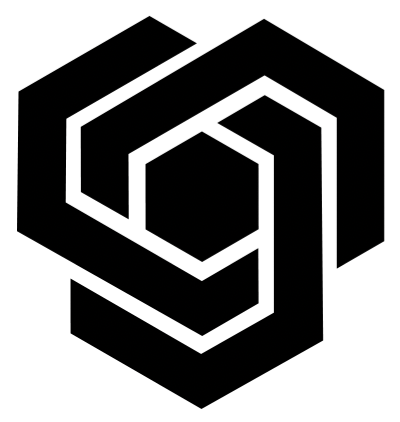
Технически университет – София

Факултет по Компютърни Системи и Технологии



Дипломна работа

Проектиране и разработка на прототип на уеб-базиран туристически наръчник

Изготвил:  
Андреа Кръстев Филчев – фак.No 121213134

Научен ръководител:

доц. д-р И. Николова

София

2020

# **Увод**

Уеб-базираните приложения са все по-често срещани във нашето всекидневие. Всеки компютър и мобилен телефон разполага с уеб-браузър, доставящ интерактивна среда с множество възможности. Модерният уеб-браузър бива постоянно оптимизиран, има достъп до множество сензори и периферни устройства, и разбира се до Интернет. Сферите, в които уеб-браузърът се използва стават все повече - статични уеб-страници, игри, онлайн пазаруване, социални мрежи, виртуална реалност и т.н. В днешно време уеб-браузърът се е превърнал в платформа, а не просто интерфейс.

С тази промяна идват и много възможности. Настоящата дипломна работа е фокусирана върху някои от тези възможности, включвайки решения, които, макар и да не са популярни, допринасят за различна гледна точка и евентуално подобрение спрямо наличните методи. Целта на уеб-приложението е да се реализира туристически наръчник с разпознаване на забележителности в браузъра. Приложението е разделено на две части. Първата част е интерактивна карта с туристически забележителности, за които потребителя може да получи повече информация. Втората част е разпознаването на забележителности чрез използване на снимки или камерата при мобилните устройства. За имплементирането на приложението ще се запознаем със структурата и начина на работа на уеб-браузъра, неговите предимства и недостатъци.

# Глава I

Тази глава разглежда основните езици и процеси необходими за работата на уеб-браузъра, както и алтернативи за евентуалното им подобряване.

# Стандартни Уеб Езици

Всяка една уеб-страница включва в себе си няколко стандартни файлове, помагащи за получаването на организирана структурата:

* HTML
* CSS
* JavaScript

## 1.1 HTML (Hypertext Markup Language)

Още от първите дни на Интернет, браузъри използват HTML, като най-основният градивен елемент. HTML **не е програмен език, а маркиращ език** използван, за да дефинира структурата на Web-страниците, които потребителя посещава. Състои се от серия елементи, информиращи браузъра за специфичният тип данни, които се съдържа в тях.

**Пример 1.1.1 HTML код**

<!DOCTYPE html>

<**html** lang="en">

<**head**>

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

<**meta** name="viewport" content="width=device-width, initial-scale=1.0">

<**title**>Document</**title**>

</**head**>

<**body**>

<**h1**>This is the Main Heading</**h1**>

<**p**>This text might be an introduction to the rest of

the page. And if the page is a long one it might

be split up into several sub-headings.

<**p**>

<**h2**>Another Sub-Heading</**h2**>

</**body**>

</**html**>

Всеки елемент започва с отварящ маркер и завършва със затварящ маркер. Между двата маркера се намира съдържанието на елемента. В отварящият маркер също така може да има и атрибути, даващи допълнителна информация.

**Пример 2.1.2 HTML елемент**

Съдържание

<p id="my-paragraph"> Hello World </p>

Отварящ маркер

Затварящ маркер

## 1.2 CSS (Cascading Style Sheets)

Докато HTML дефинира структурата и семантиката на съдържанието на Web-страниците, CSS се използва за стилистичната им обработка и/или подредба. Чрез него разработчика селективно може да променя стилът на всеки един елемент. Прилагането на промени може да стане по няколко различни начина:

* External Stylesheet (Външно) – чрез използването на отделен файл с разширение .**css** , в който дефинираме стилистичните промени.
* Internal Stylesheet (Вътрешно глобално) – чрез <**style**> елемент най-често позициониран в <**head**> елемента, в който дефинираме стилистичните промени
* Internal (Вътрешно) – чрез <**style**> елемент за всеки елемент отделно

CSS се използва чрез дефиниране на правила, които трябва да се приложат към специфичен елемент или група от елементи. Всяко правило започва със **селектор**. Негово предназначение е да избере съществуващ HTML елемент, който да променим.

Селектора е последван от двойка къдрави скоби **{ }**. Вътре в скобите се добавят една или повече декларации, които имат отношение **свойство/стойност** (property/value).

**Пример 3.2.1 CSS правило**

Стойност

h1 {

color: red;

font-size: 5em;

}

Селектор

Свойство

## 1.3 JavaScript

JavaScript e динамичен междуплатформен програмен език, който може да придаде интерактивност към Web-страницата. Освен в браузъра, може да се използва и в сървър-среда използвайки Node.js.

JavaScript комбинира процедурното, функционалното и обектно-ориентираното програмиране парадигми с лесен за употреба синтаксис. За разлика от C++, JavaScript e интерпретируем език, а не компилиращ се. При компилиращите езици кодът минава през компилатор, който го превежда до машинен код. При използването на JavaScript в Web-браузър, кодът минава през интерпретатор, или JIT(just-in-time) компилатор. Интерпретатора чете и превежда кода ред по ред. JIT комбинира скоростта на компилиран код, с гъвкавостта на интерпретатора.

