

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5384-6041

**TVORBA VIRTUÁLNEHO LABORATÓRIA S
VYUŽITÍM JAVASCRIPT-U NA STRANE SERVERA
DIPLOMOVÁ PRÁCA**

2016

Bc. Erich Stark

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5384-6041

**TVORBA VIRTUÁLNEHO LABORATÓRIA S
VYUŽITÍM JAVASCRIPT-U NA STRANE SERVERA
DIPLOMOVÁ PRÁCA**

Študijný program: Aplikovaná informatika
Číslo študijného odboru: 2511
Názov študijného odboru: 9.2.9 Aplikovaná informatika
Školiace pracovisko: Ústav informatiky a matematiky
Vedúci záverečnej práce: Ing. Pavol Bisták, PhD.

Bratislava 2016

Bc. Erich Stark



ZADANIE DIPLOMOVEJ PRÁCE

Študent: **Bc. Erich Stark**
ID študenta: 6041
Študijný program: Aplikovaná informatika
Študijný odbor: 9.2.9. aplikovaná informatika
Vedúci práce: Ing. Pavol Bisták, PhD.
Miesto vypracovania: Ústav automobilovej mechatroniky

Názov práce: **Tvorba virtuálneho laboratória s využitím JavaScript-u na strane servera**

Špecifikácia zadania:

Cieľom práce je preskúmanie možností JavaScript-u pre realizáciu serverovej časti virtuálneho laboratória a vytvorenie funkčného virtuálneho laboratória, kde klient aj server budú realizovaný s využitím JavaScript-u.

Úlohy:

1. Naštudujte problematiku virtuálnych laboratórií a zoznámte sa s existujúcimi riešeniami.
2. Navrhňte objektový model virtuálneho laboratória. Definujte jednotlivé subsystémy a ich rozhrania. Využite možnosti JavaScript-u.
3. Implementujte serverovú časť aplikácie pre virtuálne laboratórium s využitím Node.js.
4. Pre overenie funkčnosti vytvorte klientskú časť aplikácie.
5. Aplikácie otestujte a porovnajte s predošlými riešeniami založenými na technológiách Java a .NET. Vypracujte technickú dokumentáciu.

Zoznam odbornej literatúry:

1. 1. Davoli, F.; Meyer, N.; Pugliese, R.; Zappatore, S.: Remote Instrumentation and Virtual Laboratories. Springer Verlag, 2010. ISBN 978-1-4419-5597-5
2. 2. G. Rauch: Smashing Node.js: JavaScript Everywhere. Wiley, 2. vydanie, 2012, ISBN 978-1119962595.

Riešenie zadania práce od: 21. 09. 2015

Dátum odovzdania práce: 20. 05. 2016



Bc. Erich Stark

študent



prof. RNDr. Otokar Grošek, PhD.

vedúci pracoviska



prof. RNDr. Gabriel Juhás, PhD.

garant študijného programu

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Bc. Erich Stark
Diplomová práca:	Tvorba virtuálneho laboratória s využitím JavaScript-u na strane servera
Vedúci záverečnej práce:	Ing. Pavol Bisták, PhD.
Miesto a rok predloženia práce:	Bratislava 2016

Diplomová práca sa zaoberá využitím možností modernej platformy Node.js v oblasti virtuálnych laboratórií a vytvorením referenčnej aplikácie v spojení ďalších technológií ako Matlab, Simulink, Angular.js a MongoDB. V úvode práce sú popísané vlastnosti virtuálnych laboratórií a jeho možných komponentov. Taktiež pojednávame o možnostiach interakcie s experimentami. V ďalšej časti boli porovnané už existujúce riešenia a ich možné nedostatky v súčasnosti. Nasleduje sekcia, kde sme si definovali technológie a ich hlavné vlastnosti, ktoré sme plánovali využiť. Implementácia riešenia prebiehala vytvorením menších častí. Ako prvú sme implementovali referenčnú simuláciu šikmého vrhu do Matlabu a Simulinku. Bolo potrebné získať dáta zo Simulinku do Matlab workspace. Ten ich následne posiela do Node.js pomocou RESTful služieb. Na strane Node.js čaká na dáta Socket.io, ktorý ich pošle do webového prehliadača. Posledná časť hovorí o vizualizácii dát v prehliadači vo forme grafu, animácie a tabuľky a následné zapísanie do databázy. Výsledkom práce je funkčné riešenie, kde je možné implementovať vlastnú simuláciu.

Kľúčové slová: virtuálne laboratórium, web, html, javascript, angular.js, json, mongodb, node.js, npm, express.js, restful, matlab, simulink

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Author:	Bc. Erich Stark
Diploma Thesis:	Virtual laboratory using JavaScript on the server side
Supervisor:	Ing. Pavol Bisták, PhD.
Place and year of submission:	Bratislava 2016

This thesis deals with possibilities of using modern Node.js platform in virtual laboratories and create a reference application in combination of other technologies such as Matlab, Simulink, Angular.js and MongoDB. In the introduction we described the characteristics of virtual laboratories and its possible components. We also discussed the possibilities of interaction with the experiment. In the next section we compared existing solutions and their possible lack in nowadays. The following is a section where we have defined the technology and their main characteristics that we planned to use. Implementation of solution was carried out by creating smaller parts. At first we have implemented a simulation motion of projectile in Matlab and Simulink. It was necessary to get data from Simulink to Matlab workspace. Then we had to send them to Node.js using RESTful web services. On the side of Node.js was waiting Socket.io for data receiving that were sent to the web browser. The last part refers to the visualization of data in the browser in the form of graphs, animations, data table and subsequently write data into the database. The result of this thesis is a functional solution called StarkLab where is possible implement own simulation.

Keywords: virtual laboratory, web, html, javascript, angular.js, json, mongodb, node.js, npm, express.js, restful, matlab, simulink

Vyhlásenie autora

Podpísaný Bc. Erich Stark čestne vyhlasujem, že som diplomovú prácu Tvorba virtuálneho laboratória s využitím JavaScript-u na strane servera vypracoval na základe poznatkov získaných počas štúdia a informácií z dostupnej literatúry uvedenej v práci.

Vedúcim mojej diplomovej práce bol Ing. Pavol Bisták, PhD.

Bratislava, dňa 12.5.2016

.....
podpis autora

Podakovanie

Ďakujem môjmu vedúcemu práce Ing. Pavlovi Bistákovi, PhD. za jeho odbornú pomoc, snahu, zhovievavosť, pripomienky a cenné rady, ktoré mi boli poskytnuté pri vypracovaní diplomovej práce. Tiež sa chcem podakovať Petrovi Širkovi, ktorý mi vnúkol myšlienku využiť modernú technológiu Node.js. Moja veľká vďaka patrí tiež rodičom za podporu počas celého štúdia a priateľke za trpezlivosť a porozumenie v čase písania tejto práce.

Obsah

Úvod	14
1 Cieľ práce	15
2 Virtuálne laboratória	16
2.1 Prehľad existujúcich virtuálnych laboratórií	17
2.1.1 Nevýhody existujúcich riešení	18
2.2 Komponenty virtuálneho laboratória	18
3 Použité technológie	20
3.1 MATLAB R2015b	20
3.1.1 Simulink	20
3.1.2 Komunikácia medzi Matlabom a Node.js	20
3.2 Node.js	24
3.2.1 História	24
3.2.2 Architektúra	25
3.2.3 Event loop	27
3.2.4 Možnosti a využitie	28
3.3 Node Package Manager	29
3.3.1 Použitie modulov	30
3.3.2 Vstavané moduly	30
3.3.3 Vytvorenie web servera pomocou HTTP modulu	30
3.4 Express.js	31
3.4.1 Web server	31
3.4.2 Request a Response objekty	32
3.4.3 Routing	33
3.4.4 Middleware	33
3.5 MongoDB	34
3.6 Angular.js	34
3.6.1 Koncept	35
3.6.2 Directives	35
3.6.3 Scope	36
3.6.4 Expressions	38
3.6.5 Data Binding	38
3.6.6 Controller	39

3.6.7	Module	40
3.6.8	Service	40
4	Návrh a implementácia StarkLab	41
4.1	Referenčný model simulácie v Matlabe	42
4.2	Diagramy	44
4.2.1	Prípad použitia	44
4.2.2	Diagram "tried" pre StarkLab	45
4.2.3	Sekvenčný diagram	46
4.3	Databázový model MongoDB	49
4.4	Node.js s frameworkom Express.js	50
4.4.1	Vlastný middleware prihlasovania	50
4.4.2	Spustenie Matlabu z príkazového riadku	51
4.4.3	Práca s dátami simulácie	52
4.4.4	Zápis dát do MongoDB	53
4.5	Webový klient s frameworkom Angular.js	55
4.5.1	Grafy s Chart.js	56
4.5.2	Animácia pomocou html canvas	58
4.5.3	Tabuľka dát	58
4.5.4	Zobrazenie existujúcich simulácií	59
	Záver	61
	Zoznam použitej literatúry	62
	Prílohy	I
A	Štruktúra elektronického nosiča	II

Zoznam obrázkov a tabuliek

Obrázok 1	Rozdiel medzi osobne a vzdialene riadeným experimentom.	16
Obrázok 2	HTTP request.	22
Obrázok 3	HTTP POST request.	22
Obrázok 4	HTTP GET request.	23
Obrázok 5	Architektúra Node.js platformy.	25
Obrázok 6	Udalostná slučka v Node.js.	28
Obrázok 7	Potrebné vlastnosti nastavené v package.json.	29
Obrázok 8	Dependencies vlastnosť v package.json.	30
Obrázok 9	Dependencies vlastnosť v package.json.	31
Obrázok 10	Spustenie web servera na porte 8081.	32
Obrázok 11	Objekt req a res ako parametre callback funkcie.	32
Obrázok 12	Vlastnosti middleware funkcie [3].	34
Obrázok 13	JSON dokument v MondoDB.	34
Obrázok 14	Koncept frameworku Angular.js.	35
Obrázok 15	Možnosti sledovania modelu v Angular.js.	37
Obrázok 16	Data binding v Angular.js.	38
Obrázok 17	Ukážka controlleru v Angular.js a jeho volanie na HTML elemente.	39
Obrázok 18	Vytvorenie modulov a pridanie ich funkcionality.	40
Obrázok 19	Návrh komunikácie medzi komponentami VL.	41
Obrázok 20	Inicializačná funkcia šikmého vrhu v Matlabe.	42
Obrázok 21	Hodnoty v Matlab workspace po spustení funkcie.	43
Obrázok 22	Diagram prípadu použitia na prácu zo systémom.	45
Obrázok 23	Diagram "tried" pre náš systém.	46
Obrázok 24	Sekvenčný diagram pre prihlásenie pomocou LDAP.	47
Obrázok 25	Prihlásenie do aplikácie pomocou LDAP v JavaScripte.	48
Obrázok 26	Komunikácia medzi jednotlivými komponentami systému.	49
Obrázok 27	Model objektu v JavaScripte, pre vytvorenie záznamu v MongoDB.	49
Obrázok 28	Príklad záznamu simulácie v MongoDB.	50
Obrázok 29	Middleware kód v Express.js na kontrolu prihlásenia.	51
Obrázok 30	Spustenie príkazu v OS pomocou shelljs modulu.	52
Obrázok 31	Prijímanie a zasielanie dát pomocou Socket.io do prehliadača.	52

Obrázok 32	Vloženie záznamu do MongoDB pomocou JavaScriptu.	53
Obrázok 33	Vyhľadanie záznamu v MongoDB pomocou JavaScriptu.	54
Obrázok 34	Volanie vloženia záznamu do MongoDB.	54
Obrázok 35	Volanie vyhľadania záznamu v MongoDB.	55
Obrázok 36	Prihlásenie do LDAP pomocou STUBA údajov.	55
Obrázok 37	Parametre simulácie - počiatočná rýchlosť a uhol v stupňoch. . . .	56
Obrázok 38	Inicializácia hodnôt pre Chart.js.	57
Obrázok 39	Graf vykresľujúci závislosti $[x, y]$ v šikmom vrhu.	57
Obrázok 40	Animácia vykresľujúca závislosti $[x, y]$ v šikmom vrhu.	58
Obrázok 41	Tabuľka dát času, x , y a smer rýchlosti vy.	58
Obrázok 42	Zoznam uložených simulácií pre prihláseného užívateľa.	59
Obrázok 43	Graf pre vybranú vzorku dát.	60
Obrázok 44	Animácia pre vybranú vzorku dát.	60
Tabuľka 1	Porovnanie fyzických, virtuálnych a vzdialených a laboratórií. . . .	17
Tabuľka 2	Porovnanie virtuálnych laboratórií vytvorených mimo FEI STU. . .	18
Tabuľka 3	Porovnanie virtuálnych laboratórií vytvorených na FEI STU. . . .	18

Zoznam skratiek a značiek

VL - Virtual Laboratory

StarkLab - centrálny webový server nad Node.js serverom

REST - Representational state transfer

URL - Uniform Resource Locator

NPM - Node package manager

LTS - Long term support

API - Application programming interface

I/O - input and output

SSL - Secure Sockets Layer

TLS - Transport Layer Security

RDN - Relative Distinguished Name

COM - Component Object Model

MVC - Model View Controller

MVVM - Model View View Model

XML - eXtensible Markup Language

JSON - JavaScript Object Notation

HTTP - HyperText Transfer Protocol

URL - Uniform Resource Locator

CDN - Content Delivery Network

DOM - Document Object Model

LDAP - Lightweight Directory Access Protocol

TCP - Transmission Control Protocol

UDP - User Datagram Protocol

JVM - Java Virtual Machine

DNS - Domain Name System

ANSI - American National Standards Institute

TTY - teletype

IPC - inter-process communication

Zoznam algoritmov

Úvod

Praktické cvičenie v laboratóriu je dôležitá súčasť pri procese vzdelávania technicky založených ľudí. Ako aj raz povedal starý čínsky filozof Confucius: *"Povedz mi a ja zabudnem, nauč ma a ja si spomeniem, ale nechaj ma zúčastniť sa a ja pochopím."* Zo skúseností už vieme, že človek sa najrýchlejšie učí tak, že si danú vec niekoľko krát sám vyskúša a tak najlepšie pochopí ako to funguje. Nanešťastie nie je možné vždy zabezpečiť výskumníkom alebo študentom priamy prístup k reálnym zariadeniam pre vykonanie experimentu. Problémov môže byť viacero: vysoká cena vybavenia laboratória, bezpečnosť na pracovisku v závislosti od experimentu prípadne nedostatok kvalifikovaných asistentov.

V posledných rokoch sa vývoj virtuálnych systémov zvýšil hlavne vďaka technologickej evolúcii softvérového inžinierstva. Pokrok moderných technológií nám dáva solídny základ pri tvorbe, či už všeobecne virtuálnych systémov nápomocných pre online výučbu, alebo konkrétnych virtuálnych laboratórií, kde sa simulujú fyzikálne javy a procesy. Pri experimentoch vykonávaných vo virtuálnom prostredí je možné zdieľať zdroje tohto prostredia na to, aby sa k nemu pripojilo viac užívateľov, ktorí chcú vykonávať rovnaký experiment, čo by pri reálnom zariadení nebolo možné. Vďaka tomu je virtuálne laboratórium vhodným doplnkom štúdia aj výskumu, kde je možné si skúsiť rôzne variácie experimentu bez ohrozenia na zdraví, prípadne zničenia zariadenia a až potom skúšať na reálnom zariadení, ak je to potrebné.

1 Cieľ práce

Všeobecným cieľom diplomovej práce je analyzovať existujúce riešenia virtuálnych laboratórií a možností Node.js pre vytvorenie nového.

