

Slovenská technická univerzita v Bratislave

Fakulta informatiky a informačných technológií

3D UML

Finálna verzia – Inžinierske dielo

Tím č. 15

Vedúci tímu: Ing. Ivan Polášek, Phd.

Členovia tímu: Bc. Matej Vítaz, Bc. Ľubomír Jesze, Bc. Lukáš Skala, Bc. Peter Zajac, Bc. Dominik Žilka, Bc. Matúš Gáspár, Bc. Jakub Minárik

Akademický rok: 2016/2017

Obsah

1	Úvod	1
1.1	Globálne ciele projektu.....	1
1.1.1	Ciele pre zimný semester	2
1.1.2	Ciele pre letný semester	2
1.2	Celkový pohľad na 3D UML aplikáciu	2
2	Moduly systému	3
2.1	Vykresľovanie diagramov	3
2.1.1	Analýza	3
2.1.2	Návrh	6
2.1.3	Implementácia	6
2.1.4	Testovanie	7
2.2	Serverová aplikácia	8
2.2.1	Analýza.....	8
2.2.2	Návrh	13
2.2.3	Implementácia	18
2.2.4	Testovanie	19
2.3	Jadro klientskej aplikácie	19
2.3.1	Analýza.....	20
2.3.2	Návrh	23
2.3.3	Implementácia	24
2.3.4	Testovanie	26
2.4	Grafická reprezentácia častí fyzického modelu v zimnom semestri	26
2.4.1	Analýza.....	26
2.4.2	Návrh	27
2.4.3	Implementácia	27
2.5	Grafická reprezentácia častí fyzického modelu v letnom semestri	27
2.5.1	Analýza.....	27
2.5.2	Návrh	28
2.5.3	Implementácia	28
2.5.4	Testovanie	32
3	Vývojový diagramu procesu pri CRUD	36
4	Práca v budúcich etapách	36
4.1	Diskusia k optimalizácii	36

4.1.1	JSON API.....	37
4.1.2	Angular vykresľovanie.....	37
4.2	Ďalšia funkcionality.....	38
5	Používateľská príručka k 3duml – sekvenčnému diagramu.....	38
5.1	Módy.....	38
5.2	Help.....	39
5.3	Vytvorenie nového diagramu.....	39
5.4	Zobrazenie diagramu.....	40
5.5	Editovanie diagramu.....	40
5.6	Vymazanie diagramu.....	41
5.7	Pridanie vrstvy.....	41
5.8	Upraviť vrstvu.....	42
5.9	Vymazanie vrstvy.....	43
5.10	Vytvorenie/pridanie čiary života.....	44
5.11	Premiestnenie/premenovanie čiary života.....	44
5.12	Vymazanie čiary života.....	45
5.13	Vytvorenie novej správy.....	46
5.14	Upravovanie správy.....	47
5.15	Vymazanie správy.....	49
6	Inštalčná príručka.....	49
7	Technická dokumentácia.....	52

Úvod

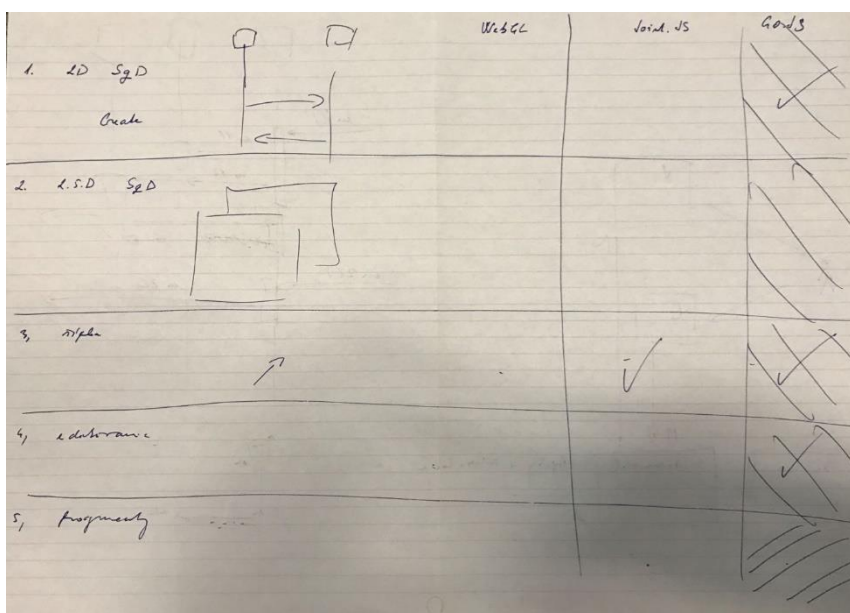
Dokument inžinierskeho diela bol vytvorený z dôvodu zdokumentovania našej tímovej práce v akademickom roku 2016/2017 v predmete Tímový projekt. Náš tím, číslo 15 s názvom Next Dimension Team, sa podieľa na projekte s názvom 3D UML, a teda vytvorenie 3D prostredia na modelovanie UML diagramov. Konkrétne sa v našom projekte špecializujeme na vytvorenie tohto prostredia pre sekvenčný diagram. Dokument je rozdelený na niekoľko častí. V úvode sa čitateľ oboznámi so všeobecnou problematikou, ktorú riešime. V prvej časti sa dozvie náš cieľ, ktorý chceme dosiahnuť v rámci zimného semestra a získa prehľad o tom, čo plánujeme v rámci projektu urobiť a aký je celkový, všeobecný, pohľad na aplikáciu. V druhej časti sú ozrejmené všetky moduly systému, ktoré používame. Postupne prejdeme každým modulom aplikácie a ozrejíme jeho časti. Moduly ktoré sa nachádzajú v našej aplikácii sú - vykresľovanie diagramov, serverová aplikácia a jadro klientskej aplikácie. Pre prehľadnosť sme rozdelili každý opis modulu na štyri časti - analýza, návrh, implementácia a testovanie.

Globálne ciele projektu

V tejto kapitole uvádzame ciele, ktoré sme si stanovili po konzultácii s pedagogickým vedúcim projektu na jednotlivé semestre. Obsahuje popis častí projektu, ktoré by sme chceli implementovať.

Aplikáciu sme na stretnutí s vedúcim si rozdelili na niekoľko častí:

1. Vytvorenie 2D sekvenčného diagramu
2. Vrstvy diagramu – 2.5D
3. Prepojenie jednotlivých plátien (3D šípka) - vytvorené ako diplomová práca a bude potrebné integrovať do nášho projektu
4. Editovanie diagramu
5. Fragменты sekvenčného diagramu – bude vytvorené ako bakalárska práca a integrované do projektu



Obrázok: Plán projektu, tak ako bol načrtnutý vedúcim projektu

Ciele pre zimný semester

V zimnom semestri je naším cieľom implementovať perzistenciu diagramu do databázy a jeho načítavanie. Navyše sa zameriame na implementáciu prvých dvoch bodov, ktoré sme uviedli v predchádzajúcej kapitole. Konkrétne ide o nasledujúce časti:

1. Integrovanie knižnice, ktorá vykresľuje 2D diagramy – ide o použitie už existujúcej knižnice, ktorú integrujeme do nášho projektu, aby bolo možné pracovať s diagramom na jednotlivých vrstvách
2. Možnosť v aplikácii pridávať vrstvy - vrstvy predstavujú 3D aspekt, kde jednotlivé diagramy sú v priestore za sebou a na jednotlivých vrstvách sú 2D diagramy (ide vlastne o 2.5D)
3. Po ukončení viacerých šprintov sa nám podarilo zistiť, že výhodnejšie bude namiesto použitia knižnice na vytváranie 2D diagramov, naprogramovať svoje vlastné jadro na vykresľovanie diagramu. Preto našim hlavným cieľom bolo, aby sme takúto funkcionálnosť naprogramovali.

Ciele pre letný semester

V letnom semestri je naším cieľom spraviť CRUD (create, retrieve, update, delete). Po prvom semestri bolo implementované jadro vykresľovania diagramu. Create predstavuje pridávanie jednotlivých elementov do diagramu. Update predstavuje editáciu elementov – zmena názvov, posúvanie, zväčšovanie, zmenšovanie. Delete predstavuje vymazávanie jednotlivých elementov v rámci celého diagramu. Dalším cieľom je taktiež pridať Menu baru, ktoré bude pre používateľa príjemné na používanie. Posledným cieľom je zobrazovať diagram v editačnom (2D) a v zobrazovacom (3D) móde.

Celkový pohľad na 3D UML aplikáciu

Náš systém sa skladá z dvoch častí aplikácie, jedna je serverová, druhá je klientská. Serverová je tá, ktorá obstaráva komunikáciu s databázou na vzdialenom zariadení, zatiaľ čo klientská časť aplikácie sa zobrazuje používateľovi v prehliadači. Klientská časť aplikácie je tá, s ktorou používateľ komunikuje, čiže v prípade UML modelovania je to tá, kde kreslí, posúva, maže, alebo upravuje diagramy, resp. jednotlivé súčasti diagramov.

Serverová a klientská časť spolu komunikujú prostredníctvom JSON API.

Na obrázku dole je znázornená základná architektúra nášho systému.



Obrázok: Architektúra aplikácie

Moduly systému

Nasledujúca kapitola obsahuje popis vykonanej analýzy, návrhu a implementácie jednotlivých modulov aplikácie, ktoré boli realizované počas prvých troch šprintov tímového projektu. Tieto moduly boli následne testované ako u vädzajú podkapitoly jednotlivých modulov.

Vykresľovanie diagramov

Táto kapitola popisuje modul aplikácie, ktorého úlohou je vykresľovanie 2D UML diagramov, ktoré budú integrované do 3D prostredia pomocou ďalších modulov aplikácie. Kapitola sa zaoberá analýzou použiteľných technológií a príslušných knižníc, zakončenou zhodnotením výhod a nevýhod jednotlivých možností. Taktiež obsahuje vysvetlenie návrhu a implementácie tohto modulu aplikácie.

Analýza

Porovnanie vykresľovacích technológií

Dôležitým krokom pri tvorbe 3D prostredia je výber vykresľovacej technológie, ktorá najviac vyhovuje našim potrebám na tvorbu 3D UML diagramov. V našom prípade prichádzajú do úvahy dve alternatívy, ktorými sú WebGL a CSS 3D.

WebGL je JavaScriptové API, ktoré umožňuje vykresľovanie 3D grafiky vo webovom prehliadači. Grafika aplikácie vytvorená prostredníctvom WebGL je renderovaná do HTML elementu canvas a je spustiteľná na grafickej karte s podporou shader renderingu, vďaka čomu je WebGL úplne hardvérovo urýchľované priamo v prehliadači. Hlavnou nevýhodou WebGL je obtiažnejšia práca s textom a zatiaľ neexistujúce knižnice pre tvorbu diagramov z ktorých by sme mohli vychádzať.

CSS 3D transformácie taktiež podobne ako WebGL umožňujú vykresľovanie 3D grafiky vo webovom prehliadači avšak iným spôsobom, konkrétne prostredníctvom 3D efektov nad bežnými HTML (DOM) elementami. V porovnaní s WebGL sú elementy ľahšie štylovateľné, resp. v CSS3D je ich vzhľad

definovaný prostredníctvom kaskádových štýlov (CSS), no vo WebGL musí mať každý objekt definovaný materiál a prípadne aj svoju textúru.

WebGL je v porovnaní s CSS 3D "silnejší" z pohľadu rozsiahlosti možností pri tvorení 3D prostredia, avšak pre tvorbu UML diagramov je postačujúce CSS 3D. Veľkou výhodou CSS 3D sú existujúce knižnice pre tvorbu 2D diagramov, ktoré je možné zužitkovať taktiež v 3D prostredí a jednoduchosť spracovania HTML elementov do XMI formátu, prostredníctvom ktorého je možné importovať vytvorené diagramy do nástroja EA, ale taktiež ich aj exportovať z EA a následne spracovať do týchto HTML elementov. Naopak veľkou nevýhodou CSS 3D je obtiažna tvorba 3D objektov (konkrétne šípky) medzi jednotlivými vrstvami diagramov.

Analýza knižníc na tvorbu 2D diagramov

JointJS

JointJS je knižnica na tvorbu a vykresľovanie rôznych druhov diagramov. Funguje na základe práce s SVG elementami, ktoré ponúka HTML5 jazyk. Knižnica poskytuje MVC (alebo skôr MV) architektúru oddeľujúcu modely grafu, vzťahov a elementov od ich vykresľovania. To uľahčuje pripojiť JointJS k backendu aplikácie. Teda pri práci s jednotlivými elementami, pri ich špecifikácií sa upravuje model nie view. JointJS využíva knižnice JQuery, Underscore, Backbone, a SVG elementy.

Model - View štruktúra je definovaná nasledovne. Diagram (Graph) v JointJS pozostáva z elementov (Element) spojených vzťahmi (Link). Vykresľovacia plocha (Paper) pozostáva z ElementView a LinkView a je v nej definovaný model diagramu (Graph), ktorý upravujeme. V praxi to znamená, že ak chceme na vykresľovaciu plochu vykresliť diagram a jeho elementy, musíme si definovať Paper, určiť mu rozmery, element v HTML, na ktorý sa ma plocha vykresliť (najčastejšie <div>) a hlavne model (Graph), v ktorom definujeme vzťahy a elementy.

JointJS ponúka široké možnosti definovania vlastných typov elementov a vzťahov a taktiež možnosť ukladania modelov elementov na backend. Chýbajú ale vytvorené elementy pre sekvenčný diagram a taktiež overenie diagramu na základe metamodelu. Keďže sa JointJS používa skôr na diagramy podobné diagramom aktivít, diagramu tried a pre sekvenčný diagram platia špecifické pravidlá pre vzťahy medzi lifelinami by sme sa mohli pri použití JointJS stretnúť s množstvom problémov, či už pri vykresľovaní alebo definovaní jednotlivých elementov pre sekvenčný diagram.

GoJS

GoJS je prispôsobiteľná knižnica pre tvorbu interaktívnych diagramov, ktoré sú následne vykresľované prostredníctvom webových prehliadačov a platforiem. GoJS vytvára JavaScriptové diagramy na základe komplexných uzlov (nodes), odkazov (links) a skupín elementov (groups), ktoré sú prispôsobiteľné prostredníctvom šablón a špecifikácií. GoJS používa čistý JavaScript a grafiku renderuje do HTML canvas alebo SVG elementov.

GoJS ponúka rozšírenú funkcionálnu interakciu, ako je napríklad drag and drop, kopírovanie a vkladanie elementov, priama editácia textu, kontextové menu, automatické rozloženie elementov, šablóny a spracovanie udalostí. Výhodou je, že je táto funkcionálna z veľkej miery prispôsobiteľná taktiež nie je závislá na žiadnych JavaScriptových knižniciach a rámcoch. Ďalšou veľkou výhodou je, že GoJS už obsahuje implementáciu šablón pre základné elementy sekvenčného diagramu, konkrétne čiary života (lifeline), správy (message) a blok akcie (action block). Tieto elementy sú už síce implementované no vytvorenie nových elementov a ich dodatočné prispôbenie môže byť

problematické. Nevýhodou GoJS môže byť taktiež jeho použitie v TypeScript, keďže základné elementy sekvenčného diagramu sú implementované v čistom JavaScripte, je potrebné kód týchto elementov transformovať do TypeScriptu.

Zhodnotenie

Prostredníctvom analýzy vykresľovacích technológií sme dospeli k poznatkom, na základe ktorých sme vytvorili porovnanie WebGL a CSS 3D. Toto porovnanie je zobrazené v tabuľke č.1.

WebGL		CSS 3D	
Výhody	Nevýhody	Výhody	Nevýhody
Hardvérovo urýchľované	Problém v práci s textom	Jednoduché selektovanie elementov	Problém s vytvorením šípky medzi vrstvami
Možnosť použitia shaderov	Neotvorený zdrojový kód	Zapojenie JS, JQuery	
Možnosť použitia osvetľovania scény	Nižšia podpora ako v prípade CSS 3D	Uloženie do XML (cez html dom)	
Silnejšie ako CSS 3D		Možnosť otáčať celé vrstvy	
Natívne pre Three.js rendering		Existujúce 2D UML knižnice	
Možná podpora pre VR			

Tabuľka: Porovnanie WebGL a CSS 3D

Na základe analýzy knižníc na tvorbu 2D diagramov sme vytvorili nasledujúce provnanie GoJS a JointJS knižníc.

GoJS

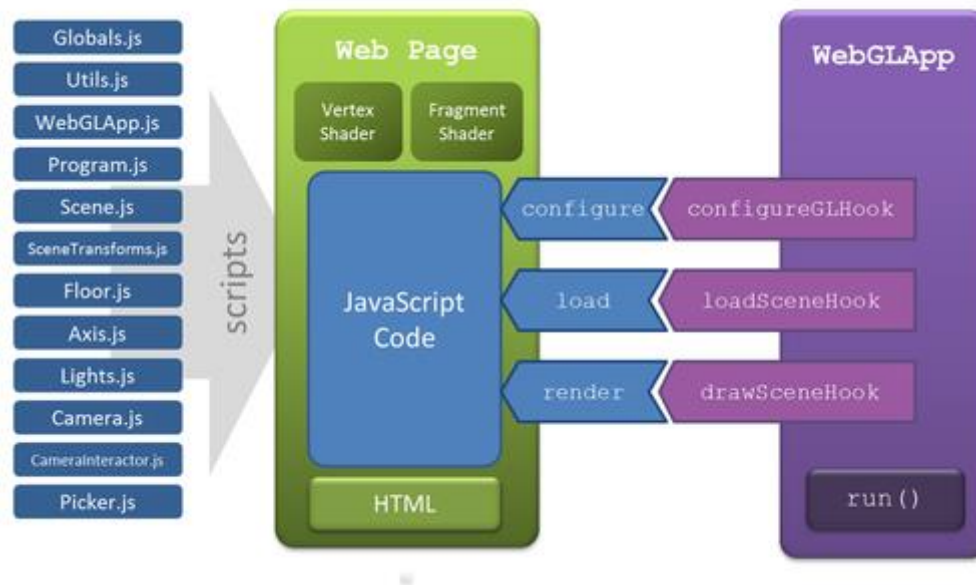
- Výhody:
 - Vytvorené šablóny pre základné elementy sekvenčného diagramu
 - Možnosť prispôbovať funkcionality knižnice
 - Možnosť definovať vlastné elementy a rozširovať existujúce
- Nevýhody
 - Chýba overenie na základe metamodelu sekvenčného diagramu
 - Možné problémy pri prispôbovaní funkcionality a vytváraní šablón

JointJS

- Výhody:
 - Možnosť definovať vlastné elementy a rozširovať existujúce
 - Jednoduché prepojenie modelov elementov s backendom
 - Prehľadná Model-View štruktúra (pracuje sa s modelmi, view len renderuje)
- Nevýhody
 - Nedefinované elementy sekvenčného diagramu (potreba definovať vlastné)
 - Chýba overenie na základe metamodelu sekvenčného diagramu
 - Možné problémy pri práci s niektorými špecifickými elementami (action block, lifeline) a ich renderovaní

Návrh

Prototyp aplikácie bude pozostávať iba z klientskej časti aplikácie. Tá sa bude implementovať pomocou jazyka Javascript a bude vykresľovaná na technológii WebGL s knižnicou Three.js. Prototyp teda nebude obsahovať žiadne serverové komponenty, žiadnu komunikáciu so serverovou časťou a žiadnu databázu. Na obrázku nižšie vidíme spôsob komunikácie WebGL a samotnej aplikácie.



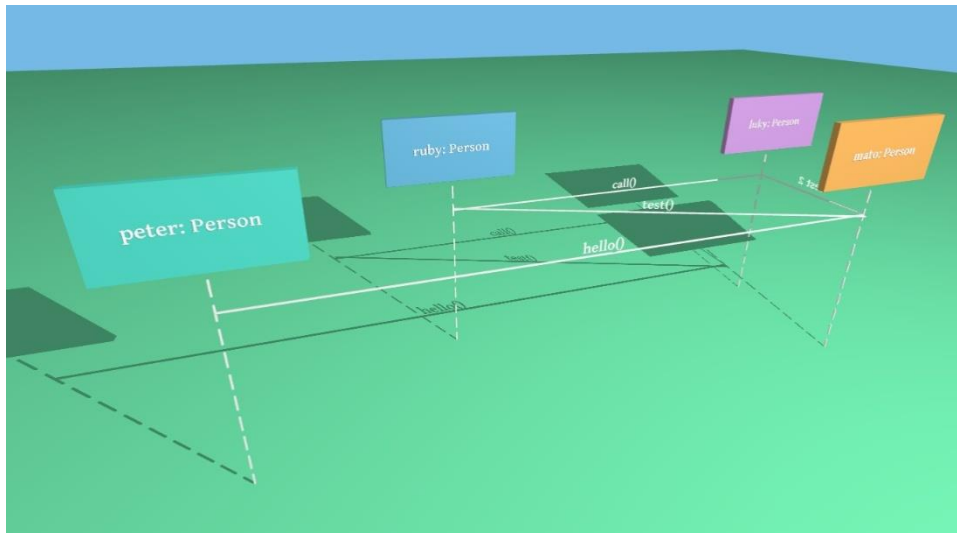
Obrázok: Prepojenie WebGL a stránky

Pre potreby overenia funkčnosti nie je potrebné zaoberať sa správnou a udržateľnou štruktúrou dát, prototyp bude slúžiť iba na overenie funkcionality vykresľovania diagramu jednotlivými knižnicami. Dáta do diagramu budú napevno určené v kóde aplikácie a nebude možné ich meniť z klientskej strany.

Implementácia

Prototyp WebGL technológie

Ako rozšírenie analýzy vykresľovacích technológií a knižníc sme sa rozhodli implementovať prototyp aplikácie založenej na technológii WebGL. Cieľom tohto prototypu bolo preukázať možnosti technológie WebGL a knižnice Three.js. Po následnej diskusii a po zvážení všetkých kladov a záporov s ohľadom na udržateľnosť a rozšíriteľnosť aplikácie sme sa však nakoniec rozhodli ďalej aplikáciu s využitím technológie WebGL nevyvíjať. Namiesto nej sme zvolili alternatívnu technológiu CSS3 3D transformácií. Nasledujúci obrázok ilustruje vytvorený prototyp aplikácie pomocou technológie WebGL.