Някои от най-честите задачи на JavaScript са:

* Добавянето/Премахването на HTML към страницата, промяна на съществуващите елементи, промяна на стила на елементите.
* Реагиране към събития – движение на мишката, натискане на клавиши, взаимодействието с форми, и т.н.
* Пращане на заявки към сървъри, обработката на върнатия отговор, свалянето и качването на файлове
* Записването, премахването и четенето на данни в локалната памет
* Създаване на 3D графики

## Critical Rendering Path

Преди браузърът да изобрази Web-страница, е необходимо да се изгради **DOM** (Document Object Model), **CSSOM** (CSS Object Model) и “**render tree**”.

## DOM (Document Object Model)

DOM има дървовидна структура. Всеки елемент, атрибут и текст от HTML файла e репрезентиран чрез DOM възли. Като всеки възел е дефиниран спрямо своята връзка със другите DOM възли.

**Пример 2.1.1 Създаване на DOM**

<!DOCTYPE html>

<**html**>

<**head**>

<**link** href="style.css" rel="stylesheet">

<**title**>Critical Path</**title**>

</**head**>

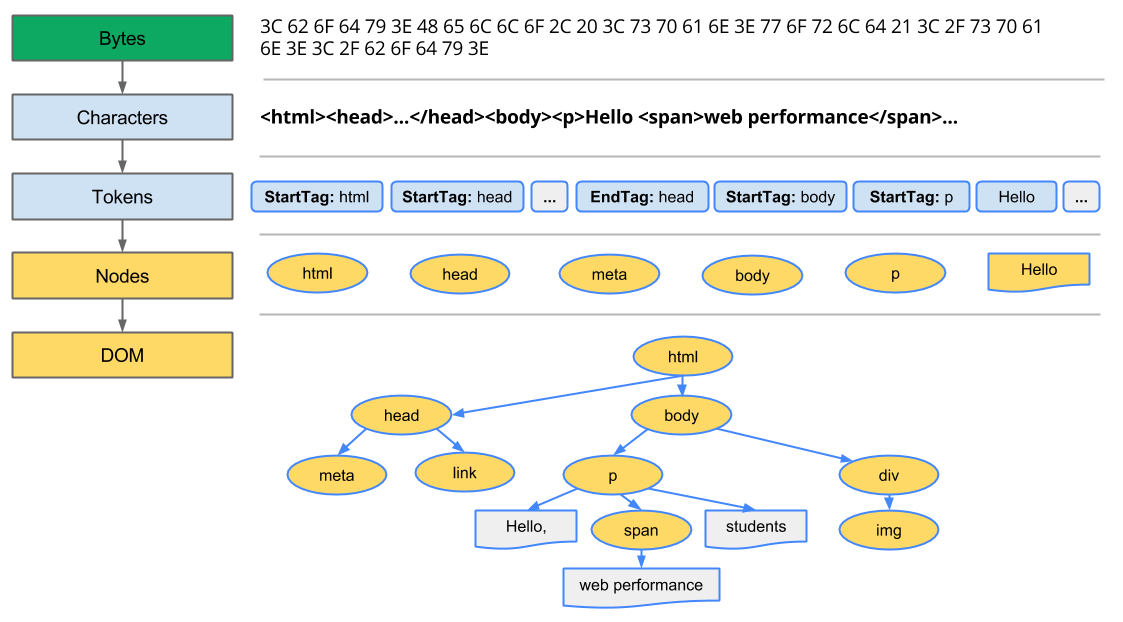
<**body**>

<**p**>Hello <**span**>web performance</**span**> students!</**p**>

<**div**><img src="awesome-photo.jpg"></**div**>

</**body**>

</**html**>



Процеса по създаването на DOM дървото се състои от 4 стъпки:

1. **Conversion** - четенето на HTLM байтове от диск или мрежа, и преобразуването им в индивидуални символи.
2. **Tokenizing** – конвертирането на низове в *token*. Като всеки *token* има свое специфично значение и собствен набор от правила за работа.
3. **Lexing** – Вече създадените *tokens* се конвертират в „обекти“, като се дефинират техните програмни свойства и правила.
4. **DOM конструиране** – Последната стъпка, е създаването на връзките между отделните обекти. Понеже HTML дефинира връзките между маркерите като дървовидна структура от данни, обектите също биват свързани със същите връзки от тип „родител-дете“.

## CSSOM (CSS Object Model)

По време на създаването на DOM дървото, браузърът следи за наличието на **stylesheet**. При наличието на **stylesheet**, браузъра трябва конвертира CSS правилата до CSSOM модел. Процеса е подобен на създаването на DOM дърво.

span

body

p

span

img

**DOM**

**CSSOM**

font-size: 16px;

font-weight: bold;

font-size: 16px;

font-weight: bold;

display: none;

font-size: 16px;

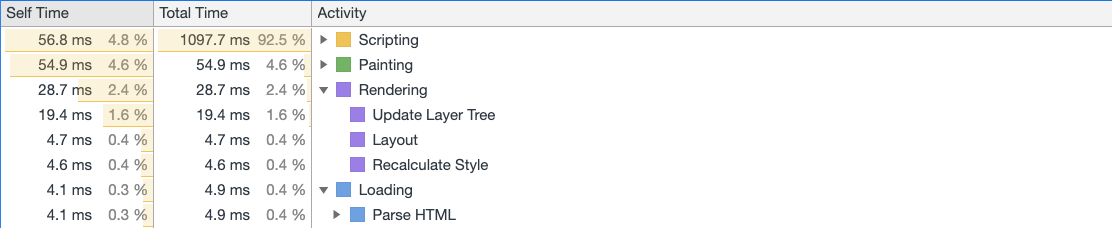
float: right;

**Пример 2.2.1 Създаване на CSSОМ**

Причината CSSOM да има дървовидна структура е, за да може след изчисляването на последните CSS правила, браузърът да започне прилагането им от най-общите към най-специфичните. Например, ако даден елемент е дете на <**body**> елемента, в такъв случай всички стилистични правила на <**body**> елемента се прилагат и към детето. Именно заради това се казва **Cascading** Style Sheets.

