OMO

13 - Patterny pro UI

- Základy UI - HTML, DOM, JavaScript, CSS
- Single Page Application
- Patterny Model View Controller, Model View Presenter, Model View View Model
- Patterny Future/Promise a monáda
- Moderní webové frameworky, React/Redux a Angular

Ing. David Kadleček, PhD
kadlecd@fel.cvut.cz, david.kadlecek@cz.ibm.com

Webový browser

Běží na všech zařízeních - desktopy, tablety, telefony, televize ...

Webový browser 2017

Google Chrome | Firefox | Safari | UC | Iron | Internet Explorer | Opera | Android

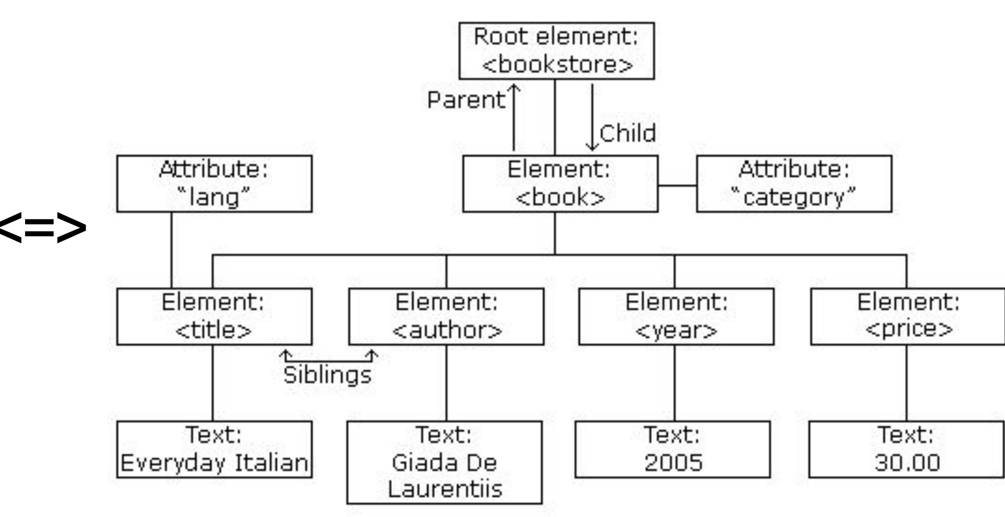
Document Object Model

Document Object Model (**DOM**) je platformě a jazykově nezávislý objektový model a API pro XML dokumenty.

- XML dokument lze převést (*parsovat*) do objektové struktury (*DOMu*), se kterou se výrazně lépe pracuje
- DOM lze převést zpět (serializace) do XML
- DOM mohu modifikovat pomocí API

```xml
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
</bookstore>
```

<=>



HTML

Zobrazený dokument

My First chapter

My Text and My Link

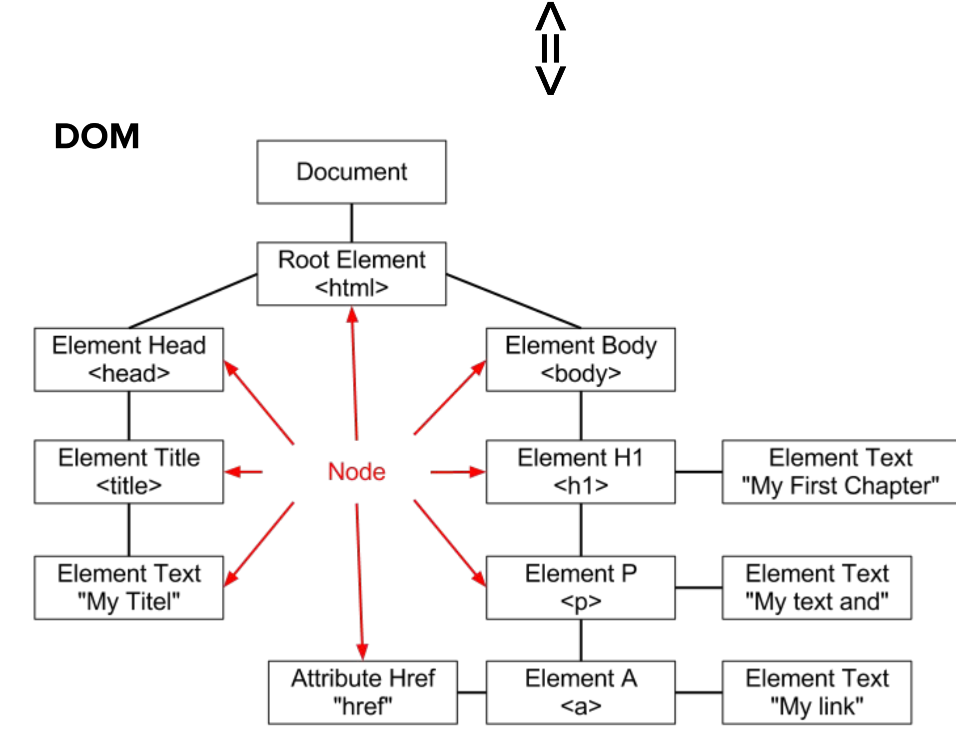
HTML je XML s předem definovanými elementy, atributy a strukturou pro popis dokumentů ve webovém browseru

HTML

```
<html>
  <head>
    <title>My Title</title>
  </head>
  <body>
    <h1>My First chapter</h1>
    <p>My Text and
      <a href="link.html">My Link</a>
    </p>
  </body>
</html>
```