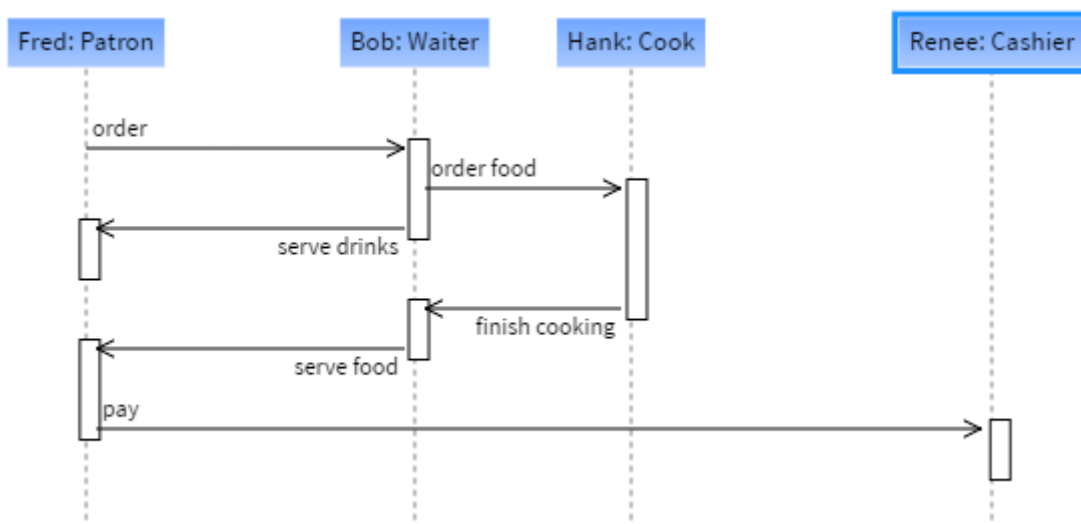


Obrázok: Ukážka WebGL prototypu aplikácie

Knižnica na tvorbu 2D diagramov

Na základe vedomostí, ktoré sme nadobudli počas analýzy vykresľovacích technológií sme sa rozhodli do nami vyvíjanej aplikácie integrovať knižnicu GoJS, kvôli jej vhodným vlastnostiam. Knižnica GoJS nám umožňuje vykresľovať UML diagramy do HTML elementu "canvas". Na docielenie 3D efektu UML diagramov je následne možné aplikovať CSS3 3D transformácie. Vďaka nim sú UML diagramy vizuálne zobrazené ako 3D prvky, medzi ktorými je následne možné vykresľovať spojenia.

Tento spôsob práce v kombinácii s viacerými knižnicami a technikami nám umožňuje využiť pozitívne vlastnosti oboch technológií, ako HTML s využitím 2D grafiky v elemente "canvas", tak aj jazyka CSS3 na zavedenie 3D aspektu do aplikácie. Dekomponovanie 3D scény na 2D vrstvy prináša výhodu z hľadiska možnosti použitia už existujúcich knižnic na tvorbu UML diagramov na jednotlivých vrstvách. Na obrázku nižšie môžeme vidieť ako vyzeral sekvenčný diagram v knižnici GoJS.



Obrázok: Príklad diagramu z knižnice GoJS

Testovanie

Po implementovaní GoJS sekvenčných elementov do AngularJS sme dospeli k záverom, že niektorá funkcionálnosť elementov je správna, ale niektoré elementy sekvenčného diagramu (napr. kombinovaný fragment, action bloky) je potrebné vytvoriť a prispôbiť.

Serverová aplikácia

V tejto kapitole opisujeme časť aplikácie, ktorej hlavnou úlohou je perzistencia dát a ich následné poskytovanie pri požiadavke. Ide o vytvorenie aplikačného rozhrania na základe protokolu, ktoré následne poskytuje dáta klientovi.

Analýza

Analýza serverových prostredí

PHP

PHP je interpretovaný jazyk na serverovej strane, ktorý pôvodne vznikol na vyvíjanie webových aplikácií, neskôr sa však začal používať aj ako všeobecne použiteľným programovací jazyk.

Tento objektovo orientovaný programovací jazyk je možné použiť spolu s HTML kódom, alebo napríklad aj v rámci frameworku, alebo web content management systému. Funkcie v PHP sú blokované, až do ukončenia, čiže nejaký príkaz sa vykoná až keď bol predchádzajúci príkaz splnený.

NodeJS

Open source, multi platformové Javascriptové prostredie na vytváranie aplikácií. Nie je to Javascript framework, aj napriek tomu, že viaceré moduly sú napísané práve v Javascripte. Má event driven architektúru schopnú asynchrónnych vstupno-výstupných operácií. Tieto vlastnosti majú za úlohu optimalizovať priepustnosť a škálovateľnosť webových aplikácií, zameraných najmä na obrovské množstvo I/O operácií, ako sú napríklad hry v prehliadači, alebo komunikácia v reálnom čase.

Aplikácie NodeJS podporujú Mac OS X, Microsoft Windows, NonStop a Unix servery. Alternatívne vedia byť tieto aplikácie vytvorené pomocou CoffeeScript, TypeScript, alebo ľubovoľného iného jazyka skompilovateľného do Javascriptu.

Využitie má podobné ako PHP, teda na tvorbu webových serverov. Hlavný rozdiel je však vo vykonávaní inštrukcií. Funkcie v NodeJS nie sú blokujúce, a teda vedia byť vykonané paralelne, pričom na signalizáciu dokončenia, alebo zlyhania použijú callback.

NodeJS používa Event driven programovanie na vytváranie rýchlych webových serverov. Vďaka tomuto programovaciemu vzoru sú servery v NodeJS vysoko škálovateľne aj bez použitia vlákien.

Zend vs Laravel

Porovnanie v dnešnej dobe populárnych frameworkov na prácu s programovacím jazykom PHP.

Vlastnosť	Laravel	Zend
licencia	MIT	BSD
databázový model	relačný	objektovo orientovaný
vývojové princípy	Test Driven Development	

	žiadne opakovania	
	konfigurácia pred pohodnosťou	konfigurácia pred pohodnosťou
programovacie paradigmy	Event driven	Event driven
	objektovo orientované	objektovo orientované
	funkcionálne	
podpora skriptovacích jazykov	PHP, Javascript	PHP
znoupoužitelnosť kódu	Model View Controller	Model View Controller
	webové služby	
podpora Maven	nie	áno
pripojená Javascript knižnica	nie	áno
čiastočné triedy	áno	nie
návrhové vzory	Active record	Active record
	Model View Controller	Model View Controller
	Dependency injection	Dependency injection
	Event driven	Event driven
	Observer	
	Facade	
		Data mapper
databázy	Sqlite, Mysql, Postgresql, Redis, Microsoft Bi	Sqlite, Mysql, Oracle, Postgresql, Microsoft Bi, Mongoddb, Mariadb

Analyza komunikačných protokolov

JSON API

Zadefinováva štruktúru, akou bude JSON formátovaný. Tým, že je presne zadefinovaná forma súboru JSON sa zvyšuje produktivita a prehľadnosť kódu.

Okrem štruktúry dokumentu definuje aj správanie klienta a serveru. Napríklad klient musí posielat všetky JSON API dáta v request dokumente s hlavičkou :

```
Content-Type: application/vnd.api+json
```

a neobsahujú žiadne parametre typu média. Pri odpovedi zase musí ignorovať všetky parametre typu média, ktoré sú v hlavičke súboru. Server má obdobne zadefinované správanie pri komunikácii s klientom.

Jednoduchý príklad :

```
{ "data": { "type": "articles", "id": "1", "attributes": { // ... this article's attributes }, "relationships": { // ... this article's relationships } } }
```

JSON RPC

V JSON formáte zakódovaný protokol na vzdialené volanie procedúr (podobný ako XML-RPC pre formát štandardu XML). Jednoduchý protokol, ktorý definuje iba základné dátové typy a príkazy. Podporuje posielanie viacerých dopytov na server, ktoré nemusia byť spracované v poradí, v akom dané

dopyty odišli. Okrem toho ponúka aj možnosť jednosmernej komunikácie formou notifikácii, kedy serveru klient odošle dáta, na ktoré nepotrebuje odpoveď.

Funguje na princípe posielania dopytov na server, kde je tento protokol implementovaný. Podporuje možnosť viacerých vstupov ako parameter volania, kedy sú tieto vstupy uložené ako pole, alebo ako objekt. Metóda takisto môže vrátiť viacero hodnôt na výstupe.

Všetky transférové typy sú objekty serializované pomocou JSON. Každé volanie musí mať následne tri vlastnosti :

- Method – meno metódy, ktorú voláme
- Params – objekt alebo pole hodnôt, ktoré sa majú predať metóde ako parameter
- Id – hodnota určená na spárovanie dopytu a odpovede

Server zase musí odpovedať validnými odpoveďami, ktoré obsahujú tieto vlastnosti:

- Result – vrátené dáta ako výstup metódy, null ak prišlo ku chybe vo vykonávaní metódy
- Error – číslo chybového hlásenia, ak ku chybe nedošlo nadobúda hodnotu null
- Id – identifikačné číslo dopytu na ktorý server odpovedá

Notifikácie majú podobnú štruktúru, akurát je vyhodnené, alebo nastavené na hodnotu null pole id, ktoré nie je potrebné, keďže neposielame odpoveď.

XML

Extensible Markup Language, ktorý definuje súbor pravidiel pre štruktúru dokumentu tak, aby boli zrozumiteľné aj pre ľudí, aj pre počítače. Obdobne ako JSON je to teda ideálny formát dát na výmenu medzi serverom a klientom.

Svojou syntaktickou skladbou pripomína HTML, kde sú tiež všetky informácie obkolesené takzvanými tagmi. Tieto tagy určujú informáciu akého typu sa jedná a samotnú informáciu. Existujú tri druhy tagov:

- Štartovacie - <section>
- Ukončovacie - </section>
- S prázdny elementom -

Element je potom informácia uchovávaná medzi štartovacím a ukončovacím tagom. V týchto tagoch ďalej môžu byť vnorené atribúty, ktorými bližšie definujeme daný element. Môže ísť o doplnujúcu informáciu, akými sú napríklad vek, farba, ...

Štandard ďalej definuje, akú hlavičku musí obsahovať XML súbor, aby bol validný, špecifikuje aké znaky sa môžu vyskytovať pri menách elementov a oveľa viac.

Porovnanie XML a JSON

JSON

Výhody:

- Jednoduchá syntax
- Jednoduché na použitie s Javascriptom, keďže obsahuje rovnaké základné dátové typy
- Vieme použiť schému aj JSON API na presné dodefinovanie štruktúry

- JsonPath na extrahovanie informácií v hlboko vnorených štruktúrach

Nevýhody:

- Jednoduchá syntax, iba základné dátové typy sú podporované

XML

Výhody:

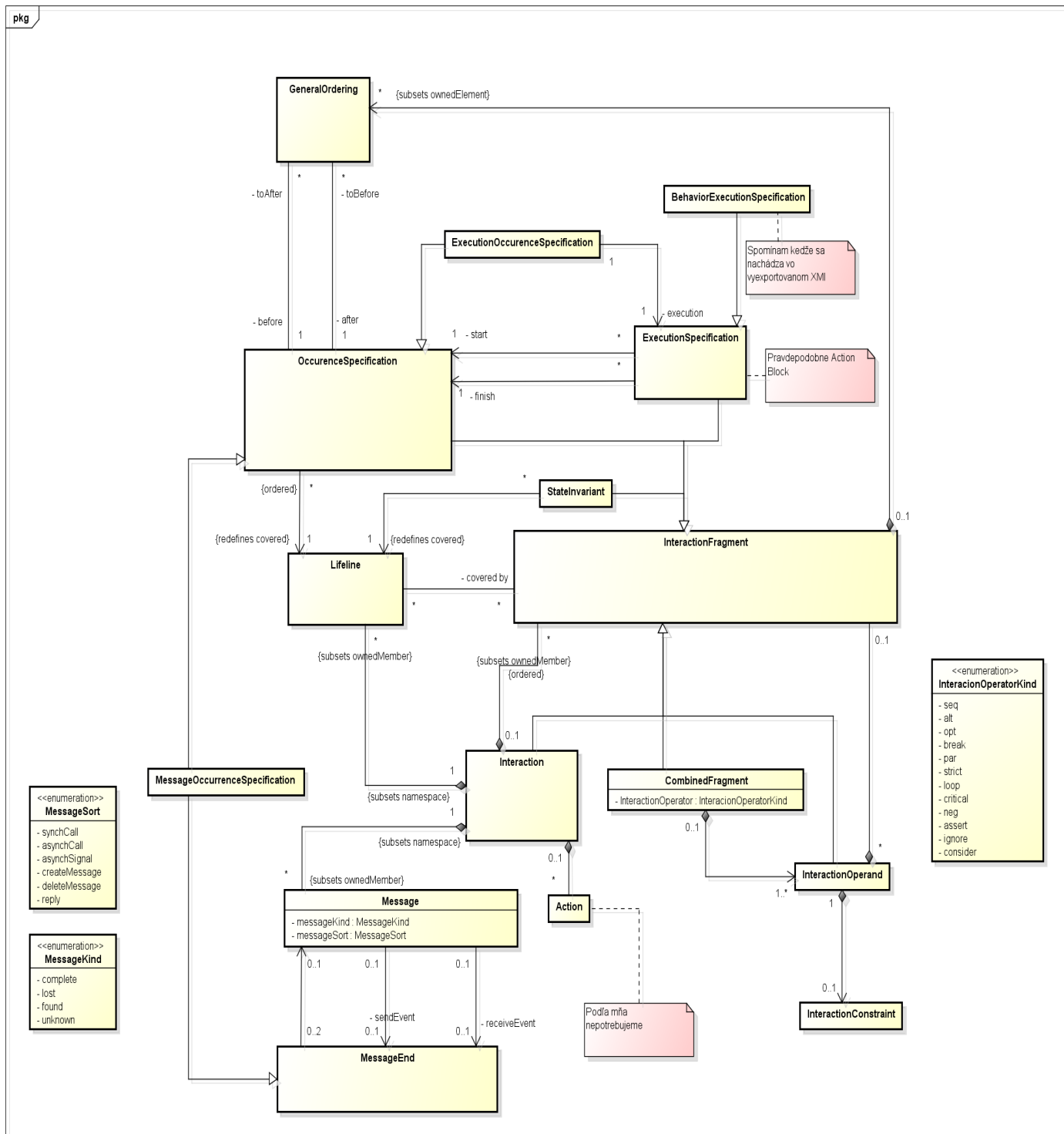
- XML schéma na validáciu štruktúry, alebo tvorbu vlastných dátových typov
- Možnosť transformácie na rozdielne výstupné formáty pomocou XSLT
- XPath na extrahovanie informácií v hlboko vnorených štruktúrach
- Vstavaná podpora pre namespace

Nevýhody:

- Obsahuje viac pomocných informácií, čím sa stáva súbor väčší, a teda potrebujeme preniesť viac dát, aby sme dostali rovnaké množstvo informácií

Metamodel sekvenčného diagramu

Metamodel sekvenčného diagramu v UML predstavuje štruktúru prepojení všetkých elementov v sekvenčnom diagrame. Práve z tohto metamodelu sme vychádzali, a to preto, aby sme dokázali správne určiť vzťahy medzi elementami a následne ich použiť pri práci a manipulácii s nimi na scéne.



Obrázok: Metamodel sekvenčného diagramu

Dôležité časti metamodelu sú:

- **Message** - správa medzi lifeline-ami, ktorá môže byť v stave complete, lost, found alebo unknown. Táto správa má taktiež aj typ a tieto typy sú: synchronne volanie, asynchronne volanie, vytvorenie, vymazanie, odpovedanie
- **Interaction** – predstavuje akoby celý sekvenčný diagram (napr. Diagram s tromi lifeline a medzi nimi 6 message tvorí jednu interakciu)

- **Lifeline** – predstavuje čiaru života v sekvenčnom diagrame. Obsahuje rolu alebo objektové inštancie, ktoré sa zúčastňujú v sekvenčnom diagrame
- **CombinedFragment** - kombinovaný fragment, ktorý môže byť napr.: alt - alternatíva, loop - cyklus... Takýto fragment môže obsahovať inštanciu samého seba (to znamená, že sú v ňom vnorené ďalšie iné combined fragmenty)
- **InteractionOperand** – je operand interakcie (napr. $A + B$, tak 'A', 'B' sú operandy a '+' je operátor)
- **InteractionConstraint** – je podmienka, ktorá ak platí, tak sa vykoná daný typ operátora
- **GeneralOrdering** – poradie výskytov daných elementov. Navrhnutý ako spájaný zoznam výskytov. To znamená že každý prvok v rámci lifeliney vie, ktorý prvok ide za ním.
- **ExecutionOccurrence** – predstavuje action block
- **OccurrenceSpecification** – táto trieda značí výskyt. Hovorí o tom kde sa element začína a kde sa končí. Dedí od nej ďalšie dve triedy, ktoré konkretizujú výskyt správy (MessageOccurrenceSpecification) a action bloku (ExecutionOccurrenceSpecification).

Zhodnotenie

Pri výbere serverového programovacieho jazyka boli analyzované hlavne PHP a Node.js. PHP je samostatný programovací jazyk, zatiaľ čo Node.js je postavené na Javascripte a poskytuje funkcionality Javascriptu aj na strane servera. PHP je starší, zaužívaný jazyk, ktorý podporuje všetky dostupné platformy operačných systémov. Za dlhé obdobie jeho používania vzniklo viacero frameworkov, ktoré pridávajú, prípadne upravujú funkcionality čistého PHP jazyka. Node.js má hlavnú výhodu, že nemusí byť postavený iba na Javascripte, ale aj nad CoffeeScriptom, Typescriptom, alebo ľubovoľným iným jazykom, ktorý je možné skompilovať do Javascriptu.

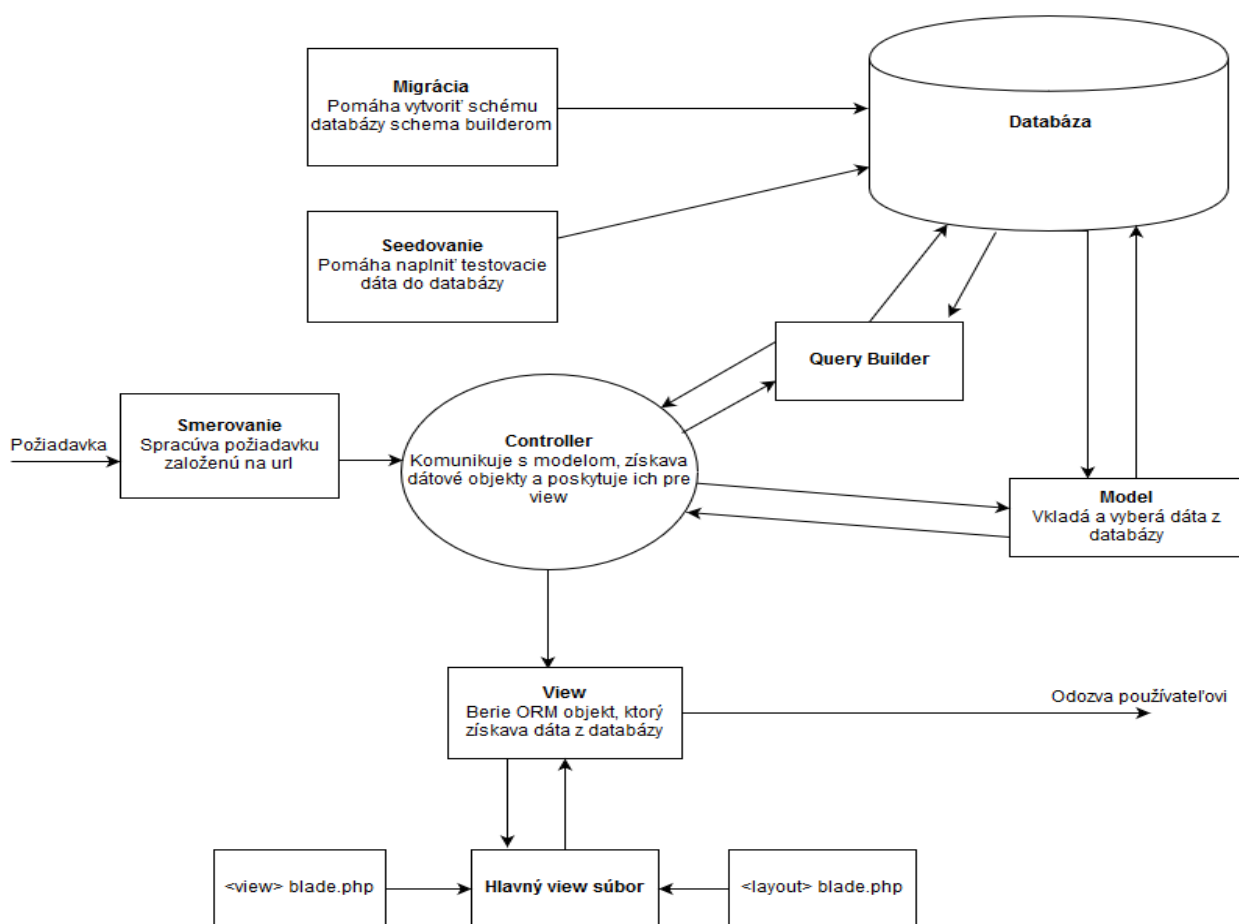
Dostupné frameworky na PHP sú napríklad Laravel, alebo Zend. Oba frameworky poskytujú Model View Controller na implementovanie používateľských rozhraní. Laravel podporuje okrem jazyka PHP aj Javascript a je teda univerzálnejším. Líšia sa možnými použiteľnými databázami, alebo aj podporou Maven.