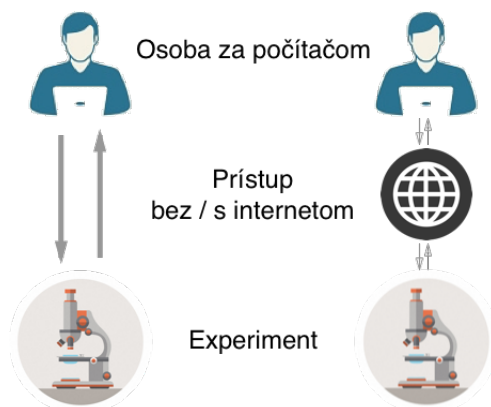
Na základe zistených možností je potrebné vytvoriť virtuálne laboratórium ako klient-server architektúru, kde server bude Node.js, klienti matlab a webová aplikácia v prehliadači. Experiment vrámci virtualného laboratória prebieha ako simulácia v Matlabe cez rozšírenie Simulink. Táto aplikácia nebude obmedzená len na lokálnu sieť, ale bude prístupná aj z internetu. Klient aj server bude vytvorený v dynamicky typovom jazyku JavaScript. Údaje z experimentu sa budú zasielať z Matlabu na server cez RESTful služby, kde môžu byť následne spracované a uložené do databázy, alebo zaslané klientovi do prehliadača.

2 Virtuálne laboratória

V dobe, keď internet ešte nebol rozšírený, sa experimenty robili vo fyzických laboratóriách. Bolo dôležité dodržiavať rôzne bezpečnostné predpisy, kvôli možnému úrazu osoby prípadne poškodeniu prístrojov.

Vzdialenosť ale aj nedostatok finančných zdrojov nám sťažuje podmienky pri vykonávaní experimentov, hlavne v prípadoch keď je potrebné mať pokrokové sofistikované nástroje. Ďalší problém, s ktorým sa stretávame, je nedostatok kvalitných lektorov. Síce v súčasnosti už existujú online kurzy, ktoré poskytujú aj video ukážky, ale to tento problém rieši len čiastočne. Vždy bolo výzvou vykonávanie spoločných experimentov viacerými inštitúciami súčasne a zároveň aj zdieľanie nákladov na prostriedky. So súčasnými možnosťami internetu a počítačových technológií už tieto obmedzenia nemusia trápiť študentov ani výskumníkov. Internet umožnil to, že experimenty môžu byť štruktúrované tak, aby boli ovládané a vizualizované na diaľku. Práve to by mohlo pomôcť v učení základných ale aj pokročilých konceptov prostredníctvom vzdialeného experimentovania. V súčasnosti veľa vybavenia už poskytuje rozhranie pre pripojenie počítača a spracovanie dát z neho. Preto je možné navrhnúť experimenty, ktoré pomôžu študentom pri učení. Experimentovanie cez internet umožňuje využívanie zdrojov, znalostí, softvéru a dát z internetu na rozdiel od fyzických experimentov, ktoré by vznikali súčasne na rôznych miestach [31].

V tejto práci sa budeme zaoberať tvorbou virtuálneho laboratória (ďalej len VL). Predtým, ako si popíšeme detailné fungovanie technológií pre vytvorenie VL, si musíme vysvetliť, čo považujeme za VL, pochopiť aké hodnoty nám môže priniesť, ale samozrejme aj tie, ktoré nemôže. Vo všeobecnosti môžeme povedať, že VL je počítačový program, kde študenti sú v interakcii s experimentom prostredníctvom počítača ako na obrázku 1.



Obrázok 1: Rozdiel medzi osobne a vzdialene riadeným experimentom.

Typický príklad je simulácia experimentu, kedy je študent v interakcii s naprogramovaným rozhraním. Ďalšia možnosť je diaľkovo ovládaný experiment, kde študent je v interakcii s reálnym zariadením cez počítačové rozhranie, napriek tomu že sa nenachádza pri ňom. Keď vylúčime druhú variantu, tak si môžeme vytvoriť definíciu nasledovne: *"Virtuálnym laboratóriom voláme to, keď je študent v interakcii s experimentom, ktorý je od neho fyzicky vzdialený alebo nemá na pozadí žiadnu fyzickú realitu"* [7].

Po vysvetlení, čo je VL sa pozrieme na výhody, ktoré nám môže priniesť. Sú popísané v bodoch v tabuľke 1. Človek často počíta medzi výhody to, že môže nahradiť fyzické laboratórium. Lenže to medzi výhody nepatrí. Nie je možné nahradiť skúsenosti z fyzickej práce so zariadením VL aj keď je to lepšie ako žiadna skúsenosť. VL by nemalo byť vnímané tak, že poskytuje maximálnu možnú skúsenosť.

Typ laboratória	Výhody	Nevýhody
Fyzické	realistické dáta interakcia s reálnym zariadením lepšia spolupráca interakcia s lektorom	obmedzenia na čas a mieste potrebné plánovanie prístupu nákladnosť experimentu potrebný lektor
Virtuálne	dobré pre vysvetlenie konceptu bez obmedzenia na čas a miesto interaktívne médium nízke náklady	idealizované dáta nedostatok spolupráce bez interakcie s reálnym zariadením
Vzdialené	interakcia s reálnym zariadením kalibrácia realistické dáta bez obmedzenia na čas a miesto stredné náklady	"virtuálna" prítomnosť v laboratóriu

Tabuľka 1: Porovnanie fyzických, virtuálnych a vzdialených laboratórií.

2.1 Prehľad existujúcich virtuálnych laboratórií

V čase písania tohto dokumentu existuje množstvo rôznych virtuálnych/vzdialených laboratórií, ktoré sú používané zahraničnými školami pre výučbu alebo výskum. V práci [9] je zoznam veľmi často používaných laboratórií, ktoré sú prístupné cez internet. Porovnanie funkcionality a využitých technológií je možné vidieť v tabuľke 2.

Názov	Klient	Server	Prevedenie
Weblab-DEUSTO	AJAX, Flash, Java applets, LabVIEW, Remote panel	Web services, Python, LabVIEW, Java, .NET, C, C++	Xilinx-VHDL, LabView
NCSLab	AJAX, Flash	PHP	Matlab, Simulink
ACT	HTML, Java Applets	PHP	Matlab, Simulink
LabShare Sahara	AJAX, Java applets	Web services, Java	Java
iLab	HTML, ActiveX, Java applets	Web services, .NET	LabVIEW
RECOLAB	HTML	PHP	Matlab, Simulink
SLD	AJAX, HTML	Web services, PHP	Matlab, Simulink

Tabuľka 2: Porovnanie virtuálnych laboratórií vytvorených mimo FEI STU.

Následne som preskúmal možnosti existujúcich riešení a vložil do tabuľky 3, ktoré boli vytvorené na Fakulte elektrotechniky a informatiky STU [4][1][8][32][30].

Rok vypracovania	Autor	Prevedenie	Spôsob komunikácie	Klient	Server
2011	Roman FARKAŠ	Matlab Simulink Reálna sústava	JMI, sockets	Java	Java
2012	Tibor BORKA	Matlab Simulink Reálna sústava	WCF	.NET, WPF	.NET
2014	Michal KUNDRÁT	Matlab Simulink	JMI, SOAP	HTML, JS	Tomcat, Java, JSF, EJB3, MySQL
2014	Tomáš ČERVENÝ	Matlab Simulink	JMI, HTTP	Mobilné HTML, JS	Jetty, Java
2015	Štefan VARGA	Matlab Simulink	COM, HTTP	HTML, JS	PHP, .NET

Tabuľka 3: Porovnanie virtuálnych laboratórií vytvorených na FEI STU.

2.1.1 Nevýhody existujúcich riešení

Pri tvorbe softwarového systému, či už všeobecne, alebo v našom prípade virtuálneho laboratória je vhodné preskúmať zo začiatku možnosti existujúcich riešení. Robí sa to z dôvodu vyvarovania rôznym chybám, ktoré môžu nastať pri návrhu, prípadne overenie technológií, ktoré boli použité a časom už zastarali. V súčasnosti je vývoj nových technológií neskutočne rýchly. Takúto analýzu existujúcich riešení sme urobili v predchádzajúcej sekcii. Naša téma je zameraná na vytvorenie multiplatformového riešenia, kde nie je možné využiť WCF ani COM technológie ako v predchádzajúcich riešeniach. JMI je zase vhodné len pre riešenie, kde sa využíva Java. Pre server nie je možné využiť technológie LabVIEW, .NET (momentálne je už vo vývoji multiplatformová verzia). Čo sa týka klientských riešení tak Flash, ActiveX, Java applets už nie sú podporované v prehliadačoch, taktiež ich nie je vhodné použiť.

2.2 Komponenty virtuálneho laboratória

Počet existujúcich laboratórií je veľký, ale väčšinou nie je možné zaručiť kompatibilitu, pretože tu neexistuje žiadny štandard. Každopádne vždy je možné identifikovať základné

komponenty, ktoré tieto VL využívajú. Niektoré z nich môžu byť dokonca využité viac krát [22].

Komponenty:

- samotný experiment,
- zariadenie umožňujúce kontrolu experimentu a získavanie hodnôt z neho,
- laboratórny server, ktorý zabezpečí kontrolu, monitorovanie a spracovanie dát z experimentu,
- server, ktorý zabezpečí prepojenie medzi vzdialenými užívateľmi a laboratórneho servera, zvyčajne prostredníctvom internetu,
- webová kamera pripojená k serveru, ktorá môže byť použitá pre vzdialeného používateľa ako vizuálna a zvuková spätná väzba o stave experimentu,
- nástroje umožňujúce viac užívateľské audio, video a chat komunikáciu.
- klientský software, pripojiteľné k experimentu.

Dôležité je uvedomiť si, ktoré z týchto komponentov chceme využiť, pretože na vytvorenie laboratória nie vždy potrebujeme všetky. Prípadne môžeme využiť aj iné, ktoré sa nám dokonale hodia na danú úlohu. Niekedy sa používa napr. aj databázový server ak chceme experimenty ukladať a spracovávať neskôr. Tak isto je potrebné uvedomiť si, aký typ VL chceme vytvoriť. Určite budú rozdiely pri návrhu jednouchádzačského VL na rozdiel od viacouchádzačského dokonca s viacerými experimentami súčasne. Treba myslieť na to, ako vhodne vyriešiť škálovateľnosť, možné problémy s bezpečnosťou, viacouchádzačský prístup, ostatné problémy prístupnosti a podobne.

3 Použité technológie

V predchádzajúcich kapitolách sme popísali ciele, ktoré chceme vlastne dosiahnuť. Ďalej sme analyzovali možnosti virtuálnych laboratórií a porovnali už s existujúcimi riešeniami. V tejto sekcii budú v krátkosti rozpísané použité technológie. Je samozrejmé, že nie je možné ku každej popísať všetky jej možnosti, ale budeme sa venovať hlavne tým, ktoré plánujeme využiť aj v implementácií.

3.1 MATLAB R2015b

Milióny inžinierov a vedcov na celom svete používajú MATLAB na analýzu a návrh systémov a produktov, ktoré menia náš svet. MATLAB sa používa v automobilových systémoch, vesmírnych staniciach, smart sieti, mobilých sieťach LTE alebo aj v škole na štúdium. Ďalej sa používa pre strojové učenie, spracovanie signálu, spracovanie obrazu, počítačové videnie, komunikácie, finančníctvo, riadenie, robotika a mnoho ďalších využití. Celá platforma MATLAB je optimalizovaná pre riešenie inžinierskych a vedeckých problémov. Jazyk MATLABu je založený na práci s maticami. Je považovaný za najprirodzenejší spôsob ako počítať matematické úlohy. Pomocou integrovanej grafickej knižnice je možné vizualizovať a získať výsledky z dát. MATLAB integruje aj množstvo toolboxov, ktoré pomáhajú priamo začať s algoritmami, ktoré potrebujeme pre našu doménu [10].

3.1.1 Simulink

Matlab obsahuje viacero integrovaných nástrojov a jeden z nich je aj Simulink. Je to grafické rozhranie, v ktorom je možné modelovať, simulovať a potom aj analyzovať dynamické systémy. Jeho hlavné rozhranie je grafické plátno, kde môžeme spájať jednotlivé bloky do diagramu. Simulink vie úzko spolupracovať s matlabom, dokonca môže byť odtiaľ aj skriptovaný, resp. počiatočná inicializácia hodnôt. Matlab a Simulink sú vlastne dve prostredia integrované do jedného software. Čiže je možné simulovať a analyzovať náš model v každom kroku simulácie v oboch prostrediach. Simulink väčšinou spúšťame priamo z Matlabu.

3.1.2 Komunikácia medzi Matlabom a Node.js

Simulácia dynamickej sústavy, ktorá sa spustí v Simulinku môže odosielať výsledné dáta do Matlab workspace. Odtiaľ ich budeme chcieť posilať do Node.js na ďalšie spracovanie. V Matlabe existuje viacero možností získania dát z workspace.

COM je prvá z technologických možností. COM bolo vytvorené spoločnosťou Microsoft, teda problémom tohto riešenia je obmedzené len na Windows platformu. Použí-

vajú sa na prepojenie rôznych aplikácií podporujúcich túto technológiu. Tieto objekty môžu byť vytvorené pomocou rôznych programovacích jazykov ako napr. C++ alebo Java [16]. Kým idea COM je celkom jasná, terminológia až toľko nie je.

COM object je softwarový komponent, ktorý zodpovedá Component Object Model. COM vnucuje zapuzdrenie objektu a tým predchádza pred priamym prístupom do dát a implementácie. COM objekt poskytne rozhranie, ktoré obsahuje premenné, metódy a udalosti.

COM client je program, ktorý používa COM objekty. COM objekty, ktoré poskytujú funkcionality pre použitie sú volané COM server.

COM server môže byť in-process alebo out-of-process. Príklad out-of-process servera je napríklad Microsoft Excel. Microsoft ActiveX control je typ in-process COM servera, ktorý vyžaduje kontajner. ActiveX zvyčajne poskytuje užívateľské rozhranie. Príkladom môže byť Microsoft Calendar control. Control container je aplikácia schopná poskytovať ActiveX prvky. Matlab figure okno alebo Simulink model sú tiež príklady control kontajnerov. Matlab môže byť použitý aj ako COM klient aj ako COM automation server [11].

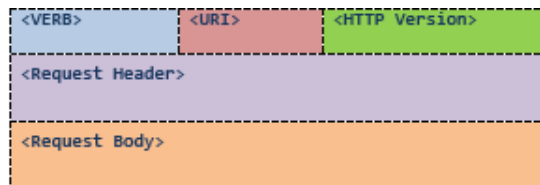
WebSockets je protokol poskytujúci obojsmernú komunikáciu cez jediné TCP spojenie. WebSocket bol vytvorený a použitý pre webové prehliadače a serveri, ale môže byť použitý akýmkoľvek klientom alebo serverom. Špecifikácia protokolu definuje **ws** a **wss** ako dve nové URI schémy, ktoré sú použité pre nešifrované a šifrované spojenia.

Nevýhodou tohto riešenia pre náš systém je ten, že nie je priamo implementovaný v Matlabe. Keďže pod Matlabom beží JVM, tak je možné implementovať WebSockets pomocou programovacieho jazyku Java a spraviť z toho knižnicu pre Matlab.

Narazili sme na jednu knižnicu vytvorenú týmto spôsobom na odkaze <https://bitbucket.org/kvasnica/wsclient/wiki/Home>. Jej nevýhodou je, že by bolo potrebné inštalovať externé závislosti, ktoré by sa mohli stať nekompatibilné pri zmene Matlabu na vyššiu verziu. Najprv je potrebný **tbxmanager**, ako manažér balíkov a následne pomocou neho inštalácia už konkrétnych balíkov **wsclient**, **matwebsocks**, **eventcollector**.