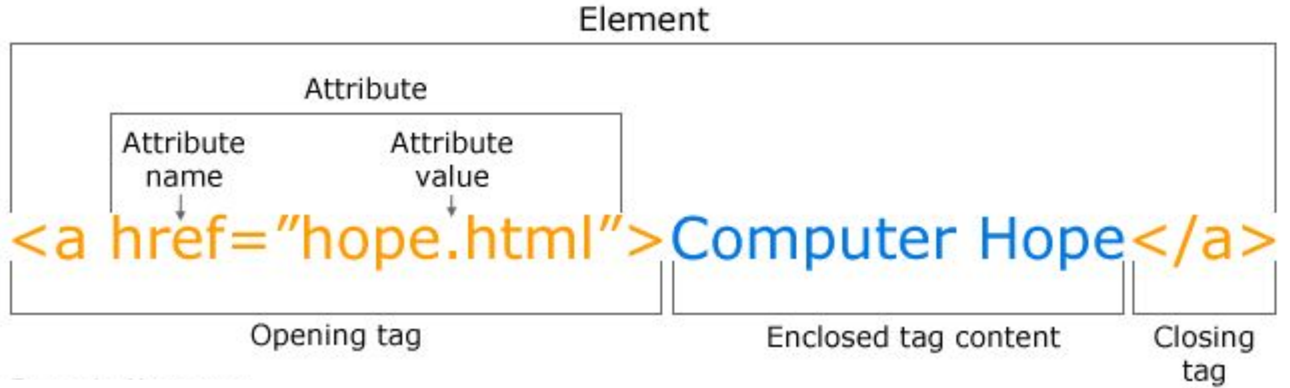
<=>

DOM



HTML Element alias HTML tag

- Má atributy
- Má obsah
- Může do sebe zanořovat další elementy

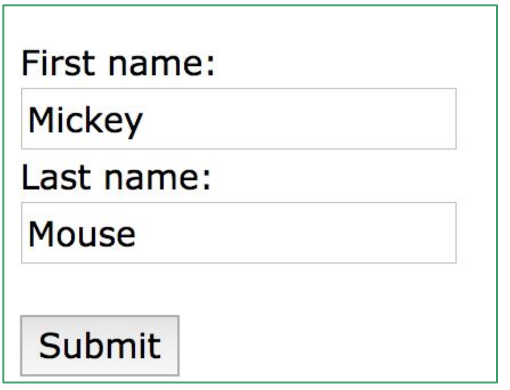


HTML

V HTML kromě struktury dokumentu mohu definovat i jednoduché chování a interakci se serverem - např. odeslání dat z HTML formuláře na server a zobrazení odpovědi ze serveru

MickeyMouse.html

```
<html>
      <body>
            <form action="/action.php">
              First name:<br>
              <input type="text" name="firstname" value="Mickey"><br>
              Last name:<br>
              <input type="text" name="lastname" value="Mouse"><br><br>
              <input type="submit" value="Submit">
            </form>
      </body>
</html>
```



Submitted Form Data

Your input was received as:

firstname=Mickey&lastname=Mouse

The server has processed your input and returned this answer.

https://www.w3schools.com/html/tryit.asp?filename=tryhtml_form_submit

CSS

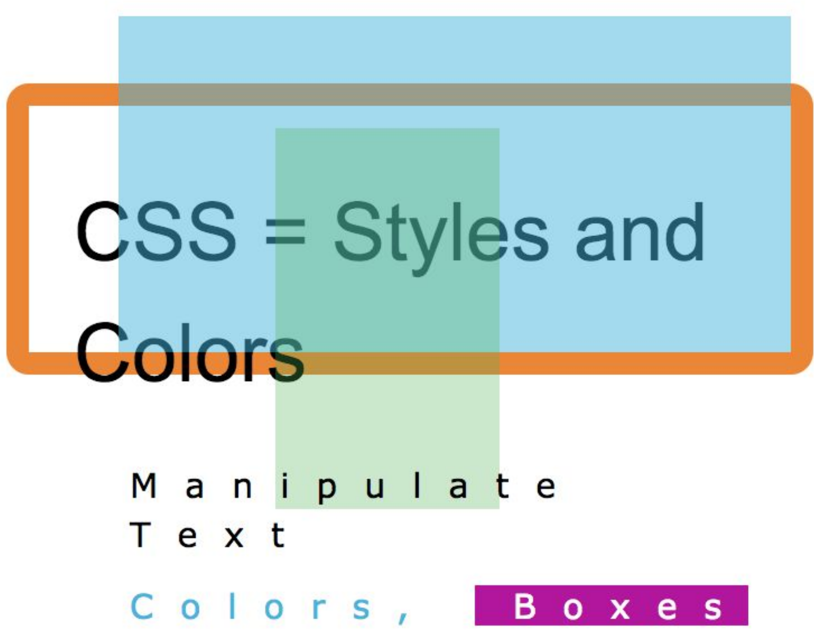
CSS znamená **C**ascading **S**tyle **S**heets.

CSS popisuje **jak budou HTML elementy zobrazeny na obrazovce, papíru a dalších médiích**

CSS šetří spoustu práce a **unifikuje vizuální stránku aplikace**.

Může být aplikován na více HTML stránek

```
<style>
body {background-color: powderblue;}
h1   {color: blue;}
p    {color: red;}
</style>
```



CSS lze připojit k HTML elementům třemi způsoby:

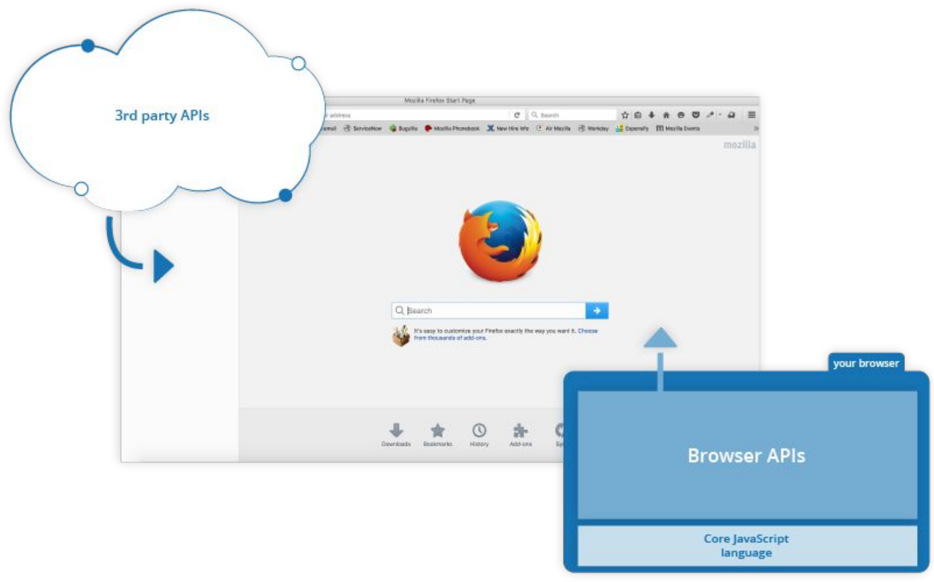
- **Inline** - použitím style atributu v HTML elementu
- **Internal** - <style> element v <head> sekci HTML
- **External** - v externím CSS souboru

Javascript

- Multi-platformní objektově orientovaný skriptovací jazyk, který je interpretován v cílovém kontaineru - např. ve webovém browseru.
- Javascript je syntakticky podobný jazyku Java.
- Narozdíl od Javy:
  - Javascript je **mnohem volnější** - není nutné deklarovat všechny proměnné a metody.
  - Javascript **není staticky typovaný** a v compile time **nedochází k silné kontrole typů** - není nutné definovat typy proměnných, funkčních parametrů a návratové hodnoty.
  - Javascript **nemá třídy** - má pouze objekty, dědičnost se obchází prototypováním (klonováním původních objektů), do objektů je možné v runtime přidávat nové atributy a metody.

Javascript - příklady

Client-side JavaScript - běží v internetovém browseru, může přistupovat k HTML stránkám renderovaným browserem přes jejich DOM. Také může přistupovat k objektům a funkcím vlastního browseru. Např. vkládat vlastní elementy do HTML, reagovat na kliknutí myši, vyplnění formuláře, navigaci mezi stránkami atd.



Server-side JavaScript - běží na serveru a má přístup k objektům a funkcím serveru - např. Komunikovat s databází, manipulovat se soubory atd. Např. *NodeJS*.