Metamodel tak, ako je vyššie zadaný bude použitý v ďalších fázach vývoja aplikácie, kedy z neho bude vychádzať napríklad fyzický model databázy, ale aj jednotlivé triedy, ktoré potrebujeme vytvoriť na správnu prácu s dátami.

Komunikačné protokoly sú dva hlavné (XML a JSON). XML definuje štruktúru predávania informácií v zrozumiteľnom tvare aj pre stroje, aj pre človeka. Štruktúra sa skladá z tagov, v ktorých sú vložené informácie. Tento jazyk však obsahuje viacero pomocných informácií, ktoré zvyšujú potrebné množstvo dát na prenesenie. Oproti tomu je JSON API, ktoré presne definuje štruktúru a konvencie pri vytváraní JSON súboru. Je jednoduchší, ľahší a vďaka štandardizácii aj dobre čitateľný a použiteľný.

Návrh

Architektúra serverovej aplikácie



Obrázok: Dekompozícia aplikácie backendu

Na obrázku vyššie je znázornená architektúra, ktorú používame pri vývoji naše aplikácie. Odvíja sa od návrhového vzoru MVC (model-view-controller). Takto sme schopní oddeliť aplikačnú logiku od zobrazovania komponentov. Ktokoľvek bude teda schopný zmeniť stratégiu vykresľovania diagramu a nebude nutné meniť dátový model. Popri architektonickom vzore na obrázku vyššie využívame aj iné, už vopred vyvinuté a overené programátorské prístupy.

ORM

Využívame mapovaciú techniku ORM, ktorou zaistíme synchronizáciu medzi použitými objektami v aplikácii tak, aby bola zaistená perzistencia dát. Taktiež je použitá aj z dôvodu, že často redukuje množstvo kódu, ktorý je potrebný písať.

Lazy Loading

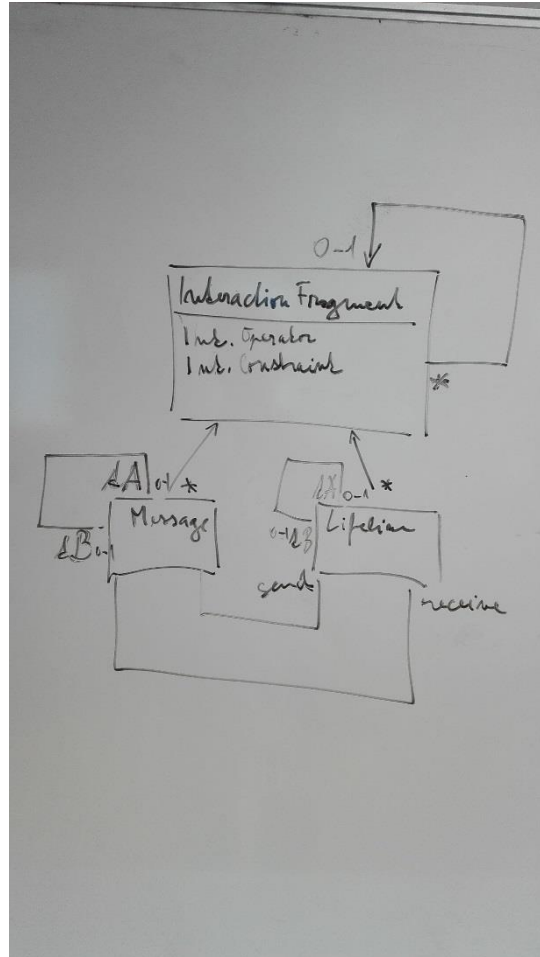
Jeho využitie je dôležité z toho hľadiska, že nemusíme načítať do pamäte všetky dáta z databázy, ale vždy len práve tie, ktoré potrebujeme. Tento prístup je vhodný, keď používame viacero tabuliek v databáze a potrebujeme sa dopytovať do nich naraz a nie vtedy ak máme všetky údaje v jednej tabuľke.

Dependency injection

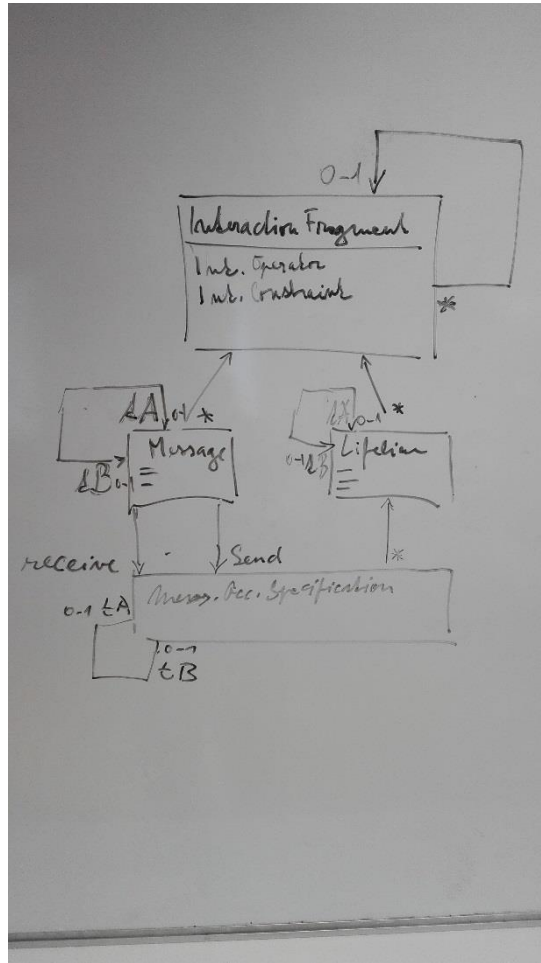
Aby sme udržali kód čo najčistejší a zbavený závislostí, využívame taktiež dependency injection. Zaručíme tým teda dekompozíciu aplikácie a taktiež zjednodušíme testovanie.

Fyzický dátový model

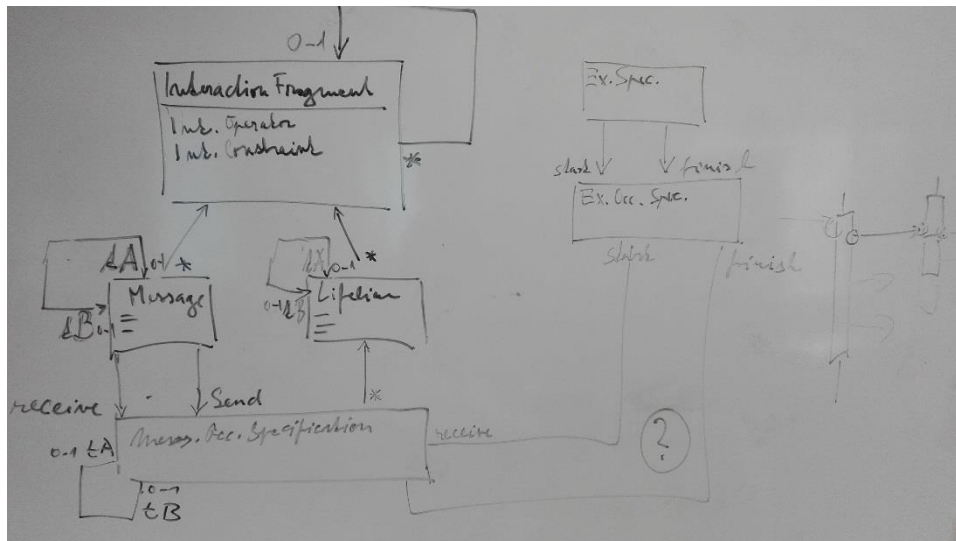
V nižšie uvedenom diagrame sa nachádza fyzický dátový model. Na obrázkoch nad diagramom sa nachádza postup ako sme ho vypracovávali v spolupráci s našim vedúcim. Pri jeho vytváraní sme sa držali metamodelu z UML superstructure. Jednotlivé triedy reprezentujú dané elementy v superstructure a vzťahy medzi nimi.



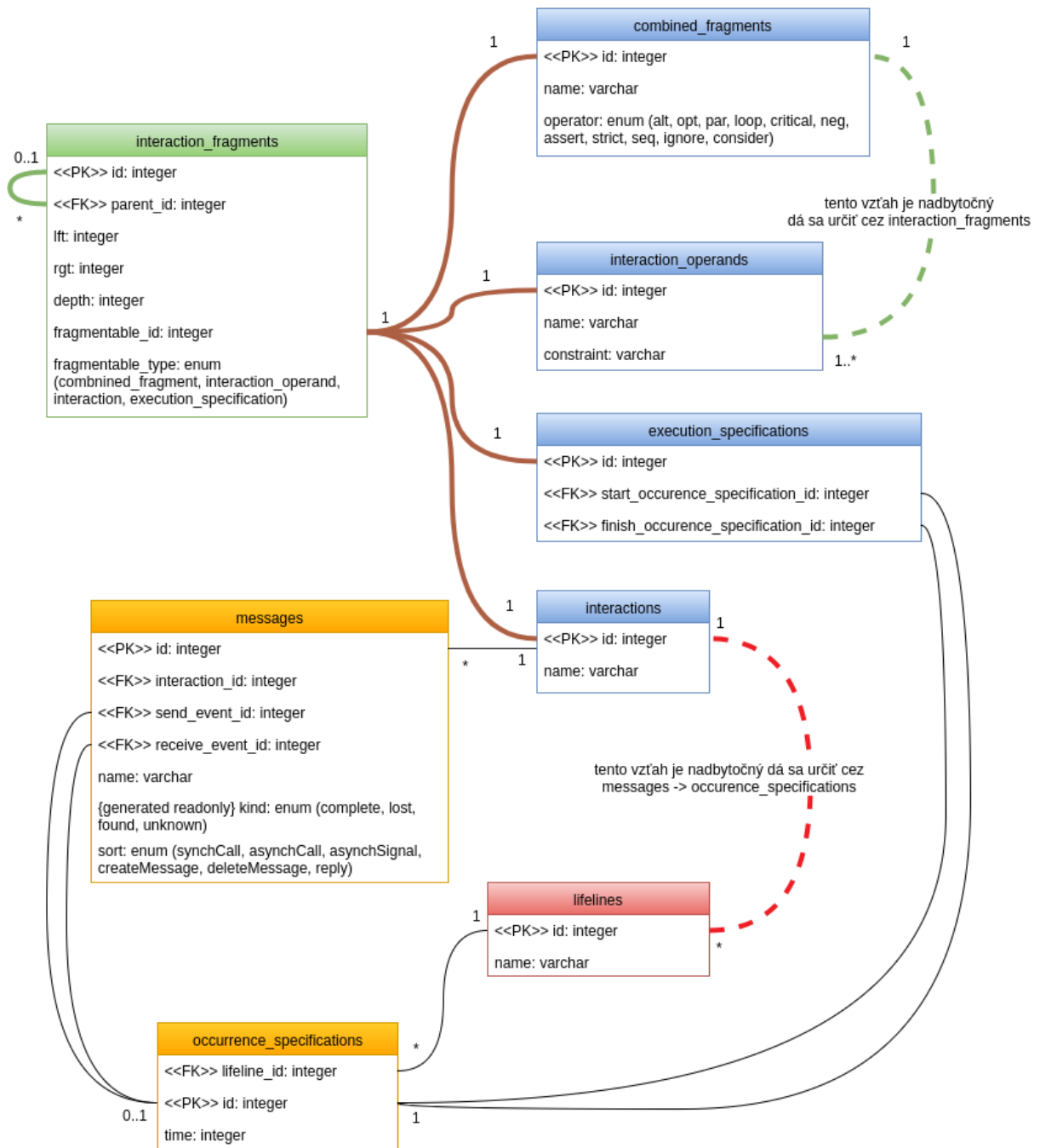
1.krok - identifikácia tried



2. krok – doplnenie vzťahov a entity



Obr.: Postup vytvárania fyzického dátového modelu



Obrázok: Fyzický model

- **interaction_fragments** - abstraktná trieda reprezentujúca fragment v sekvenčnom diagrame. Vďaka nej sme schopní vnímať sekvenčný diagram ako kompozitnú štruktúru. Všetky fragmenty diagramu majú vzťah s touto tabuľkou 1 k 1. Následne teda každý fragment môže mať vzťah samého so sebou a diagram sa teda chová ako kompozit.
- **combined_fragments** - kombinovaný fragment, ktorý v sebe môže niesť ďalšie kombinované fragmenty. Vo fyzickom modeli riešime túto skutočnosť tak, že sa odkážeme cudzím kľúčom v interaction_fragments na samého seba. Tým pádom sa vieme rekurzívne vnárať do podfragmentov. Kombinované fragmenty sú typu alt, opt, par a pod.
- **interaction_operands** – predstavuje operand vo fragmente. Jeden operand môže v sebe obsahovať aj iné operandy, a taktiež viacero kombinovaných fragmentov.

- **interactions** – interakcia, ktorá predstavuje celý sekvenčný diagram
- **messages** – predstavujú správy medzi jednotlivými lifeline-ami. Obsahuje cudzí kľúč na interakciu, keďže správa existuje len v rámci interakcii a v jednej interakcii môže byť viacero správ. Správa má niekoľko typov ako: complete, lost, found, unknown a taktiež niekoľko rozdelení: synchronná, asynchrónna, vytvorená, vymazaná, alebo ako odpoveď. Viacero správ prislúcha iba jednej lifeline-ne. Cudzí kľúče send_event_id a receive_event_id predstavujú správu, ktorá je odoslaná a správu, ktorá je prijatá.
- **lifelines** - znázorňuje čiaru života. Každá čiara života má id interakcie, do ktorej patrí ako cudzí kľúč. Má teda vzťah s interakciou viac ku 1. Taktiež má vzťah so správami, ktoré na čiare života začínajú alebo končia. Teda vzťah je vyjadrený ako 1 ku viac. Je to dvojnásobne zastúpený vzťah keďže sa zvlášť zameriava na začiatok správy a zvlášť na jej koniec.

Implementácia

V tejto kapitole popisujeme programové prostredie, databázu a protokol komunikačného rozhrania, ktoré sme sa rozhodli využiť pri realizovaní implementácie aplikácie.

Programové prostredie

Ako programovací jazyk serverovej aplikácie sme si zvolili jazyk PHP. Tento jazyk nám v kombinácii s frameworkom Laravel poskytol všetky požadované vlastnosti na serverové prostredie nami vyvíjanej aplikácie. Architektúra frameworku Laravel je založená na kombinácii viacerých návrhových a architektonických vzorov. Vďaka tomu je možné aplikáciu jednoduchým a prehľadným spôsobom dekomponovať a organizovať na samostatné moduly.

Základným stavebným vzorom je MVC (Model-View-Controller). Ten nám poskytuje možnosť oddelenia biznis logiky od zobrazení a vrstvy dát. Vrstva dát obsahuje dátové modely. Framework Laravel tiež obsahuje implementovaný návrhový vzor Dependency Injection, ktorý tiež napomáha pri dekomponovaní a testovaní aplikácie. Aplikácia samotná je implementovaná ako IoC (Inverse of Control) kontajner. Injekcia závislostí jednotlivých modulov je následne realizovaná naprieč všetkými časťami frameworku.

Framework Laravel navyše obsahuje funkcionality, ktorá nám uľahčí prácu s databázou. Ide napríklad o systém databázových migrácií spolu s naplnením databázy testovacími údajmi, ktoré nezávisia na použítom type databázy ani na príslušnom SQL dialekte. Rovnako je obsiahnutý aj objektovo relačný mapovač Eloquent, ktorý umožňuje mapovanie dát na dátové modely aplikácie.

Perzistencia údajov

Je potrebné, aby nami vytvorená aplikácia bola schopná dlhodobej perzistencie UML diagramov. S ohľadom na budúcu možnosť importovania a exportovania UML diagramov do zaužívaných formátov sme sa rozhodli údaje o UML diagramoch taktiež ukladať aj do relačnej databázy.

Zvolili sme si databázu PostgreSQL kvôli jej kvalite, rozšírenosti a otvorenosti. Ukladanie UML diagramov do databázy vo forme fyzického dátového modelu nám tiež umožňuje automatické overovanie ukladaných dát voči metamodelu.

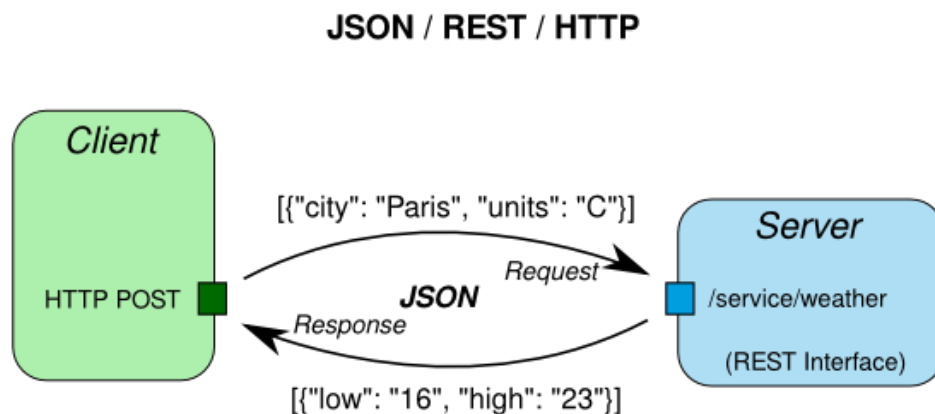
Prepojenie medzi databázou PostgreSQL a frameworkom Laravel je realizované pomocou ORM mapovača Eloquent. Tento mapovač obsahuje aj implementáciu užitočných návrhových vzorov ako napríklad lazy loading, čím je dosiahnutý vyšší výkon pri prístupe k údajom uloženým v databáze.

V ostatných častiach aplikácie je možné pracovať s dátovými entitami ako s objektami bežných tried, ktoré sa nazývajú dátové modely. Dátové modely obsahujú relácie v podobe metód, ktoré umožňujú prácu s príslušnými väzbami na iné dátové modely.

Protokol komunikačného rozhrania

Aby sme vedeli zabezpečiť výmenu informácií medzi serverovou časťou aplikácie a klientami potrebujeme presne stanovený formát a pravidlá. Z analýzy možných formátov sme si ako formát na výmenu dát vybrali JSON, najmä pre jeho rozšírenosť, jednoduchosť použitia a interoperabilitu.

Špecifikácia JSON API je sada generických konvencií formátovania požiadaviek a odpovedí na zdroje z angl. resources. Jeho hlavnou výhodou je definovanie všeobecných pravidiel, ktoré možno použiť na akúkoľvek doménu riešenia. Špecifikácia neobsahuje len pravidlá na dopytovanie na zdroje ale taktiež aj na ich aktualizáciu, či mazanie. Obsahuje nielen atribúty zdroja ale taktiež aj jeho vzťahy na iné zdroje spolu s odkazmi na dané zdroje. Umožňuje aj filtrovanie výsledkov či stránkovanie. Ďalšou významnou výhodou je jeho štandardizácia a teda možnosť použitia knižnice do zvoleného jazyka, ktorá spĺňa špecifikáciu. To nám umožňuje jednoducho implementovať aplikačné rozhranie a toto rozhranie konzumovať klientami opäť použitím knižnice na strane klienta pre transparentnú prácu nad zdrojmi. Nasledujúci obrázok zobrazuje komunikáciu medzi klientskou aplikáciou so serverom pomocou špecifikácie JSON API.



Obrázok: Komunikácie klientskej aplikácie so serverom pomocou JSON API

Testovanie

Testovanie funkcionality migrácií v Laraveli prebehlo úspešne. Dané tabuľky boli v databáze vytvorené. Taktiež prepojenie JSON API medzi frontendom a backendom prebehlo úspešne na základe manuálnych testov porovnania požadovaných výsledkov.

Jadro klientskej aplikácie

Nasledujúca kapitola opisuje aplikáciu, ktorá beží v prehliadači a zabezpečuje celkovú interakciu s používateľom aplikácie.

Analýza

Analýza klientských prostredí

Typescript

Je striktným supersetom Javascriptu, ktorý pridáva statické písanie a objektovo orientované programovanie do jazyka. Vieme ho použiť na vytváranie klientských aplikácií, ale aj serverových založených na NodeJS.

Typescript je navrhnutý na vývoj veľkých aplikácií a ich následnej kompilácie do Javascriptu. Keďže je Typescript supersetom Javascriptu, všetky existujúce Javascript programy sú kompatibilné aj s Typescript programami.

Typescript obohacuje Javascript napríklad pridaním anotácií, interface, enum typov, namespace,...

Výhody:

- Superset Javascriptu, čiže vie pracovať aj s budúcimi vlastnosťami pridanými do Javascriptu
- Odhalenie chybových hlásení priamo pri kompilácii do Javascriptu
- Dedenie, interface a ďalšie OOP štandardy
- Syntax podobná pre Java, alebo C# vývojárov
- Využívané napríklad aj Angular frameworkmi

Nevýhody:

- Náročnejšie pre začiatočníkov objektovo orientovaného programovania
- Nutnosť použiť Typescript definície aj pre knižnice, ktoré nie sú v Typescripte
- Vyžaduje najskôr skompilovanie kódu, nevieme teda kód priamo spustiť
- Náročnejšie vytváranie, buildovanie, testovanie projektu, hlavne keď viaceré automatizačné nástroje ešte nepodporujú úplne Typescript

Javascript

Vysoko úrovňový, dynamický a interpretovaný programovací jazyk, ktorý bol štandardizovaný v rámci ECMAScript špecifikácie. Javascript nie je úplne procedurálny jazyk, má toho veľa podobného aj s funkcionálnymi jazykmi, ako napr. Lisp. Obsahuje aj viacero funkcionalít, ktoré pripomínajú objektovo orientované programovanie, ale v skutočnosti neobsahuje triedy.