Можем да видим колко време отнема прилагането на всеки един от процесите по създаването на DOM и CCSOM в Developer Tools на всеки един браузър.

**Пример 2.2.2 Необходимо време за създаване на DOM и CSSOM в Google Chrome**

**

Създаване на DOM

Създаване на CSSOM

## Render tree

След създаването на DOM и CSSOM, двете структури се комбинират в “**render tree**”, което се използва за изчисляването на **layout** (оформлението) на всеки елемент. За конструирането на “**render tree**” браузърът преминава през 3 стъпки:

1. Започвайки от корена на DOM дървото, се преминава през всеки елемент.
   1. Има елементи, които не са видими (като например <**body**> маркера) и те не се добавят.
   2. Има елементи, които не са видими заради CSS правила (като например **display: none**), които също се не добавят.
2. За всеки елемент се намира съвпадащият CSS селектор и дефинираните правила, които се прилагат към него.
3. Като резултат се връщат всички видими елементи с приложени им CSS правила.

С вече конструираното “**render tree**” се преминава към “**layout**” фазата, която има за цел да изчисли позицията, на която ще бъде нарисуван всеки отделен елемент и с какъв размер. Резултатът от “**layout**” фазата се използва за рисуването на изчислените елементи. Тази стъпка се нарича “**painting**” или „**rasterizing**”. Операцията за рисуване на елементите върху екрана е една от най-бавните, и трябва да се вземе в предвид, за да се подобри интерактивността на Web-приложението.

## Elm и Virtual DOM

В тази глава ще разгледаме алтернативa на работата със DOM елементи наречена **Virtual DOM**, както и програмни езици като **Elm**, които се компилират до JavaScript.

## Virtual DOM

Въпреки че, DOM дървото прави работата с елементи лесна, операциите включващи директни манипулации върху дървото често се оказват много бавни.

За да видим разликите ще разгледаме два различни начина за промяна на DOM дървото.

В първият пример обновяваме низ 100 000 пъти, без да изразяваме промяната в DOM дървото.

**Пример 3.1.1 Код за намирането на продължителността за променяне на низ**

***let*** *testString = "test string";*

***function*** *updateString() {*

*for (****let*** *i = 0; i <= 100000; i++) {*

*testString = "Update testString";*

*}*

*}*

***let*** *t1 = performance.now();*

*updateString();*

***let*** *t2 = performance.now();*

*console.log("Време за изпълнение: " + (t2 - t1) + " milliseconds");*

Във вторият пример обновяваме низ, който е „дете“ на <div> елемент, 100 000 пъти.

**Пример 3.1.1 Код за намирането на продължителността за променяне на низ намиращ се в <div> елемент**

newDiv.appendChild(testText);

document.body.appendChild(newDiv);

**function** updateDОМ() {

for (**let** i = 0; i <= 100000; i++) {

newDiv.innerHTML = "updated testText";

}

}

**let** t3 = performance.now();

updateDОМ();

**let** t4 = performance.now();

console.log("Време за изпълнение: " + (t4 - t3) + " milliseconds");

Представеният код е изпълнен върху машина с 4.2 GHz Intel Core i7, 16GB памет и Google Chrome 83. Резултатите са следните:

**Пример 3.1.1 Времена за променяне на низ**



Нека разгледаме резултатите, за да намерим обяснение:

1. Промяната на низ е проста операция. JavaScript знае къде е съхранен низът в паметта на компютъра, така че единственото което трябва да направи е да пренапише битове в паметта.
2. Промянаta на низ в <**div**> елемент изисква повече работа. В този случай променяме не само низът в паметта, а и DOM и CSSOM дърветата. Това означава, че всяка промяна изисква преминаването през **CRP** (Critical Rendering Path):

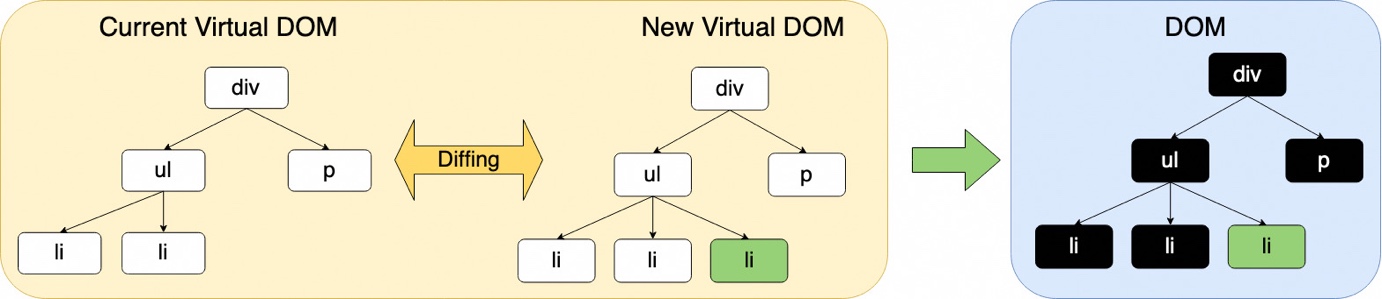
В зависимост от това колко сложна е промяната, която се прави, може да се наложи браузъра да изтрие и пресъздаде съществуващите връзки на елементите, които трябва да се обновят. В последствие, ще трябва да се изчислят наново позициите и размерите на пресъздадените елементи, както и съществуващите елементи засегнати от промяната. В даденият пример, промяната не е сложна, и въпреки това отнема **280 пъти** повече време, за да се извърши.