RESTful web service patrí medzi modernejšie možnosti komunikácie Matlabu s vonkajším svetom rovnako ako WebSockets. REST je softwarový architektonický štýl, pomocou ktorého vieme posilať a získavať dáta zo serveru. REST komunikuje pomocou HTTP/HTTPS protokolu a zväčša sa používa na CRUD aplikácie, čiže tam kde chceme robiť CREATE, READ, UPDATE, DELETE operácie. Dáta je možné vymieňať medzi klientom a serverom cez JSON alebo XML správ. Pre jednoduchšie projekty sa používajú

skôr JSON, hlavne ak sa majú spracovávať JavaScriptom. Výhodou RESTful v tomto riešení je to, že je priamo implementované v Matlabe a nie je potrebné inštalovať ďalšie knižnice a toolboxy.



Obrázok 2: HTTP request.

Na obrázku 2 je štruktúra HTTP requestu. V položke **<VERB>** sa nachádza jedna z HTTP metód GET, PUT, POST, DELETE, OPTIONS, ...

<URI> slúži na učenie zdroju, nad ktorým sa bude vykonávať operácia a je tam uložený jeho odkaz.

<HTTP version> je verzia HTTP, vo všeobecnosti to bude "HTTP v1.1" ale môže byť aj "HTTP v2.0" v novších systémoch.

<Request Header> obsahuje metadáta ako kolekciu párov "key" : "value" v hlavičke. Tieto nastavenia obsahujú informáciu o správe a jej odosielateľovi ako typ klienta, aký formát podporuje klient, formát správy tela, nastavenia cache pre odpoveď a mnohé ďalšie informácie.

<Request Body> je aktuálny obsah správy. V RESTful službách je toto miesto, kde sa nachádza obsah správy, ktorý sa vymieňa medzi klientom a serverom. V tejto časti nie sú žiadne tagy ani značky pre učenie začiatku alebo konca správy [29].

V ukážke POST requestu na obrázku 3 zasielame serveru JSON s jednoduchým párom.

```
POST /test HTTP/1.1
Host: localhost:3000
Content-Type: application/json
Content-Length: 120
Cache-Control: no-cache
Postman-Token: c9903c6e-42b8-0bb0-37fb-b311a6b40a9b
{
  "key": "value"
}
```

Obrázok 3: HTTP POST request.

A zase na obrázku 4 v prípade GET requestu vidíme, že server vráti zdroje, ktoré

server vracia, alebo v našom prípade celý obsah HTML stránky na ktorú bola požiadavka vytvorená.

```
HTTP/1.1 200 OK
X-Powered-By: Express
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Fri, 06 May 2016 20:09:32 GMT
ETag: W/"51d-15487b08060"
Content-Type: text/html; charset=UTF-8
Content-Length: 1309
Date: Mon, 09 May 2016 18:34:53 GMT
Connection: keep-alive

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Login</title>
```

Obrázok 4: HTTP GET request.

Teraz, keď sme si vysvetlili v stručnosti ako fungujú RESTful služby, tak sa dostávame k tomu, ako ich je možné využiť v Matlabe. Matlab poskytuje viacero funkcií na prácu s REST ako `weboptions`, `webread`, `webwrite`.

Objekt `weboptions` slúži na špecifikáciu parametrov pre RESTful službu. V Matlabe sa volá príkazom `options = weboptions` alebo `options = weboptions(Name,Value)` pričom `Name` je názov parametra, ktorý chceme nastaviť a `Value` jeho hodnota. Je možné nastaviť tieto parametre: `CharacterEncoding`, `UserAgent`, `Timeout`, `Username`, `Password`, `KeyName`, `KeyValue`, `ContentType`, `ContentReader`, `MediaType`, `RequestMethod`, `ArrayFormat`. Ak chceme zobraziť heslo v objekte `weboptions`, tak na pozícií hesla síce budú hviezdičky, každopádne objekt ukladá heslo ako čistý text. V prípade keď zavoláme túto vlastnosť v Matlabe cez `options.Password` tak heslo bude viditeľné [13].

Objekt `webread` číta obsah z REST služby, na ktorú sme mu poskytli URL cestu a vráti obsah ako štruktúru do požadovanej premennej. Existujú tri najpoužívanejšie možnosti použitia. `data = webread(url)` kde parameter `url` je refazec, v ktorom sa nachádza odkaz na REST službu. `data = webread(url, QueryName1, QueryValue1,`

..., QueryNameN, QueryValueN) kde vložené parametre pridá do url volania. Posledná možnosť je `data = webread(___, options)`, kde je možné špecifikovať posledný objekt ako `weboptions` [14].

A posledný objekt `webwrite`. Tak ako v predchádzajúcom prípade obsahuje zhodné metódy s parametrami, len využitie funkcie je iné. `response = webwrite(url, PostName1, PostValue1, PostNameN, PostValueN)` zapíše obsah na špecifikovanú url a vráti response. `response = webwrite(url, data)` zapíše obsah na špecifikovanú url a vráti response. Vstupný parameter `data` špecifikuje obsah, ktorý je uložený ako pole. `response = webwrite(___, options)` nastaví `weboptions` objekt, zapíše obsah na špecifikovanú url a vráti response.

3.2 Node.js

Na stránke platformy (<http://www.nodejs.org>) je definovaný Node ako "platforma založená na JavaScript runtime, ktorý je v Chrome pre jednoduchú tvorbu rýchlych a škálovateľných sieťových aplikácií. Node.js používa udalosťami riadený, neblokujúci I/O model, ktorý ho robí nenáročný a efektívny, perfektný pre real-time aplikácie." V súčasnosti patrí medzi najpopulárnejšie JavaScript technológie.

Vďaka jeho súčasnej stabilite ho používa v produkcii mnoho svetových firiem, napríklad eBay, GoDaddy, Microsoft, PayPal, Uber, Yahoo...

3.2.1 História

Vytvoril ho Ryan Dahl v roku 2009 a bol dostupný iba pre Linux. Vývoj a údržba bola vedená jeho zakladateľom a neskôr sponzorovaná firmou Joyent. Node.js sa skladá z JavaScriptového engine V8 od Google, event loop a nízkoúrovňového I/O API. V roku 2011 bol vytvorený správca balíkov pre Node.js zvaný NPM. Umožňuje programátorom publikovať, zdieľať zdrojový kód modulov a bol navrhnutý tak, aby zjednodušoval inštaláciu, aktualizáciu alebo odinštalovanie modulov. Neskôr v júni 2011, Microsoft a Joyent spolupracovali na implementácii natívnej verzii Node.js pre Windows. V roku 2014 vznikli nezhody pri vývoji, tak programátor Fedor Indutny spravil fork Node.js a vytvoril io.js. Na rozdiel od Node.js autorov, chcel udržiavať io.js aktuálny súčasne s poslednou verziou V8 JavaScript engineu.

Po dohode bola vytvorená Node.js Foundation, ktorá zastrešila vývoj, a spojila Node.js v0.12 a ioJS v3.3 do Node.js 4.0, aby znovu spojila roztrieštenú komunitu. Táto verzia priniesla ES6 novinky z V8 do Node.js a súčasne bola vytvorená aj LTS verzia vhodná pre produkčné nasadenie, ktorá má dlhší vývojový cyklus a príjma len opravy chýb [18].

3.2.2 Architektúra

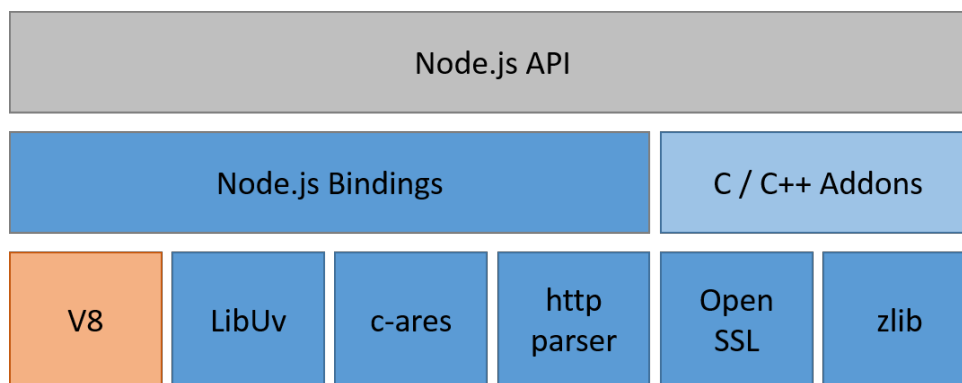
Tak ako pri iných platformách aj Node.js má svoju architektúru a v tejto sekcii si popíšeme jeho kľúčové vlastnosti. Síce ich nebudeme priamo používať, ale je dobré o nich vedieť.

Má **asynchronné a udalosťami riadené** API, čo znamená, že neblokuje vykonávanie nasledujúcich volaní. V podstate ide o to, že server založený na Node.js nikdy nečaká, kým volaná služba/funkcia vráti dáta. Server sa presunie na ďalšie volanie a pomocou notifikačného mechanizmu udalostí získa odpoveď z prechádzajúceho volania, keď bude ukončené a vyzdvihne jeho výsledok.

Vďaka udalostnej slučke je **jednovláknový a vysoko škálovateľný**. Udalostný mechanizmus pomôže serveru vrátiť odpoveď tak, aby nebol blokovaný a tak robí server vysoko škálovateľný oproti tradičným serverovým riešeniam, ktoré vytvárajú limitovaný počet vlákien na spracovávanie požiadaviek. Node.js spustí jednovláknový program, ktorý môže poskytnúť službu omnoho väčšiemu počtu požiadaviek ako tradičný server Apache httpd.

Aplikácie sú **bez vyrovnávajúcej pamäte**, čiže jednoducho posielajú údaje na výstup v malých blokoch.

Ako sme už spomenuli, tak platforma Node.js sa skladá z viacerých častí. Bolo by možné ju rozdeliť ešte na menšie časti, ale z nášho pohľadu stačí pre ilustráciu zobrazenie na obrázku 5.



Obrázok 5: Architektúra Node.js platformy.

Na vrchole obrázku máme základné Node.js API. Je napísané v JavaScripte a je priamo dostupné programátorom, pre využitie v ich aplikáciach. Pod týmto základným API je knižnica, ktorá viažš C/C++ s JavaScriptom. Node.js tiež poskytuje doplnky (addons), čo sú dynamicky linkované zdieľané objekty. Tie sa viažu na C/C++ knižnice. To

znamená, že môžeme využiť skoro akúkoľvek C/C++ knižnicu a vytvoriť z nej doplnok, ktorý použijeme v Node.js.

Pod týmto všetkým máme už len natívne knižnice vytvorené v C/C++:

V8 je open source JavaScript engine, ktorý bol vytvorený pre internetový prehliadač Google Chrome. Je napísaný v C++ a je možné ho spustiť samostatne alebo v ktorejkoľvek C++ aplikácii. V podstate kompiluje JavaScript kód do natívneho strojového kódu, namiesto toho, aby bol interpretovaný.

Libuv je multiplatformová podporná knižnica so zameraním na asynchrónne I/O operácie. Zo začiatku Node.js začal používať **libuv** ako abstrakčnú vrstvu pre **libev** a **libio**, ale neskôr sa **libuv** stala robustnejšou a nahradila túto funkcionality, aby sa mohla stať multiplatformovou. Keď V8 spravuje vykonávanie JavaScriptu, tak **libuv** spravuje udalostnú slučku (event loop) a asynchrónne I/O operácie. V tomto zozname sú všetky možnosti **libuv**:

- plnohodnotná udalostná slučka, ktorú tvorí epoll, kqueue, IOCP a udalostné porty,
- asynchrónne TCP a UDP sockety,
- asynchrónne DNS,
- asynchrónne operácie nad súbormi a súborovým systémom,
- udalosti nad súborovým systémom,
- preklad ANSI kódov kontrolovaných cez TTY,
- IPC so zdieľaním socketu s využitím Unix socketov alebo pomenované kanály (vo Windows),
- detské procesy,
- vlákna a synchronizácia,
- riadenie signálov,
- hodiny s presným časovaním (high resolution clock).

c-ares je C knižnica pre asynchrónne DNS žiadosti vrátane prekladania názvov. Je určená pre aplikácie, ktoré potrebujú vykonávať dotazy na DNS bez blokovania, alebo ak potrebujú vykonať niekoľko dotazov paralelne.

http_parser je parser pre požiadavky a odpovede HTTP napísaný v C. Nerobí žiadne systémové volania ani alokácie, neukladá dáta do vyrovnávacej pamäte a môže byť zrušený okamžite. Jeho hlavné vlastnosti sú:

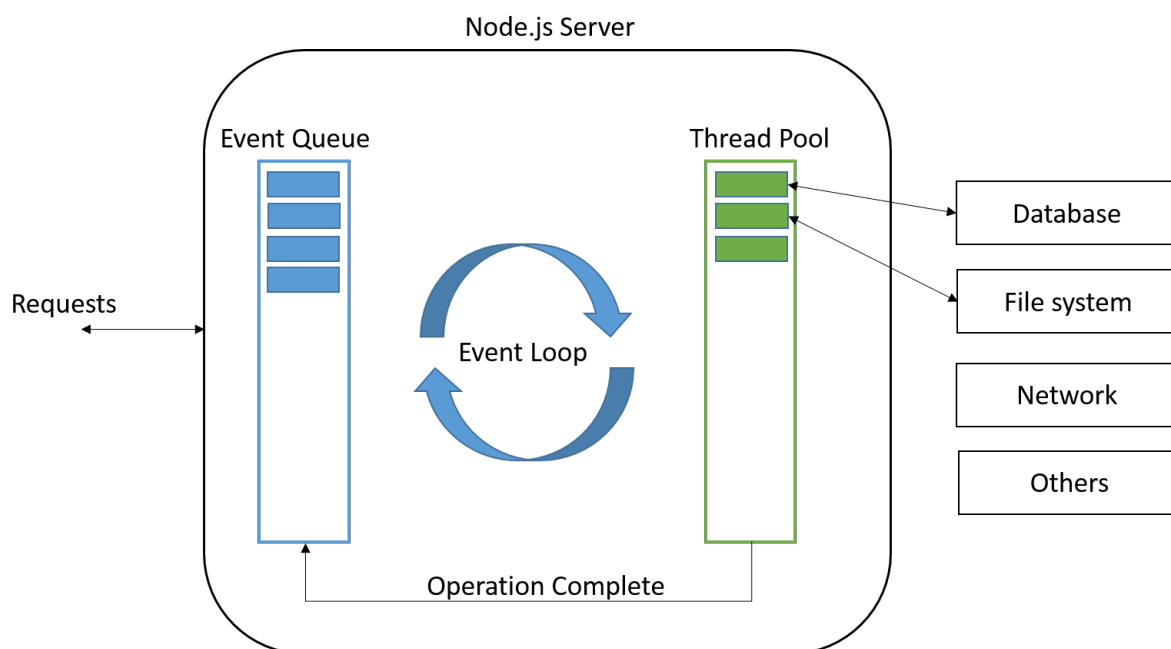
- nemá žiadne závislosti,
- udržiava trvalý stream
- dekodovanie blokového kódovania,
- chráni buffer proti útokom.

OpenSSL je open source implementácia SSL v2/v3 a TLS v1 protokolov ako aj kryptografická knižnica pre všeobecné účely. Je založená na SSLeay knižnici. Tá poskytuje všetky potrebné kryptografické metódy ako je hash, hmac, cipher, decipher, sign a verify metódy.

Zlib je všeobecná knižnica na kompresiu dát napísaná v C [19].

3.2.3 Event loop

Udalostná slučka dáva Node.js možnosť zvládnuť veľké množstvo súčasných požiadaviek aj keď je spustený v "jednom vlákne". V každej udalostne riadenej aplikácii je vo všeobecnosti hlavná slučka, ktorá počúva a čaká na udalosti a keď udalosť je zaregistrovaná, tak sa zavolá callback funkcia. Na obrázku 6 je zjednodušený pohľad na to, ako to funguje vrámci Node.js.



Obrázok 6: Udalostná slučka v Node.js.