Ladění HTML + CSS + JavaScript

- JSFiddle - https://jsfiddle.net/7tj0cxn2/
- Codepen - https://codepen.io/anon/pen/wpPZjw

HTML ▼

```html
<p>Player 1: Chris</p>
```

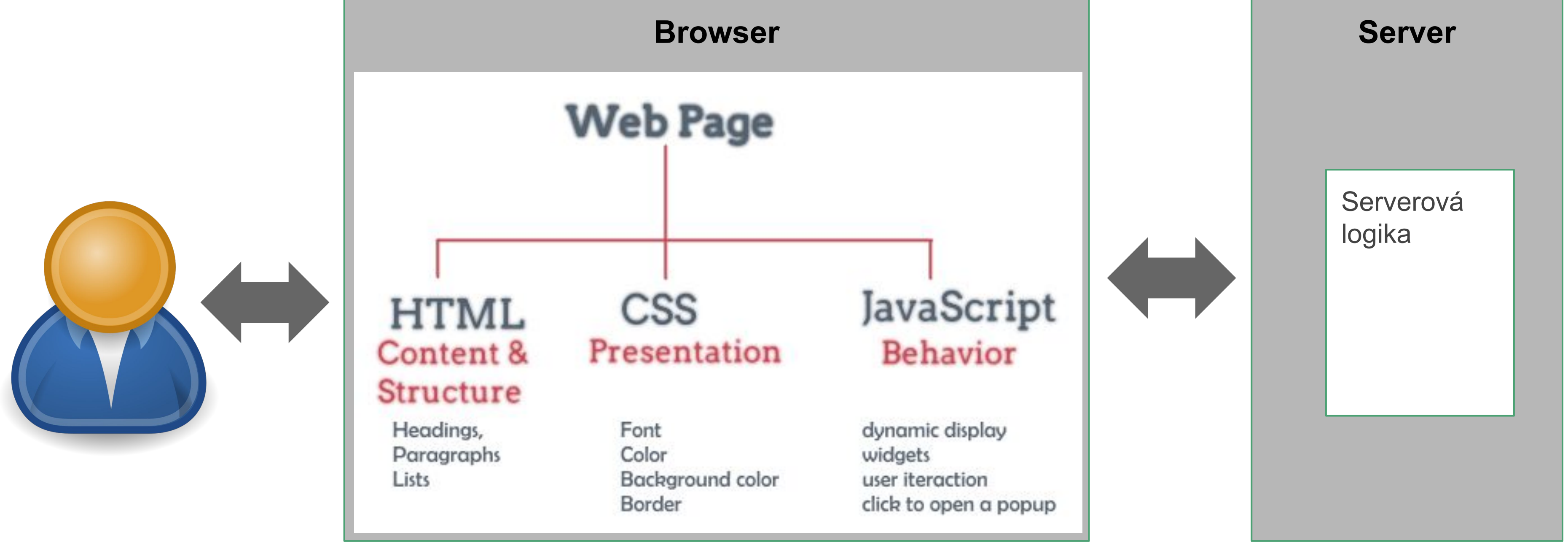
JavaScript (No-Library (pure JS)) ▼

```javascript
var para = document.querySelector('p');

para.addEventListener('click', updateName);

function updateName() {
  var name = prompt('Enter a new name');
  para.textContent = 'Player 1: ' + name;
}
```

CSS ▼

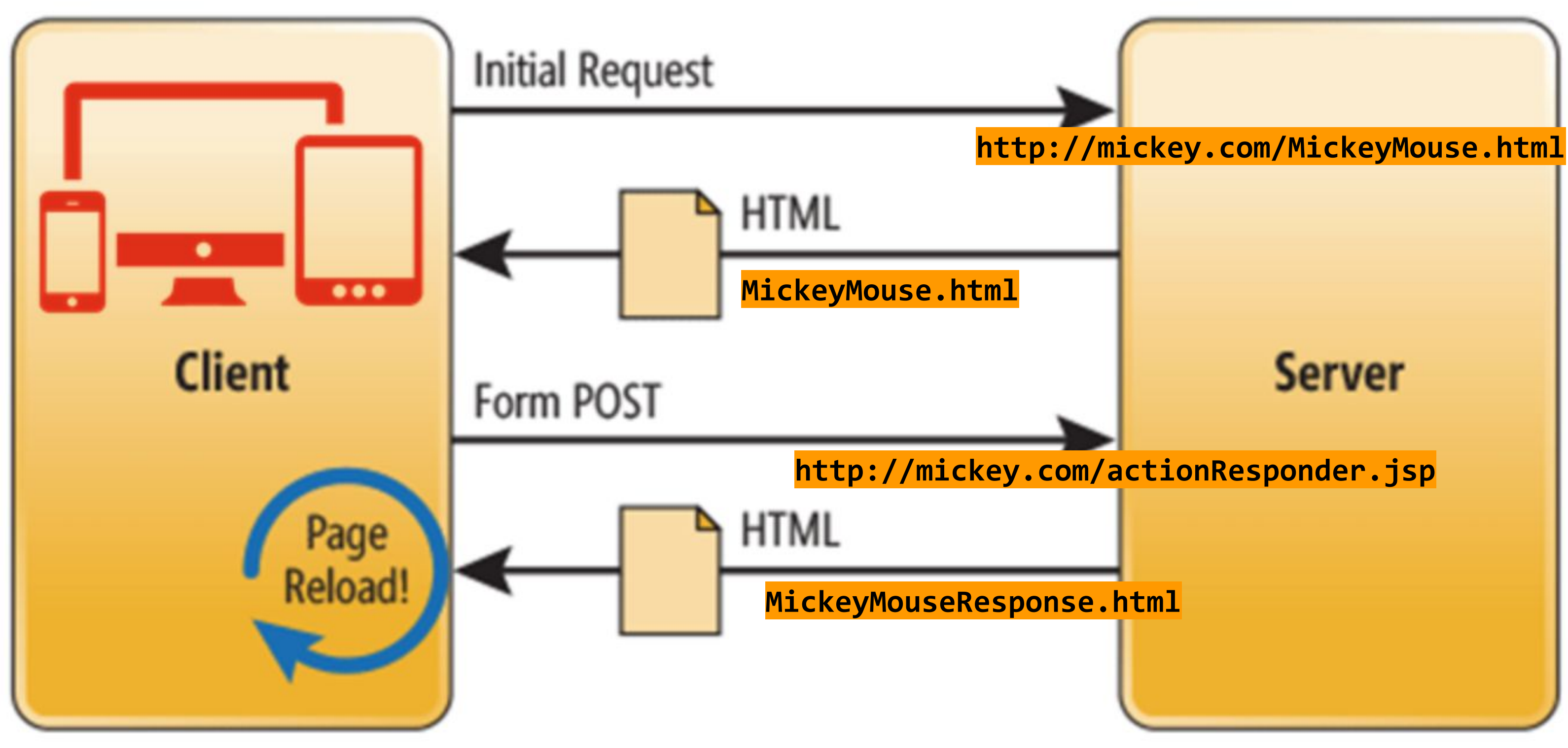
```css
p {
  font-family: 'helvetica neue', helvetica, sans-serif;
  letter-spacing: 1px;
  text-transform: uppercase;
  text-align: center;
  border: 2px solid rgba(0,0,200,0.6);
  background: rgba(0,0,200,0.3);
  color: rgba(0,0,200,0.6);
  box-shadow: 1px 1px 2px rgba(0,0,200,0.4);
  border-radius: 10px;
  padding: 3px 10px;
  display: inline-block;
  cursor:pointer;
}
```