Тук идва Virtual DOM. Най-просто казано, Virtual DOM представлява опростенарепрезентация на потребителският интерфейс, която се съхранява в паметта. Това означава, че всеки Virtual DOM елемент има същите свойства като реален DOM елемент, но не може директно да въздейства върху изобразяването му на екрана. При промяна на Virtual DOM елемент, се променя единствено репрезентацията на елемента в паметта, без да се обновява DOM дървото, “render” дървото, без да се извиква “layout” фазата и без да се изрисуват промените на екрана. Използвайки този начин за манипулиране на DOM дървото носи доста оптимизации.

* Обновяването на елементи може да се случва на партиди, избягвайки скъпи и излишни пречертавания, преизчислявания на позициите и размерите на елементите при всяка промяна.
* Състоянието на приложението е по-малко вероятно да бъде извън синхрон от интерфейсът.

Оптимизирането продължава с още две операции на Virtual DOM:

* **diffing** – това е процесът отговорен за това как да бъде обновено реалното DOM дърво при промяна. Това се постига чрез създаване на нов Virtual DOM инстанция при всяка промяна. След това се прави сравнение между ***сегашната*** и ***новата*** версия на виртуалният DOM и се търсят разликите. След като всички разлики са намерени, те се изпращат на партида като инструкции за модифицирането на реалното DOM дърво, използвайки функцията **requestAnimationFrame()**.



* **lazy** – ако се знае, че има елементи които няма да се променят, то тези елементи могат да се маркират и да бъдат пропуснати при **diffing** процеса.

## Elm

В днешно софтуерните програмисти и архитекти се целят в това да разработват системи, които са както лесни за ползване, така и надеждни. Например извън Web-браузърът съществуват езици от **ML фамилията** (“Meta Language”), като например **Haskell**, **OCaml** и **F#,** които са известни със своята надеждност. Компилаторите на тези езици могат да намерят почти всички грешки, които иначе биват пропускани. Но тези езици се използват предимно в *back-end* системи, работещи на сървър. За разлика от тях *front-end* системите, или клиентските системи работят в браузър. А браузърът разбира единствено JavaScript. Именно затова най-популярните платформи, като **React.js** и **Angular**, използват JavaScript. Но в последните години се хвърлят доста усилия, за да се донесе надеждността на ML езиците в Web-браузъра. Един от тези езици е **Elm**.

**Elm** e *чист* *функционален език* за създаване Web-приложения, който се компилира до JavaScript. Това означава, че притежава няколко черти, които го различават значително от JavaScript :

1. **Чисти функции** - *Функционалното програмиране*, за разлика от *императивното,* на практика не съдържа присвояване на стойности, като по този начин, щом една променлива получи стойност, тя повече не се променя. Това означава, че функционалните програми *нямат странични ефекти*. Извикване на дадена функция не може да има друг ефект, освен изчисляването на резултата. Това спомага за премахването на голям брой грешки, както и се обезсмисля реда на изпълнение. Поради липсата на странични ефекти, няма как да се промени стойността на дадена променлива, съответно няма значение кога ще се изпълни дадена функция.
2. **Частично приложение на функции (Currying)** -Техника за трансформирането на функция с няколко аргумента в последователност от функции, всяка от които приема един аргумент.
3. **Първокласни функции и функции от по-висок ред** - Първокласните функции се приемат за първокласни променливи. Това означава, че функциите както и първокласните променливи могат да бъдат аргументи на функции, да бъдат връщани от функции и могат да бъдат съхранявани в различни структури от данни.
4. **Рекурсия** - Повечето функционални езици не използват *for* и *while цикъл.* Итерациите се извършват чрез рекурсия. Рекурсивните функции извикват себе си, повтаряйки дадена операция докато се достигне базов случай.
5. **Типова система**  - Elm предразполага за използването на типови анотации при дефинирането на функции. При добавянето на типови анотации, се дава допълнителна информация на компилатора, какво точно се опитваме да направим. Това спомага както за четливостта на кода, така и за качеството на съобщенията за грешки, които компилаторът връща.

**Пример 3.2.1 Типова анотация**

**half : Float -> Float**

**half n =**

**n / 2**

Представеният код е функция, която приема един аргумент от тип Float и връща Float.

1. **Справяне с грешки** – Една от гаранциите, които дава Elm е, че на практика не може да има грешки по време изпълнение на програмата. Грешките в Elm са очакван резултат, а не изключение. Това позволява да се промени дизайна на езика и архитектурата, така че да се отрази наличието на грешки в специален тип данни. В Elm има два такива типа:
   1. **Maybe**

**Пример 3.2.2 Maybe тип**

**type Maybe a**

**= Just a**

**| Nothing**

**-- Just 3.14 : Maybe Float**

**-- Just "hi" : Maybe String**

**-- Just True : Maybe Bool**

**-- Nothing : Maybe a**

**Maybe** представлява тип данни с два варианта, като в зависимост от резултата може да съществува само един от двата варианта. Наличието на специален тип за грешки е доста полезен в някои случаи, като например при трансформиращи функции:

**Пример 3.2.3 Трансформиране на String към Float**

> String.toFloat

<function> : String -> Maybe Float

> String.toFloat "3.1415"

**Just** 3.1415 : Maybe Float

> String.toFloat "abc"

**Nothing** : Maybe Float

Функцията **toFloat** от модула **String** е дефинирана да приема аргумент от тип **String** и да връща **Maybe Float**.

* Ако аргумента е **Float**, се връща варианта **Just Float**.
* Ако аргумента не е **Float**, се връща варианта **Nothing**.

Всъщност Maybe може да се разглежда като “контейнер” за единична стойност, която може да бъде или **Nothing** или **Just**.

Друго често приложение на типа Maybe е при опционални полета.

