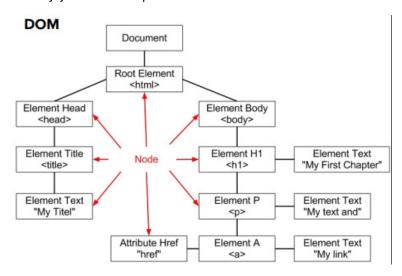
# Patterny pro tvorbu UI

# Základy UI

- webový browser může běžet na desktopech, tabletech, telefonech, TV,...

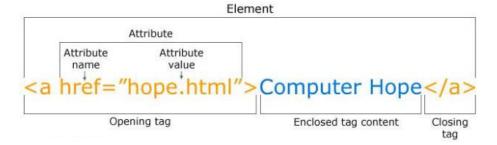
### DOM (Document Object Model)

- = platformě nezávislý objektový model a API pro XML dokumenty
  - XML lze převést (parsovat) do objektové struktury DOM s ní se výrazně lépe pracuje
  - serializací lze převést zpět do XML
  - mohu jej modifikovat pomocí API



#### HTML (HyperText Markup Language)

- = XML s předem definovanými elementy, atributy a strukturou pro popis dokumentů ve webovém prohlížeči
  - kromě struktury dokumentu lze definovat i jednoduché chování a interakci se serverem
    - o např. odeslání HTML formuláře na server a zobrazení odpovědi ze serveru
  - HTML tag má atributy, obsah a lze do zanořovat další elementy



#### CSS (Cascading Style Sheets)

- popisuje, jak budou HTML elementy zobrazeny na obrazovce, papíru apod.
- šetří práci, unifikuje vizuální stránku aplikace
- může být aplikován na více HTML stránek

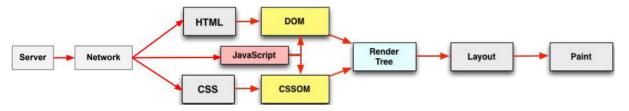
- lze jej připojit k HTML třemi způsoby:
  - o inline použitím style atributu v HTML elementu
  - o internal <style> element v <head> sekci HTML
  - o external v externím CSS souboru

#### **Javascript**

- multiplatformní OO skriptovací jazyk
- je interpretován v cílovém containeru (např. webový prohlízeč)
- syntakticky podobný Javě, ale narozdíl od ní:
  - o **je volnější** není nutné deklarovat všechny proměnné a metody
  - o není staticky typovaný, v compile time nedochází k silné typové kontrole
  - o **nemá třídy** pouze objekty, lze do nich přidávat atributy a metody v runtime
- **client-side JS** běží v prohlížeči, může přistupovat k HTML stránkám renderovaných DOMem
- **server-side JS** běží na serveru, přístup k objektům a funkcím serveru (komunikace s DB,...)

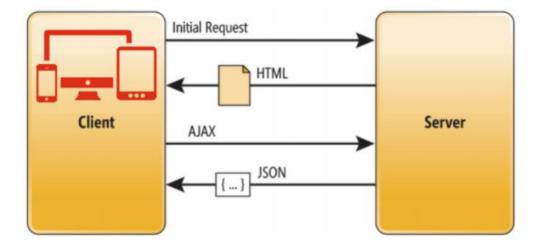
#### Procesování webové stránky

- zdrojové kódy z webu/aplikace jsou z HTML a CSS transformovýny do objekt. reprezentací
- pak zkombinovány do tzv. RenderTree
  - o poskytuje prohlížeči dostaten info pro finální opatření layoutem a vykreslení
- JS může zasahovat do procesu paralelního vytváření DOM a CSSOM (obdoba DOM pro CSS)
  - o modifikuje výstup, připojuje sám sebe na elementy DOM/CSSOM
  - o může oba procesy synchronizovat



# Single Page Application

- v tradiční webové aplikaci se po každé odpovědi ze serveru nahrává nová HTML stránka
- v SPA probíhá mezi aplikací v prohlížeči a serverem plynulá komunikace
  - o ze serveru přichází data měnící různé části stejného HTML dokumentu (DOMu)



# MVC, MVP, MVVM

## MVC (Model View Controller)

#### Model

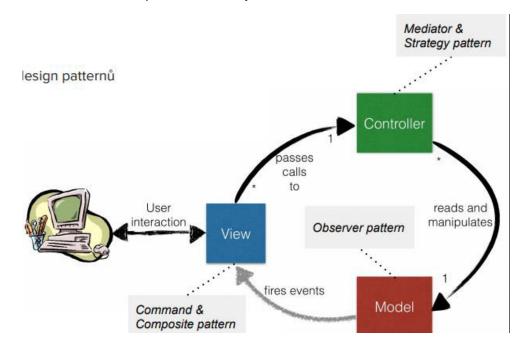
- objektový graf, který popisuje business logiku
- tzn. model aplikace, přístup k datům a pravidla, jak je možné s tímto modelem manipulovat

#### View

- reprezentuje to, co je zobrazené uživateli
- zobrazuje data z modelu
  - o ta jsou přenášena do View pomocí změnových eventů (binding)