Výhody:

- Jednoduchý jazyk
- Vykonáva sa na klientovej strane, čím šetrí záťaž servera
- Rozšírená funkcionalita na webových stránkach, vrátane širokej možnosti podporovaných prehliadačov
- Rýchle vykonávanie kódu
- Má prístup k DOM štruktúre

Nevýhody

- Bezpečnostné problémy, kedy takýto kód sa vykonáva u klienta a môže obsahovať škodlivý kód
- Vykreslenie závisí od jednotlivých prehliadačov

Porovnanie Angular 1 a Angular 2

Angular 2 je nová verzia frameworku Angular, ktorá sa od predchádzajúcej líši najmä tým, že už nepracuje s Javascriptom, ale Typescriptom. V prípade, že projekt bude postavený na Typescripte, Angular 1 nie je použiteľný a je treba prejsť na Angular 2. Hlavné zmeny, ktoré nastali vo frameworku s príchodom verzie 2:

- Využíva Hierarchical Dependency Injection systém na zvýšenie výkonu
- Širšia podpora jazykov, vrátane ES5, ES6, Typescript, Dart
- Webové štandardy sú implementované ako komponenty
- Komplikovanejšie nasadenie Angular 2
- Controller je nahradený komponentom
- Rozdielny spôsob definície lokálnych premenných
- Použitie camelCase na vstavané príkazy
- Priamo používa validné HTML DOM vlastnosti a udalosti elementov
- Zmena syntaxe pri Dependency Injection

Knockout.js

Samostatná Javascript implementácia Model-View-View model patternu. Je to teda webový framework, v ktorom sa ale ťažšie definuje infraštruktúra, resp. musí si ju používateľ zdefinovať sám. To môže tvoriť problémy v neskorších fázach vývoja aplikácie hlavne v prípade, bola štruktúra zle vytvorená. Využíva sa hlavne na kontrolu reprezentácie UI rozhrania pri menej komplexných aplikáciach. Nepodporuje natívne routing, ale dá sa externe pripojiť pomocou ďalších webových frameworkov.

Ďalšie vlastnosti:

- Dependency tracking
- Jednoduché asociovanie DOM elementov s modelovými dátami
- Pracuje na čistom Javascripte
- Deklaratívne väzby
- Jasné oddelenie doménových dát, viewov a dát na zobrazenie
- Data binding

Analýza komunikačných knižníc

Na implementáciu komunikácie klientskej aplikácie so serverom, kde rozhranie je implementované pomocou špecifikácie JSON API, potrebujeme knižnicu, ktorá dokáže na základe tejto špecifikácie vymieňať informácie so serverovou časťou. Do úvahy vzhľadom na výber frameworku Angular2 pripadajú dve alternatívy:

angular2-jsonapi

Knižnica jednoducho poskytuje volania na server pričom odpovede priamo mapuje na modely (DAO) triedy vo frameworku Angular 2. Medzi výhody patrí najmä jednoduché použitie v projekte. Ďalej poskytuje expresívnu definíciu JSON API modelov. Implementuje všetky základné časti špecifikácie a teda je možné nielen dopytovanie, ale aj aktualizácia a mazanie.

ngrx-json-api

Knižnica je rozšírením knižnice ngrx (Reactive Extensions for Angular), ktorá slúži na komunikáciu s aplikačnými rozhraniami a je inšpirovaná frameworkom React. Implementuje taktiež špecifikáciu JSON API, no neposkytuje tak expresívne a jednoduché rozhranie. Definícia modelov je len na úrovni opisu zdroja.

Analýza 3D grafických knižníc

Three.js

Open source projekt Javascriptovej knižnice zameraný na vytvorenie WebGL frameworku, ktorý je v súčasnosti aj jedným z najpoužívanejších. Vytvára sa ako ďalšia vrstva nad WebGL, ale zároveň umožňuje aj vykreslenie priamo na canvas pri zariadeniach, ktoré nepodporujú WebGL. Knižnica je dostatočne nízko úrovňová, aby ponúkala možnosti zmeny a úpravy kódu, ale zase nie je potrebné starať sa o komunikáciu s prehliadačom. Ponúka možnosť vytvorenia 3D animácií, ktoré sú akcelerované pomocou GPU.

Obsahuje obsiahlu dokumentáciu a príklady, ale jeho nevýhodou môže byť väčšia spotreba prostriedkov. Preto je vhodná pre menšie projekty, kde nároky na prostriedky nebudú až také vysoké. Knižnica má problémy so spätnou kompatibilitou. V prípade, že vyjde novšia verzia knižnice, staré demá, návody a kusy kódu môžu prestať fungovať.

Vlastnosti:

- Nízko úrovňový API
- Časovo náročnejší na vytvorenie základných vecí
- Možnosť vytvorenia vlastnej logiky a vlastných komponentov
- Nemožnosť exportu do 3D modelovacieho programu
- Dodávané ako minimalistická verzia, všetko ostatné je potrebné doprogramovať

Scene.js

Jednoduchá knižnica, ktorá sa špecializuje na rýchle vykresľovanie obrovského množstva objektov, bez herných efektov, ako sú odrazy alebo tieň. Ponúka menej flexibility, nakoľko je zameraná hlavne na CAD, medicínsku anatómiu, inžinierske vizualizácie a iné.

Babylon.js

Rovnako ako knižnica Three.js, aj Babylon.js je knižnica založená na jazyku Javascript a ponúka rozšírenie WebGL. zameraná hlavne na menšie projekty. Podporuje možnosti práce s grafikou, ako sú napríklad osvetlenie, kamery, materiály, alebo fyzikálne prostredie, ktoré má svoje využitie hlavne pri vyvíjaní hier. Pri jednoduchšej grafickej aplikácii, akou je 3D UML nie je potrebné riešiť textúry, alebo fyziku.

Vlastnosti:

- Vyššie úrovňové API
- Menšia flexibilita pri vytváraní vlastných modulov
- Export do viacerých 3D programov
- Obsahuje integrovanú logiku pre hry
- Náročná dekompozícia

Zhodnotenie

Pri výbere technológie na klientské prostredia prichádzali do úvahy technológie Javascript, alebo Typescript. Typescript je novší, ako Javascript, čo znamená, že je pre neho menej návodov a dostupného kódu, ale obsahuje takisto vyladené chyby z Javascriptu. Hlavný rozdiel je, že Typescript sa musí pred spustením skompilovať na Javascript. Ponúka však lepšiu prácu s objektami, prehľadnejšiu štruktúru a lepšiu prácu s objektami.

Podľa toho, či zvolíme Typescript alebo Javascript sú k dispozícii viaceré frameworky na prácu s daným programovacím jazykom. V prípade, že by sme zvolili Typescript je najvhodnejšie použiť Angular 2, ktorý bol vytvorený priamo pre Typescript. Na rozdiel od neho, Angular 1, alebo Knockout.js pracujú nad čistým Javascriptom. Hlavná nevýhoda Knockout.js je nedefinovaná štruktúra projektu a preto sa využíva hlavne pri menej zložitých aplikáciách a na kontrolu reprezentácie UI rozhrania.

V prípade 3D knižníc bolo rozdielne ich zameranie. Scene.js bol vhodnejší na CAD projekty, alebo inžiniersku vizualizáciu, preto neobsahoval zložitejšie možnosti. Babylon.js zase na druhej strane toho obsahoval príliš veľa, od tieňov až po fyziku, ktoré sú vhodné najmä pri vytváraní hier. Menšie úpravy a zásahy do preddefinovaných metód boli náročné. Three.js je nízko úrovňová knižnica, ktorá slúži na všeobecné účely, keďže je možné všetko potrebné do nej doimplementovať.

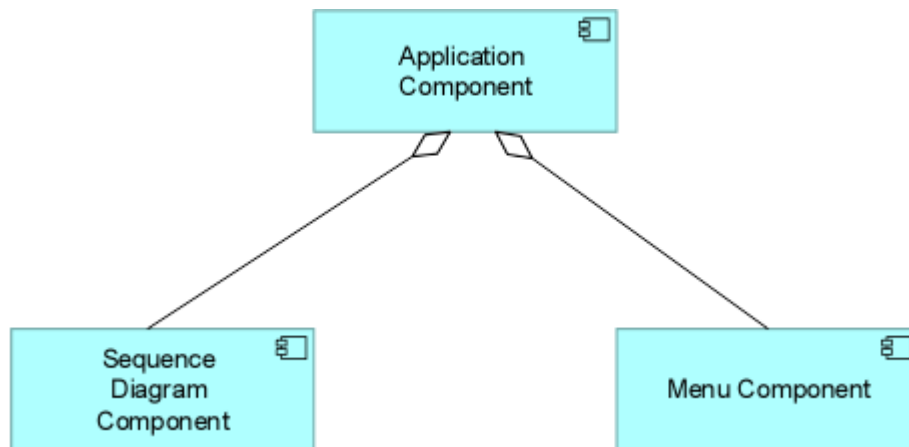
Návrh

Architektúra klientskej aplikácie

Aby správne fungovala celá aplikácia je potrebné, aby spolu komunikovali klientská a serverová časť aplikácie. Táto komunikácia prebieha pomocou JSON API, ktoré spája Javascript, resp. Typescript v podobe na klientskej strane s frameworkom na serverovej strane. Existuje viacero knižníc, ktoré implementujú JSON API do Angularu, napríklad Angular 2 JSON API.

Angular 2 JSON API je jednoduchá knižnica, ktorá interaguje priamo s triedami a modelmi, nie čistým JSONom. Umožňuje tak mapovať všetky dáta do modelov a vzťahov. V prvom kroku musíme pridať `JsonApiModule` do importov, aby sme boli schopní knižnicu používať. Keď je knižnica pripravená, vytvoríme si vlastné modely, v ktorých zdefinujeme štruktúru dát, ktoré budeme pomocou JSON posielat'.

Na obrázku nižšie vidíme dekompozíciu Angular knižnice na jednotlivé komponenty. Celá aplikácia je `Application Component`, ktorá v sebe obsahuje dva ďalšie komponenty. `Sequence Diagram Component`, ktorého úlohou je zabezpečiť chod klientskej strany aplikácie a `Menu Component`, ktorý bude obstarávať menu položky, kde bude mať používateľ možnosti práce s aplikáciou.



Obrázok Diagram komponentov v architektúre

Komunikácia so serverom

Na komunikáciu so serverom využívame triedu datastore, na ktorej vieme volať metódy. Tieto metódy na základe vopred vytvorených modelov vedia, aké typy informácií majú aplikácii poskytnúť ako odpoveď na dopyt.

Použijeme DAO (Data access object) objekty, ktoré nám poskytnú interface ku databáze, resp. ku dátam. Aplikáčnne volania sú namapované na perzistentnú vrstvu, kde DAO poskytuje niektoré špecifické dátové operácie bez odhalenia detailov databázy. Je takto zabezpečené oddelenie dát, ktoré potrebuje aplikácia od samotnej implementácie databázovej schémy. Z takto navrhutej schémy nie je jasné, aké dáta sa v databáze nachádzajú, ani ako databáza vyzerá, vieme iba s akými dátami chceme pracovať. Keby sa zmenila databáza, je potrebné prerobiť modely, nemusíme však meniť metódy na klientskej strane aplikácie.

Príklad použitia triedy datastore na prístup k dátam z databázy, prostredníctvom DAO:

Vytvorenie záznamu:

```

this.datastore.createRecord(Post, {
  title: 'My post',
  content: 'My content'
});
  
```

Nájdenie záznamu:

```

this.datastore.findRecord(Post, '1').subscribe(
  (post: Post) => {
    post.title = 'New title';
  }
);
  
```

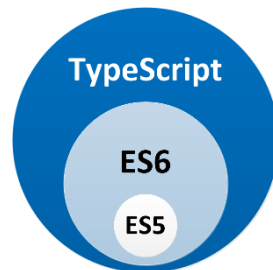
Implementácia

Programové prostredie

Vzhľadom na organizovanosť kódu a s ohľadom na jednoduchšiu udržateľnosť vytváratej aplikácie do budúcnosti sme sa rozhodli ako programovací jazyk pre implementovanie klientskej aplikácie zvoliť jazyk TypeScript. Tento jazyk nám ponúka viac možností ako bežný JavaScript (ECMAScript). Veľmi

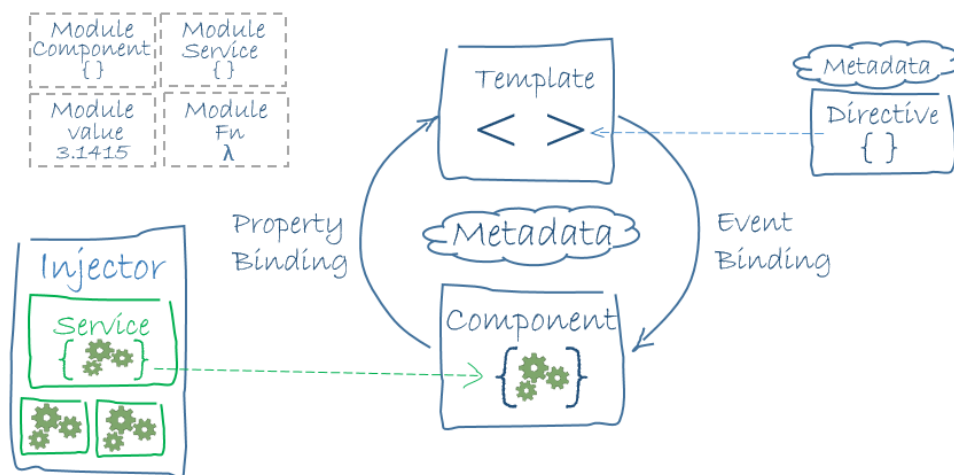
užitočnou funkcionalitou je tiež možnosť dekomponovať jednotlivé časti zdrojového kódu do modulov, vďaka čomu sa zvyšuje prehľadnosť a čitateľnosť kódu.

Z dôvodu nekompatibility TypeScriptu a bežne rozšírených internetových prehliadačov je však potrebné zdrojový kód z jazyka TypeScript skompilovať do jazyka ECMAScript 5. Následne je možné aplikáciu spustiť bežným spôsobom priamo v internetovom prehliadači. Nasledujúci obrázok porovnáva jazyk TypeScript s dvoma verziami jazyka ECMAScript.



Obrázok: Porovnanie TypeScriptu so štandardom ECMAScript

Na prekonanie implementačných prekážok sme sa rozhodli nami vyvíjanú aplikáciu postaviť na frameworku Angular 2 od spoločnosti Google. Angular 2 je kompatibilný s jazykom TypeScript, čo presne splnilo naše požiadavky. Okrem toho framework Angular 2 disponuje viacerými návrhovými vzormi, ktoré nám pomohli aplikáciu implementovať s menšou námahou a robustnejšie. Taktiež komunikáciu so serverovou aplikáciou bolo možné vyriešiť využitím knižnice, ktorá je prepojená s týmto frameworkom. Nasledujúci obrázok zachytáva zjednodušený pohľad na architektúru našej klientskej aplikácie založenej na frameworku Angular 2.



Obrázok: Architektúra Angular 2 aplikácie

Zavedenie 3D aspektu diagramov

Nami vyvíjaná aplikácia obsahuje zobrazenie UML diagramov v 3D prostredí. Do tohto 3D prostredia sú umiestnené bežné 2D vrstvy, medzi ktorými sú vykresľované jednoduché 3D prvky. Napríklad v prípade sekvenčného diagramu sa jedná o interakciu medzi dvoma čiarami života v dvoch 2D vrstvách.

Na 2D vrstvách sú vykresľované samotné UML diagramy. Vďaka tomuto princípu je možné na vykresľovanie diagramov použiť externé knižnice, ktoré sú prispôbené na prácu s bežným 2D

prostredím. Celkový 3D efekt je docielený rotovaním 2D plátien s UML diagramami do 3D scény. Na docielenie tejto funkcionality sme využili grafickú knižnicu Three.js.

Testovanie

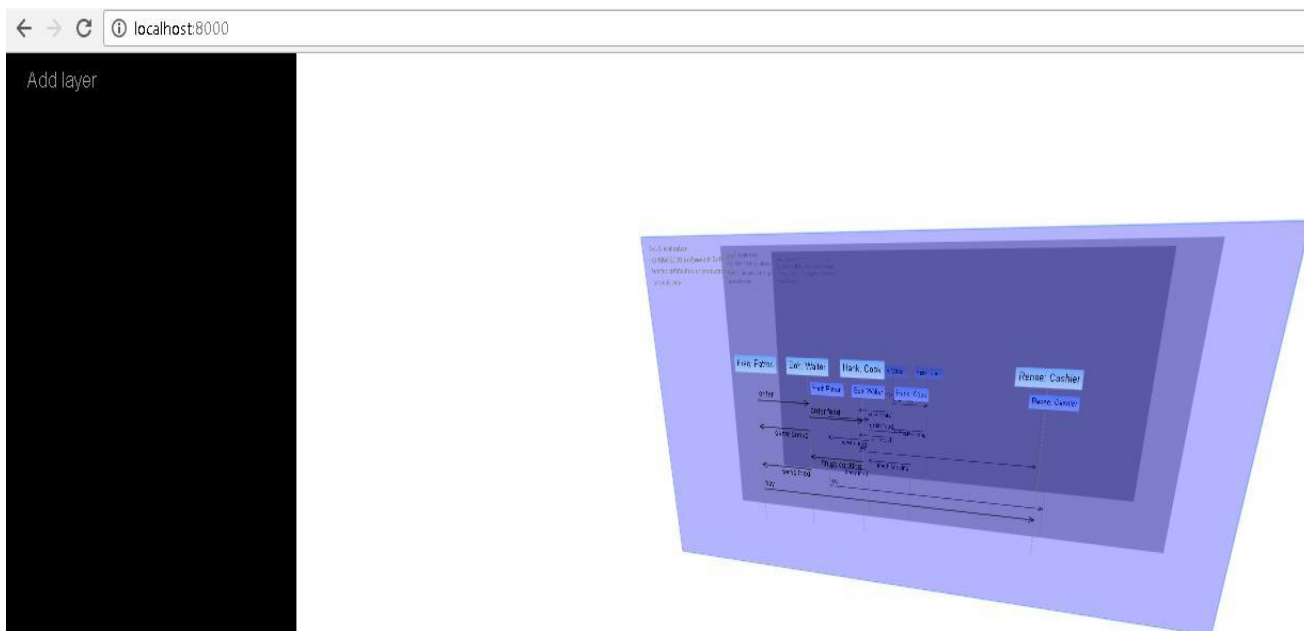
Testovanie funkčnosti JSON API na frontende (AngularJS/Typescript) prebehlo úspešne a je možné komunikovať s backendom. Taktiež sa podarilo prepojenie GoJS knižnice s AngularJS.

Grafická reprezentácia častí fyzického modelu v zimnom semestri

Analýza

Pri programovaní s knižnicou GoJS sme narazili na problém spojený s 3D transformáciou jednotlivých objektov. Po pridání viacerých plátien a ich následnom zoomovaní alebo rotovaní vznikol problém s následným označovaním objektov v vrstve. Taktiež bol problém s pohybom kamery pri jej rotácií. Bolo to spôsobené jadrom samotného GoJS, v ktorom sú elementy implementované v HTML elemente "canvas" (aj vrstva je div, atď.). Týmto elementom chýba 3D aspekt, ktorý bol pre nás nevyhnutným (bez neho sa nedalo správne robiť rotácie, atď.).

Dospeli sme k záveru, že je potrebné preprogramovať jadro aplikácie buď zmenou priamo v knižnici GoJS alebo implementáciou vlastných elementov reprezentujúcich objekty vo fyzickom modeli. Tie si môžeme vkladať do HTML tagu "div", pre ktorý už platia 3D transformácie. Na obrázku je možné vidieť sekvenčný diagram vo vrstvách z knižnice GoJS - nebolo možné rotovať jednotlivé objekty v sekvenčnom diagrame ani posúvať (GoJS nepočítal s 3D aspektom).



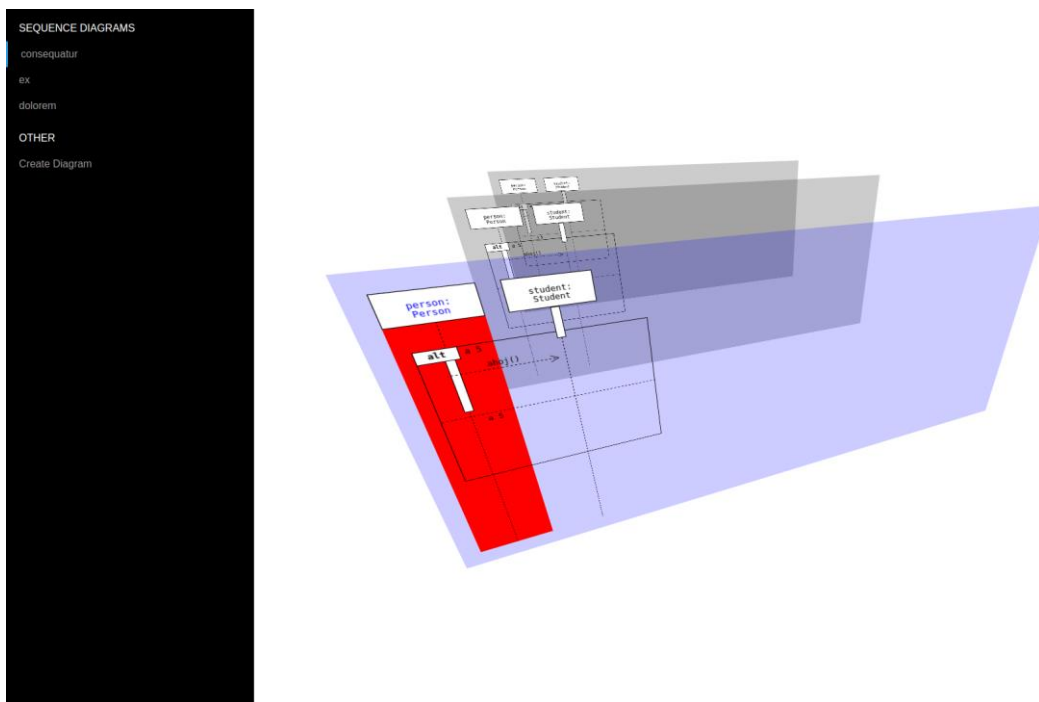
Obrázok: Implementovaná knižnica GoJS vo vrstvách

Návrh

Aplikáciu je potrebné dekomponovať a oddeliť view od modelu, aby sme pri zmenách vzhľadu aplikácie nemuseli meniť zdrojový kód backendu, keďže nami navrhnutý fyzický model už meniť nebudeme. Usúdili sme, že by bolo vhodné implementovať two-way data binding. To zaručuje, že zmeny vo view sa okamžite prejavujú v modeli a naopak.