Webový browser



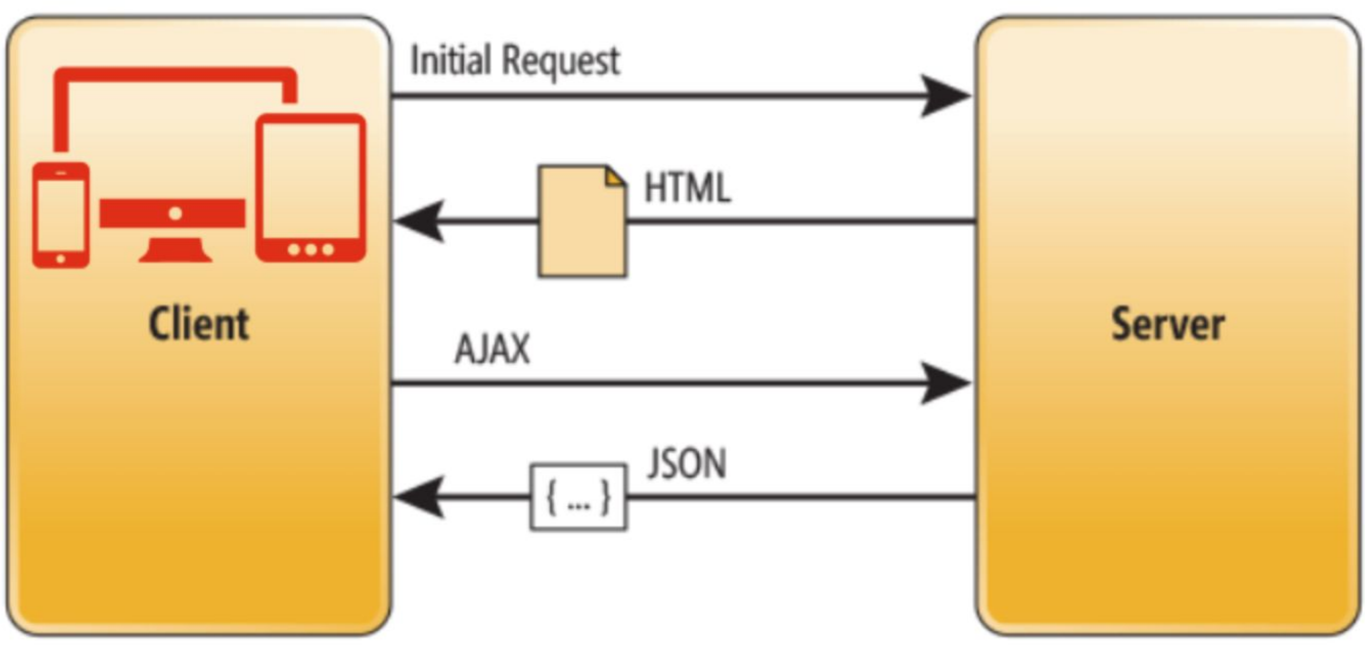
Architektura webové aplikace - Tradiční

V tradiční webové aplikací se po každé odpovědi ze serveru nahrává i nová HTML stránka.

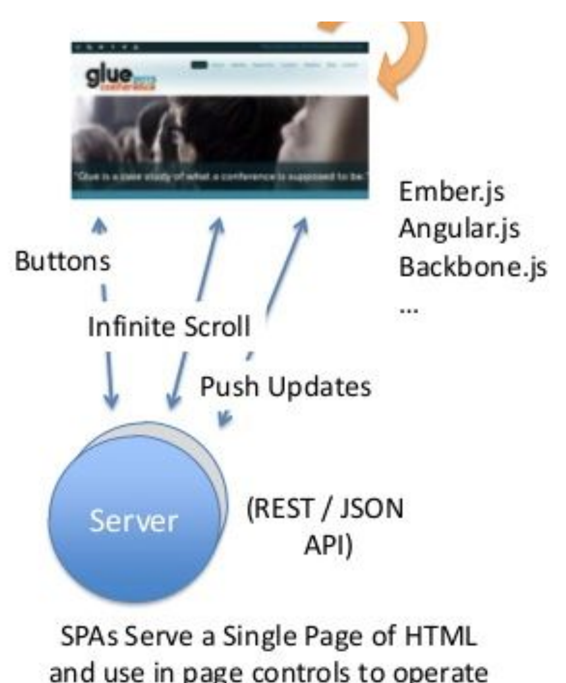


Architektura webové aplikace - Single Page Application

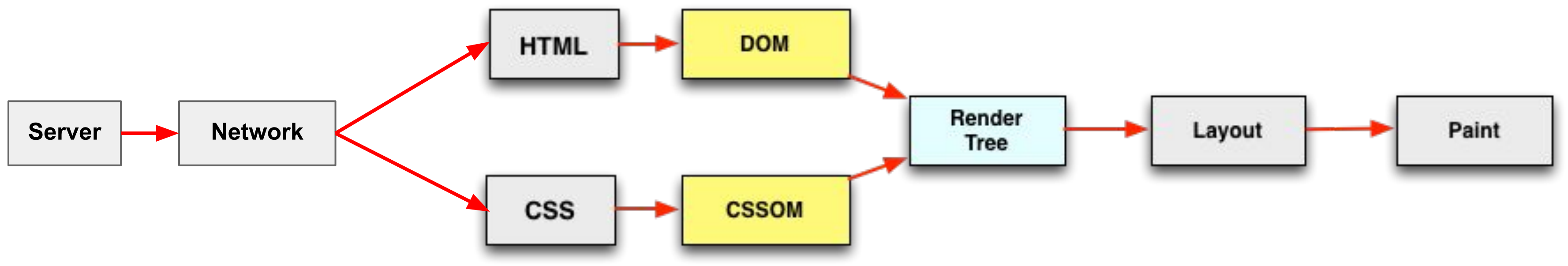
Single Page Application = mezi aplikací v browseru a serverem probíhá plynulá komunikace - ze serveru přichází data a ta mění různé části stejného HTML dokumentu (DOMu).