**Пример 3.2.4 Опционални полета с Maybe**

type alias Person =

{ name : String

, age : Maybe Int

, id : Int

}

george : Person

george =

{ name = “George”

, age = Just 21

, id = 012345678

}

pamela : Person

pamela =

{ name = “Pamela”

, age = Nothing

, id = 123456789

}

canDriveCar : Person -> Bool

canDriveCar person =

case person.age of

Nothing ->

False

Just age ->

age >= 18

В *пример 3.2.4* е дефиниран тип **Person**, който съдържа поле **age** от тип **Maybe Int**. След това се създават две инстанции **george** и **pamela** от тип **Person**. С помощта на функцията **canDriveCar** може да се провери дали даденo лице е навършило нужните години, за да кара МПС. Функцията **canDriveCar** приема един аргумент от тип **Person** и връща **Bool**. С помощта на *pattern matching* се “разопакова” **Maybe** полето **age**, за да се върне правилният резултат.

* 1. **Result**

Типът **Result** позволява да се разбере защо има грешка, а не само наличието на такава.

**Пример 3.2.5 Result тип**

**type Result error value**

**= Ok value**

**| Err error**

Едно от най-честите места, при които се използва **Result** типът са **HTTP** заявките. Много могат да бъдат причините, поради които една заявка да не завърши успешно. В Elm, стойността, която се връща при заявка изглежда така:

**Пример 3.2.6 Резултат от HTTP заявка**

**Result Error String**

**type Error**

**= BadUrl String**

**| Timeout**

**| NetworkError**

**| BadStatus Int**

**| BadBody String**

**-- Ok "This is valid" : Result Error String**

**-- Err Timeout : Result Error String**

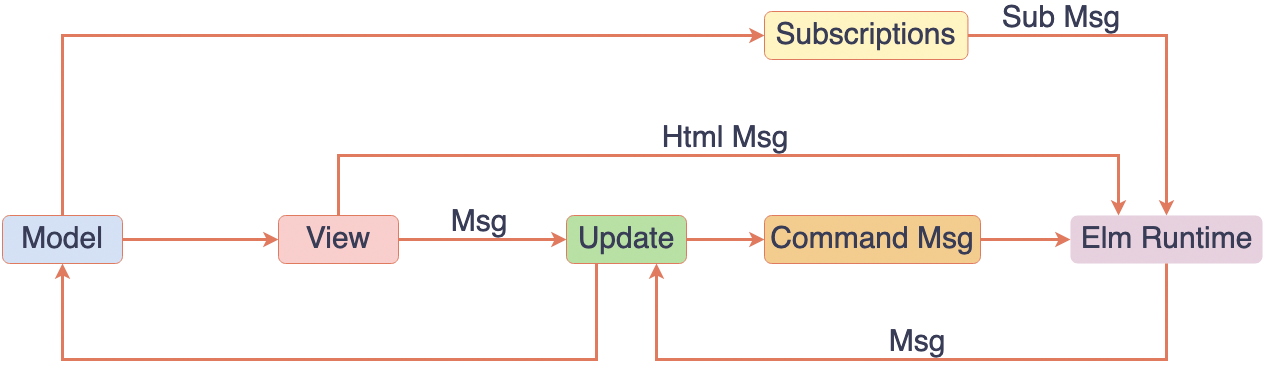
**-- Err NetworkError : Result Error String**

Това позволява по-фин контрол върху изпълнението на програмата, както и за по-информативни съобщения за потребителите.



Elm не е само програмен език, но също така и архитектура. Докато JavaScript предлага различни подходи за контролиране на потокa от данни, при Elm има само един начин. Самата архитектура се състои от три главни части – **Model**, **View** и **Update**.

**Пример 3.2.7 Elm Архитектура**

****

# **Model**

**Model** се използва за съхраняване състоянието на приложението. Първо се дефинира нов комплексен тип данни с помощта на **type alias Model**, в която се описва как ще изглежда структурата на приложението. След това се използва вече създадената структура, за да се създаде първоначален модел.

# **Update**

**Update** се използва за обновяване състоянието на приложението чрез използването на съобщения. Първо се дефинират съобщенията описващи *какво се е случило* с помощта на **type Msg**. Ако е нужно, съобщенията могат да носят информация заедно със себе си. След това се използват вече дефинираните съобщения, за да се обнови моделът. Важно е да се отбележи, че при компилиране се проверява дали са взети в предвид всички възможни съобщения, за да се гарантира правилното изпълнение на програмата.

# **View**

**View** се използва за представянето на състоянието под формата на HTLM. Това става чрез помощта на функцията **view**, която приема **Model** като аргумент и връща **Html Msg**. Това е едно от местата, от които може да се подават съобщения обратно към **Update**.

Примерното течение на поток от данни с Elm Архитектурата би изглеждало по следният начин:

1. Инициализиране на моделът

Първоначална инициализация

1. Извикване на **view** функцията с вече създаденият модел
2. Изчакване на входни съобщения
3. Изпращане на съобщение към **update**
4. Обновяване на моделът

Повтарящ се цикъл

1. Извикване на **view** функцията с вече обновеният модел
2. Показване на новият HTML на екрана
3. **Повтори стъпка 3**

**Пример 3.2.8 Главни части на Elm архитектурата**

**-- MODEL**

type alias Model =

{ count : Int }

init : **Model**

init =

{ count = 0 }

**-- UPDATE**

type Msg

= Increment

| Decrement

update : **Msg** -> **Model** -> (**Model**, **Cmd Msg**)

update msg model =

case msg of

Increment ->

( { model | count = count + **1** }

, Cmd.none

)

Decrement ->

( { model | count = count – **1** }

, Cmd.none

)

**-- VIEW**

view : **Model** -> **Html Msg**

view model =

div []

[ button [ onClick Decrement ] [ text "-" ]