Implementácia

Aplikáciu sme okrem dekomponovania na MVC dekomponovali taktiež na Angular komponenty. Každý komponent reflektuje objekt z fyzického modelu, taktiež Angular samotný poskytuje two-way data binding. Ako vrchná vrstva nám slúži HTML + CSS a jej 3D transformácie, ktoré vkladáme do tagu "div".



Obrázok: Implementovaná aplikácia vo vrstvách – HTML + CSS

Grafická reprezentácia častí fyzického modelu v letnom semestri

Analýza

V letnom semestri nadväzujeme na implementáciu zobrazenia diagramu uloženého do databázy vo vrstvách. Pri editácii vzniká problém výberu elementov kvôli ovládaniu pohybu kamery. Preto je pre úpravu diagramu a mazanie komponentov potrebný takzvaný editačný mód, ktorý obsahuje 2D pohľad na jednotlivé vrstvy, ktoré možno prepínať kolečkom myši. Pridanie novej správy chceme realizovať len na určité body čiar života medzi, ktorými je konštantná vzdialenosť. To možno realizovať pridaním

viditeľných a klikateľných bodov na čiary života, ktoré budú predstavovať začiatkové a koncové body správy.

Návrh

Naším návrhom je vytvoriť aplikáciu, v ktorej budú dva zobrazovacie módy. Jeden zobrazovací mód bude v 3D (na prezeranie diagramu), druhý mód bude 2D pre editáciu diagramu. Editácia diagramu bude predstavovať CRUD pre správu, čiaru života, diagram a vrstvu. Oproti minulému semestru chceme pridať CUD (create, update, delete) všetkých elementov, keďže sme mali iba R (retrieve diagramu z databázy).

Používame naďalej Angular, HTML + CSS a typescript, ktorý sme si zvolili už konečných fázach zimného semestra.

Menu

Vytvorenie menu side baru je dôležité na interakciu s aplikáciou. Preto je našou víziou vylepšiť menu, ktoré by bolo rozdelené na jednotlivé akcie v zobrazovacom režime.

Vrstvy (layers), Správy (messages), Čiary života (lifeline)

Dorábanie vrstiev, správ a čiar života bude prebiehať vo vylepšovaní ich funkcionalít, keďže väčšinu ich vizuálnych častí sme zvládli v zimnom semestri. Vo všetkých týchto elementoch je potreba nastavovanie názvov. Navrhli sme úpravu dizajnu vrstvy, pričom nový dizajn zahŕňa aj zobrazenie názvu vrstvy. Tento návrh taktiež zahŕňa prispôbovanie veľkosti vrstvy k veľkosti diagramu. Návrhom pri správach je pridávať ich postupne, ukladať do databázy a posúvať horizontálne a vertikálne po vrstve. Návrhom vedúceho tímového projektu bolo, aby správy odskakovali, ak chce používateľ pridať jednu správu medzi dve iné. Pri čiare života bol návrh jednoduchý, novú čiaru života pridáme na koniec aktívnej vrstvy, za všetky ostatné. Následne je pozícia, ako aj meno novej čiary života uložené do databázy. Zmenenie pozície čiary života bolo navrhnuté chytením a posúvaním danej čiary života. Pri takomto presune bolo navrhnuté, že ostatné čiary života odskočia, aby boli od seba vždy fixne vzdialené.

Implementácia

Držali sme sa návrhu a pripomienok od nášho vedúceho a implementovali sme funkcionalitu vytvorenia jednotlivých elementov (hlavne pridávanie, editovanie a vymazávanie). Aplikáciu sme aj dekomponovali na MVC.

Menu

Menu je implementované ako postranný panel, ktorý využíva Material Design knižnicu, ktorá umožňuje dobrú spoluprácu s AngularJS a Typescriptom.

Menu je rozdelené na **View** a **Edit** mód. Vo view móde sú načítané používateľove uložené a vytvorené diagramy, ktoré sú renferované v 3D móde pomocou CSS3D rendereru. V edit móde

sú diagramy renderované v 2D priestore ako HTML elementy s tým, že sa medzi vrstvami diagramu dá prepínať skrolovaním myši. Taktiež sú v menu implementované funkcionality na editáciu diagramu.

Vrstva

V projekte sme implementovali dva typy vrstiev.

Prvý typ je vytvorený pre zobrazenie sekvenčného diagramu v 3D móde, pričom každá vrstva je implementovaná ako HTML element zobrazený v 3D priestore prostredníctvom CSS3D.

Druhý typ je vytvorený pre editáciu sekvenčného diagramu v 2D móde. Pre editáciu v 2D móde sme sa rozhodli kvôli zjednodušeniu ovládania a lepšej orientácii v rámci vrstiev. Vrstvy editačného módu sú taktiež implementované prostredníctvom HTML elementov, ktoré sú následne zobrazené v 2D priestore. Tieto vrstvy už nie sú zobrazené v jednej scéne, ale každá vrstva je zobrazená samostatne pričom používateľ môže prechádzať medzi vrstvami pomocou kolieska myši.

Každá vrstva sa skladá z názvu vrstvy a samotného tela vrstvy ktoré obsahuje sekvenčný diagram alebo jeho časť. Toto telo je implementované tak, aby sa veľkosťou prispôbovalo diagramu ktorý obsahuje. Táto funkcionality je založená na odstupoch, ktoré sú definované pre každý element, pričom pri zmene v diagrame sa prostredníctvom týchto odstupov elementov diagramu, vypočíta veľkosť vrstvy.

Čiara života (lifeline)

Pri pridaní lifeline sme sa držali jednoduchého návrhu. Používateľ sa prepne do editačnej polohy, kde môže vytvoriť novú lifeline jednoduchým stlačením tlačidla. Samozrejme lifeline sa pridá na tú vrstvu, ktorá je v popredí v editačnom móde. Po stlačení tlačidla sa zobrazí okno, do ktorého používateľ zadá požadovaný názov pre novú lifeline. Ak názov potvrdí, lifeline sa pridá na danú vrstvu, za všetky ostatné lifeline a uloží sa do databázy.

V prípade, že používateľ chce zmazať lifeline, opäť sa musí prepnúť do editačného režimu. Tu zvolí z kontextového menu možnosť zmazania elementu. Následne musí kliknúť na lifeline, ktorú chce zmazať. Zmazanie stačí už len potvrdiť a lifeline sa zmaže z diagramu, ako aj z databázy. Okrem zmazania samotného záznamu o lifeline z databázy sa zmažú aj všetky Occurrence Specification a Message, ktoré boli na danú lifeline naviazané.

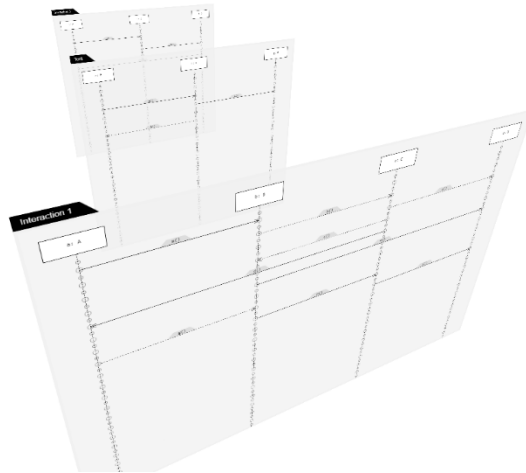
V rámci editácie môže používateľ lifeline premenovať (jednoduchým dvojklikom na názov lifeline v diagrame), alebo popresúvať. Keďže vieme pridávať lifeline iba na koniec diagramu, je vysoko pravdepodobné, že bez možností presúvania poradia jednotlivých lifeline sa nezaobídeme. Presúvanie, podobne ako aj mazanie, alebo pridávanie funguje iba v editačnom móde. Celé presúvanie je založené na princípe drag and drop, kedy používateľ stlačí tlačidlo myši nad danou lifeline a nepustí, kým sa nedostane na požadované miesto. Nové miesto lifeline sa uloží do databázy a poradie ostatných lifeline sa prepočíta. Tieto nové hodnoty sú tiež aktualizované v databáze.

Správa (message)

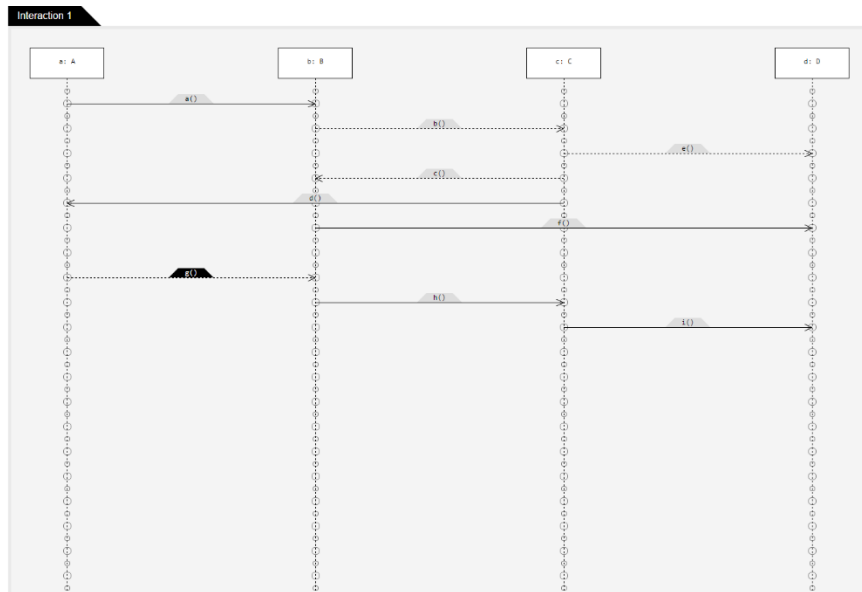
Pri vytváraní funkcionality pre správu, sme začali jej pridávaním do diagramu. Funguje tak, že používateľ si vyberie editačný mód a v nej má možnosť pridania. Správa sa pridáva tak, že sa klikne ľavým tlačidlom myši na čiaru života odkiaľ má správa ísť (konkrétne na occurrence, ktorý je vyznačený na čiare života guľičkou), následne zadá názov vytváratej správe a zvolí si taktiež jej typ.

Následne sme sa zamerali na jej editáciu, ktorá má 3 možnosti:

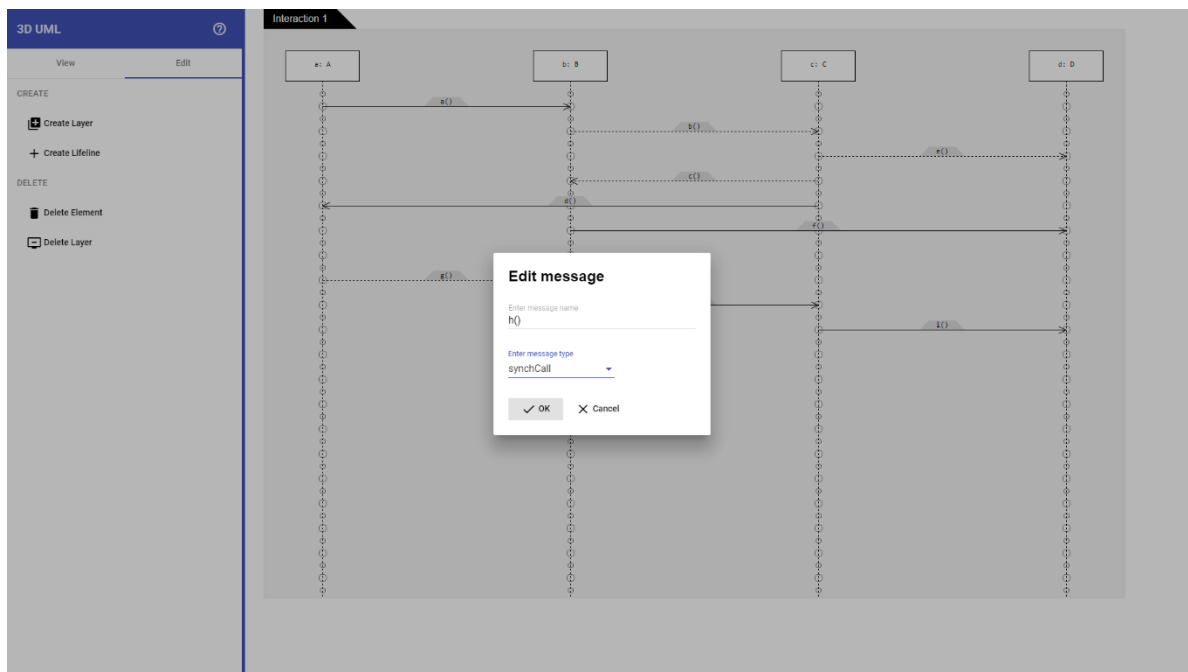
- *Zmena názvu a typu správy* (double-clickom na správu)
- *Odskakovanie správ* - Odskakovanie je potrebné, aby neboli správy na sebe a neprekrývali sa. Funguje to tak, že ak sa pridá správa na "malú guľičku" na čiaru života, resp. Vertikálne posunú medzi dve správy, alebo nejaká vymaže, tak sa zistí či pod danou správou sa nachádzajú ešte nejaké, ak áno tak všetky body na čiare života, ktoré obsahujú správy (vzťah v databáze), sa posunú o jeden nadol a tým podskočia aj správy s danými bodmi naviazanými na ne.
- *Horizontálne posúvanie správy* - horizontálne posúvanie funguje tak, že je potreba kliknúť pravým tlačidlom myši na jej začiatok alebo koniec, a pravým tlačidlom si následne zvolit' kam sa má tento bod správy premiestniť. Implementované je to tak, že program počúva na right-click a ak používateľ klikol na nejaký bod na čiare života tak sa zisťuje, či na tom bode je uložená nejaká správa, ak nie nič na sa nerobí, ak áno, zistíme si informácie o správe. Samozrejme stále sa počúva na right-click, keďže používateľ musí druhýkrát kliknúť, aby umiestnil tento bod správy inde. Ak ho umiestni inde, zisťuje sa či neklikol náhodou na to isté miesto, ak klikol nič sa nestane, ak nie, tak je nutné pozrieť či neklikol ešte na opačný bod správy, ktorý je na čiare života, na ktorú sa správa pripája. Ak to je ten bod, tak sa jednoducho vymenia tieto body a správa bude ukazovať opačným smerom. Ak to bol iný bod (musí byť iba v rovnakej výške - keďže je to horizontálne posúvanie), tak sa jednoducho daný bod zmení na ten, kam klikol a správa sa predĺži alebo skráti.
- *Vertikálne posúvanie správy* - Vertikálne posúvanie správu slúži na to, aby ju používateľ mohol premiestniť, ak ju pridá na iné miesto. Funguje to tak, že sa počúva na kliknutie ľavého tlačidla myši nadol, pohyb myšou a následné uvoľnenie tlačidla myši, vtedy vieme, že ju "ťahal". Zapamätá sa pozícia správy pri jej začatí premiestňovaní a porovná sa pri jej uvoľnení na inom mieste. Ak sa pozícia zmenila tak, sa skontroluje, či nebola posunutá medzi dve iné správy (vtedy správy odskočia nadol). Ak nebol posunutá medzi dve iné, tak sa len zmení jej pozícia v databáze a refreshne diagram. Následne už bude správa na posunutom mieste. Kontroluje sa taktiež aj to, či používateľ nevyšiel mimo vrstvu, alebo či nemá pridané maximálne množstvo správ, resp. pridanú správu na poslednom mieste na vrstve, v tom prípade je upozornený.



Obrázok: 3D UML View mód



Obrázok: 3D UML Edit mód



Obrázok: 3D UML Edit mód - dialógové okno

Testovanie

Testovanie funkcionality aplikácie bolo vykonané Bc. Miroslavom Novotným aby sme dosiahli objektívne posúdenie funkcionality i použiteľnosti aplikácie. Testovanie bol vykonané na vopred vykonaných testovacích scenároch.

Test scenario 1 – Vytvorenie nového sekvenčného diagramu

Test case 1.1- Otestuj odozvu aplikácie pri vložení názvu diagramu

Test steps: 1. Zvoľ možnosť „Create Diagram“

2. Vlož názov sekvenčného diagramu

3. Potvrď tlačidlom „OK“

Test data: Meno diagramu: „Novy diagram“

Expected result: Diagram bol vytvorený.

Testing result: Diagram bol úspešne vytvorený.

Test case 1.2- Otestuj odozvu aplikácie bez vloženia názvu diagramu

Test steps: 1. Zvoľ možnosť „Vytvor sekvenčný diagram“

2. Nevkladaj názov sekvenčného diagramu.

3. Potvrď tlačidlom „OK“

Expected result: Aplikácia neumožní používateľovi stlačiť „OK“.

Testing result: Diagram nie je možné vytvoriť bez mena.

Test scenario 2 – Načítanie uloženého sekvenčného diagramu

Test case 2.1- Otestuj odozvu aplikácie pri zvolení sekvenčného diagramu v menu

Test steps: 1. Vyber sekvenčný diagram z menu kliknutím na meno alebo tlačidlo „Select“ pri ňom.

Expected result: Diagram bol načítaný.

Testing result: Diagram bol načítaný.

Test scenario 3 – Úprava názvu sekvenčného diagramu

Test case 3.1- Otestuj odozvu aplikácie pri vložení nového názvu diagramu.

Preconditions: Načítaný sekvenčný diagram.

Test steps: 1. Dvojklikni na názov sekvenčného diagramu.

2. Vlož nový názov.

3. Potvrď tlačidlom „OK“.

Expected result: Zmenený názov diagramu.

Testing result: Meno diagramu bolo úspešne upravené.

Test case 3.2- Otestuj odozvu aplikácie pri vložení nového názvu a zrušení zmeny.

Preconditions: Načítaný sekvenčný diagram.

Test steps: 1. Dvojklikni na názov sekvenčného diagramu.

2. Vlož nový názov.

3. Zruš tlačidlom „Cancel“.

Expected result: Nezmenený názov diagramu.

Testing result: Nezmenený názov diagramu.

Test scenario 4 – Zmazanie sekvenčného diagramu

Test case 4.1- Otestuj odozvu aplikácie pri zmazaní diagramu (načítaný diagram).

Preconditions: Načítaný sekvenčný diagram.

Test steps: 1. Klikni na možnosť „Delete“ pri diagrame, ktorý chceš zmazať

2. Potvrď v okne zmazanie diagramu tlačidlom „Yes.“

Expected result: Zmazaný diagram odstránený zo zoznamu.

Testing result: Úspešne zmazaný diagram.

Test scenario 5 – Pridanie vrstvy do sekvenčného diagramu

Test case 5.1- Otestuj odozvu aplikácie pri pridaní vrstvy do diagramu.

Preconditions: Načítaný sekvenčný diagram.

Test steps: 1. Vyber z menu možnosť „Create Layer.“

2. Zadáaj meno vrstvy a potvrd' s „OK“.

Expected result: Vrstva pridaná v sekvenčnom diagrame.

Testing result: Pridaná vrstva, tester mal pripomienku, že by bolo dobré previesť diagram do editačnej polohy pre pridanú vrstvu.

Test scenario 6 – Zmazanie vrstvy zo sekvenčného diagramu

Test case 6.1- Otestuj odozvu aplikácie pri zmazaní vrstvy.

Preconditions: Načítaný sekvenčný diagram.

Test steps: 1. Prepni sa skrolovaním myši na vrstvu, ktorú chceš zmazať.

2. Vyber možnosť „Delete Layer.“

3. Potvrd' tlačidlom „OK“. (Tlačidlo „Cancel“ na zrušenie)

Expected result: Vrstva zmazaná zo sekv. diagramu spolu s lifelinami a správami.

Testing result: Úspešne zmazaná vrstva, tester našiel problémy pri prepínaní vrstvy skrolovaním myšky v prehliadači Mozilla Firefox.

Test scenario 7 – Pridanie lifeline na vrstvu sekvenčného diagramu.

Test case 7.1- Otestuj odozvu aplikácie pri pridaní lifeline na vrstvu, s vložením názvu.

Preconditions: Načítaný sekvenčný diagram s pridanou vrstvou.

Test steps: 1. Preskroluj sa na vrstvu na ktorú chceš pridať lifeline.

2. Vyber možnosť „Create lifeline.“

3. Zadáaj názov.

4. Potvrd' Enterom alebo tlačidlom „OK“.

Expected result: Pridaná lifeline.

Testing result: Úspešne pridaná lifeline.

Test scenario 8 – Zmazanie lifeline z vrstvy diagramu.

Test case 8.1- Otestuj odozvu aplikácie pri zmazaní lifeliny.

Preconditions: Načítaný sekvenčný diagram s pridanou vrstvou a 2 lifelinami.

Test steps: 1. Prepni sa do vrstvy na ktorej chceš zmazať lifeline.

2. Vyber možnosť „Delete element.“
3. Klikni na lifeline, ktorú chceš zmazať.
4. Potvrď tlačidlom „OK“.

Expected result: Zmazaná lifeline a správy na ňu pripojené.

Testing result: Úspešne zmazaná lifeline, tester navrhol, že pre lepší prehľad by sa mohol zmeniť kurzor na symbol ruky napríklad.

Test scenario 9 – Presunutie lifeline na vrstve diagramu.

Test case 9.1- Otestuj odozvu aplikácie pri presunutí lifeliny na prvé miesto.

Preconditions: Načítaný sekvenčný diagram s pridanou vrstvou a viacerou lifelinami.

Test steps: 1. Prepni sa do vrstvy na ktorej chceš presunúť lifeline.

2. Potiahni ju technikou drag and drop na miesto na, ktoré chceš.

Expected result: Presunutá lifeline a správy na ňu pripojené.

Testing result: Úspech.

Test scenario 10 – Pridanie správy medzi 2 lifelinami.

Test case 10.1- Otestuj odozvu aplikácie pri pridaní správy v rámci 1 vrstvy.