Single Page Application



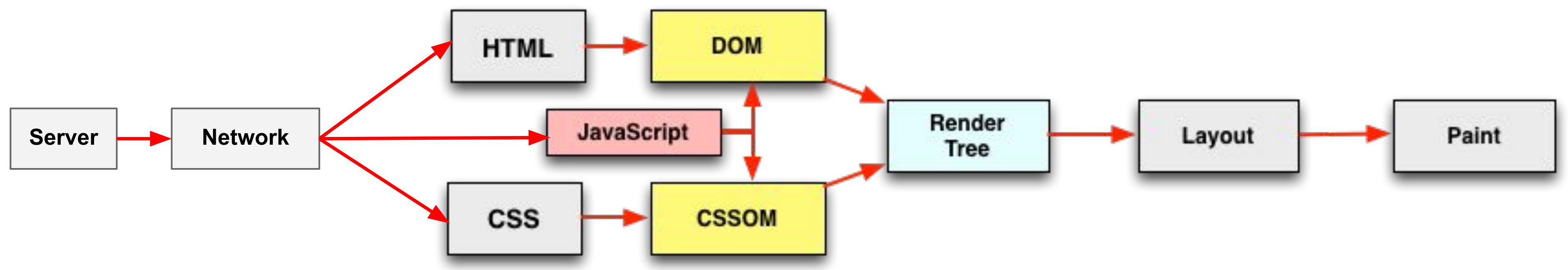
Procesování webové stránky



Zdrojové kódy vaší stránky nebo aplikace jsou z HTML a CSS transformovány do svých objektových reprezentací a zkombinovány do tzv. ***RenderTree*** - ten poskytuje browseru dostatek informací pro finální opatření layoutem (rozmístění atd.) a vykreslení.

CSSOM - Obdoba DOM pro CSS

Procesování webové stránky



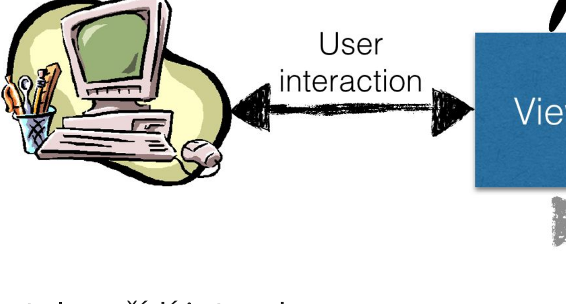
Javascript může zasahovat do procesu paralelního vytváření DOM a CSSOM a to tak, že modifikuje výstup, připojuje sám sebe na elementy DOMu nebo CSSOMu. Může dokonce oba procesy synchronizovat.

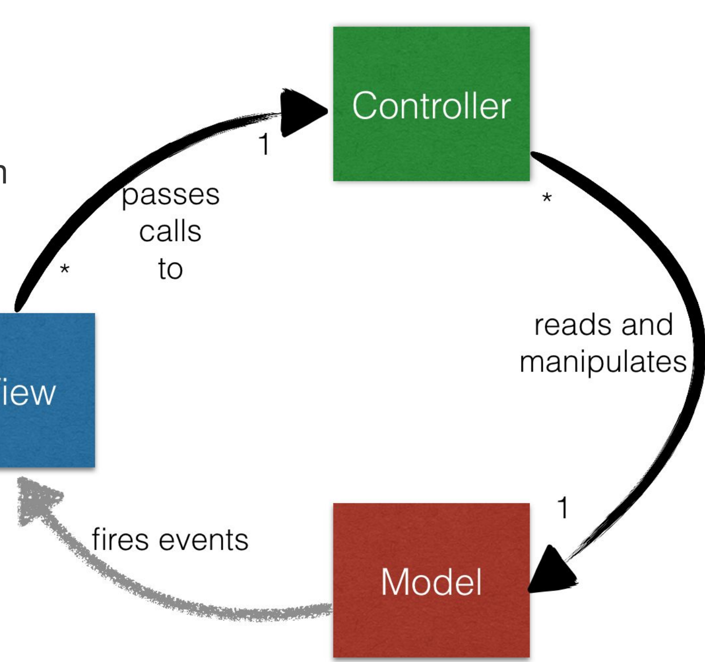
V místě, kde je do procesování vložen Javascript “sedí” i moderní UI frameworky jako je **Angular, React**, Backbone, Ember…

Model View Controller (MVC) Pattern

Model: Objektový graf, který popisuje business logiku - model aplikace, přístup k datům a pravidla jak je možné s tímto modelem manipulovat (číst a upravovat).

View: Reprezentuje to, co je zobrazené uživateli. Zobrazuje data z modelu. Data z modelu jsou přenášena do view pomocí změnových eventů (binding).





Controller: Reaguje na vstupy uživatele a řídí interakce s modelem. Controller přijme vstup od uživatele, ten zvaliduje, provede business operaci (lokálně nebo provoláním vzdálené služby) a na základě výsledku modifikuje model.

Model View Controller (MVC) Pattern

MVC využívá několik standardních design patternů

Mediator & Strategy pattern

Controller

1

passes calls to

*

*

View

User interaction

Observer pattern

reads and manipulates

1

Model

fires events

Command & Composite pattern

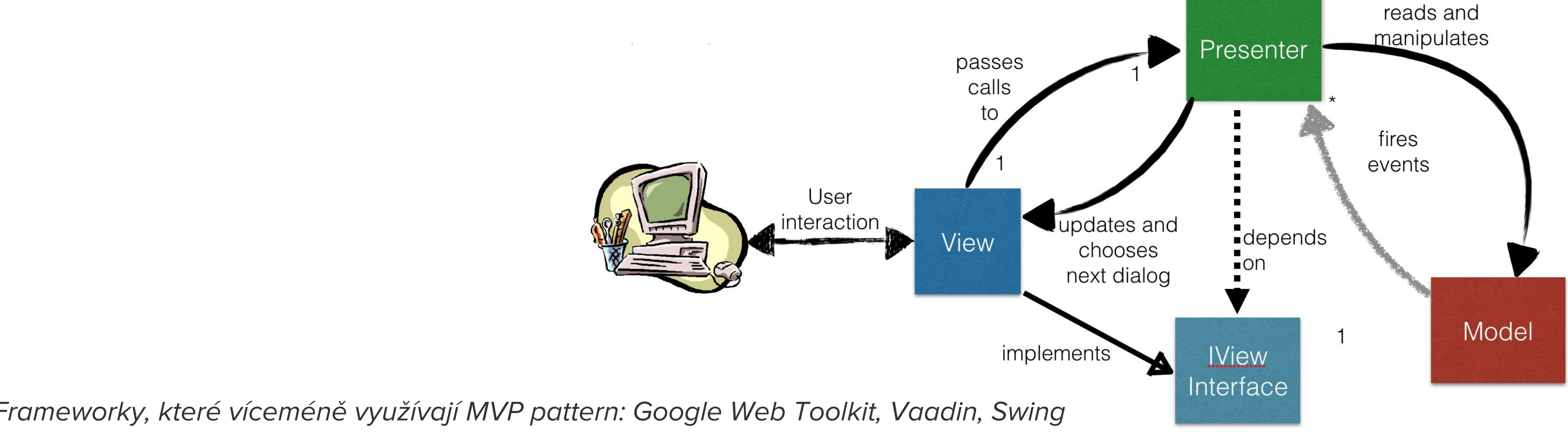
Model View Presenter (MVP) Pattern

Narozdíl od MVC je view oddělen od modelu přes tzv. ***Presenter***. Přes Presenter tedy probíhají veškeré interakce mezi view a modelem.

Používá se kvůli jednoduššímu unit testování nebo tam, kde se využívá více UI technologií najednou.