Udalostná slučka jednoducho prechádza cez frontu čo je vlastne zoznam udalostí a callbackov vykonaných operácií. Všetky I/O operácie sú vykonané asynchrónne vláknami vo "vláknovom stacku" (thread pool). Tu zohráva dôležitú rolu libuv. Ak nejaká položka vyžaduje I/O operáciu, tak udalostná slučka jednoducho prenechá operáciu do vláknového stacku. Udalostná slučka pokračuje vo vykonávaní položiek v udalostnej fronte. Keď je I/O operácia hotová, tak callback je zaradený na spracovanie. Udalostná slučka vykoná callback a poskytne požadované výstupy. A takto sa celý proces opakuje [20].

3.2.4 Možnosti a využitie

Na základe popísaných základných častí Node.js v predchádzajúcich sekciách si ukážeme možnosti využitia, resp. výhody a nevýhody [19].

Výhody

- asynchrónne I/O - vhodné pre webové a sieťové aplikácie,
- rôzne možnosti škálovania,
- programovací jazyk je JavaScript, čiže rovnaký jazyk pre backend aj frontend aplikácie,
- rýchly prechod od Javy, .NET stačí len zmeniť myslenie na asynchrónne,

- veľmi aktívna komunita, ktorá zdieľa množstvo kódu na verejných repozitároch ako github,
- rýchlo rastúca NPM komunita s množstvom modulov pripravených na použitie.

Nevýhody

- veľmi neefektívne pre úlohy náročné na CPU, ako generovanie reportov, analýzy, výpočty...
- použitím udalosťami riadenej metodológie bez pochopenia prístupu môže viesť k nevhodne napísaným kódom (napr. "callback hell"),
- neexistuje toľko štandardných knižníc ako pri Java, .NET platforme ako sú napr. XML parsery, alebo zložitejšie dátové štruktúry.

Ako teda vidíme, Node.js nie je vhodný na všetky úlohy, ale vždy záleží od konkrétnej potreby. Čiže ak potrebujeme streaming alebo rýchly upload súborov, real-time získavanie údajov, single page aplikácie, websockety tak je na takéto úlohy veľmi vhodný. Vo všeobecnosti všade kde sa používajú I/O operácie, tak vie zvládnuť veľké množstvo súčasných spojení [24].

3.3 Node Package Manager

Node.js po nainštalovaní obsahuje aj **NPM**. NPM je program, ktorý sa spúšťa z príkazového riadku, pomocou ktorého vieme sťahovať moduly z centrálného repozitára <https://www.npmjs.org>. Rovnako je možné aj vytvoriť vlastný modul a uložiť ho do repozitára.

Každý modul by mal mať vlastný adresár, ktorý tiež obsahuje súbor s metadátami volaný **package.json**. V tomto súbore musia byť nastavené vlastnosti a ich hodnoty ako na obrázku 7, čiže: **name** a **version** [23].

```
{
  "name": "my-awesome-nodejs-module",
  "version": "0.0.1"
}
```

Obrázok 7: Potrebné vlastnosti nastavené v package.json.

Ďalšia dôležitá vec je, že pri **version** musíme používať sémantické verziovanie (<http://semver.org/>), kde pri verzii 1.2.3 je prvá major verzia, druhá minor a tretia len oprava chýb.

3.3.1 Použitie modulov

Vo všeobecnosti existujú tri možnosti ako použiť moduly, ktoré sú už publikované na npmjs.org. Všetky tri možnosti si vyžadujú použitie Node manažéra na prácu s balíkmi: [23]

- môžeme nainštalovať špecifický modul manuálne tak, že sa presunieme do požadovaného priečinka a zavoláme v termináli `npm install nazov-modulu`. Správca balíkov automaticky nainštaluje poslednú verziu modulu a vloží ju do priečinka `node_modules`. V prípade použitia, nevoláme celú cestu, ale len `require(nazov-modulu)`.
- inštalácia modulu globálne pre celý systém s použitím `-g` prepínača v príkaze. `npm install nazov-modulu -g`. Toto použitie slúži skôr na inštaláciu vývojových nástrojov, ktorých verzia sa tak často nemení a nie na bežné moduly.
- posledná možnosť je príkaz `npm install` v priečinku, kde sa nachádza už spomínaný súbor `package.json` a má v sebe vlastnosť `dependencies` ako na obrázku 8.

```
{
  "name": "another-module",
  "version": "0.0.1",
  "dependencies": {
    "my-awesome-nodejs-module": "0.0.1"
  }
}
```

Obrázok 8: Dependencies vlastnosť v package.json.

3.3.2 Vstavané moduly

Node.js je považovaný za technológiu, pomocou ktorej môžeme programovať serverové aplikácie. Často potrebujeme robiť rôzne úlohy. V Node.js existujú veľmi nápomocné moduly, ktoré môžeme využiť [23].

3.3.3 Vytvorenie web servera pomocou HTTP modulu

Modul `http` patrí asi medzi najviac používané, pretože vďaka nemu spustíme web server na konkrétnom porte - obrázok 9.

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(9000, '127.0.0.1');
console.log('Server running at http://127.0.0.1:9000/');
```

Obrázok 9: Dependencies vlastnosť v package.json.

Máme dostupnú metódu `createServer()`, ktorá vracia nový web server objekt. Často hneď za tým voláme aj `listen()` metódu pre nastavenie portu. Ak je potrebné tak pomocou funkcie `close()` zastavíme prijímanie nových požiadaviek. Telo callback funkcie, ktorú používame, má vždy `request` a `response` objekty. Prvý objekt sa používa na získanie informácií o prichádzajúcich požiadavkách ako napr. parametre z GET a POST [23].

3.4 Express.js

Express.js patrí medzi minimalistické a flexibilné Node.js webové frameworky. Poskytuje robustné možnosti pre vývoj webových a mobilných aplikácií. Uľahčuje vývoj webových aplikácií založených na Node.js [26].

Medzi jeho hlavné možnosti patrí:

- možnosť nastaviť middleware funkciu pre odpovede na HTTP požiadavky,
- definuje smerovaciu tabuľku, ktorá je použitá na vykonanie požiadaviek založených na HTTP,
- umožňuje dynamicky vykreslovať HTML stránky, tak aby bolo možné vkladať argumenty do šablóny.

Vo väčšine prípadov web aplikácií s použitím Expressu inštalujeme aj **body-parser**, ktorý slúži ako middleware pre narábanie s JSON, Raw, Text a URL parametrami.

3.4.1 Web server

Následujúca ukážka na obrázku 10 je základná kostra Express aplikácie, ktorá spustí web server a počúva na porte 8081. Po príchode na stránku bude vždy odpovedať textom **Hello word!** [26].


```

var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World');
})

var server = app.listen(8081, function () {

  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)

})

```

Obrázok 10: Spustenie web servera na porte 8081.

Na spustenie tohto kódu je potrebné ho vložiť do súboru s koncovkou **app.js** a následne spustiť **node app.js** čoho výsledkom bude výstup v termináli **Example app listening at http://0.0.0.0:8081**.

3.4.2 Request a Response objekty

Ak webová aplikácia používa smerovanie tak obsahuje objekty request a response z Node.js, ktoré sú dostupné v callback funkcii. Na obrázku 11 je smerovanie na hlavnú stránku [26].

```

app.get('/', function (req, res) {
  // --
})

```

Obrázok 11: Objekt req a res ako parametre callback funkcie.

Request objekt reprezentuje HTTP request a obsahuje isté vlastnosti a funkcie. Ich zoznam je celkom dlhý a nebudeme ich potrebovať všetky. Ale niektoré stoja za zmienku ako **req.body**, obsahujúci páry "kľúč": "hodnota", ktoré boli vytvorené ako telo správy. **req.cookies** umožňuje získať cookies z klienta na server. **req.params** slúži na získanie parametrov z URL, keď používame dynamické volania ako napr. **/user/:id** tak **id** je parameter dostupný v **req.params.id**. Čo sa týka funkcií request objektu, tak tie zvyčajne nie sú používané [27].

Response objekt predstavuje HTTP response, ktorý Express odošle po prijatí GET prípadne POST požiadavky. Pri tomto popise zase nie sú pre nás zaujímavé vlastnosti, ale skôr funkcie. Pomocou **res.cookie(name, value [, options])** vieme nastaviť hodnotu

cookie a s `res.clearCookie(name [, options])` ju zase vymazať. Ďalej môžeme odoslať kód, aby klient vedel akým statusom skončila požiadavka. Používa sa na to funkcia `res.status(200).end()`, kde číselný kód sa určuje podľa štandardných HTTP status hodnôt. Užitočná je aj funkcia `res.json({ user: 'xstark' })` pre zaslanie odpovede vo formáte JSON. V prípade, že potrebujeme sa presmerovať na inú stránku, tak použijeme `res.redirect([status,] path)` funkciu [28].

3.4.3 Routing

Routing, teda smerovanie slúži na určenie ako bude aplikácia reagovať na požiadavku klienta, čo zahŕňa URI a špecifické metódy (GET, POST, PUT, DELETE, ...) HTTP požiadavky [26]. Každý **route** môže mať viacero callback funkcií, ktoré sa vykonajú ak je zavolaná daná route.

Definícia route má nasledujúcu štruktúru `app.METHOD(PATH, CALLBACK)`, kde:

- **app** je inštancia express,
- **METHOD** je jedna z dostupných HTTP metód,
- **PATH** je cesta na serveri,
- **CALLBACK** je funkcia, ktorá sa vykoná v prípade, že sa zhoduje cesta [2].

3.4.4 Middleware

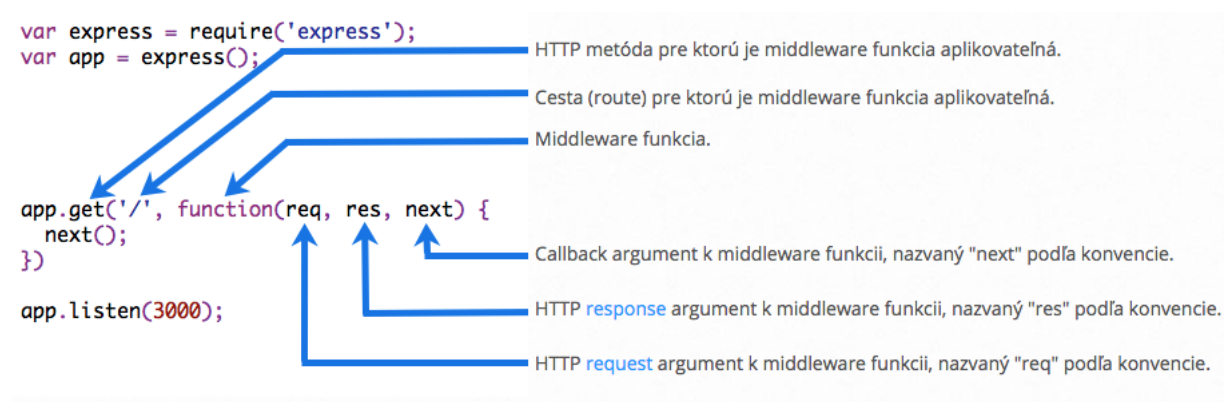
Funkcie označované middleware, sú také, ktoré majú prístup do request (**req**), response (**res**) objektu a následujúcej middleware funkcie. Následujúca funkcia sa bežne označuje názvom **next** [3].

Úlohy middleware funkcií:

- vykonávať akýkoľvek kód,
- robiť zmeny na **request** a **response** objektoch,
- ukončiť request-response cyklus,
- zavolať ďalšiu middleware funkciu v poradí.

Obrázok 12 popisuje vlastnosti middleware funkcie. V prípade, že middleware funkcia neukončí volanie, musí posunúť obsluhu ďalšej middleware funkcií zavolaním `next()`. Inak

sa request zasekne a klient bude stále čakať na odpoveď a po určitom omeškaní sa ukončí [3].



Obrázok 12: Vlastnosti middleware funkcie [3].

3.5 MongoDB

MongoDB je multiplatformová dokumentovo orientovaná databáza. Je klasifikovaná ako NoSQL databáza, čo znamená, že nepoužíva klasickú tabuľkovú štruktúru ako v relačnej databáze, ale objekty vo forme JSON dokumentov s dynamickými schémami (v MongoDB sa volá tento formát BSON). Vďaka formátu JSON dokumentov je možné ukladať dáta z rôznych aplikácií rýchlejšie a jednoduchšie [17].