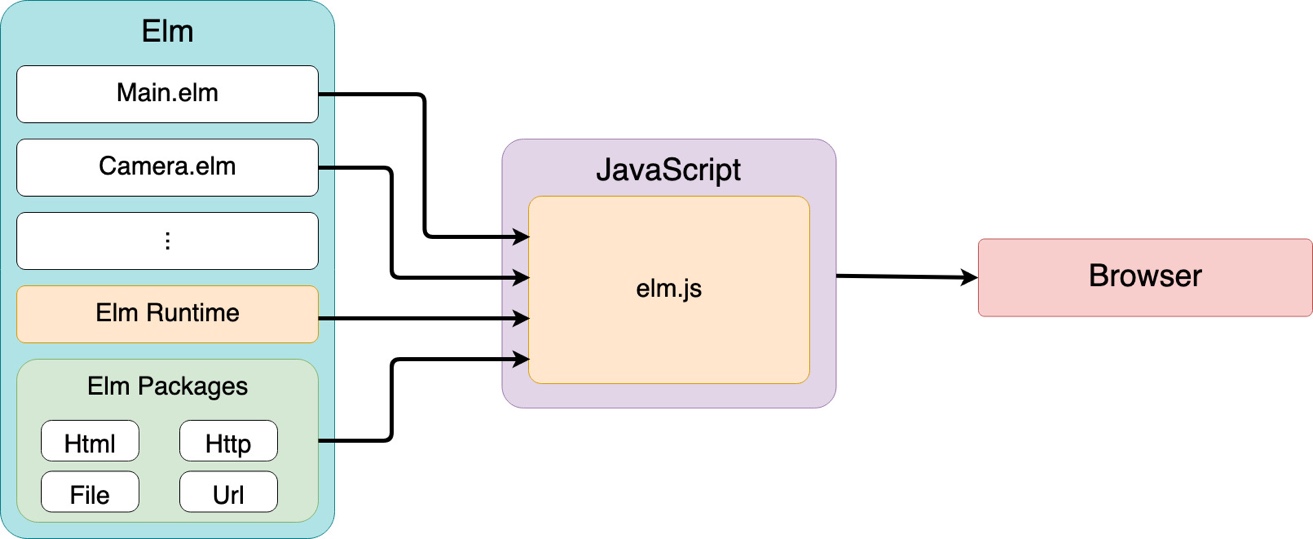
, div [] [ text (String.fromInt model) ]

, button [ onClick Increment ] [ text "+" ]

]

За да може кодът написан на Elm да работи в JavaScript среда, то кодът трябва да бъде компилиран до JavaScript. Това става като се комбинират всички нужни файлове – *програмната логика*, *Elm Runtime* и *Elm пакетите*. При успешно компилиране, изходният файл е с разширение js. След това компилираният файлът се указва в HTML скрипта, който го зарежда в Web-браузърът.

**Пример 3.2.8 Elm Runtime**

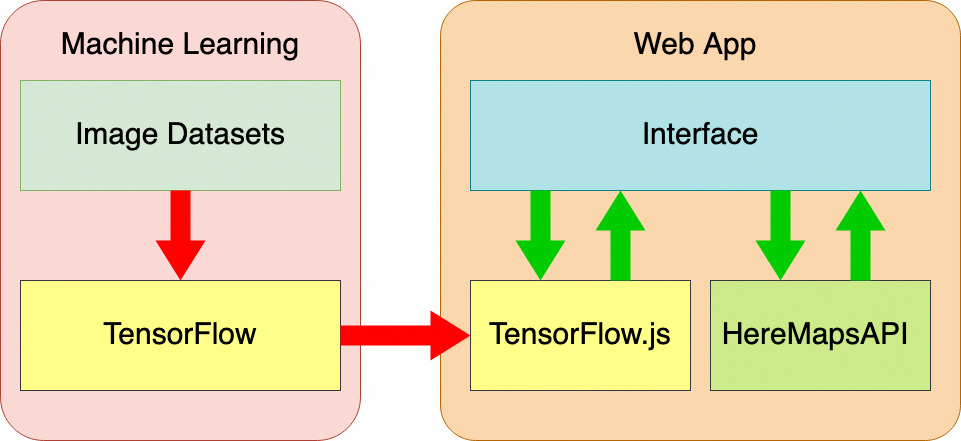


# Глава II

## 1. Архитектура на приложението

Приложението е разделено на три части :

1. Използването на Machine Learning за създаването на модел за разпознаване на забележителности
2. Интерактивна карта със забележителности и информация за тях
3. Уеб-интерфейс, който обединява интерактивната картата и разпознаването на забележителности.



**Фиг. 1.1 Архитектура на приложението**

### Създаване на модел за разпознаване на забележителности с TensorFlow

Разпознаването на обекти е дисциплина от машинното обучение наречена компютърно зрение. В тази дисциплина се обхващат различни проблеми. Те могат да бъдат разделени на няколко поддисциплини :

* Класификация на обекти
* Идентификация на обекти
* Откриване и локализация на обекти
* Сегментиране на обекти
* Разпознаване на пози

За реализацията на текущата работа ще се използва класификация на обекти чрез използването на изображения. Класификацията на изображения представлява задача, при която се определя подходящият клас към когото принадлежи дадено изображение, чрез използването на предефиниран набор от класове.