Model View Presenter (MVP) Pattern

View a presenter by měly být kompilovatelné nezávisle jeden na druhém. To dosáhneme tím, že obě vrstvy sdílí interfacy, které implementují. Vzájemná interakce probíhá přes tyto interfacy, implementace je možné prohazovat (dependency injection)



Frameworky, které víceméně využívají MVP pattern: Google Web Toolkit, Vaadin, Swing

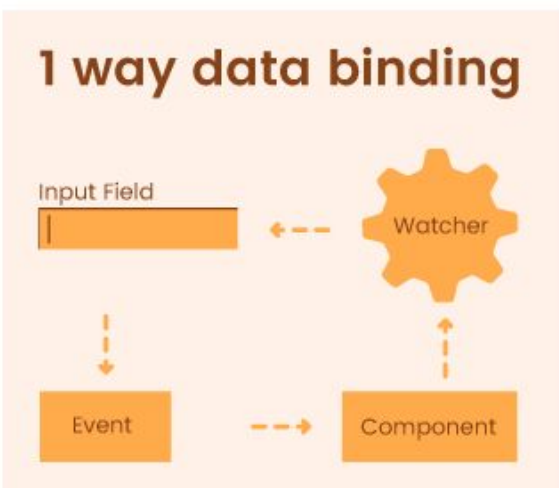
...

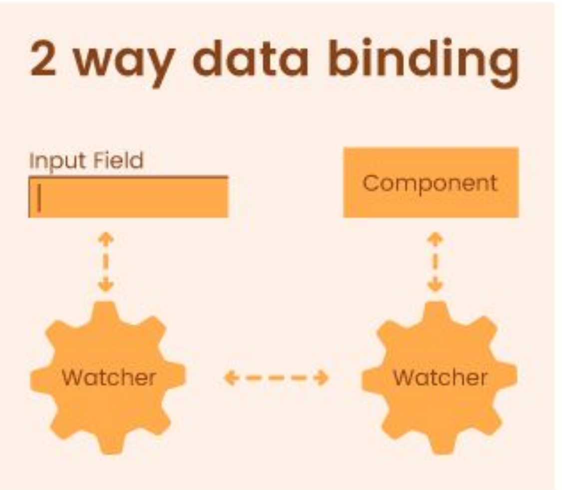
1 way versus 2 way data binding

Interakce mezi komponentami může být prováděna automaticky UI frameworkem - *data binding*. Narozdíl od programátorem explicitně vytvořeného volání mezi synchronizovanými komponentami např. pomocí eventu.

Data binding je tedy automatický proces, kterým jsou synchronizovány změny např. mezi View (tím co vidí uživatel) a dalšími vrstvami - ViewModel, Model.

Binding může být jedním směrem nebo oběma směry.





Pozn. Two Binding je kontroverzní pattern, který při nerozmyšleném použití vede ke spaghetty propojení komponent mezi sebou, kde změna v jedné komponentě iniciuje stovky vzájemných notifikací

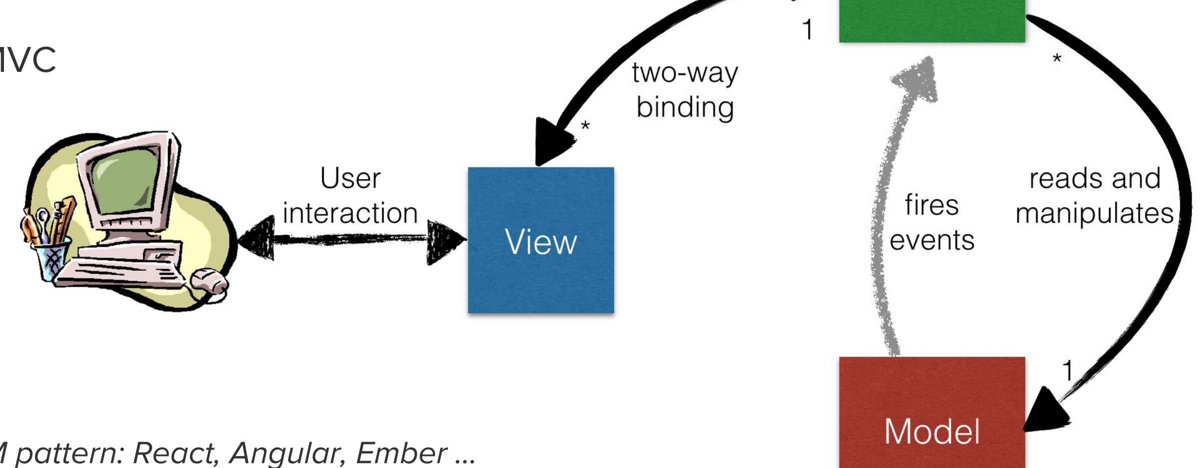
Model View ViewModel (MVVM) Pattern

View nepracuje přímo s modelem, ale s tzv. *ViewModelem*, který je jednodušší a bližší k vlastnímu view.

Mezi oběma funguje *two way binding*, který zajišťuje obousměrnou synchronizaci změn mezi View a ViewModelem.

ViewModel je podobný Controlleru v MVC

ViewModel drží stav modelu.



Frameworky, které víceméně využívají MVVM pattern: React, Angular, Ember ...

Moderní webové frameworky

- V současné době se používají moderní webové frameworky jako např.: React, AngularJS, Backbone, Ember, Polymer. Nejpoužívanější v poslední době je právě **React** a **AngularJS**
- Vesměs se označují jako **MV* - Model View Whatever,** protože umožňují použít více UI patternů nebo vytvářet hybridní paterny. *Pozn: Implementace MV* se liší podle toho jaká část MV* běží na klientu a jaká na serveru (např. Angular - téměř vše na klientu, Django - téměř vše na serveru a na klientu je pouze jednoduchý view)*
- Používají koncept **SPA** - Single Page Application
- Jsou interpetovány v Browseru pomocí Javascriptu buď přímo nebo jsou do něj kompilovány např. z **JSX** nebo **TypeScriptu** - lepší obdoby Javascriptu
- Kombinují se s layoutovacími knihovnami, které umožňují tvořit aplikace se responsivním UI. Responsivní UI - přizpůsobuje se tak, aby stejný kód mohl běžet v browseru na různých zařízeních (desktop, telefon, tablet):
  - **Material Design** - https://material.io/guidelines/material-design/introduction.html#
  - **Bootstrap** - https://getbootstrap.com/
- Common UX (User Experience) Patterny - http://ui-patterns.com/patterns

Future/Promise Pattern

Future je read-only view na proměnnou, která je nastavena někdy v budoucnu pomocí asynchronní funkce - ***Promise***. Před vlastním nastavením i po nastavení proměnné mohou vznikat různé situace, jejichž ošetření pattern také podporuje - jako např. timeout, vrácení chyby

Použití tohoto patternu výrazně snižuje blokování/latenci v distribuované nebo mnoha threadové aplikace.