```
{
  name: "sue",
  age: 26,
  status: "A",
  groups: [ "news", "sports" ]
}
```

Annotations with arrows pointing to the JSON fields:

- `name: "sue"`: field: value
- `age: 26`: field: value
- `status: "A"`: field: value
- `groups: ["news", "sports"]`: field: value

Obrázok 13: JSON dokument v MondoDB.

Obrázku 13 je príklad dokumentu, ktorý sa ukladá do MongoDB. V jednoduchých prípadoch ako tento nie je vidieť žiadny rozdiel oproti klasickému JSON.

3.6 Angular.js

Angular.js, bežne označovaný aj ako Angular je open source framework pre web vytvorený Slovákom Miškom Hevery do verzie 1.0. Neskôr vývoj a údržbu frameworku vzal pod seba Google. Bol vytvorený za účelom tvorby SPA, teda takých webových

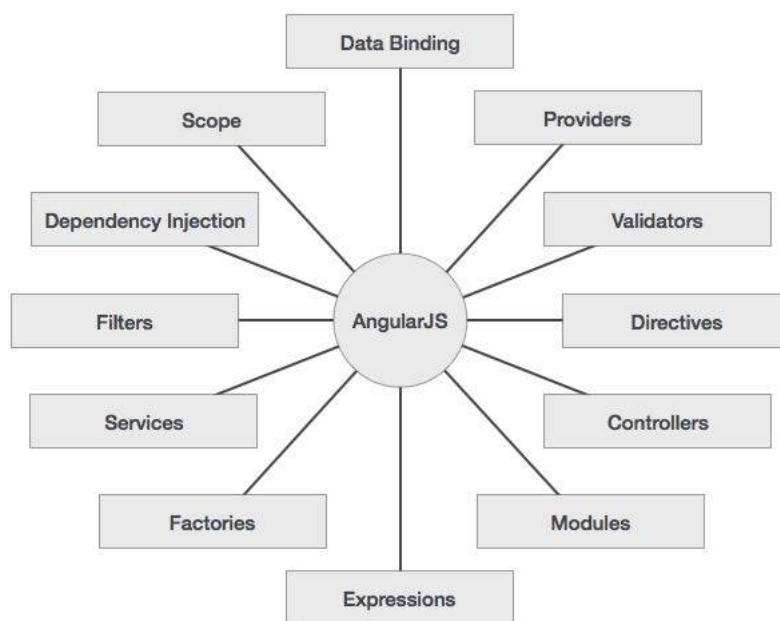
aplikácii, v ktorých je možné aktualizovať svoj obsah bez toho, aby si užívateľ toho vo väčšej miere všimol. Cieľom je zjednodušiť vývoj, ale aj testovanie front-end web aplikácií na báze MVC alebo MVVM architektúre.

Framework funguje tak, že pri prvom načítaní HTML stránky vloží k elementom vlastné atribúty. Angular interpretuje tieto atribúty ako direktívy a viaže ich vstupno-výstupné časti ako model, ktorý je reprezentovaný štandardnými premennými JavaScriptu. Hodnoty týchto premenných môžu byť manuálne nastavené v kóde, alebo získané zo statických (väčšinou uložených na súborovom systéme), dynamických (získaných z RESTful služby) JSON súborov.

Slúži ako front-end súčasť MEAN vývojarského stacku, čo je vlastne MongoDB databáza, Express.js web server framework, Angular.js a Node.js ako aplikačný web server [5][6].

3.6.1 Koncept

Angular ako framework sa skladá z viacerých častí, z ktorých každá má svoju úlohu. Na obrázku 14 sú zobrazené základné komponenty, z ktorých sa skladá. Nižšie si popíšeme hlavne tie, ktoré chceme využiť v našom softwarovom riešení [25].



Obrázok 14: Koncept frameworku Angular.js.

3.6.2 Directives

V jednoduchosti direktívy sú značky na DOM elemente (napr. atribút, element, komentár alebo CSS trieda), ktorá povie Angular HTML kompilátoru - `$compile`, aby priradil špecifické správanie na daný DOM element (napr. pomocou **event listenerov**),

alebo dokonca transformoval DOM element na svoje dieta. V šablónach sa spájajú informácie z modelu a kontroleru pre renderovanie dynamického obsahu, ktorý potom vidí užívateľ v prehliadači.

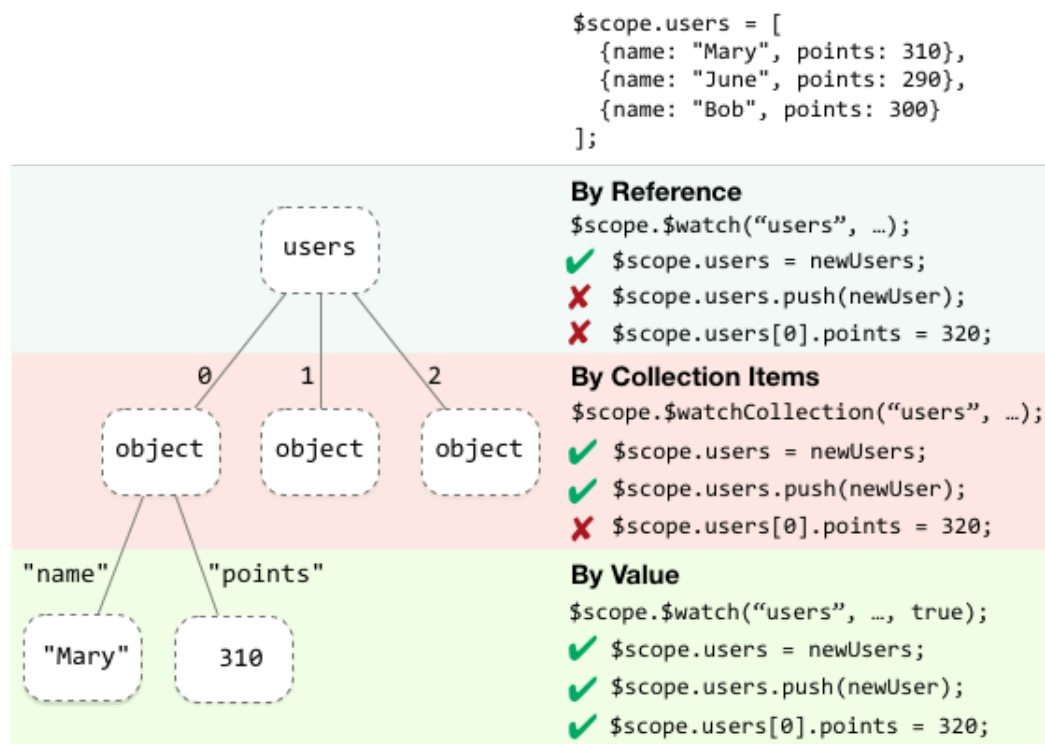
Angular je dodávaný so vstavanými direktívami, od ktorých sa očakáva, že budú využívané často. Napr. `ng-bind`, `ng-model`, `ng-class` [6].

3.6.3 Scope

Scope je objekt, ktorý odkazuje na model aplikácie. V tomto kontexte sa vykonávajú výrazy. Scopes sú usporiadané do hierarchickej štruktúry, ktorá napodobňuje DOM štruktúru aplikácie. V scope je možné sledovať model pomocou `$watch` a vykonávať udalosti s `$apply` cez celý systém do view. Slúži na previazanie aplikačného controlleru a viewu. Rovnako ako controller aj directive majú prístup k scope, ale nie navzájom medzi sebou. Vďaka tomu je controller izolovaný od directive a tak isto aj od DOM.

Každá aplikácia má práve jeden `$rootScope`, ale môže mať viacero detských `$scope`. Aplikácia môže mať viacero scopes, pretože niektoré directive môžu vytvoriť nový detský scope, ak to potrebujeme. Keď je nový scope vytvorený, tak je priradený ako detský k rodičovskému scope. Tento systém vytvára stromovú štruktúru paralelnú k DOM, v ktorej sú vytvorené [6].

Často používaná operácia pre zistenie zmien na objekte v scope sa volá **dirty checking** a preto by mala byť efektívna. V závislosti od potreby môže byť dirty checking využité týmito tromi stratégiami: referenciou na objekt, obsah poľa alebo na hodnoty objektu - obrázok 15.



Obrázok 15: Možnosti sledovania modelu v Angular.js.

Líšia sa v spôsobe sledovania zmeny a výkonnosnými rozdielmi:

- **podľa referencie:** `$scope.$watch(watchExpression, listener)` - detekcia zmeny, keď sa na sledovanú hodnotu nastaví nová. Ak sa zmení vnorený objekt alebo pole objektu ktorý sledujeme, zmeny vnútri nie su detekované. Toto je najúčinnnejšia stratégia.
- **na celej kolekci:** `$scope.$watchCollection(watchExpression, listener)` - detekcia zmien, ku ktorým dochádza vo vnútri poľa alebo objektu. Napr. keď sú položky pridané, odstránené alebo sa v nich zmenilo poriadie. Detekcia je plytká, čiže nesleduje vnorené polia. Sledovanie celého obsahu kolekcie je výkonovo náročnejšie ako sledovanie na referenciu, pretože treba uchovávať kópiu obsahu. Avšak, táto stratégia sa snaží minimalizovať množstvo potrebných kopírovaní.
- **všetky položky objektu:** `$scope.$watch(watchExpression, listener, true)` - detekcia akékoľvek zmeny v ľubovolnej vnorenej štruktúry. Táto stratégia má najväčšie možnosti detekcie zmien, ale za to aj výkonovo a pamäťovo najnáročnejšia. Je potrebné uchovávať kópiu celej vnorenej štruktúry a pri každej zmene, sa musí skopírovať do pamäte.

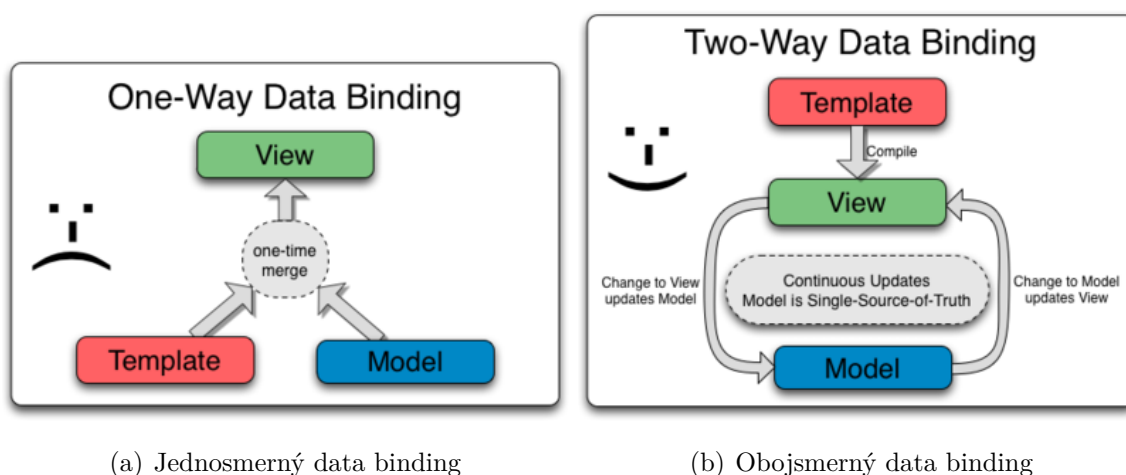
3.6.4 Expressions

Angular výrazy sú kúsky kódu v JavaScripte, ktoré sú umiestnené medzi dvojitémi kučeravými zátvorkami.

Príklad: `{{ textBinding }}`. Vyhodnotenie môže rovnako prebehnúť aj vo funkcii na kliknutie `ng-click="functionExpression()"`. Pre príklad pridávam zopár ďalších platných výrazov, ktoré sa často môžu používať: `{{ 1 + 2 }}`, `{{ a + b }}`, `{{ user.name }}`, `{{ items[index] }}` [6].

3.6.5 Data Binding

Data binding v Angulari funguje ako automatická synchronizácia dát medzi modelom a view. Ak sa zmení model, tak zmena sa automaticky prejaví aj do view, alebo aj naopak. Na obrázkoch môžeme vidieť pre porovnanie aký je rozdiel medzi jednosmerným a obojsmerným data bindingom.



(a) Jednosmerný data binding

(b) Obojsmerný data binding

Obrázok 16: Data binding v Angular.js.

Mnoho šablónovacích enginov nastavujú dáta len v jednom smere: spoja spolu šablónu a komponenty modelu do view. Keď sa dokončí spojenie, zmeny v modeli alebo jeho príslušných sekcií vo view sa neprejaví automaticky vo view. Horšie je, že zmeny ktoré užívateľ urobí vo view sa neprejaví do modelu. To znamená, že vývojár musí napísať kód, ktorý sústavne synchronizuje view s modelom a model s view.

Šablóny v Angulari fungujú inak. Šablóna (čo je neskompilovaný HTML kód spolu s ďalšími značkami a directívami) je najprv kompilovaná vo webovom prehliadači. Tento krok už produkuje živý view. Akékoľvek zmeny vo view sa okamžite prejaví v modeli, a akékoľvek zmeny v modeli sú poslané do view. Vďaka tomuto spôsobu môžeme o view hovoriť ako o okamžitej projekcii modelu.

Pretože view iba je projekcia modelu, tak controller je kompletne separovaný od view a nevedia o sebe. Vďaka tomu je testovanie jednoduchšie, pretože je možné testovať controller nezávisle od view [6].

3.6.6 Controller

V Angulari je controller definovaný funkciou konštruktoru v JavaScripte, ktorý je používaný pre rozšírenie Scope. Keď je controller pripojený k DOM cez `ng-controller` directívu, tak sa vytvorí inštancia objektu konštruktoru. Bude vytvorený detský Scope a je dostupný ako parameter do konštruktorovej funkcie controlleru ako `$scope`.

Obrázok 17 ukazuje ako je potrebné definovať controller v javascript súbore a jeho následné volanie v HTML šablóne.

```
JavaScript:

var myApp = angular.module('myApp',[]);

myApp.controller('LabController', ['$scope', function($scope) {
    $scope.name = 'StarkLab!';
}]);

HTML:

<div ng-controller="LabController">
    {{ name }}
</div>
```

Obrázok 17: Ukážka controlleru v Angular.js a jeho volanie na HTML elemente.

Vo všeobecnosti by controller nemal toho robiť veľa. Mal by obsahovať iba business logiku potrebnú pre konkrétny view. Najlepší spôsob ako zachovať controller čo najmenší, je zabalenie niektorej úlohy do service, ktorá tam nepatrí. Potom tieto úlohy zo services voláme v controlleri ako závislosť [6].

Controller je možné používať na:

- nastavenie počiatočného stavu v `$scope` objekte.
- pridanie vlastností a správania do `$scope` objektu.

Nepoužívať controller na:

- manipuláciu s DOM, pretože controller by mal obsahovať iba business logiku. Vložením akékoľvek prezentačnej logiky do controlleru vo veľkej miere ovplyvňuje jeho testovanie. Na manipuláciu slúžia directívy.

- formátovanie vstupu - použiť angular form controls.
- filtrovanie výstupu - použiť angular filter.
- zdieľanie kódu alebo stavu naprieč controllermi - použiť angular service.
- správa životného cyklu ostatných komponentov (napr vytváranie inštancií service).

3.6.7 Module

Module môžeme brať do úvahy ako kontajner pre rozličné časti našej aplikácie - **controller**, **service**, **filter**, **directive**... Každý module môže obsahovať zoznam iných modulov ako svoju závislosť. Keď závisíme na nejakom module, tak tento modul musí byť načítaný ešte pred našim modulom. Toto poriadie môžeme manuálne nastaviť v konfigurácii modulov. Každý modul môže byť načítaný iba raz aj keď na ňom závisí viacero modulov [6].

Pri programovaní modulov musíme dávať pozor na to, aby sme si neprepísali už existujúci v pamäti. Postup vytvorenia na obrázku 18.

```
var myModule = angular.module('myModule', []);

// pridanie directive a sluzby do modulu
myModule.service('myService', ...);
myModule.directive('myDirective', ...);

// toto volanie prepise uz vytvorene myService
// a myDirective vytvorenim noveho modulu
var myModule = angular.module('myModule', []);