Preconditions: Načítaný sekvenčný diagram s pridanou vrstvou a 2 lifelinami.

Test steps: 1. Klikni na ľubovoľnú guľičku na lifeline, od ktorej chceš aby správa išla.

2. Klikni na guľičku na 2. lifeline, kam má správa ísť.
3. Pridaj názov správy a zvol' typ.

Expected result: Pridaná správa.

Testing result: Úspech, tester uviedol, že chvíľu bol zmetený na akú guľičku má kliknúť (veľká /malá).

Test scenario 11 – Upravenie správy.

Test case 11.1- Otestuj odozvu aplikácie pri upravení správy.

Preconditions: Načítaný sekvenčný diagram s pridanou správou.

Test steps: 1. Označ správu dvojkliknutím.

2. Zmeň meno správy alebo typ.
3. Potvrď tlačidlom OK.

Expected result: Upravená správa.

Testing result: Úspech.

Test scenario 12 – Zmazanie správy.

Test case 12.1- Otestuj odozvu aplikácie pri zmazaní správy.

Preconditions: Načítaný sekvenčný diagram s 2 pridanými spravami.

Test steps: 1. Dostaň sa skrolovaním do vrstvy, na ktorej chceš mazať správu.

2. Zvoľ možnosť „Delete Element“.

3. Zvoľ správu.

4. Potvrď tlačidlom „OK“.

Expected result: Zmazaná správa.

Testing result: Úspech.

Test scenario 13 – Presunutie správy.

Test case 12.1- Otestuj odozvu aplikácie pri presunutí správy.

Preconditions: Načítaný sekvenčný diagram s pridanou správou.

Test steps: 1. Dostaň sa skrolovaním do vrstvy, na ktorej chceš presunúť správu.

2. Presuň správu na miesto kam potrebuješ, technikou drag and drop.

Expected result: Presunutá správa.

Testing

result:

Úspech.

Vývojový diagramu procesu pri CRUD

Práca v budúcich etapách

V tejto kapitole opisujeme možné smery, ktorými je možné sa uberať v budúcej práci na nami rozpracovanom projekte.

Diskusia k optimalizácii

Jeden z možných smerov v budúcej práci na nami rozpracovanom projekte môže byť optimalizácia aplikácie. Jedným z kľúčových aspektov, nad ktorým je možné uvažovať z pohľadu optimalizácie je komunikácia pomocou JSON API. Druhým aspektom je vykresľovanie aplikácie, ktoré je momentálne realizované pomocou komponentov vo frameworku Angular.

JSON API

Súčasná architektúra nami vytvorenej aplikácie pozostáva z dvoch častí. Samotná aplikácia 3D UML na strane browsera (frontend) a PHP aplikácia (backend), ktorá slúži na ukladanie dát do databázy PostgreSQL. Tieto dve časti navzájom spolu komunikujú. Frontend posiela XHR dopyty na server, ktorý tieto dopyty spracováva a následne odosiela odpoveď (tiež vo formáte JSON) na frontend.

Backend je pripojený k relačnej databáze PostgreSQL. Vzhľadom na to, že štandard JSON API je implementovaný vo veľkom počte backendových riešení a programových prostredí, je možné v budúcnosti kombináciu PHP + PostgreSQL vymeniť za rýchlejšie riešenie napríklad Node.js + MongoDB. Bude však potrebné dodržať špecifikáciu JSON API¹.

Na strane frontendu aj backendu je nutné modely (či už TypeScriptové alebo PHP modely) pred odoslaním serializovať do JSON API a pred dorúčením deserializovať na opačnej strane. Backend na serializáciu a deserializáciu využíva knižnicu s názvom `cloudcreativity/laravel-json-api`². Táto knižnica je veľmi kvalitne navrhnutá a implementovaná a s jej použitím sme nenarazili na žiadne väčšie problémy.

Naopak na strane frontendu využívame knižnicu `ghidoz/angular2-jsonapi`³. Táto knižnica je navrhnutá na prácu priamo v prostredí jazyka TypeScript vďaka čomu umožňuje využívať možnosti, ktoré sa v čistom JavaScripte nenachádzajú. Napríklad anotácie vzťahov a podobne. Z tohto pohľadu je knižnica vhodná pre naše riešenie a v tomto ohľade sme nenašli žiadnu lepšiu alternatívu. Táto knižnica je však veľmi jednoduchá a na vykonávanie zložitejších operácií je potrebné väčšie množstvo kódu. Zo samotnej dokumentácie tejto knižnice vyplýva, že nie sú implementované niektoré základné operácie ako je napríklad upravovanie a synchronizovanie vzťahov medzi entitami. Napríklad pri zmene vzťahu occurrence specification pri modeli lifeline sme museli implementovať vlastnú funkcionality, pretože táto knižnica spomínanú operáciu nepodporuje. Preto do budúca silno odporúčame využiť kvalitnejšie navrhnutú a implementovanú knižnicu na JSON API na strane frontendu. Bohužiaľ v dobe realizácie nami vytvárajúcej aplikácie sa nám nepodarilo nájsť vhodnejšiu implementáciu.

Angular vykresľovanie

Vzhľadom na to, že na základe analýzy sme sa rozhodli využiť riešenie využívajúce technológiu CSS 3D transformácií, je potrebné vytvárať a spravovať jednotlivé HTML elementy, na ktoré sú následne aplikované CSS transformácie, vďaka čomu vizuálne vyzerajú 3D. Samozrejme pri zmene modelu je potrebné tieto zmeny reflektovať aj v týchto HTML elementoch, ktoré tvoria z pohľadu architektonického vzoru Model-View-Controller zložku View.

Na túto synchronizáciu využívame vstavaný princíp s názvom two-way-binding, ktorý je súčasťou frameworku Angular. Do budúca je však možné synchronizáciu medzi zložkami Model a View implementovať inak než s použitím frameworku Angular, čím by mohla byť docielená lepšia odozva aplikácie. Vzhľadom na dobrú podporu tohto frameworku, jeho rýchlosť a modernosť sme sa ho rozhodli využiť, avšak už počas práce na tomto projekte sa Angular vyvinul z verzie 2 na verziu 4 s prísľubom autorov, že frameworku sa bude ďalej lineárne vyvíjať. Vďaka tomu je možné predpokladať,

¹ <http://jsonapi.org/format/>

² <https://github.com/cloudcreativity/laravel-json-api>

³ <https://github.com/ghidoz/angular2-jsonapi>

že do budúcnosti bude možné aj našu aplikáciu upgradovať spolu s verziou Angularu, čím môže byť tiež zvýšená odozva aplikácie.

Ďalšia funkcionality

V nami vytvorenej aplikácii sa nám podarilo navrhnuť a implementovať funkcionality pre zobrazovanie, vytváranie, úpravu a odstraňovanie nasledujúcich častí sekvenčného diagramu:

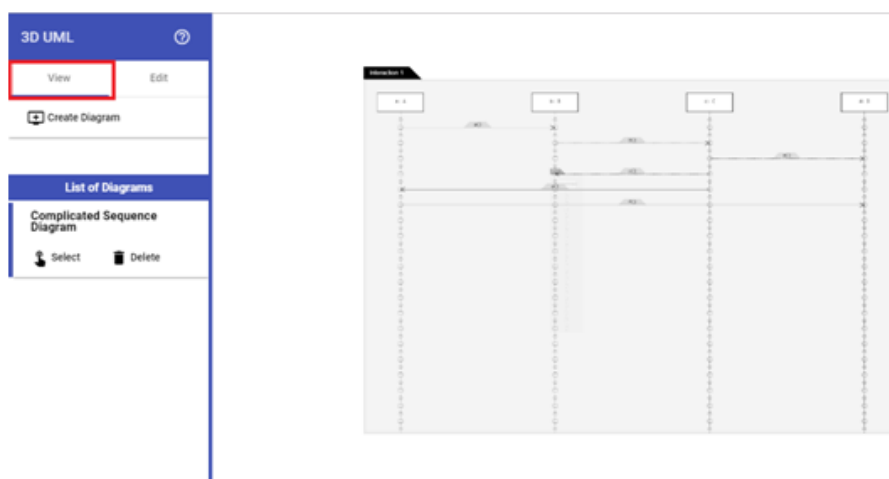
1. Sekvenčný diagram
2. 3D vrstva
3. Čiary života
4. Správy v rámci jednej 3D vrstvy

Tiež sa nám poradilo implementovať zobrazovanie fragmentov. V budúcich etapách je teda možné nadviazať na našu prácu a pokračovať v implementácii zatiaľ nespracovaných častí, medzi ktoré patrí hlavne správa medzi dvoma 3D vrstvami alebo export do špecifikovaných formátov ako je napr. XML.

Používateľská príručka k 3duml – sekvenčnému diagramu

Módy

Táto aplikácia obsahuje dva módy zobrazení:



1. View mode – slúži na 3D zobrazenie diagramu. Dostaneme sa do neho po kliknutí na „View“ v hornej časti menu.
2. Edit mode – slúži na editáciu diagramu. Zobrazujú sa v ňom vrstvy v 2D zobrazení. Premiestňujeme sa medzi nimi pohybom koliečka na myši. Do editačného módu sa môžeme

dostať dvomi spôsobmi. Dvojklikom na vrstvu vo View mode alebo kliknutím na „Edit“ v hornej časti menu.



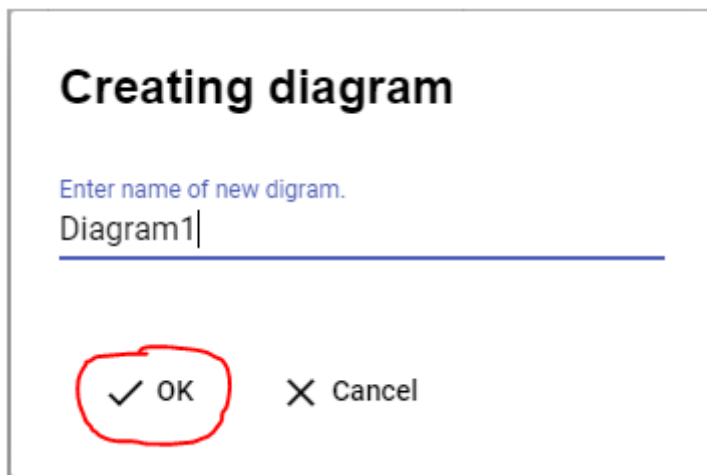
Help

Ak potrebujete rýchlu nápovedu kliknite na ikonku otáznika v hornej časti menu. Otvorí sa vám intuitívny help.



Vytvorenie nového diagramu

Pre vytvorenie nového diagramu choďte do View mode. Následne kliknite v menu na „Create Diagram“. Potom vyplňte formulár (zadajte meno diagramu) a stlačte „OK“.



Zobrazenie diagramu

Pre zobrazenie vami zvoleného diagramu choďte do View mode, v menu v položke „List of Diagrams“ kliknite na tlačidlo „Select“ (označené ručičkou s prstíkom), ktoré sa nachádza v kolónke diagramu, ktorý chcete zobraziť.



Editovanie diagramu

Aby ste mohli editovať diagram, najprv si ho musíte zvoliť, ako bolo písané v kapitole „Zobrazenie diagramu“. Následne prejdite do Edit modu alebo dvojkliknite na viditeľnú vrstvu.

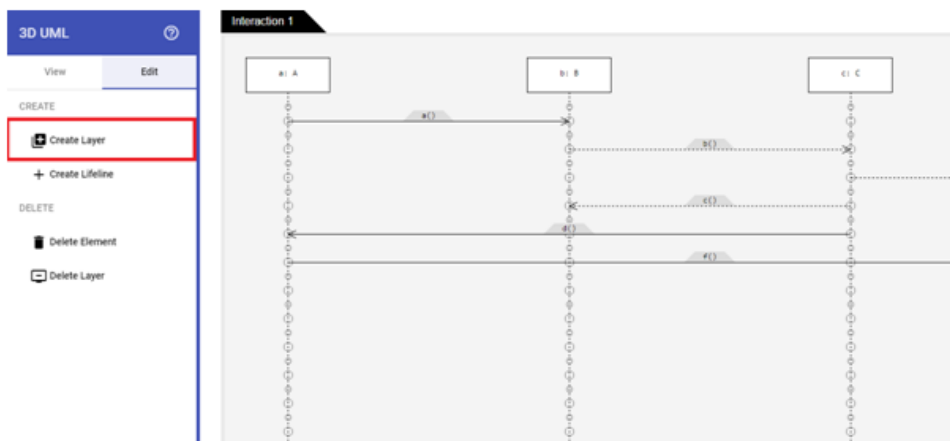
Vymazanie diagramu

Pre vymazanie vami zvoleného diagramu choďte do View mode, v menu v položke „List of Diagrams“ kliknite na tlačidlo „Delete“ (označené smetným košom), ktoré sa nachádza v kolónke diagramu, ktorý chcete zobraziť.



Pridanie vrstvy

Pre pridanie novej vrstvy choďte do Edit mode. Následne kliknite na tlačidlo „Create Layer“, ktoré sa nachádza v „Create“ časti menu. Vyplňte formulár a stlačte „OK“.



Creating layer

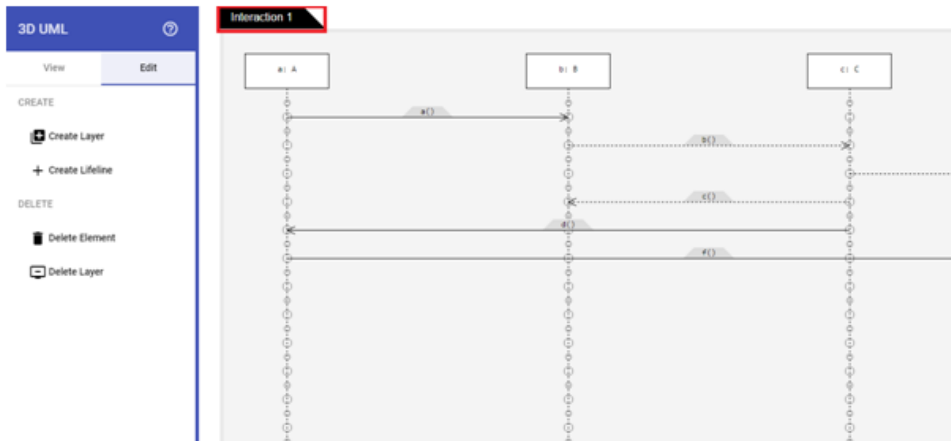
Enter name of new layer.

Layer1|

✓ OK ✕ Cancel

Upraviť vrstvu

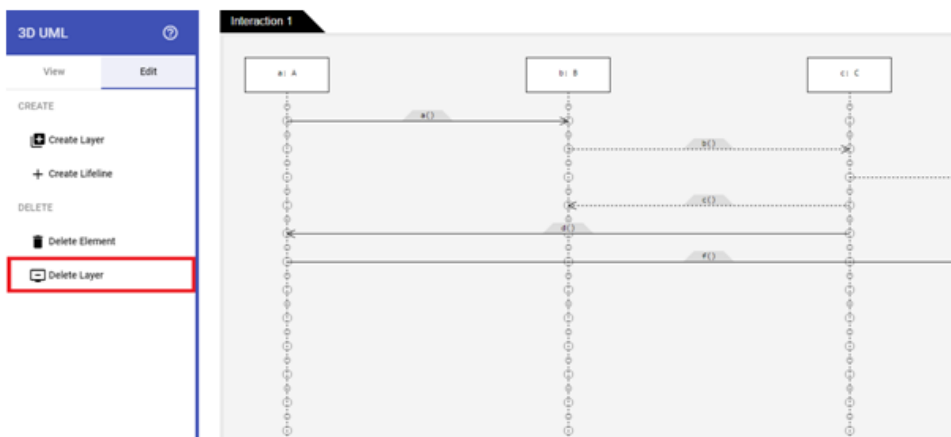
Vrstvu môžete kedykoľvek premenovať. Nastavte sa v Edit mode na vrstvu, ktorú chcete premenovať, pomocou koliečka na myši. Potom dvojkliknite na názov vrstvy a vyplňte formulár. Potvrďte tlačidlom „OK“.

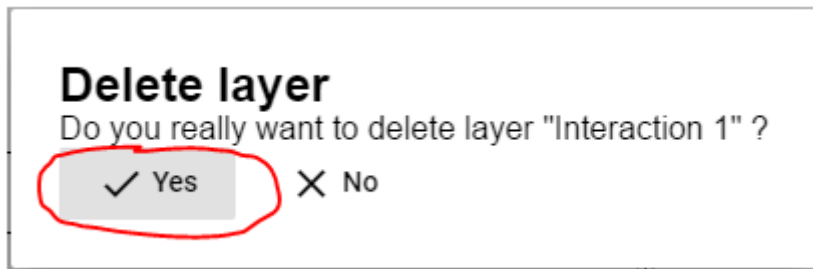


The 'Edit layer' dialog box is shown. It has a title 'Edit layer' and a text input field labeled 'Enter Layer name' containing the text 'Layer1'. Below the input field are two buttons: 'OK' (with a checkmark icon) and 'Cancel' (with an 'X' icon). The 'OK' button is circled in red.

Vymazanie vrstvy

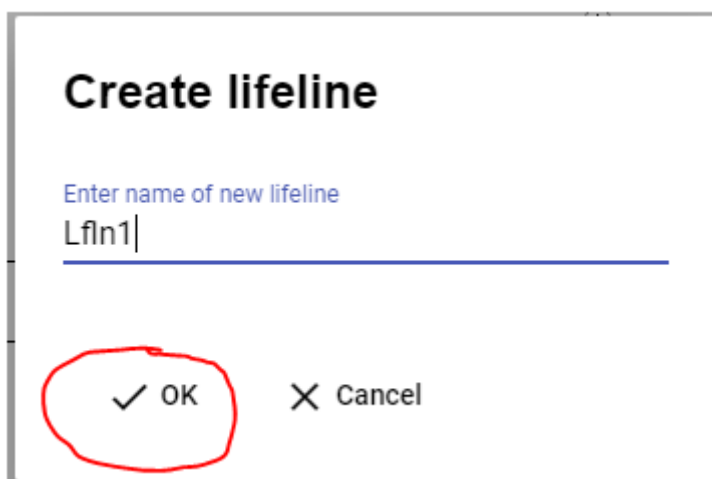
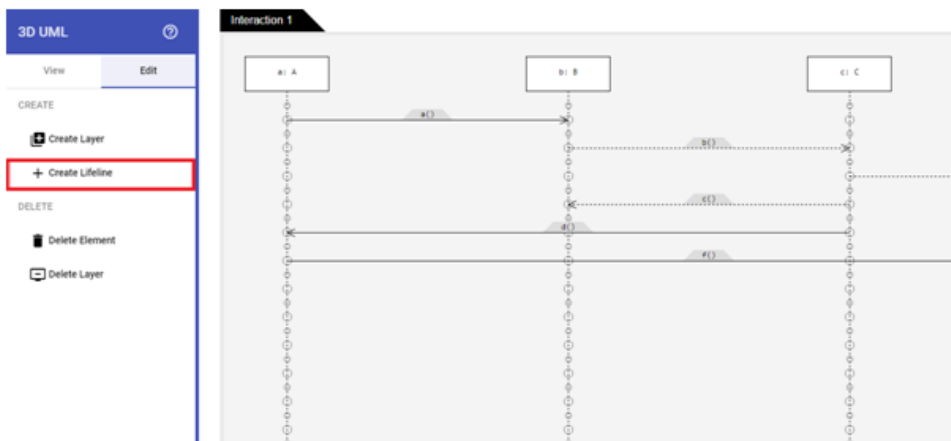
Vrstvu môžete taktiež vymazať. Nastavte sa v Edit mode na vrstvu, ktorú chcete vymazať, pomocou koliečka na myši. Potom kliknite na tlačidlo „Delete Layer“ v „Delete“ menu. Potvrďte tlačidlom „Yes“.





Vytvorenie/pridanie čiary života

Pre vytvorenie čiary života sa v Edite mode nastavíme na vrstvu, do ktorej chceme čiaru života pridať. Následne klikneme na tlačidlo „Create Lifeline“ (označené jednoduchým znakom „+“), v „Create“ menu. Vyplníme formulár a klikneme na tlačidlo „OK“.



Premiestnenie/premenovanie čiary života

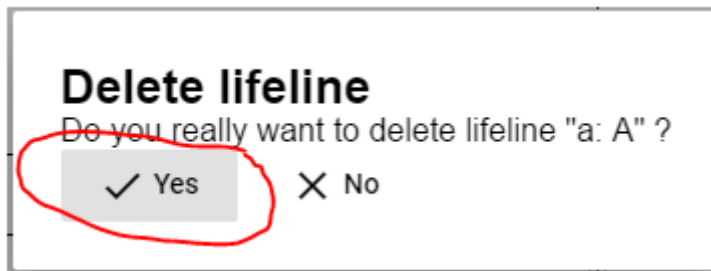
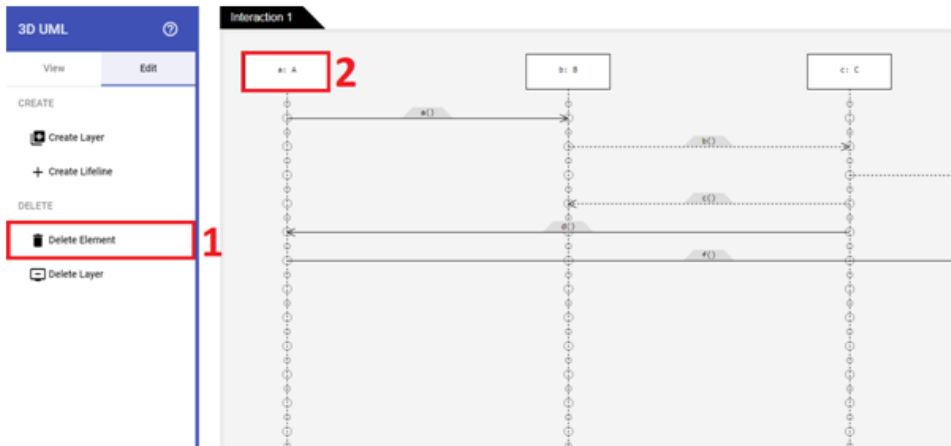
Pre pohybovanie čiarou života musíme byť v Edit mode. Následne ľavým kliknutím na hlavičku čiary života a podržaním tlačítka dole môžeme čiarou života horizontálne hýbať. Keď sa čiara života nachádza v polohe, ktorú chceme, jednoducho pustíme tlačítko a diagram sa prekreslí. Vzťahy ostajú zachované.