Java realizace

Java realizuje tento design pattern pomocí frameworku okolo tříd *Future* a *CompletableFuture*. *Future* interface byl přidán do Java 5 pro lepší podporu asynchronního zpracování. Nicméně až v Java 8 byl přidán interface CompletableFuture, který navíc umožňuje výpočetní operace řetězit, zpracovávat různé druhy chyb (včetně timeoutu)

Future/Promise Pattern

Metoda *complete()* - nastaví hodnotu proměnné, do té doby jsou blokovány všechny pokusy o získání její hodnoty

```
public Future<String> calculateAsync() throws InterruptedException {
   CompletableFuture<String> completableFuture = new CompletableFuture<>();
   Executors.newCachedThreadPool().submit(() -> {
       Thread.sleep(5000);
       completableFuture.complete("Hello");
       return null;
   });
   return completableFuture;
}
..
Future<String> completableFuture = Complete.calculateAsync();
String result = completableFuture.get(); //Program gets blocked at this line until result provided
..
```

Zpracování chyby

V porovnání s try/catch blokem (syntaktický blok), v *CompletableFuture* se použije speciální metoda *handle()*. Jejími parametry jsou výsledek zpracování jestliže vše doběhlo OK a výjimka pokud nastal problém.

```
String name = null;
...
CompletableFuture<String> completableFuture
      = CompletableFuture.supplyAsync(() -> {
            if (name == null) {
                  throw new RuntimeException("Computation error!");
            }
            return "Hello, " + name;
      })}).handle((s, t) -> s != null ? s : "Hello, Stranger!");

assertEquals("Hello, Stranger!", completableFuture.get());
```

Future a Promise Pattern

Runnable a Supplier rozhraní umožňují se dále zbavit kódu pro multi-threading pomocí metod *runAsync()* nebo *supplyAsync()* s pomocí lambda expressions následovně:

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Hello");
assertEquals("Hello", future.get());
```

Pokud chceme výsledek výpočtu dále zprocesovat funkcí, tak použijeme metodu *thenApply*, která vezme výsledek výpočtu, ten zprocesuje funkcí o vrátí CompletableFuture, který drží nový výsledek.

```
CompletableFuture<String> compFuture = CompletableFuture.supplyAsync(() -> "Hello");
CompletableFuture<String> future = compFuture.thenApply(s -> s + " World");
assertEquals("Hello World", future.get());
```

Jestliže nepotřebujeme vracet future bez návratové hodnoty - CompletableFuture<Void> future, tak se použije thenAccept()

```
CompletableFuture<Float> compFuture = CompletableFuture.supplyAsync(() -> 120.0);
CompletableFuture<Void> future = compFuture.thenAccept(s -> ShoppingList.setPrice(s));
future.get();
```

Design pattern **Monáda** = kombinování futures

Zpracováváme více futures najednou, skládáme/vyhodnocujeme je sekvenčně, paralelně nebo dokonce hybridně.

Sekvenční zřetězení více futures

1) *thenCompose()* - výsledek zpracování funkce dáváme jako vstup do následující:

```
 CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() -> "Hello")
     .thenCompose(s -> CompletableFuture.supplyAsync(() -> s + " World"));
  assertEquals("Hello World", completableFuture.get());
```

Pozn. Java Stream map() je realizována kombinací thenCompose a thenApply
Rozdíl mezi thenCompose() a thenApply() je jako mezi Java Stream map() a flatMap(). Prosím prostudujte.

2) *thenCombine()* - výsledek dvou funkcí zkombinujeme do následující:

```
CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() -> "Hello")
      .thenCombine(CompletableFuture.supplyAsync(() -> " World"), (s1, s2) -> s1 + s2));
assertEquals("Hello World", completableFuture.get());
```

Design pattern **Monáda** = kombinování futures

Paralelní zřetězení více futures

1) *allOf()* - blokuje se dokud nejsou vyhodnoceny všechny výsledky:

```
CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> "Hello");
CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() -> "Beautiful");
CompletableFuture<String> future3 = CompletableFuture.supplyAsync(() -> "World");
CompletableFuture<Void> combinedFuture = CompletableFuture.allOf(future1, future2, future3);
combinedFuture.get();
assertTrue(future1.isDone());
assertTrue(future2.isDone());
assertTrue(future3.isDone());
```

2) *anyOf()* - blokuje se dokud není vyhodnocen jeden z výsledků:

```
CompletableFuture<String> future0 = createCFLongTime(0);
CompletableFuture<String> future1 = createCF(1);
CompletableFuture<String> future2 = createCFLongTime(2);
CompletableFuture<Object> future = CompletableFuture.anyOf(future0, future1, future2);
System.out.println("Future result>> " + future.get());
System.out.println("All combined>> " + future0.get() + "|" + future1.get() + "|" + future2.get());}
```

Další příklad na allOf()

```
StringBuilder result = new StringBuilder();
List<String> messages = Arrays.asList("a", "b", "c");

List<CompletableFuture<String>> futures = messages.stream()
      .map(msg -> CompletableFuture.completedFuture(msg).thenApply(s -> delayedUpperCase(s)))
      .collect(Collectors.toList());
CompletableFuture.allOf(futures.toArray(new CompletableFuture[futures.size()])).whenComplete((v, th) ->
{
      futures.forEach(cf -> assertTrue(isUpperCase(cf.getNow(null))));
      result.append("done");
});
```

Design patterny Future/Promise a monáda jsou masivně využívány při psaní frontendových aplikací, kdy veškerá většina interakce mezi UI komponenty (model, view, controller), interakce se serverem je asynchronní.

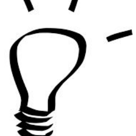
Angular

```
  $scope.getRequest = function () {
          console.log("I've been pressed!");
          $http.get("http://urlforapi.com/get?name=Elliot")
              .then(function successCallback(response){
                  $scope.response = response;
              }, function errorCallback(response){
                  console.log("Unable to perform get request");
              }
      .catch(error => this.setState({ error, isLoading: false }));
```

React