// vyhodi chybu pretoze volame myOtherModule,
// ktory este nebol definovany
var myModule = angular.module('myOtherModule');
```

Obrázok 18: Vytvorenie modulov a pridanie ich funkcionality.

3.6.8 Service

Angular service je nahraditeľný objekt, ktorý je možné použiť v aplikácii pomocou dependency injection. Services môžeme použiť pre lepšiu organizáciu a zdieľanie kódu naprieč celej aplikácie. Framework poskytuje mnoho užitočných services, ako napr. **\$http**, ale väčšina aplikácií má potrebu si vytvoriť vlastné [6]. Poznáme dva typy services:

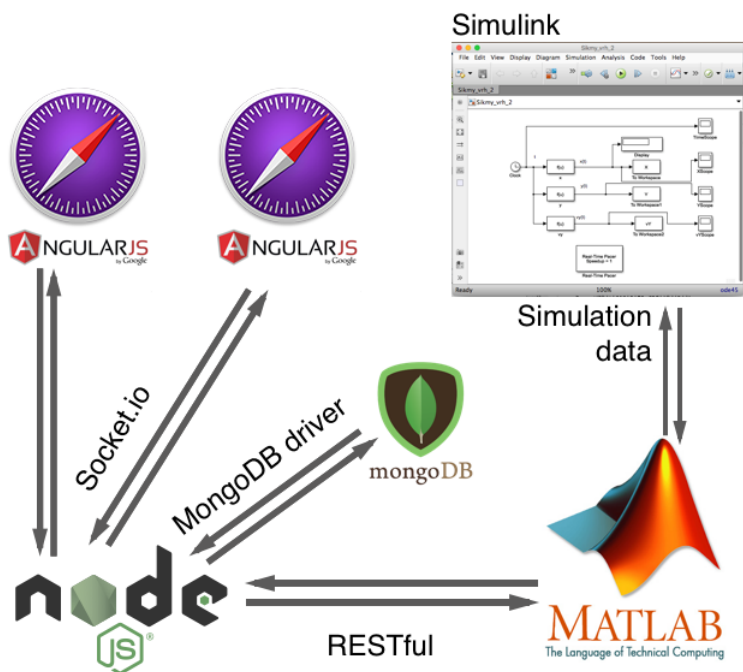
- lazily instantiated - angular vytvorí inštanciu service iba v prípade, že na nej závisí komponent/modul,
- singleton - každý komponent závisiaci na service získa referenciu na jedinú inštanciu generovanú z service factory.

4 Návrh a implementácia StarkLab

Témou práce je navrhnuť a implementovať virtuálne laboratórium s využitím JavaScriptu na strane servera. V tejto kapitole si ukážeme návrh celého virtuálneho laboratória s využitím technológií na jednotlivých komponentoch. O ich využití a konkrétnejšom popise sme si popísali viacej v sekcii 3.

V zadaní sme sa dohodli o využití Node.js platformy ako náš server. Teda centrálny server (ďalej len StarkLab), ktorý spracováva dáta z Matlab workspace. Do workspace Matlabu sa dáta získavajú v intervaloch zo Simulinku, v ktorom bola spustená referenčná schéma generujúca dáta. Zo začiatku nebolo isté či bude možné docieľiť spustenie Simulinku v reálnom čase multiplatformovo. Našli sme riešenie Simulink Real-time priamo od Mathworks, ktorá toto umožňuje ale bohužiaľ len pre operačný systém Windows. Lenže neskôr sme našli knižnicu `tos_lib.mdl`, ktorá nám túto funkcionality umožňuje aj pod MAC OS X.

Na komunikáciu s workspace používame volanie RESTful služieb, ktoré podporuje aj Matlab R2015b [12]. Keď si užívateľ spustí simuláciu cez klienta, tak dáta budú uchované aj v databáze MongoDB pre neskoršie spracovanie.

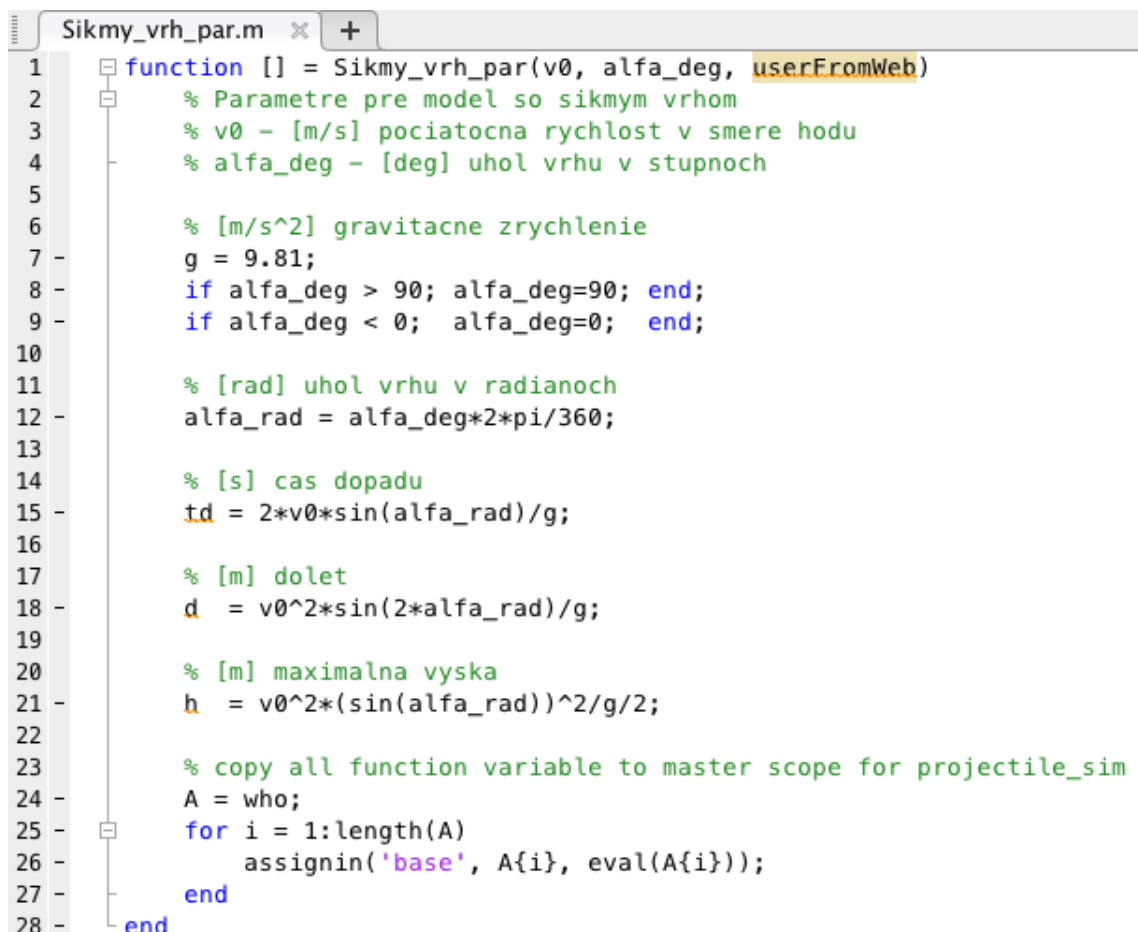


Obrázok 19: Návrh komunikácie medzi komponentami VL.

4.1 Referenčný model simulácie v Matlabe

Funkčné softwarové riešenie síce bude mať možnosť spolupracovať s reálnymi zariadeniami, no pre jeho vývoj sme použili simuláciu dynamického systému šikmého vrhu.

Pre načítanie simulácie je potrebné spustiť dva súbory. Účel prvého je inicializácia premenných potrebných pre výpočet súradníc polohy bodu v šikmom vrhu. Podľa kódu na obrázku 20 vieme, že je potrebné zadať parametre parametre pre spustenie a jej návratová hodnota nie je žiadna. Prvý z parametrov v_0 predstavuje počiatočnú rýchlosť telesa v metroch za sekundu, akou bude vymrštené z bodu $(0,0)$. Druhý parameter α_{deg} zase určuje, pod akým uhlom v stupňoch, bude teleso vymrštené. Posledný parameter $userFromWeb$ nie je potrebný k samotnej simulácii, ale skôr pre identifikáciu užívateľa, ktorý spustil simuláciu. Vďaka tomu je možné mu priradiť výsledky simulácie pri neskoršom spracovaní z databázy.



```
1 function [] = Sikmy_vrh_par(v0, alfa_deg, userFromWeb)
2     % Parametre pre model so sikmym vrhom
3     % v0 - [m/s] poiatocna rychlost v smere hodu
4     % alfa_deg - [deg] uhol vrhu v stupnoch
5
6     % [m/s^2] gravitacne zrychlenie
7     g = 9.81;
8     if alfa_deg > 90; alfa_deg=90; end;
9     if alfa_deg < 0; alfa_deg=0; end;
10
11     % [rad] uhol vrhu v radianoch
12     alfa_rad = alfa_deg*2*pi/360;
13
14     % [s] cas dopadu
15     td = 2*v0*sin(alfa_rad)/g;
16
17     % [m] dolet
18     d = v0^2*sin(2*alfa_rad)/g;
19
20     % [m] maximalna vyska
21     h = v0^2*(sin(alfa_rad))^2/g/2;
22
23     % copy all function variable to master scope for projectile_sim
24     A = who;
25     for i = 1:length(A)
26         assignin('base', A{i}, eval(A{i}));
27     end
28 end
```

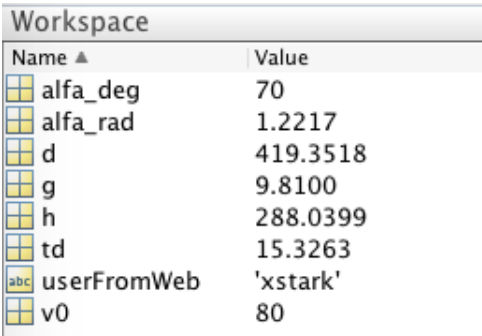
Obrázok 20: Inicializačná funkcia šikmého vrhu v Matlabe.









Keď sú parametre funkcie správne zadané a spustí sa telo funkcie, tak sa nastaví

a vypočítajú hodnoty ako `alfa_rad`, `td`, `d`, `h`, ktorých popis vidíme na obrázku a sú potrebné ako vstupné parametre do simulácie.

Posledná časť kódu od riadku č. 23 slúži pre prekopírovanie premenných do hlavného scope matlabu, čiže workspace. Keby túto časť vynecháme, tak všetky hodnoty po skončení funkcie zaniknú pretože sú tam lokálne premenné. Bolo by možné sa tomuto vyhnúť, v prípade že by sme nevolali funkciu `Sikmy_vrh_par(80, 70, 'xstark')`, ale volali priamo len súbor, vďaka čomu by zostali všetky hodnoty v hlavnom workspace. Lenže pri tomto spôsobe riešenia by nebolo možné nastavovať inicializačné hodnoty ako parametre funkcie z webovej aplikácie, ale by museli byť natvrdo nastavené v kóde Matlabu.

Ako sme si už povedali, že hodnoty sa po spustení úspešne uložia do Matlab workspace a to vidieť aj na obrázku č. 21. Tieto hodnoty boli vygenerované po zavolaní funkcie zo špecifickými hodnotami parametrov `Sikmy_vrh_par(80, 70, 'xstark')`.



Workspace	
Name ▲	Value
 <code>alfa_deg</code>	70
 <code>alfa_rad</code>	1.2217
 <code>d</code>	419.3518
 <code>g</code>	9.8100
 <code>h</code>	288.0399
 <code>td</code>	15.3263
 <code>userFromWeb</code>	'xstark'
 <code>v0</code>	80

Obrázok 21: Hodnoty v Matlab workspace po spustení funkcie.

Druhý súbor, ktorý potrebujeme spustiť po tomto je obslužný kód, vďaka ktorému vieme zaslať údaje do Node.js. Kvôli jeho dĺžke a vlastnej implementácii ho nie je možné zobrazit celý, tak si popíšeme len kľúčové vlastnosti.

Pri inicializácii sa nastavi URL cesta pre Express.js API, na ktorú bude Matlab posielat dáta.

Prednačítavanie modelu sa vykoná pomocou funkcie `load_system('Sikmy_vrh_2')`. Táto funkcia vyhledá v aktuálnom priečinku súbor `Sikmy_vrh_2.mdl` a nastaví ho ako aktuálny top-level model. Po nastavení je potrebné ešte simuláciu aj spustiť. Ovládanie simulácie sa robí pomocou príkazu `set_param(model, 'SimulationCommand', 'Start')`, kde posledný parameter znamená spustenie simulácie. Následne sa spustí **nekonečný while** cyklus, vďaka ktorému je možné zbierať údaje zo simulácie až do stavu kým nie je ukončená. Na začiatku cyklu sa volá `set_param(model, 'SimulationCommand',`

'WriteDataLogs') čo vlastne pristúpi k aktuálne najvyššie spustenej simulácii a snaží sa realtime zapisovať vypočítané hodnoty do Matlab workspace. Bez tohto príkazu by sa zapísali až po skončení simulácie.

Medzitým sa údaje upravujú na potrebný formát a pred odoslaním na REST API je vhodné ho zabaliť do JSON štruktúry. Tú vieme vytvoriť pomocou knižnice `jsonlab` v aktuálnej verzii 1.2. Pomocou sekvencie týchto dvoch príkazov vytvoríme požadovaný JSON formát a odošleme ho na Express.js API. Vytvorenie JSON `json = savejson('result', struct('user', userFromWeb, 'status', 'running', 'data', struct('time', timeFinal, 'vy', vyFinal, 'y', yFinal, 'x', xFinal)))` a jeho následné zaslanie do služby `response = webwrite(url, json, options)`.

Pomocou príkazu `get_param(model, 'SimulationStatus')` vieme získať aktuálny stav simulácie. V prípade, že simulácia stále beží tak získame výstup "running" inak je ukočená a vráti reťazec "stopped". Hneď ako dostaneme status "stopped" tak ukončíme cyklus pomocou `break` a vieme že všetky dáta su prenesené do Node.js.

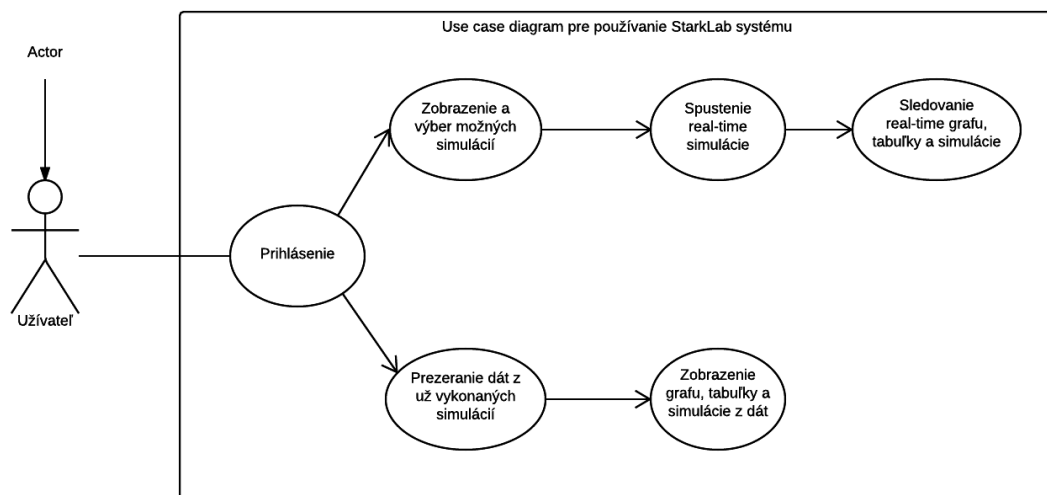
4.2 Diagramy

Pred tvorbou rôznych softwarových systémov zväčša nezačneme hneď programovať, ale najprv si zadefinujeme požiadavky, ktoré by mal spĺňať. Následne na základe požiadaviek vytvoríme diagramy, podľa ktorých sa neskôr budeme riadiť pri tvorbe software. Niektoré nám hovoria o správaní len zo všeobecnosti čo by sa mohlo dať v systéme robiť ako napr. **prípady použitia**. Potom existuje aj **diagram tried**, ktorý slúži na vizualizáciu jednotlivých tried, ktoré sa tvoria v objektovo orientovaných jazykoch. Diagram tried v našom prípade nie je možné vytvoriť, pretože JavaScript nie je klasický objektovo orientovaný jazyk. Ale zase môžeme využiť **sekvenčné diagramy**, pretože tie nám hovoria o interakciách medzi jednotlivými komponentami, prípadne časťami kódu.

4.2.1 Prípad použitia

Diagram prípadov použitia sa používa k popisu chovania systému z hľadiska užívateľa a zachytáva, ktoré typy užívateľov so systémom pracujú a aké činnosti v rámci systému vykonávajú [21].

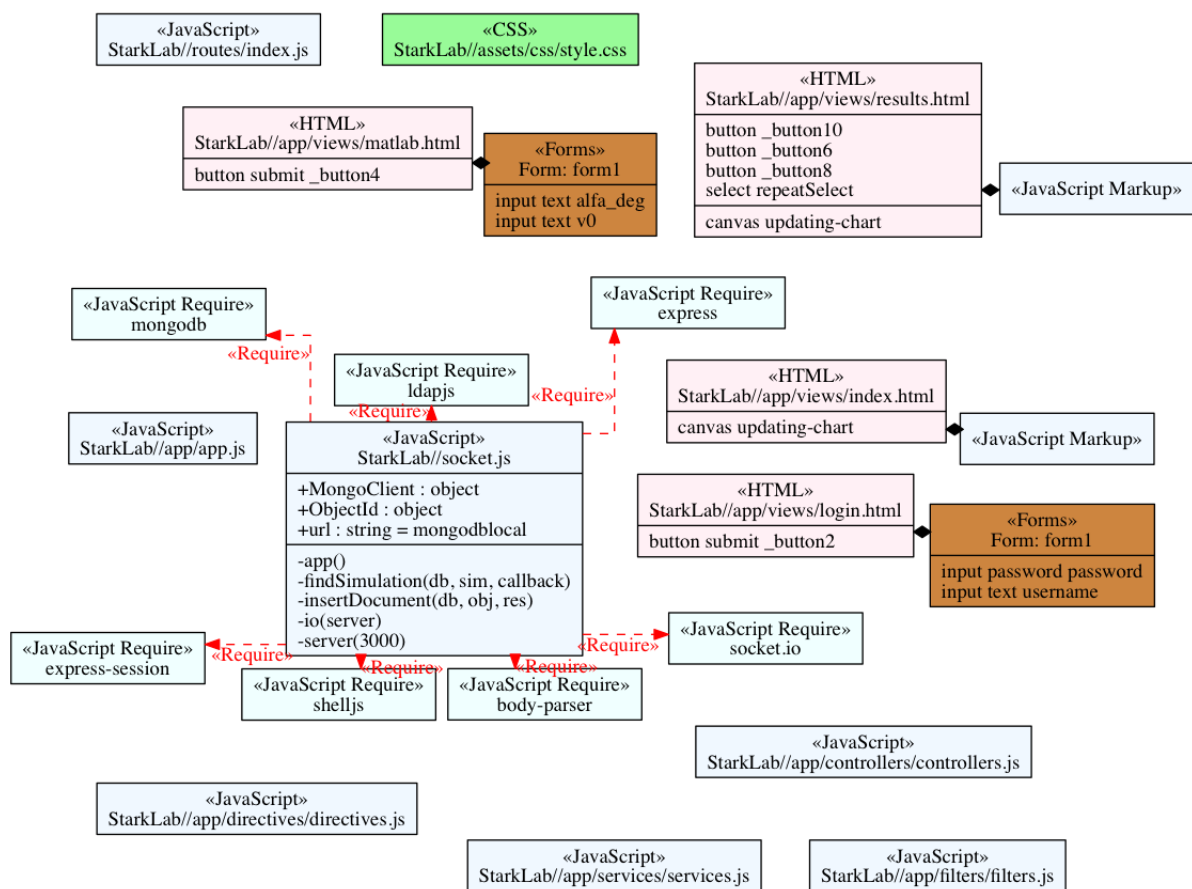
Na obrázku 22 vidíme možnosti používania systému na diagrame prípadov použitia. V systéme vystupuje len jeden typ používateľa a musí to byť člen STUBA LDAP. Bez prihlásenia sú jeho možnosti obmedzené a nemôže nič pozeráť ani vykonávať. Po prihlásení môže vykonať dve úlohy. Zobrazenie už existujúcich simulácií alebo vybrať simuláciu, ktorú chce spustiť s počiatočnými parametrami ak sú potrebné.



Obrázok 22: Diagram prípadu použitia na prácu zo systémom.

4.2.2 Diagram "tried" pre StarkLab

Síce sme hovorili o tom, že klasický diagram tried nie je možné vytvoriť pre JavaScript aplikáciu, ale našli sme knižnicu `wavi` (<https://www.npmjs.com/package/wavi>), ktorá dokáže vygenerovať takýto diagram. Síce to nie je prehľadné ako pri klasickom diagrame, ale je vhodné ho vygenerovať už len pre to, aby mal programátor informáciu o základnej štruktúre aplikácie - obrázok 23.

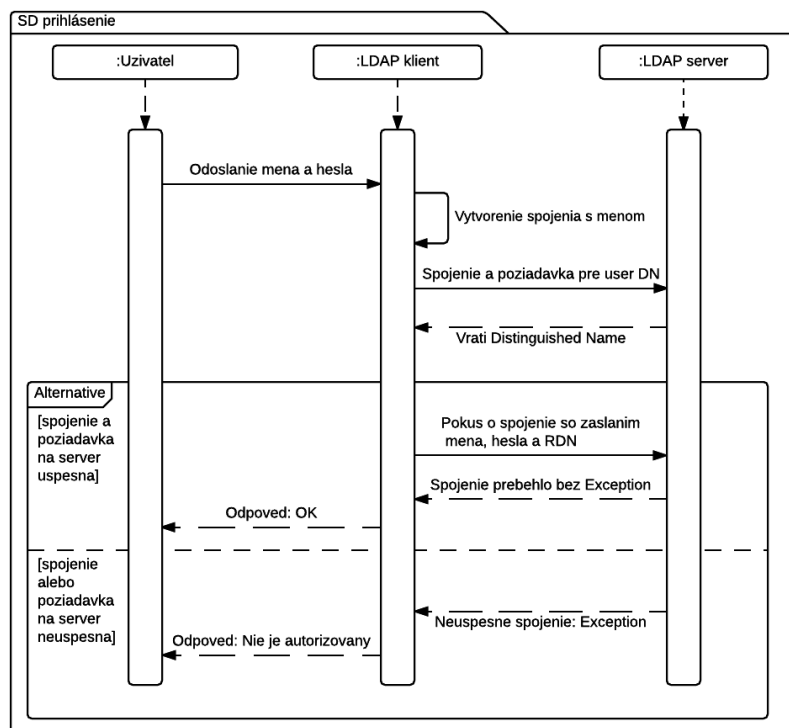


Obrázok 23: Diagram "tried" pre náš systém.

4.2.3 Sekvenčný diagram

Sekvenčný diagram patrí medzi najviac používané diagramy pre zobrazenie interakcií. Zachytáva priebeh spracovania v systéme v podobe posielania správ.

Prihlásenie do LDAP



Obrázok 24: Sekvenčný diagram pre prihlásenie pomocou LDAP.

Na obrázku 24 je spôsob prihlásenie pomocou mena a hesla do LDAP systému. Nižšie si popíšeme kód z obrázka 25, ktorým to bolo implementované v JavaScripte. Toto je len časť kódu, ktorá sa určena pre prihlásenie do systému. Na začiatku súboru je volaná požadovaná knižnica pre prácu s LDAP ako `var ldap = require("ldapjs")`.

Keď užívateľ príjde na stránku a vyplní prihlasovacie meno a heslo do stuba `ldap`, tak formulár ho presmeruje na `/login`, kde už sa postaral o routovanie Express.js. Z `request` parametra získame zadaný `username` - `req.body.username` a `password` - `req.body.password`, ktoré boli vyplnené pred odoslaním formulára. Čiže ak sú oba vyplnené dostaneme sa do vnútra podmienky, kde sa vytvorí spojenie na linku `ldap://ldap.stuba.sk`, a zašle RDN reťazec v tvare `"uid=xstark", ou=People, DC=stuba, DC=sk`.

V momente keď je zostavený reťazec a máme získané heslo užívateľa, tak zavoláme `ldap` funkciu `bind` s parametrami RDN a heslom. V prípade ak overovanie prebehlo bez chyby, tak autorizácia prebehla úspešne. Potom vytvoríme `session` a `cookie` pre užívateľa a presmerujeme ho na stránku, kde už môže vidieť simulácie.