Pokiaľ chceme čiaru života premenovať dvojklíkneme na jej hlavičku (zobrazená na obrázku vyššie). Následne vyplníme formulár a klikneme na tlačítko „OK“.

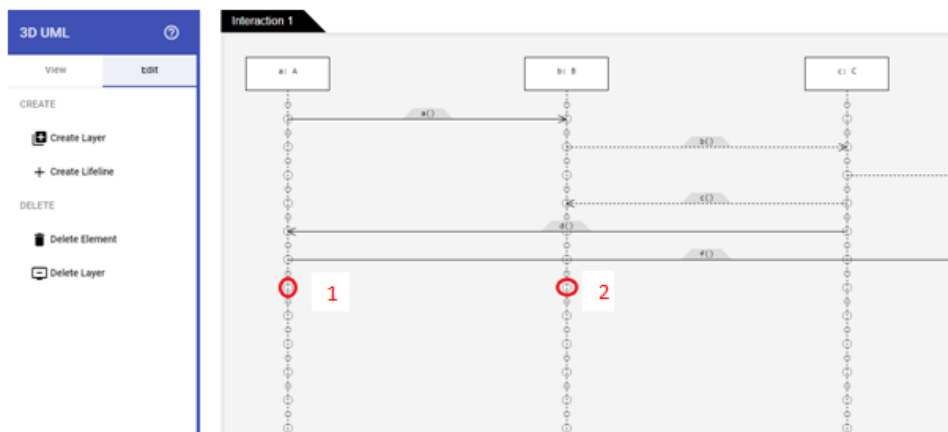
Vymazanie čiary života

Pre vymazanie čiary života sa musíme nachádzať v Edit mode. Následne klikneme na tlačidlo „Delete Element“ (znázornené ikonou smetného koša), ktoré sa nachádza v „Delete“ menu. Následne klikneme na hlavičku čiary života. Potom potvrdíme vymazanie čiary života tlačidlom „Yes“.



Vytvorenie novej správy

Pre vytvorenie novej správy sa musíme nachádzať v Edit mode. Následne klikneme ľavým tlačítkom na guľičku na čiare života vo výške, kde chceme správu pridať. To vytvorí začiatkový bod správy v pamäti. Následne to isté vykonáme pre cieľ správy na inej čiare života, kde chceme aby sa správa končila. Potom vyplníme formulár a stlačíme tlačidlo „OK“.



Creating message

Enter message name
 Message1

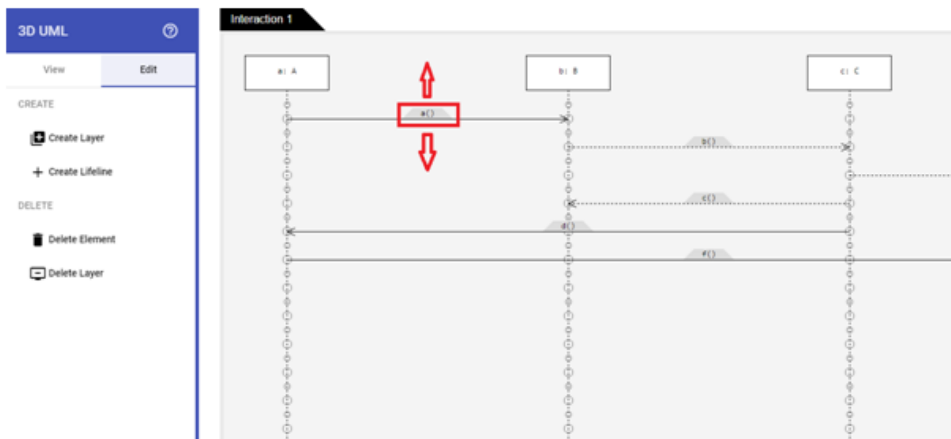
Enter message type
 synchCall

OK Cancel

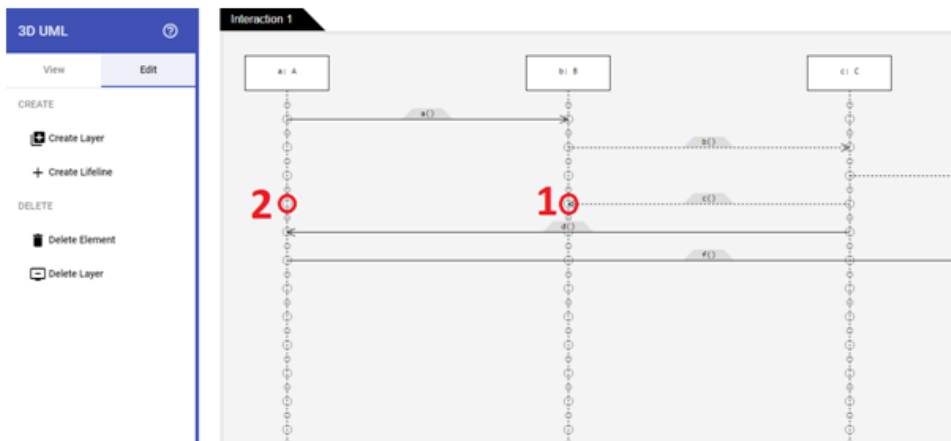
Upravovanie správy

Správou sa dá pohybovať vo viacerých smerom a taktiež sa dá premenovať.

Ak chceme hýbať správou vertikálne, klikneme na jej názov a držíme myšku stlačenú. Následne môžeme správou pohybovať hore a dole medzi dvomi čiarami života, ku ktorým náleží. Keď máme správu v pozícii, v ktorej ju chceme mať, jednoducho pustíme tlačítko na myši.



Ak chceme správou pohybovať horizontálne, musíme meniť jej zdrojové a cieľové body. To urobíme pomocou pravého tlačítka na myši. Klikneme na bod, ktorý chceme premiestniť pravým tlačítkom a následne klikneme na iný bod na inej čiare života, kde chceme aby sa správa končila/začínala.



Pre premenovanie správy musíme dvojkliknúť na jej názov a názov a vyplniť formulár. Následne potvrdíme tlačidlom „OK“. Takto môže taktiež meniť jej typ zo synchronnej na asynchrónnu a naopak.



Edit message

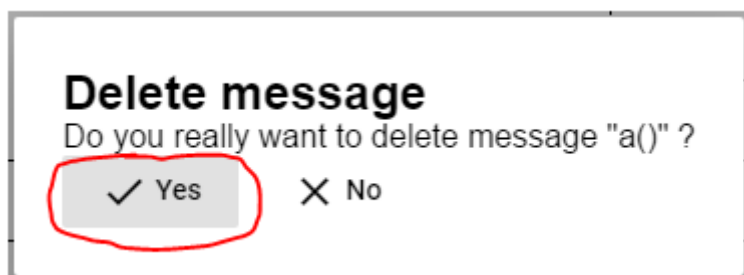
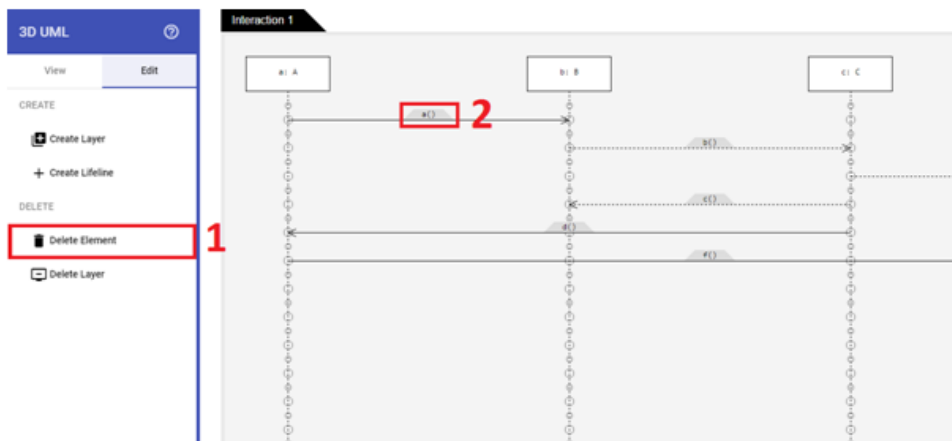
Enter message name

Enter message type

✓ OK
✗ Cancel

Vymazanie správy

Pre vymazanie správy sa musíme nachádzať v Edit mode. Následne klikneme na tlačidlo „Delete Element“ (znázornené ikonou smetného koša), ktoré sa nachádza v „Delete“ menu. Následne klikneme na hlavičku správy. Potom potvrdíme vymazanie správy tlačidlom „Yes“.



Inštalčná príručka

V tejto kapitole opisujeme požiadavky a použitie nástrojov potrebných na inštaláciu a údržbu produktu. Ďalšie informácie možno nájsť v súbore readme.md priloženom ku samotnému kódu produktu.

Požiadavky

Pre spustenie projektu sú potrebné nasledujúce závislosti:

- PHP >= 5.6.4 - <https://secure.php.net/>
- Npm > 3 (node >= 4.2.1) - <https://nodejs.org/>
- Composer - <https://getcomposer.org/>
- Apache 2 (so zapnutým modulom mod_rewrite) <https://httpd.apache.org/> alebo iný webservice - potrebné len pri nasadení na produkciu, pre potreby vývoja stačí integrovaný PHP development server, ktorý možno spustiť príkazom `php artisan serve`
- PostgreSQL prípadne iná podporovaná databáza opísaná v dokumentácii frameworku Laravel 5.4 (<https://laravel.com/docs/5.4/database>)

Postup inštalácie

Inštaláciu projektu možno vykonať nasledujúcimi krokmi:

1. Prejdeme do priečinka webservera, kde chceme aplikáciu uložiť, napríklad /var/www/3duml
Následne aplikáciu naklonujeme z repozitára, poprípade nakopírujeme z priloženého disku DVD.

```
$ git clone https://github.com/next-dimension-team/3duml
```

Zadáme nasledujúce príkazy do terminálu (pre Unixové systémy, pre OS Windows sa môžu niektoré príkazy odlišovať).

```
$ cd 3duml  
$ composer install  
$ sudo chmod -R 777 storage/  
$ sudo chmod -R 777 bootstrap/cache/  
$ cp .env.example .env  
$ php artisan key:generate  
$ npm install
```

2. Vytvoríme v konfigurácii webservera nový virtual host tak, že nasmerujeme DocumentRoot do priečinka public aplikácie (napr. /var/www/3duml/public/). Návod pre webserver Apache 2 - <https://www.digitalocean.com/community/tutorials/how-to-set-up-apache-virtual-hosts-on-ubuntu-14-04-lts>
3. Vytvoríme prázdnu databázu pre dáta aplikácie buď pomocou programu (pgAdmin) alebo zadaním príkazu do terminálu (<https://www.postgresql.org/docs/9.1/static/app-createdb.html>)

```
$ createdb 3duml
```

4. Následne je potrebné nastaviť údaje potrebné pre pripojenie k databáze v konfiguračnom súbore .env, napríklad ak ide o databázu PostgreSQL:

```
DB_CONNECTION=pgsql  
DB_HOST=localhost  
DB_PORT=5432  
DB_DATABASE=3duml  
DB_USERNAME=postgres  
DB_PASSWORD=postgres
```

5. Potom spustíme build príkazom (s nastavením prostredia development) nasledovne:

```
$ npm run build
```

Pre spustenie buildu s produkčným nastavením (optimalizácia, minifikovanie a pod.) použijeme nasledovný príkaz.

```
$ npm run build:prod
```

Pre zmenu url pre požiadavky na backend API podľa vytvoreného virtual hostu na webserveri (predvolene: <http://localhost/>) musíme pri kompilácii zadať premenné prostredia nasledovne. Napríklad ak chceme spustiť aplikáciu napríklad na Laravel development serveri

```
$ HOST=127.0.0.1 PORT=8000 npm run build
```

Development server

Pri vývoji možno spustiť development server s možnosťou detekcie zmien, rekompilácie a automatického reloadu v prehliadači bez potreby opakovaného spúšťania buildu (hot module replacement)

Spúšťa sa na <http://localhost:8080/> súčasne s Laravel development serverom ako proxy pre backend na <http://127.0.0.1:8000>. Spustíme ho zadaním nasledujúceho príkazu:

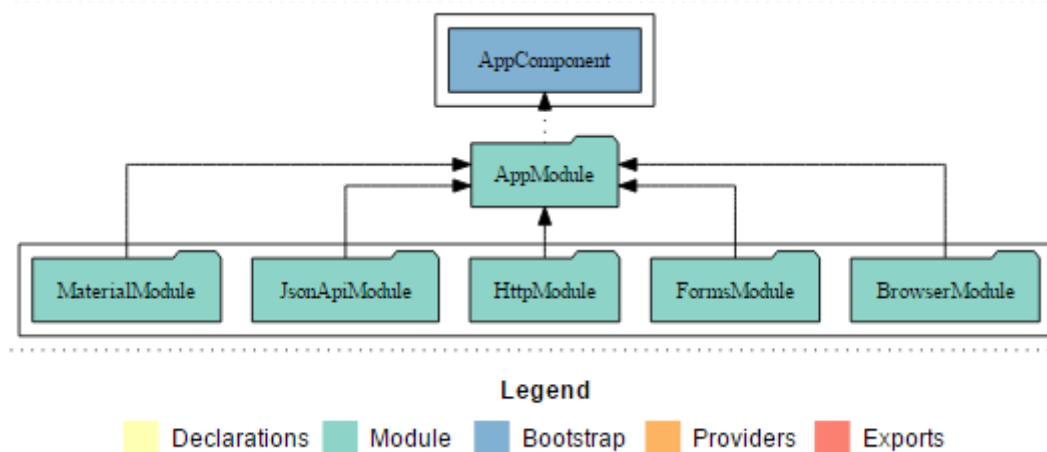
```
$ npm run start:hmr
```

Technická dokumentácia

Moduly

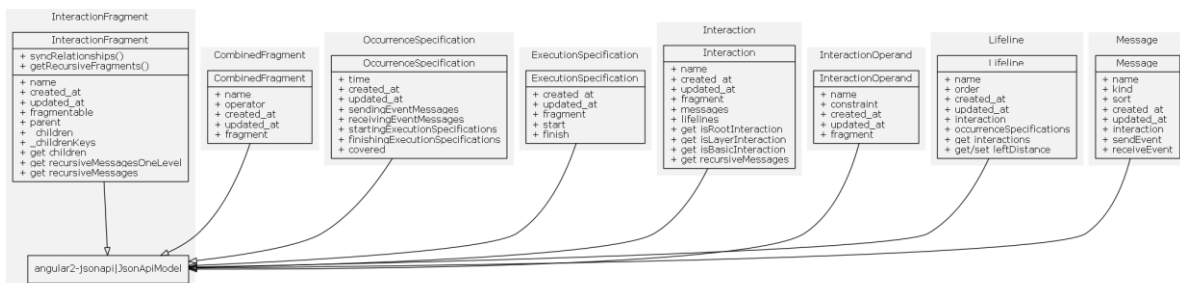
Aplikácia pozostáva z koreňového modulu AppModule, ktorý importuje nasledujúce moduly:

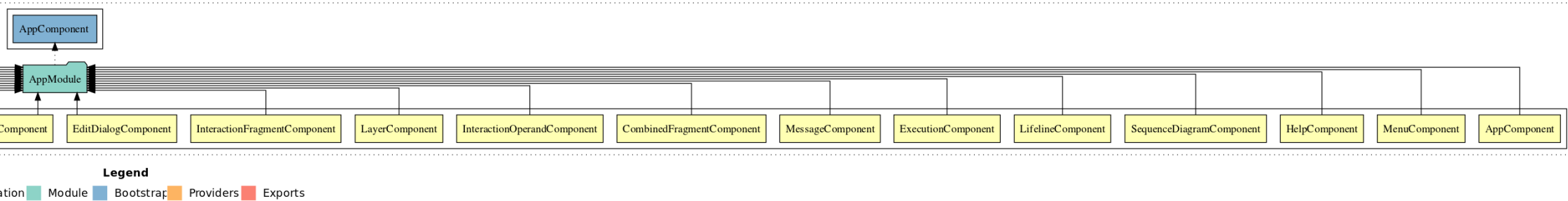
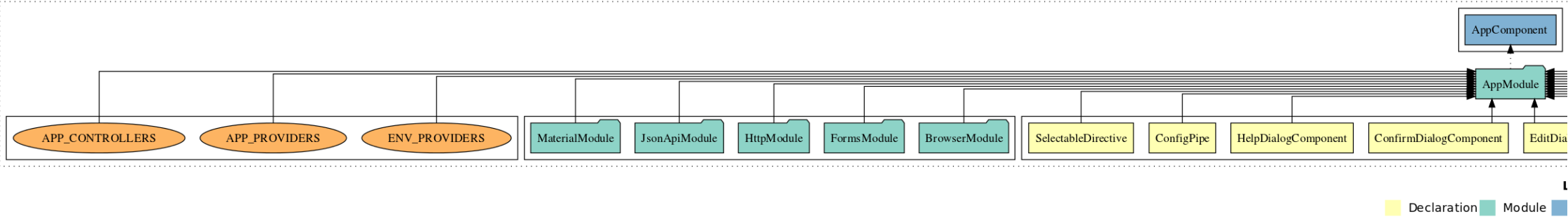
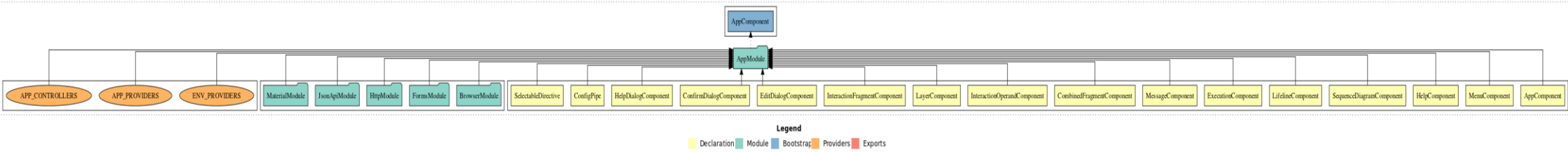
- *MaterialModule* - potrebný pre použitie Material design
- *JsonApiModule* – poskytuje implementáciu pre prácu s JSON API
- *HttpModule* - základný modul na prácu s HTTP požiadavkami
- *FormsModule* - základný modul potrebný na prácu s formulármi
- *BrowserModule* – modul potrebný pre beh aplikácie vo webovom prehliadači



Modely

Aplikácia mapuje entity na modely, ktoré sú totožné s databázovým modelom ale obsahujú niektoré špecifické metódy na prácu s nimi.