```
fetch("http://urlforapi.com/get?name=Elliot")
      .then(response => {
        if (response.ok) {
          return response.json();
        } else {
          throw new Error('Something went wrong ...');
        }
      })
      .then(data => this.setState({ hits: data.hits, isLoading: false }))
      .catch(error => this.setState({ error, isLoading: false }));
```

Pro zajímavost (nepovinné)

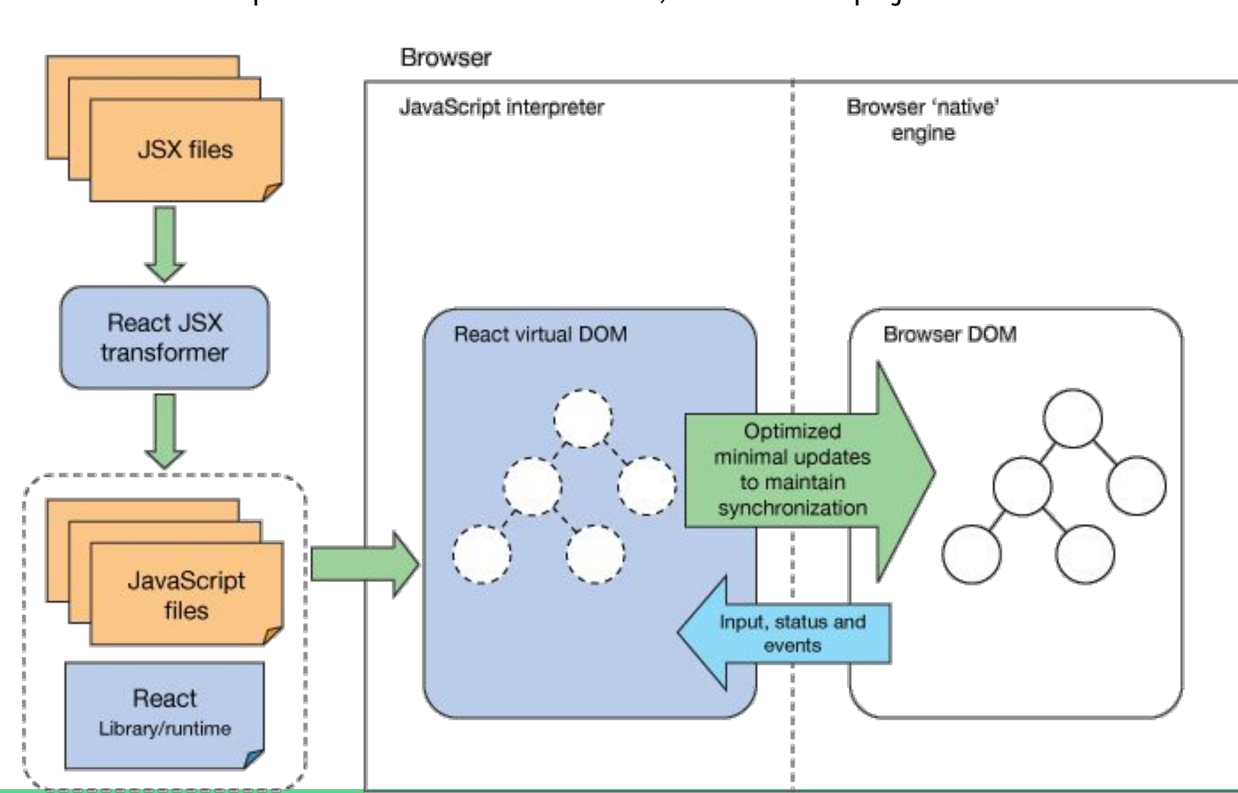


React

React je *view* knihovna. Používá koncept UI komponent, které mají stav stejný jako odpovídající část UI. Nepracují tedy s odděleným modelem

Abstrakce do komponent v Reactu je velmi dobrá na provádění změn v DOMu, ale horší v přijímání změn z DOMu.

JSX je silněji typovaná a rychlejší obdoba Javascriptu



React

HTML stránka - obsahuje import JavaScript knihoven ReactJS:

```
<script src="http://fb.me/react-0.10.0.min.js"></script>
<div id="container">
    <!-- This element's contents will be replaced with your component. -->
</div>
```

Javascript soubor obsahuje vlastní kód:

- ***ReactDOM.render()*** aplikuje React kód na DOM HTMLElement s ***id = container*** a inicializuje stav komponenty *Hello* na hodnotu “World”.
- ***change()*** metoda mění stav komponenty na vstupní parametr
- ***render()*** metoda vrací vlastní HTML kód, který se vkládá do HTML elementu s id *container*. Závorky {} se používají na propojení HTML markupu s proměnnými ve skriptu - proměnná ***name***

```
class Hello extends React.Component {
  state = {
    name: this.props.initialName
  }
  change(e) {
    this.setState({ name: e.target.value });
  }
  render() {
    const { name } = this.state;
    return (
      <div>
        Name: <input value={name} onChange={this.change.bind(this)} /><br/>
        Hello {name}!
      </div>
    );
  }
}

ReactDOM.render(
  <Hello initialName="World"/>,
  document.getElementById('container')
);
```

Test => http://jsfiddle.net/protron/560fyjnp/

Angular 2

Angular je kompletní framework pro psaní webových aplikací

Obsahuje celou řadu knihoven, soubory lze lze místo Javascripty psát i v **TypeScriptu** - silně typovaná obdoba Javascriptu

HTML stránka - obsahuje element na který se budeme napojovat:

```
...
<body>
  <app-hello-world></app-hello-world>
</body>
...
```

Typescript soubor:

- Importuje Angular knihovny a definuje komponentu, která se aplikuje na element ***app-hello-world***.
- Závorky {} se používají na propojení HTML markupu s proměnnými ve skriptu - proměnná ***name***

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-hello-world',
  template: '<h1>Hello {{name}}!</h1>',
})
export class AppComponent {
  name = 'World';
}
```

Angular 2

Test => https://jsfiddle.net/rautamar/1y0tw6zf/

Implementace TODO listu - seznamu úkolů:

```
<script src="https://code.angularjs.org/2.0.0-alpha.26/angular2.sfx.dev.js">
<script>
    function AppComponent() {
        this.todos = [];
        this.addTodo = function(todo) {
            this.todos.push(todo.value);
            todo.value = null;
            return false;
        }
    }
    AppComponent.annotations = [
      new angular.ComponentAnnotation({
        selector: 'my-app'
      }),
      new angular.ViewAnnotation({
        template: '<h3>TODO</h3>' +
                  '<ul><li *ng-for="#todo of todos">{{ todo }}</li></ul>' +
                  '<form (submit)="addTodo(todotext)"><input #todotext><button>add #{{todos.length + 1}}</button></form>',
        directives: [angular.NgFor]
      })
    ];
    document.addEventListener('DOMContentLoaded', function() {
      angular.bootstrap(AppComponent);
    });
</script>
<my-app></my-app>
```

TODO

- Dnes: Koupit maso
- Zítra: Dojít na přednášku
- Zítra: Sníst maso

Pozítří: Kou [add #4]

React

React se často používá v kombinaci s dalšími frameworky jako je např. Redux. Redux přidává k reactu správu dat a jejich stavu.

Redux (Store)
Connection between the two
React (View)

Service
Calls
Returns
Action & Action Creator
Container
Template
Event
Re-renders
Triggers
Sent To
Reducer
Is Used By
Selector
Components
Updates
Updates
Is Exposed by
Store
Map Store to Props
Props

Srovnání

Technology	React	AngularJS	Angular 2
Author	Facebook community	Google	Google
Type	Open source JS library	Fully-featured MVC framework	Fully-featured MVC framework
Toolchain	High	Low	High
Language	JSX	JavaScript, HTML	TypeScript
Learning Curve	Low	High	Medium
Packaging	Strong	Weak	Medium
Rendering	Server Side	Client Side	Server Side
App Architecture	None, combined with Flux	MVC	Component-based
Data Binding	Uni-directional	Bi-directional	Bi-directional
DOM	Virtual DOM	Regular DOM	Regular DOM
Latest Version	15.4.0 (November 2016)	1.6.0	2.2.0 (November 2016)