```

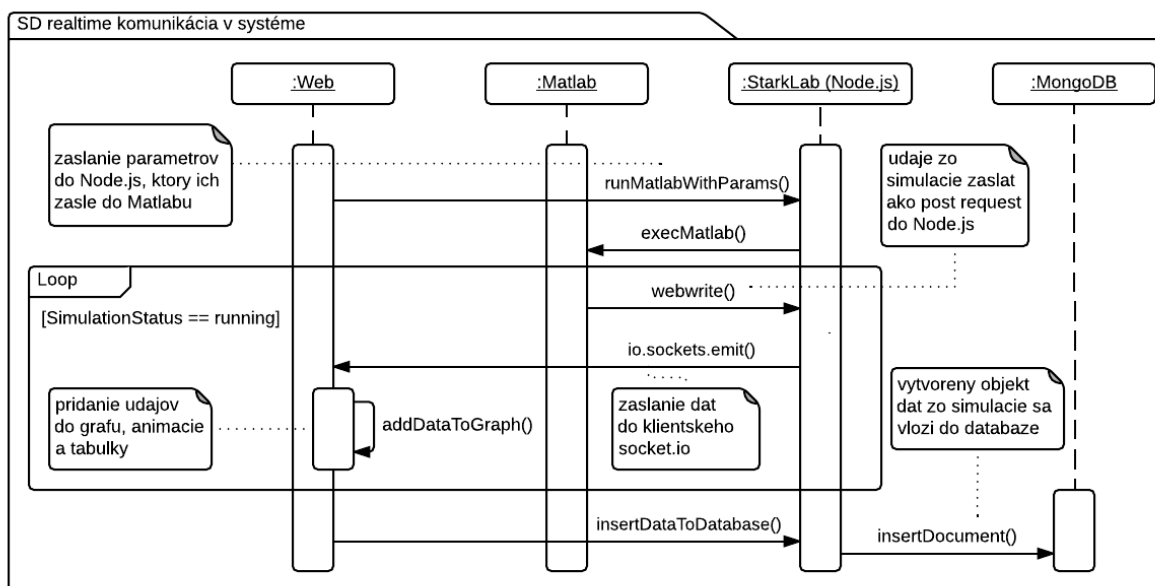
1  app.post('/login', function (req, res) {
2      if (req.body.username && req.body.password) {
3          var client = ldap.createClient({
4              url: 'ldap://ldap.stuba.sk'
5          });
6          var rdn = "uid=" + req.body.username + ", ou=People, DC=stuba, DC=sk";
7          var password = req.body.password;
8
9          client.bind(rdn, password, function (err) {
10             if (err != null) {
11                 if (err.name === "InvalidCredentialsError") {
12                     console.log("Credential error");
13                     res.redirect('/');
14                 }
15                 else {
16                     console.log("Unknown error: " + JSON.stringify(err));
17                     res.redirect('/');
18                 }
19             }
20             else {
21                 console.log("Login successful!");
22
23                 req.session.user = req.body.username;
24                 res.cookie('username', req.body.username);
25                 res.redirect('/matlab');
26             }
27         });
28     } else {
29         res.redirect('/');
30     }
31 });

```

Obrázok 25: Prihlásenie do aplikácie pomocou LDAP v JavaScripte.

Komunikácia komponentov v systéme

Medzi jednotlivými komponentami systému prebieha komunikácia. Síce v každej časti prebieha inak, ale ich spoločný menovateľ je komunikácia založená na HTTP protokole. Na obrázku 26 vidíme, že komunikácia začína od webového klienta v prehliadači. Užívateľ zadá parametre simulácie a tie budú zaslané do StarkLab. Ten následne spustí Matlab na aktuálnom operačnom systéme s potrebnými súbormi simulácie a ich parametrami. Medzitým užívateľ čaká kým sa na pozadí spustí Matlab. Hneď po spustení začne vykonávať simuláciu a posielat údaje do StarkLab, ktorý ich pošle priamo webovému klientovi, odkiaľ bola spustená simulácia. Každé prijatie údajov sa prejaví zapísaním do grafu, animácie a tabuľky vo webovom rozhraní. Táto sekvencia sa opakuje pokiaľ platí podmienka, že je simulácia stále spustená. Po zastavení simulácie klient zašle požiadavku na zapísanie kompletných dát cez StarkLab priamo do dokumentovej databáze MongoDB.



Obrázok 26: Komunikácia medzi jednotlivými komponentami systému.

4.3 Databázový model MongoDB

V našom zadaní sme nepoužívali štandardnú SQL databázu, čiže nie je možné využiť bežné modelovanie cez ERD diagramy. Ako už vieme MongoDB neukladá dáta ako tabuľky, ale JSON dokumenty. Keďže sa jednalo o jednoduchý model šikmého vrhu, tak v tomto prípade nebude model extra zložitý. V tejto časti si ukážeme do akého objektu ho v JavaScripte ukladáme a potom konkrétny príklad záznamu z databázy.

```

1  function ProjectileDataObject(user, time, x, y, vy) {
2      this.user = user;
3      this.experiment = 'projectile';
4      this.executed = new Date();
5      this.time = time;
6      this.x = x;
7      this.y = y;
8      this.vy = vy;
9  }
  
```

Obrázok 27: Model objektu v JavaScripte, pre vytvorenie záznamu v MongoDB.

Ako vidíme v JavaScripte nepotrebujeme špecifikovať nejaké štandardné typy. Hodnoty premenných `user`, `experiment` budú vždy typu `String`, `executed` je typu `Date` v štandardizovanom formáte ISO-8601, ktorého formát pre zobrazenie je YYYY-MM-

DDTHH:mm:ss.sssZ. Hodnoty premenných `time`, `x`, `y`, `vy` sú závislé od simulácie a každá z nich je pole čísiel.

Obrázok 28 je záznam z MongoDB, kde konkrétne hodnoty `time`, `x`, `y`, `vy` nie sú zobrazené, pretože simulácia vygenerovala až 788 hodnôt.

Key	Value	Type
▼ (1) {_id : 570976a056b5eff515993525}	{ 8 fields }	Document
_id	570976a056b5eff515993525	ObjectId
"user"	xstark	String
"experiment"	projectile	String
"executed"	2016-04-09T21:39:44.382Z	String
time	[788 elements]	Array
x	[788 elements]	Array
y	[788 elements]	Array
vy	[788 elements]	Array

Obrázok 28: Príklad záznamu simulácie v MongoDB.

4.4 Node.js s frameworkom Express.js

Hlavná časť tejto práce je sústredená na komunikáciu medzi Matlabom a Node.js, resp Express.js. Využívame ich posledné verzie Node.js verzie 6.1.0 a Express.js verzie 4.13.4. Je to vlastne len framework umožňujúci jednoduchšiu tvorbu REST služieb. Na začiatok si povieme čo sme použili ako `middleware`. Ako už bolo napísané je to funkcia, ktorá má prístup do request a response objektu. Použili sme `body-parser`, ktorý nám umožnil spracovávať telo POST požiadavky, ktorá bola zasielaná z Matlabu. Každý `middleware`, ktorý chceme použiť je potrebné ešte aj aktivovať v kóde `app.use(bodyParser.json({limit: '2mb'}))`. Po viacerých testoch som zistil, že základný limit je nastavený na 1MB, kde sa vyskytoval problém s prijímaním POST requestu, keď boli dáta veľmi veľké.

4.4.1 Vlastný middleware prihlasovania

V aktuálnom softwarovom riešení nebolo potrebné vytvoriť prihlasovanie, pretože neskôr bude integrované do **LMS Moodle**, ktorý prihlasovanie rieši vlastným spôsobom. Ale pre ukážku možností frameworku Express.js sme implementovali `middleware express-session`, vďaka ktorému môžeme vytvoriť reláciu pre úspešne autorizovaného užívateľa.

Pri implementácii prihlasovania cez session sme zistili, že v každom smerovaní, bolo potrebné kontrolovať na začiatku kódu, či tam pristupuje prihlásený používateľ alebo požiadavka z Matlab služby. Pri väčšom projekte, prípadne viacerých smerovaní by to začalo byť neefektívne a neprehľadné.

Vytvorili sme vlastný `middleware` na obrázku 29, ktorý slúži na overenie prihláseného

užívateľa pri každom prístupe na akékoľvek smerovanie.

```
// middleware to check logged user through browser or result from matlab
app.use(function (req, res, next) {
  if ((req.session && req.session.user) || req.body && req.body.result && req.body.result.user) {
    next();
  } else {
    res.redirect('/');
  }
});
```

Obrázok 29: Middleware kód v Express.js na kontrolu prihlásenia.

Volanie `app.use` nám zabezpečí, že sa táto funkcia zaregistruje ako middleware a bude volaná pri každom requeste. Prvá časť podmienky overuje, či prichádzame z webového prehliadača a je nastavená session. Keďže Matlabu nevieme nastaviť session, pretože tá sa ukladá len na webserveri tak sme to vyriešili iným spôsobom. V prichádzajúcom tele POST requestu od Matlabu je objekt `result`, kde sú uložené simulované dáta. Priradili sme mu ďalšiu vlastnosť `user`, ktorá keď je nastavená, tak vieme, že požiadavka prichádza z Matlabu a vieme jej povoliť prístup. V prípade splnenej aspoň jednej podmienky sa zavolá callback `next()`, čo teda vyvolá ďalší zaregistrovaný middleware, alebo už smerovanie, na ktoré sme posielali požiadavku. Ak nie je splnená ani jedna z dvoch podmienok, tak nás server presmeruje na hlavnú stránku, kde sa od nás požaduje znovu prihlásenie.

4.4.2 Spustenie Matlabu z príkazového riadku

Zo začiatku nebolo isté ako budeme spúšťať simuláciu, pretože sme nevedeli či Node.js umožňuje vykonávať príkazy nad operačným systémom, resp. spúšťať programy. Vtedy sme komunikáciu riešili takým spôsobom, že Matlab sa otvoril ručne a v ňom sme spustili potrebné inicializačné súbory a následne samotnú simuláciu. Takéto riešenie nie je postačujúce z hľadiska automatizácie a samostatnosti.

Potom sme zistili, že Node.js má možnosti na spustenie hocijakého software, ktorý je možné spustiť cez terminál. Pre zjednodušenie práce bol využitý modul `shelljs`, ktorý takúto možnosť poskytuje.

Na obrázku 30 je funkcionálna, ktorá sa vykoná po prístupe na route `/matlab/run`. Táto časť sa volá po odoslaní formuláru s parametrami počítačovej rýchlosti a uhlu z webového prehliadača.