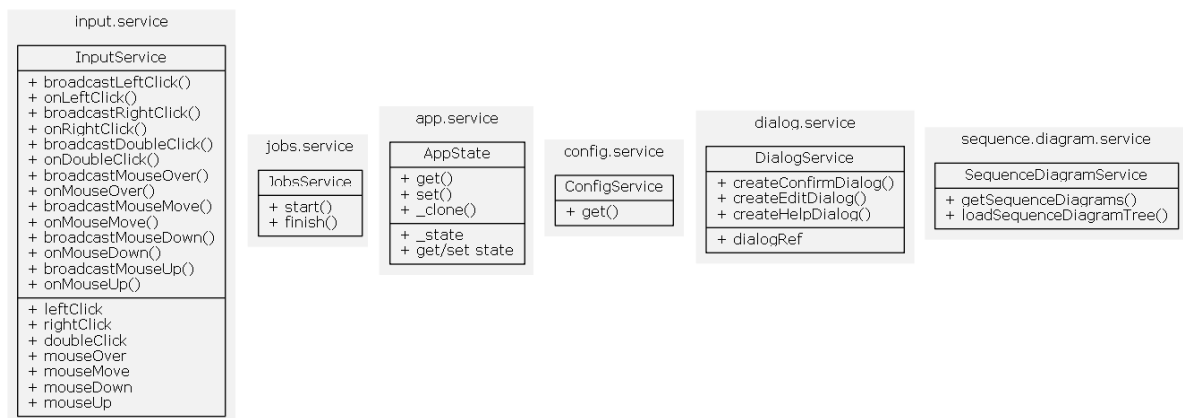




Služby

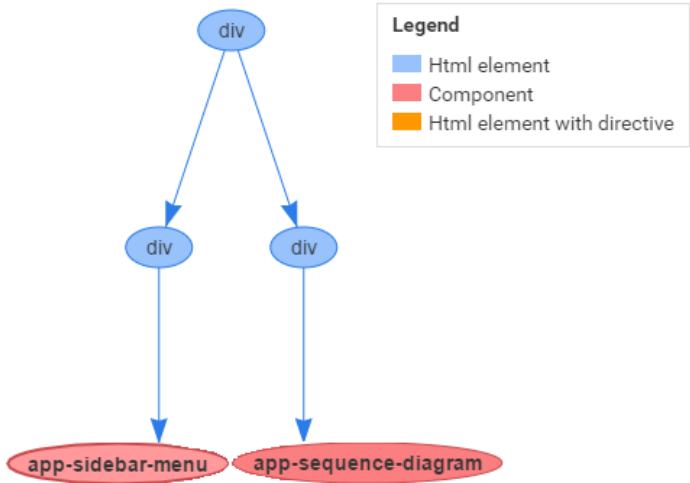
Znovupoužitelná funkcionálna aplikácie je implementovaná v aplikačných službách, ktoré je možné injektovať do konštruktora, kde chceme službu použiť. Aplikácia obsahuje nasledujúce služby:

- *InputService* - predstavuje centrálnu miesto na počúvanie na jednotlivé udalosti rozhrania (klikanie, pohyb myši) a zabezpečuje ich distribúciu na naviazaných subscriberov.
- *JobsService* - zabezpečuje evidenciu práve prebiehajúcich činností a tak napr. zmenu kurzora myši pri vykonávaní
- *AppState* - udržiava stav aplikácie a zabezpečuje podporu HMR – hot module replacement
- *ConfigService* - zabezpečuje získanie hodnoty z konfigurácie
- *DialogService* - slúži na dynamické vytvorenie komponentov pre rôzne druhy dialógových okien
- *SequenceDiagramService* – implementuje základné metódy na načítavanie potrebných dát z backendu aplikácie



Komponenty

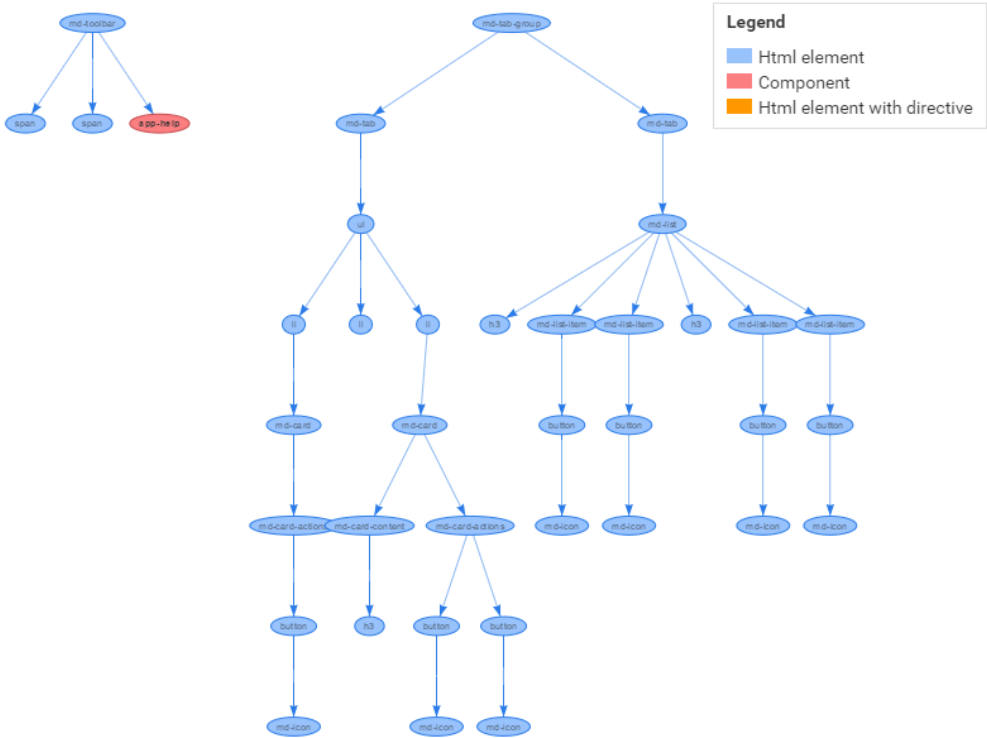
AppComponent:



SequenceDiagramComponent



MenuComponent:



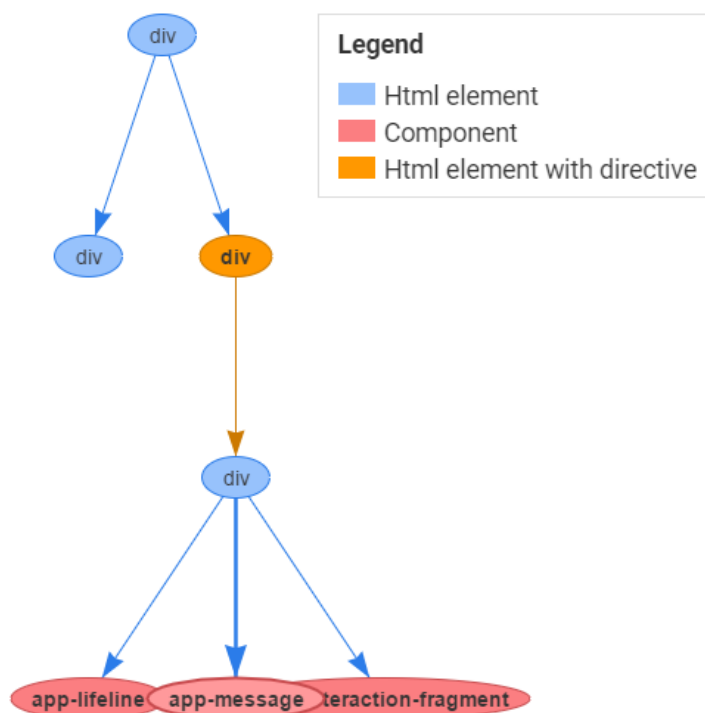
Vrstva (LayerComponent)

Šablóna (template): `ng-app/app/sequence-diagram/components/layer.component.html`

Komponent: `ng-app/app/sequence-diagram/components/layer.component.ts`

Šablóna (angl. template) komponentu LayerComponent je tvorená HTML elementom div (layer-wrapper), ktorý pozostáva z dvoch div elementov. Prvý div element (layer-title) slúži na vizualizáciu názvu vrstvy a druhý div element (layer) vizualizuje samotnú vrstvu a obsahuje div element (sequence-diagram), ktorý obsahuje komponenty app-lifeline (čiara života), app-message (správa) a app-interaction-fragment. Na obrázku nižšie je vizualizovaná štruktúra šablóny tohoto komponentu.

Tento komponent je súčasťou komponentu SequenceDiagramComponent pričom jednotlivé vrstvy sú vykresľované do 3D scény prostredníctvom three.js a CSS3D. Logika tohoto vykresľovania sa nachádza v zdrojovom súbore komponentu, ktorý je uvedený na začiatku tejto kapitoly.



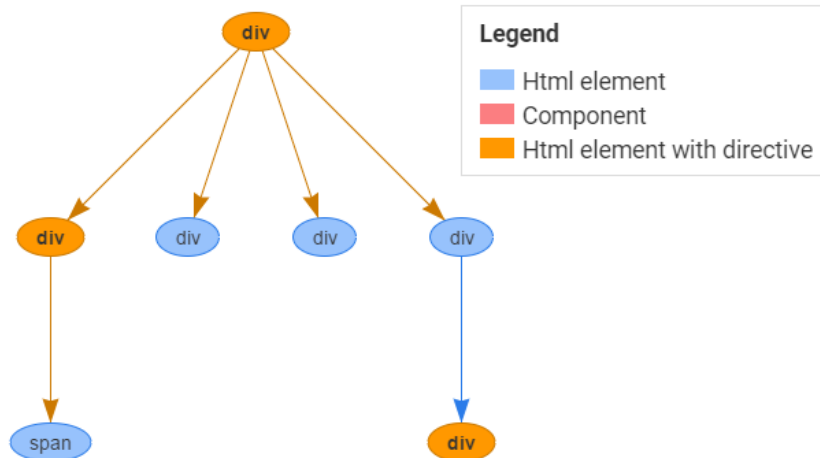
Obrázok: Štruktúra šablóny komponentu LayerComponent (vrstva)

Čiara života (LifelineComponent)

Šablóna (template): `ng-app/app/sequence-diagram/components/lifeline.component.html`

Komponent: `ng-app/app/sequence-diagram/components/lifeline.component.ts`

Šablóna komponentu LifelineComponent je tvorená zo 4 div elementov. Prvý div element slúži na vypísanie názvu lifeline (čiary života), ktorý je editovateľný. Druhý element je vertikálna čiara, čiary života. Tretí element sú vykreslené body na čiare života, z ktorých idú správy medzi čiarami života. Štvrtým elementom div je na vykreslenie „hlavičky“ čiary života, ktorá obklopuje jej názov.



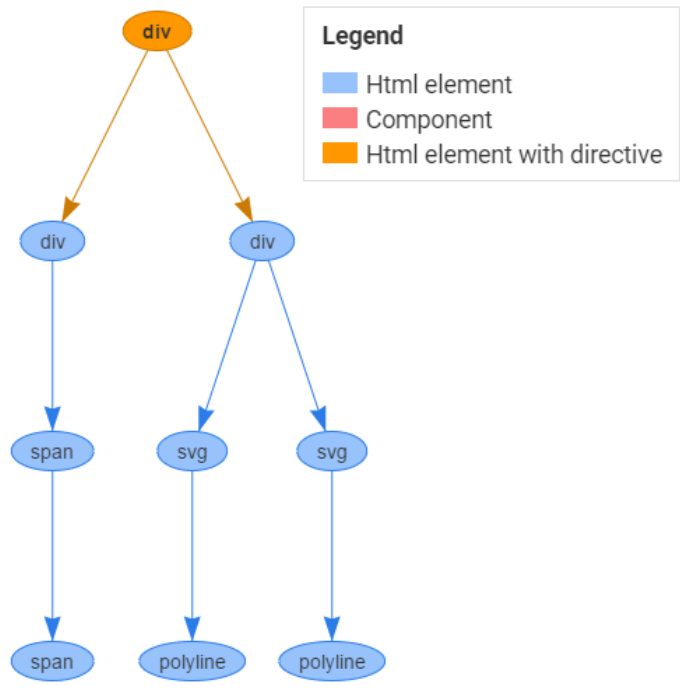
Obrázok: Štruktúra šablóny komponentu LifelineComponent (čiara života)

Správa (MessageComponent)

Šablóna (template): [ng-app/app/sequence-diagram/components/message.component.html](#)

Komponent: [ng-app/app/sequence-diagram/components/message.component.ts](#)

Šablóna komponentu MessageComponent je tvorená z 2 div elementov. Prvý div element je tvorený zo span elementu, ktorý má v sebe ďalší span element, v ktorom sa vizualizuje názov správy. Druhý div element obsahuje svg elementy pre vizualizáciu šípky, ktorá reprezentuje správu, a to 2 typy: sprava doľava a zľava doprava.



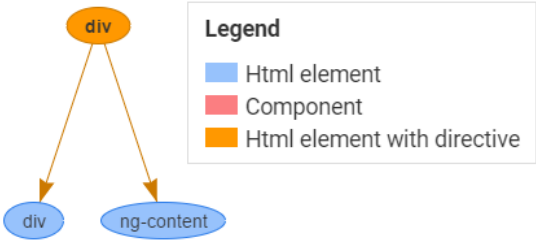
Obrázok: Štruktúra šablóny komponentu MessageComponent (správa)

Interakčný operand (InteractionOperandComponent)

Šablóna (template): ng-app/app/sequence-diagram/components/interaction-operand.component.html

Komponent: ng-app/app/sequence-diagram/components/ interaction-operand.component.ts

Šablóna komponentu InteractionOperandComponent je zložená z elementu div, v ktorom sa vykresľuje podmienka pri interakčnom operande (napríklad $a > 0$) a z elementu ng-content, ktorý obsahuje dynamický obsah, ktorý môže byť pridaný v komponente InteractionFragmentComponent, čiže napríklad ďalší kombinovaný fragment, alebo ďalší operand.



Obrázok: Štruktúra šablóny komponentu InteractionOperand

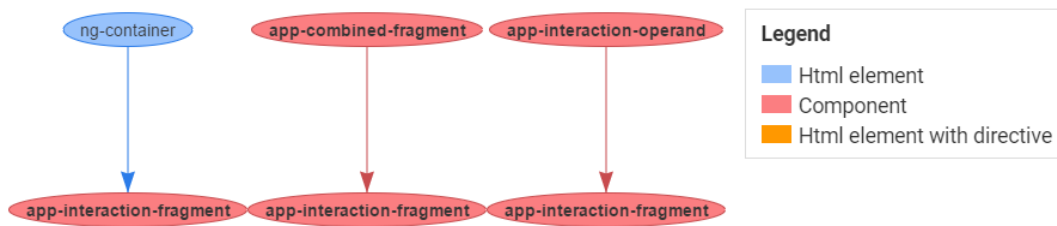
Interakčný fragment (InteractionFragmentComponent)

Šablóna (template): ng-app/app/sequence-diagram/components/interaction-fragment.component.html

Komponent: ng-app/app/sequence-diagram/components/ interaction-fragment.component.ts

Šablóna komponentu InteractionFragmentComponent je vlastne abstraktným komponentom nad komponentami kombinovaného fragmentu, interakčného operandu a ďalšieho interakčného fragmentu.

Tým pádom element ng-container môže obsahovať ďalší interakčný fragment. App-combined-fragment element obsahuje telo komponentu CombinedFragmentComponent a app-interaction-operand telo komponentu InteractionOperandComponent.



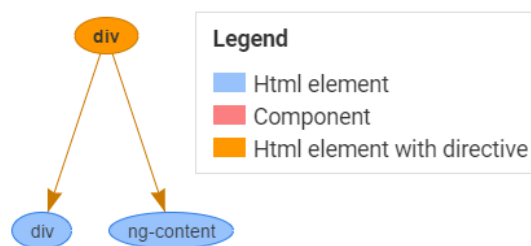
Obrázok: Štruktúra šablóny komponentu InteractionFragment

Kombinovaný fragment (CombinedFragmentComponent)

Šablóna (template): [ng-app/app/sequence-diagram/components/combined-fragment.component.html](#)

Komponent: [ng-app/app/sequence-diagram/components/combined-fragment.component.ts](#)

Šablóna komponentu CombinedFragmentComponent je tvorená z div elementu a ng-content elementu. V div elemente sa vypisuje názov operátora kombinovaného fragmentu. Ng-content je dynamický element pre možnosti pridania obsahu do vnútra kombinovaného fragmentu (napríklad správy).



Obrázok: Štruktúra šablóny komponentu CombinedFragment

Lifeline

Pri čiarach života, ako aj pri ostatných súčiastiach sekvenčného diagramu máme dva rozdielne modely. Jeden je model, ktorý je uložený v databáze, druhý je komponent, ktorý používame na zobrazenie čiary života v aplikácií.

Model uložený v databáze pozostáva z nasledujúcich súčastí:

- ID čiary života
- Čas a dátum vytvorenia záznamu v databáze
- Čas a dátum poslednej úpravy čiary života
- Meno čiary života
- Poradie danej čiary v rámci vrstvy, v ktorej sa nachádza

- ID vrstvy, ku ktorej čiara patrí

Tieto modely sa pomocou knižnice JSON API ukladajú a načítavajú z PostgreSQL databázy. V okamihu, keď je model načítaný, vytvorí sa z neho komponent, ktorý je následne možné zobrazit'.

Takýto komponent sa oproti modelu odlišuje miernymi zmenami. Neobsahuje údaje, ako dátum vytvorenia a zmeny, alebo poradie čiary života v rámci vrstvy. Poradie je tu nahradené premennou left. Tá určuje, o koľko pixelov má byť zľava má byť na stránke zobrazená daná čiara. Okrem toho obsahuje aj nový atribút pole bodov. To sú body, ktoré sa nachádzajú na celej čiare, na ktorých sú napojené správy medzi viacerými čiarami života.

Pridanie Lifeline

Pridanie novej čiary života funguje jednoduchým spôsobom. Po stlačení tlačidla na vytvorenie novej čiary a zadaní názvu systém prejde všetkými čiarami, zistí najväčšiu pozíciu a následne pridá novú čiaru s pozíciou o jedna vyššou.

```
for (let lifeline of lifelines) {
  if (lifeline.order > maxOrder) {
    maxOrder = lifeline.order;
  }
}

// Vytvoríme lifelinu úplne nakonci
this.datastore.createRecord(M.Lifeline, {
  name: result.name,
  order: maxOrder + 1,
  interaction: this.sequenceDiagramComponent.editingLayer.fragmentable
}).save().subscribe(() => {
  this.sequenceDiagramComponent.refresh(() => {
    this.jobsService.finish('createLifeline');
  });
});
```

Premenovanie lifeline

K premenovaniu dôjde v prípade, že používateľ dva krát klikne na názov čiary. Túto zmenu možno vykonať iba v prípade, že sa používateľ nachádza v editačnom móde. Zobrazí sa mu kontextové okno, do ktorého zadá nový názov a ten následne potvrdí.

```
this.dialogService.createEditDialog("Edit lifeline", lifeline, "Enter lifeline name", "lifeline")
  .componentInstance.onOk.subscribe((result) => {
    // Start job
    this.jobsService.start('renameLifeline');
    // Rename lifeline
```

```

    lifeline.name = result.name;
    lifeline.save().subscribe(() => {
        // Finish job
        this.jobsService.finish('renameLifeline');
    });
});

```

Tento nový názov čiary prepíše starý názov v databáze.

Zmazanie lifeline

K zmazaniu dôjde podobne ako aj pri ostatných elementoch. Z kontextového menu používateľ zvolí, že chce zmazať element a klikne na čiara. Vyskočí potvrdzovacie okno, po ktorom sa čiara, spolu so všetkými naviazanými správami zmaže z databázy a zobrazený diagram sa prekreslí.

```

for (let occurrence of occurrences) {
    // Označime occurrence specifications, ktoré su na lifeline na zmazanie
    occurrencesToDelete.push(occurrence);
    let messages = occurrence.sendingEventMessages;
    for (let message of messages) {
        // Pridame occ spec na prijimacej strane message
        occurrencesToDelete.push(message.receiveEvent);
    }
    messages = occurrence.receivingEventMessages;
    for (let message of messages) {
        // Pridame occ spec na odosielacej strane message
        occurrencesToDelete.push(message.sendEvent);
    }
}
// Zmaz všetky occurrence specification, ktoré su ovplyvnenie zmazaním
lifeline
for (let occurrence of occurrencesToDelete) {

this.jobsService.start('deleteOccurrenceSpecification.occurrenceSpecification.'
+ occurrence.id);
    this.datastore.deleteRecord(M.OccurrenceSpecification,
occurrence.id).subscribe(() => {

this.jobsService.finish('deleteOccurrenceSpecification.occurrenceSpecification
.' + occurrence.id);
    });
}
// Start job
this.jobsService.start('deleteLifeline');
this.datastore.deleteRecord(M.Lifeline, lifeline.id).subscribe(() => {
    this.sequenceDiagramComponent.refresh(() => {
        // Finish job
        this.jobsService.finish('deleteLifeline');
    });
});

```



```
});  
});
```

Samotná čiara sa zmaže až v okamihu, keď už boli vymazané všetky správy, ktoré do nej smerovali, alebo z nej vychádzali.

Zmenenie poradia lifeline

Vzhľadom na to, že je možné pridať čiaru iba na koniec vrstvy je potrebné, aby bolo možné presúvať čiary. Toto posúvanie funguje jednoduchým chytením čiary. Čiara sa pohybuje spolu s myšou a stačí ju pustiť iba na požadované miest. Pre prehľadnosť sa spolu s čiarou pohybujú aj všetky správy, ktoré sú na ňu naviazané.

V okamihu, keď čiaru pustíme na nové miesto, dôjde k prepočítaniu pozícií všetkých ostatných čiar. Tieto nové pozície sú pre všetky čiary aktualizované v databáze a diagram sa prekreslí podľa aktuálnych pozícií.

```
for (let lifeline of lifelinesInInteraction) {  
    // Lifeline, ktoru posuvam prehodim na jej nove miesto a ulozim do  
    DB  
    if (lifeline.id == draggingLifelineModel.id) {  
        lifeline.order = position;  
        this.jobsService.start('reorderLifeline.lifeline.' + lifeline.id);  
        // Ulozime lifeline na jej nove miesto  
        lifeline.save().subscribe(() => {  
            this.jobsService.finish('reorderLifeline.lifeline.' +  
lifeline.id);  
        });  
        continue;  
    }  
    // Lifeline musim posunut o 1 dolava, aby za nu mohla ist nova  
    else if (lifeline.order >= originalOrder && lifeline.order <=  
position) {  
        lifeline.order--;  
        this.jobsService.start('reorderLifeline.lifeline.' + lifeline.id);  
        // Ulozime lifeline s o 1 nizsou poziciou do DB  
        lifeline.save().subscribe(() => {  
            this.jobsService.finish('reorderLifeline.lifeline.' +  
lifeline.id);  
        });  
        continue;  
    }  
    // Posunul som lifeline pred tuto, teda musim ju posunut o 1 do  
    prava  
    else if (lifeline.order < originalOrder && lifeline.order >=  
position) {  
        lifeline.order++;  
        this.jobsService.start('reorderLifeline.lifeline.' + lifeline.id);  
    }  
}
```

```
    // Ulozime lifeline s o 1 vacsou poziciou do DB
    lifeline.save().subscribe(() => {
      this.jobsService.finish('reorderLifeline.lifeline.' +
lifeline.id);
    });
    continue;
  }
}
```

Backend API

JSON API na serverovej časti je riešené implementáciou potrebných tried, ktoré rozširujú triedy balíka <https://github.com/cloudcreativity/laravel-json-api>. API sa nachádza na adrese s prefixom `/api/v1/`. Obsahuje nasledujúce dostupné zdroje (resources), ktoré nájdeme v prečinku `app/JsonApi/`:

- CombinedFragments - `/api/v1/combined-fragments`
- ExecutionSpecifications - `/api/v1/execution-specifications`
- InteractionFragments - `/api/v1/interaction-fragments`
- InteractionOperands - `/api/v1/interaction-operands`
- Interactions - `/api/v1/interactions`
- Lifelines - `/api/v1/lifelines`
- Messages - `/api/v1/messages`
- OccurrenceSpecifications - `/api/v1/occurrence-specifications`

Celý zoznam dostupných ciest (routes) spolu s metódami možno zobrazit' spustením nasledujúceho príkazu v koreňovom adresári aplikácie:

```
$ php artisan route:list
```

Každý zdroj je opísaný implementáciou 4 tried:

- Schema – obsahuje opis zdroja, teda ktoré atribúty a dáta zo vzťahov sa posielajú pri požiadavke na zobrazenie zdroja
- Request - obsahuje opis povolených vzťahov, filtrov a zorad'ovacích parametrov
- Hydrator – obsahuje opis atribútov a vzťahov, ktoré môžu byť ukladané pri jednotlivých zdrojoch
- Validators – obsahuje validáciu atribútov a vzťahov pri ukladaní a modifikácii zdroja a validáciu filtrov