Класификация

Котки





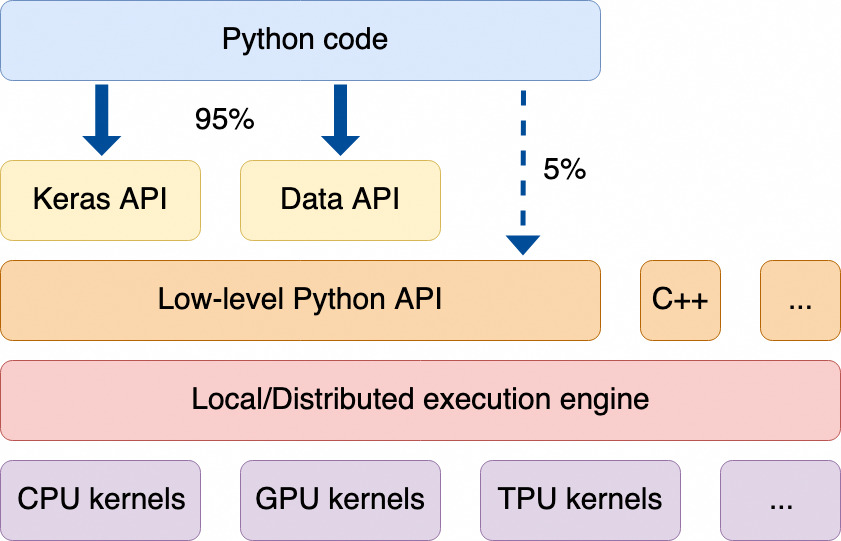
Кучета



**Фиг. 1.2 Класификация на обекти/изображения**

За целта е избрана библиотеката за машинно обучение TensorFlow. Макар да има и други библиотеки за машинно обучение, това което различава TensorFlow e броят програмни среди, в които може да се използва. Към момента TensorFlow може да се използва в Python, JavaScript и C++ среди. Освен това модел създаден от една среда може да се използва и от останалите. Например модел обучен с Python може да бъде използван с JavaScript в Уеб-браузър на Android устройство.

На най-ниско ниво, TensorFlow използва оптимизиран C++ код, като голяма част от операциите имат различни имплементации наречени *kernels*. Всяка от имплементациите е специфично оптимизирана да работи върху различен хардуер – CPU, GPU или TPU(Tensor Processing Unit). Именно с помощта на GPU може да се постигнат големи увеличения в скоростта на трениране, чрез разбиване на изчисленията на по-малки части и паралелното им изпълнение. В днешно време може да се използват и TPU ASIC (Application-specific integrated circuit), които са специфично създадени за Машинно Обучение и съответно са по-ефективни от GPU.



**Фиг. 1.3 Архитектура на TensorFlow**

Преди да стигнем до самото обучаване на модела, имаме нужда от данни, с които да работим. Това е важен аспект, както в компютърното зрение така и за машинното обучение като цяло. Количеството и качеството на данните, с които се обучава моделът е от огромно значение. Затова трябва да вземем в предвид най-често срещаните проблеми при събирането на данни за класифициране на изображения:

* **Брой данни**

За ефективното обучаване на модел е нужен огромен брой от данни. Например един от най-използваните набор от данни използван при обучение за разпознаване на изображения е ImageNet. Към днешна дата в него има 14 милиона изображения разделени в 21 хиляди категории. Това означава, че всяка категория съдържа поне 500 снимки.

* **Вариативност на данните**

Набор от изображения като ImageNet, както и всеки друг набор от данни трябва да се справи и с друг важен проблем, а именно прекомерното нагаждане(overfitting) към данните. Това означава, че изображенията трябва да бъдат колкото може по-разнообразни. Това включва наличието на различни ъгли на снимане, вътрешно-класова вариативност, както и да се вземе в предвид различните размери на изображенията и наличието на оклузии и деформации в изображенията.

В случая, за тренирането на модела са използвани снимки, направени от същата камера в кратък диапазон от деня. Съответно данните не предлагат добра вариативност. За да се намали ефекта от еднаквостта на изображенията са използвани няколко подхода, които се представени по-късно.

Първо започваме със подреждането на изображенията в подходяща структура. В случая подреждаме изображенията на забележителностите в отделни директории, като всяка директория представлява отделен клас.

**photos**

├── alexander\_nevsky

├── banya\_bashi

├── battenberg\_mausoleum

├── monument\_to\_the\_soviet\_army

├── national\_archeological\_museum

├── national\_opera\_and\_ballet

├── paraskeva

├── regional\_history\_museum

├── russian\_church

├── saint\_joseph

├── saint\_sofia

├── saint\_sofia\_monument

├── seven\_saints

├── sofia\_synagogue

├── sofia\_university

├── st\_nedelya\_church

└── tsar\_osvoboditel

**Фиг. 1.4 Структура на директорията със изображения**

След това се прехвърляме в облачната среда **Google Colab**, която предоставя виртуална машина за работа с Python и TensorFlow, както и достъп до висок клас GPU и дори TPU. Използването на GPU/TPU е безплатно, но лимитирано до 12 часа на ден, което е напълно достатъчно за нуждите на текущата дипломна работа.

Преди да започнем можем да проверим какви са характеристиките на предоставената виртуална машина:

**Пример 1.1 Хардуерна характеристика на виртуалната машина в Google Colab**

|  |
| --- |
| #GPU info  !nvidia-smi  #memory that we can use  !free -h --si | awk '/Mem:/{print $2}' |
| +-----------------------------------------------------------------------------+  | NVIDIA-SMI 450.51.05 Driver Version: 418.67 CUDA Version: 10.1 |  |-------------------------------+----------------------+----------------------+  | GPU Name Persistence-M| Bus-Id Disp.A | Volatile Uncorr. ECC |  | Fan Temp Perf Pwr:Usage/Cap| Memory-Usage | GPU-Util Compute M. |  | | | MIG M. |  |===============================+======================+======================|  | 0 Tesla K80 Off | 00000000:00:04.0 Off | 0 |  | N/A 31C P8 28W / 149W | 0MiB / 11441MiB | 0% Default |  | | | ERR! |  +-------------------------------+----------------------+----------------------+  Mem:13G |

Продължаваме с качването на снимките в Google Colab и подготовката на изображенията. След качването на изображенията, можем да инициализираме масив съдържащ класовете. За тази цел можем да използваме имена на директориите съдържащи изображенията.

