp>

[7][https://medium.com/hongbeomi-dev](https://medium.com/hongbeomi-dev/create-android-app-with-mvvm-pattern-simply-using-android-architecture-component-529d983eaabe)

[8] <https://blog.webf.zone/contemporary-front-end-architectures-fb5b500b0231>

[9] <https://www.ibm.com/developerworks/web/library/wa-aj-multitier/>

[10] "Jepsen: MongoDB stale reads"

[11] Nate Barbettini, The little ASP.NET Core book

[12] <https://www.tutorialsteacher.com/core/dotnet-core>

[13] <https://docs.microsoft.com/en-us/nuget>

[14] <https://www.entityframeworktutorial.net>

[15] https://www.simplilearn.com