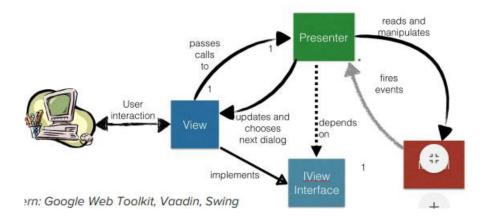
#### Controller

- reaguje na vstupy uživatele a řídí interakce s modelem
  - o přijme vstup od uživatele
  - o zvaliduje jej
  - o provede business operaci (lokálně či provoláním vzdálené služby)
  - o na základě výsledku modifikuje model



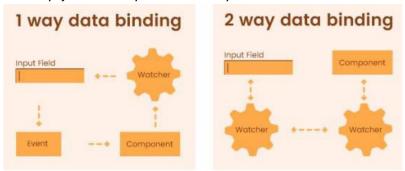
#### MVP (Model View Presenter)

- narozdíl od MVC je view oddělen od modelu přes tzv. Presenter
- přes něj probíhají veškeré interakce mezi view a modelem
- používá se kvůli jednoduššímu unit testování nebo tam, kde se využívá více UI technologií
- View a Presenter by měly být kompilovatelné nezávisle na sobě
  - o dosaženo tím, že obě vrstvy sdílí interfaces, které implementují
  - o vzájemná interakce probíhá přes ně
  - o implementace je možné prohazovat (dependency injection)
- využívá jej např. Google Web Toolkit, Swing, Vaadin,...



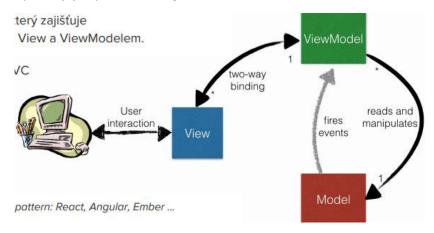
#### 1-way vs 2-way data binding

- interakce mezi komponentami může být prováděna automaticky UI frameworkem
- např. pomocí eventu mezi synchronizovanými komponentami
- DB je to automatický proces, kterým jsou synchronizovány změny např. mezi View (tím, co vidí uživatel) a dalšími vrstvami (ViewModel, Model,...)
- může být jednosměrný či obousměrný



#### MVVM (Model View ViewModel)

- View nepracuje přímo s modelem, ale s tzv. ViewModelem
- ten je jednodušší a bližší k vlastnímu View
- mezi oběma funguje 2-way binding (obousměrná synchronizace mezi View a VM)
- ViewModel je podobný Controlleru z MVC
- ViewModel drží stav modelu
- využívá jej např. React, Angular,...



# Future/Promise pattern

- **Future** je read-only view na proměnnou, která je nastavena někdy v budoucnu pomocí asynchronní funkce (**Promise**)
- před/po nastavení proměnné mohou vznikat situace (timeout, vrácení chyby,...), které pattern také podporuje
- výrazně snižuje blokování/latenci v distribuované/multithreadové aplikaci

#### Java realizace

- pomocí frameworku okolo tříd Future a CompletableFuture
- complete() nastaví hodnotu proměnné (do té doby blokovány pokusy o získání její hodnoty)
- CompleTableFuture používá metodu handle() ke zpracování chyby
  - o parametry:
    - výsledek zpracování, proběhlo-li vše OK
    - výjimka, nastal-li problém
- runAsync() a supplyAsync() umožňují zbavit se kódu pro multithreading
  - o thenApply() další zprocesování výsledku výpočtu
  - o thenAccept() není-li třeba vracet future bez návratové hodnoty

#### Monáda (kombinování futures)

- zpracováváme více futures najednou
- skládáme/vyhodnocujeme je sekvenčně, paralelně nebo hybridně

#### Sekvenční

- thenCompose() – výsledek zpracování funkce dáváme jako vstup do:

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

- **thenCombine()** – výsledek dvou funkcí zkombinujeme do:

#### Paralelní

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());
```

- **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()
```

- Future/Promise a Monáda se masivně používají při psaní front-end aplikací
- většina interakce mezi UI komponenty (Model, View, Controller) a se serverem je asynchronní

# Moderní webové frameworky

- např. React, AngularJS, Backbone, Ember, Polymer,...
- označují se jako MV\* (Model View Whatever)
  - o umožňují použít více UI patternů, popř. vytvářet hybridní
- používá se koncept SPA Single Page Application
- jsou interpretovány v prohlížeči pomocí **JS** (buď **přímo**, nebo jsou do něj kompilovány např. z **JSX** či **TypeScriptu**)
- kombinace s layoutovacími knihovnami
  - o umožňují vytvářet aplikace s responsivními UI
  - o přizpůsobuje se tak, aby stejný kód mohl běžet v prohlížeči na různých zařízeních
  - Material Design, Bootstrap,...

#### React

- je to View knihovna JS
- používá koncept UI komponent, které mají stav stejný jako odpovídající část UI
- nepracuje s odděleným modelem
- abstrakce do komponent skvělá na provádění změn v DOM, ale horší na jejich přijímání
- často v kombinaci s dalšími frameworky (např. Redux přidává správu dat a jejich stavu)
- 1-way data-binding
- od Facebooku

#### Angular 2

- kompletní framework pro psaní webových aplikací
- obsahuje řadu knihoven
- soubory lze místo JS psát i v TypeScript
- 2-way data binding
- od Googlu

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

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

@Component({
   selector: 'app-hello-world',
   template: '<h1>Hello {{name}}!</h1>',
})

export class AppComponent {
   name = 'World';
}
```