**Пример 1.2 Инициализиране на класове**

|  |
| --- |
| data\_dir = pathlib.Path(base\_dir\_photos)  classes = np.array([item.name for item in data\_dir.glob('\*')]) |
| ['alexander\_nevsky',  'banya\_bashi',  'battenberg\_mausoleum',  'monument\_to\_the\_soviet\_army',  'national\_archeological\_museum',  'national\_opera\_and\_ballet',  'paraskeva',  'regional\_history\_museum',  'russian\_church',  'saint\_joseph',  'saint\_sophia',  'saint\_sofia\_monument',  'seven\_saints',  'sofia\_synagogue',  'sofia\_university',  'st\_nedelya\_church',  'tsar\_osvoboditel'  ] |

След това разделяме изображенията на три части.

* *Тренировъчен сет* (70% от всички изображения)

Това са изображения върху които се тренират и оптимизират параметрите на модела.

* *Валидиращ сет* (10% от всички изображения)

Това са изображения, с помощта на които се настройват хиперпараметрите на класификатора с цел да се намали прекомерното нагаждане. Важно е да се отбележи, че изображенията от този сет няма да се използват за да се тренира модела.

* *Тестващ сет* (20% от всички изображения)

Това са изображения, с които се прави крайна оценка за точността на модела.

|  |
| --- |
| """  Devides all photos to three folders:  /train - training set - results to 70% of all photos  /val - validation set - results to 10% of all photos  /test - test set - results to 20% of all photos  """  for cl in classes:  img\_path = os.path.join(base\_dir\_photos, cl)  images = glob.glob(img\_path + '/\*.jpeg')  print("{}: {} Images".format(cl, len(images)))  num\_train = int(round(len(images)\*0.7))  val\_num\_train = int(round(len(images)\*0.1)) + num\_train  train, val, test = images[:num\_train], images[num\_train:val\_num\_train], images[val\_num\_train:]  for t in train:  if not os.path.exists(os.path.join(base\_dir, 'train', cl)):  os.makedirs(os.path.join(base\_dir, 'train', cl))  shutil.copy2(t, os.path.join(base\_dir, 'train', cl))  for v in val:  if not os.path.exists(os.path.join(base\_dir, 'val', cl)):  os.makedirs(os.path.join(base\_dir, 'val', cl))  shutil.copy2(v, os.path.join(base\_dir, 'val', cl))  for t in test:  if not os.path.exists(os.path.join(base\_dir, 'test', cl)):  os.makedirs(os.path.join(base\_dir, 'test', cl))  shutil.move(t, os.path.join(base\_dir, 'test', cl)) |

След като изображенията са разделени, започваме да ги подготвяме за тренирането на модела.

Първо дефинираме размерността, с която нашият модел очаква да е входния вектор, за да преоразмерим изображенията от тренировъчният сет.

|  |
| --- |
| IMG\_SHAPE = 224  BATCH\_SIZE = 64 |

Използваме ImageDataGenerator, за да подготвим изображенията за трениране.

|  |
| --- |
| image\_gen\_train = ImageDataGenerator(  rescale=1./255,  rotation\_range=45,  width\_shift\_range=.15,  height\_shift\_range=.15,  horizontal\_flip=True,  zoom\_range=0.5  )  validation\_gen = ImageDataGenerator(rescale=1./255)  test\_gen = ImageDataGenerator(rescale=1./255)  train\_data\_gen =  image\_gen\_train.flow\_from\_directory(  batch\_size=BATCH\_SIZE,  directory=train\_dir,  shuffle=True, target\_size=(IMG\_SHAPE,IMG\_SHAPE),  class\_mode='sparse'  )  val\_data\_gen =  validation\_gen.flow\_from\_directory(  batch\_size=BATCH\_SIZE,  directory=val\_dir,  shuffle=True,  target\_size=(IMG\_SHAPE, IMG\_SHAPE),  class\_mode='sparse'  )  test\_data\_gen =  test\_gen.flow\_from\_directory(  directory=test\_dir,  batch\_size=BATCH\_SIZE,  target\_size=(IMG\_SHAPE, IMG\_SHAPE),  class\_mode="sparse",  save\_to\_dir="/content/test\_augmented"  ) |
| Found 852 images belonging to 17 classes. 🡨 Training set  Found 206 images belonging to 16 classes. 🡨 Validation set  Found 421 images belonging to 17 classes. 🡨 Test set |

Добавят се следните параметри, за да се подобри вариативността на изображенията при тренировъчните изображения:

|  |  |
| --- | --- |
| **rescale** | Нормализиране на размерността на RGB каналите от диапазона  0-255 към 0-1 float |
| **rotation\_range** | Произволно завъртане на снимките до 45° |
| **width\_shift\_range** | Произволно хоризонтално разместване на изображението със специфирана фракция |
| **height\_shift\_range** | Произволно вертикално разместване на изображението със специфирана фракция |
| **horizontal\_flip** | Произволно вертикално разместване на изображението със специфирана фракция |
| **zoom\_range** | Произволно увеличение на изображението |

Преди да продължим правим инспекция на промените, като прегледаме партида от тренировъчните изображения:

|  |
| --- |
| def inspect\_batch(images, labels):  fig = plt.figure(figsize=(40,40))  for n in range(15):  ax = plt.subplot(5,3, n+1)  ax.set\_title(classes[tf.cast(labels[n], tf.int8)], fontsize=40)  plt.imshow(images[n])  plt.axis('off')  plt.subplots\_adjust(hspace=0.5)  image\_batch, label\_batch = next(train\_data\_gen)  inspect\_batch(image\_batch, label\_batch) |
|  |

Нормално обучението на модели за класифициране на изображения става чрез огромен голям набор от тренировъчни данни, за да се постигне по-голяма точност и п