```

app.post('/matlab/run', function (req, res) {
  var cmd = '\\Applications\\MATLAB_R2015b.app\\bin\\matlab -nosplash -nodesktop -noFigureWindows ' +
    '-r \\cd \\Users\\Erich\\Desktop\\DP\\Matlab\\diploma-matlab\\;Sikmy_vrh_par(' +
    + req.body.v0 + ', ' + req.body.alfa_deg + ', \\ ' + req.session.user + '\\);projectile_sim;exit;\\';

  shell.exec(cmd, function (code, stdout, stderr) {
  });

  res.redirect('/dashboard');
});

```

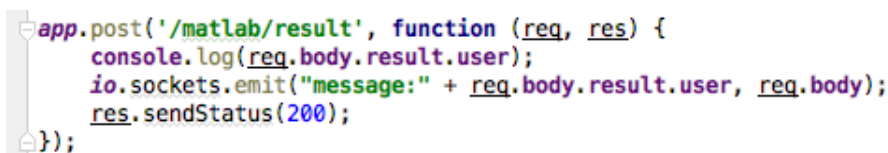
Obrázok 30: Spustenie príkazu v OS pomocou shelljs modulu.

Vstupný príkaz vo forme reťazca, ktorý chceme vykonať, musí mať správne zadanú cestu k Matlabu na súborom systéme. Matlab spúšťame v termináli s parametrami, ktoré umožňujú spustenie bez nepotrebných častí [15]. Ako napr. nepotrebujeme vidieť výsledky v okne, alebo animáciu Matlabu pri spustení. Pomocou parametra `-r` vieme volať príkazy v príkazovom riadku Matlabu, napr. špecifikovať cestu, kde sa nachádza naša simulácia a potrebné súbory. Následne zavoláme funkciu `Sikmy_vrh_par()`, s ktorou sa pošlú parametre počiatočnej rýchlosti, uhla a zároveň meno prihláseného používateľa. Po vykonaní tejto funkcie sa spustí súbor simulácie `projectile_sim`. Po jej skončení zatvoríme Matlab pomocou príkazu `exit`.

Po zostrojení tohto reťazca ako parametra funkcie `exec()` a jeho spustení sa čaká na vykonávanie Matlabu. Medzitým nás server presmeruje na stránku `/dashboard` kde sa čaká na údaje zo simulácie. V momente keď sa načíta simulácia a začne jej vykonávanie so vstupnými parametrami, tak sa dáta zobrazia realtime vo webovom prehliadači.

4.4.3 Práca s dátami simulácie

Keď je simulácia spustená, Matlab je pripravený zasielať výsledky na smerovanie, ktoré vidíme na obrázku 31.



```

app.post('/matlab/result', function (req, res) {
  console.log(req.body.result.user);
  io.sockets.emit("message:" + req.body.result.user, req.body);
  res.sendStatus(200);
});

```

Obrázok 31: Prijímanie a zasielanie dát pomocou Socket.io do prehliadača.

Výsledky z Matlabu máme uložené v tele každého requestu, ktorý príjde. Následne môžeme k nim prístup pomocou `req.body`. Pôvodne **Socket.io** odosielať údaje len na adresu "message". V takom prípade by sme ale nevedeli identifikovať, ku ktorému užívateľovi sa majú posilať údaje. Po implementácii session sme zvažili, že by ich bolo

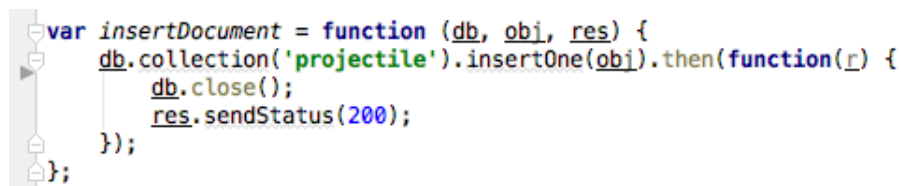
lepšie posielat priamo na adresu "message:xstark", teda s menom prihláseného užívateľa. Vždy po prijatí je potrebné odoslať status kód 200, čo v HTTP znamená OK po úspešnom prijatí dát.

4.4.4 Zápis dát do MongoDB

Po prijatí a zobrazovaní dát nasleduje spracovanie. Ich spracovanie sa vykonáva na úrovni Express.js za pomoci MongoDB ovládača.

Na začiatku je potrebné najprv zavolať modul a následne jeho klientskú časť cez `var MongoClient = require('mongodb').MongoClient`. Potom nasleduje definovanie URL, na ktorej máme spustenú databázu `mongodb://localhost:27017/test`.

Najprv si ukážeme funkcie, ktoré vykonávajú svoju úlohu, až potom ich použitie. Prvá z nich je vloženie dokumentu do databázy na obrázku 32. Táto funkcia slúžiaca na vytvorenie jedného záznamu, sa pokúsi vložiť `obj` do kolekcie **projectile**, kde sú vlastne naše kompletne záznamy zo simulácií. Po úspešnom vložení zatvorí prihlásenie na databázu a vráti HTTP status kód 200.



```
var insertDocument = function (db, obj, res) {  
  db.collection('projectile').insertOne(obj).then(function(r) {  
    db.close();  
    res.sendStatus(200);  
  });  
};
```

Obrázok 32: Vloženie záznamu do MongoDB pomocou JavaScriptu.

Druhú funkciu, ktorú využívame na prácu s databázou je na obrázku 33. Slúži na vyhľadanie záznamu v závislosti od parametrov. Jednu funkciu vieme využiť na viacero účelov. V prípade ak zadáme parameter funkcie `sim`, tak v podmienke sa skontroluje, či je dostupná vlastnosť `id`. Ak áno, tak vyhľadanie v databáze prebehne na základe parametrov užívateľského mena a konkrétneho id simulácie, čo nám vyhledá iba jeden záznam ak existuje. V opačnom prípade vyhledá všetky záznamy simulácií pre daného užívateľa. Táto funkcionalita sa využíva pri zobrazení všetkých záznamov u klienta.

```

var findSimulation = function (db, sim, callback) {
  var query = {};
  var user = sim.user;
  var simType = sim.experiment;
  var simulationId = sim.id;
  if (simulationId) {
    query = {
      "_id": ObjectId(simulationId),
      "user": user
    }
  } else {
    query = {
      "user": user
    }
  }
  var cursor = db.collection("projectile").find( query ).toArray(function(err, results){
    callback(results);
  });
};

```

Obrázok 33: Vyhľadanie záznamu v MongoDB pomocou JavaScriptu.

Po ukážke implementácií funkcií, ktoré vykonávajú zápis/čítanie si ukážeme ich volanie. Na obrázku 34 je vloženie záznamu po prístupe na smerovanie /mongo/insert/one. Pre vykonanie nejakej operácie nad databázou musíme vytvoriť spojenie medzi klientom a MongoDB. Služi na to funkcia `connect(url, callback)`, ktorej prvý parameter je URL kde sa nachádza DB a druhý je callback, kde sa vracajú údaje.

```

app.post('/mongo/insert/one', function (req, res) {
  console.log("mongo insert one");

  MongoClient.connect(url, function (err, db) {
    if (err === null) {
      console.log("Connected correctly to server.");
      insertDocument(db, req.body, res);
    }
  });
});

```

Obrázok 34: Volanie vloženia záznamu do MongoDB.

Pre vyhľadanie záznamu na obrázku 35 sa s pripojením nič nemení. Rozdiel je iba vo využití volania smerovania. Dvojbodka pred názvom `:user` znamená, že tento parameter môže byť v URL dynamický. Otáznik pri parametri ako pri `:id?` znamená, že tento nie je povinný, a URL môže byť volaná aj bez neho. Ak sa teda v URL nachádzajú parametre, pristupujeme k nim pomocou poľa `req.params`.

```

app.get('/mongo/:user/:simulation?/:id?', function (req, res) {
  MongoClient.connect(url, function (err, db) {
    if (err) {
      console.log(err);
      res.send(err);
    } else {
      console.log("Connected correctly to server.");
      var checkedId = undefined;

      if (req.params.id && req.params.id.length === 24) {
        checkedId = req.params.id;
        // should be send bad param
      }

      var simulationParams = {
        user: req.params.user,
        experiment: req.params.simulation,
        id: checkedId
      };

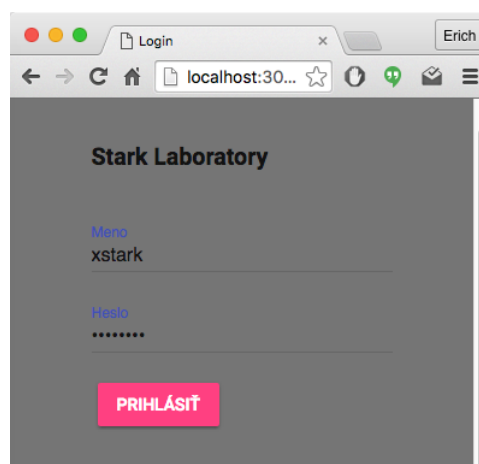
      findSimulation(db, simulationParams, function (results) {
        res.setHeader('Content-Type', 'application/json');
        res.send(JSON.stringify(results));
        db.close();
      });
    }
  });
});

```

Obrázok 35: Volanie vyhľadania záznamu v MongoDB.

4.5 Webový klient s frameworkom Angular.js

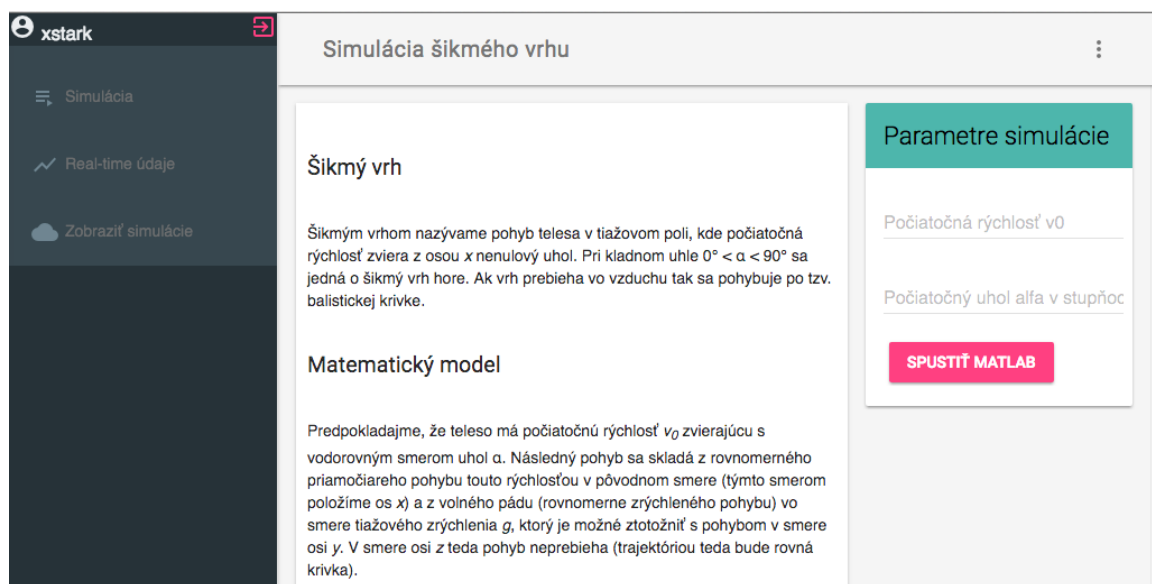
Na klientskú časť práce sme využili JavaScriptový framework Angular.js v poslednej stabilnej verzii 1.5.5. Úloha webového klienta, bolo overiť funkčnosť vytvoreného prostredia, ktorý posiela simulované dáta. Funkčnosť sme overili a popíšeme si konkrétne obrazovky.



Obrázok 36: Prihlásenie do LDAP pomocou STUBA údajov.

Ako to funguje na pozadí už sme si popísali v sekcii 4.2.3, kde sa pojednávajú sekvenčné diagramy, konkrétne prihlásenie do LDAP. Po prihlásení sa zobrazí stránka vyhradená pre zadanie parametrov šikmého vrhu, počiatočnú rýchlosť v_0 a uhol vystrelenia α_{deg} . Rovnako sú tu popísané aj základné informácie o simulácii a jeho vzorcov.

Po zadaní rýchlosti $v_0=40$ a $\alpha_{deg}=60$ sme presmerovaní na stránku /matlab.



Obrázok 37: Parametre simulácie - počiatočná rýchlosť a uhol v stupňoch.

4.5.1 Grafy s Chart.js

Pre vizualizáciu prichádzajúcich dát používame knižnicu na grafy `Chart.js`, ktorá je vytvorená pomocou HTML canvas. Využívame poslednú verziu jednotkovej rady 1.1.1. Používame ju ako skript v HTML stránke `<script src="/node_modules/chart.js/Chart.min.js">`. Ako si môžeme všimnúť tak nie je závislá od žiadnej online CDN služby, ale jej inštalácia prebehla lokálne pomocou NPM.

Vyvorili sme Angular.js directívu `uiGraph`, ktorá slúži pre vytvorenie grafu na potrebnom mieste. Ako template má v obsahu `<canvas id="updating-chart" width="600" height="300"></canvas>`. Na obrázku 38 vidíme vytvorenie grafu a nastavenie hodnôt. Táto časť kódu sa nachádza v `postLink` funkcii directívy.

Na začiatku je potrebné získať element z DOM, získať kontext canvasu a vytvoriť objekt `startingData` s nastaveniami pre inicializačné dáta. Následne je potrebné vytvoriť inštanciu grafu, kde ako parameter grafu použije kontext canvasu, a volá sa funcia čiarového grafu, ktorý tam potrebujeme. Tá zase vyžaduje ako vstupný parameter inicializačné dáta. Túto directívu zavoláme v HTML ako `<ui-graph></ui-graph>` element.

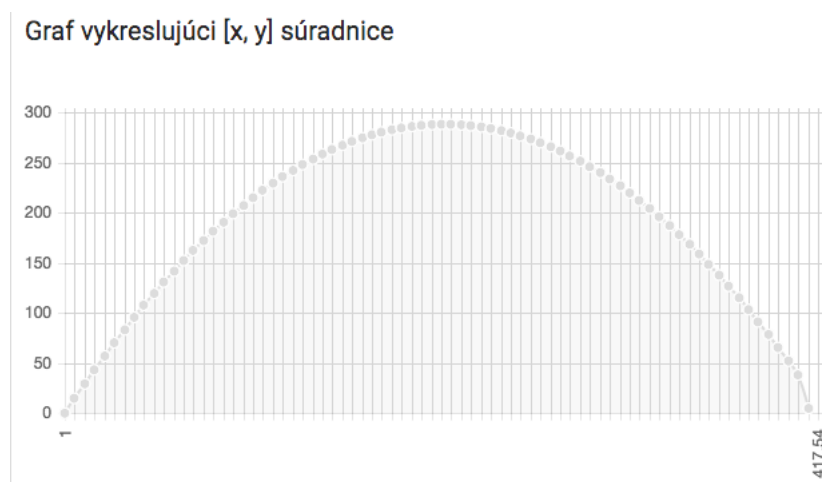
Ak by sme chceli rozšíriť graf, resp. ak by bolo rozumné súčasne vykreslovať viacero závislých veličín, tak je možné zdefinovať ďalší dataset.

```
var canvas = document.getElementById('updating-chart'),
    ctx = canvas.getContext('2d'),
    startingData = {
      labels: [1],
      datasets: [
        {
          fillColor: "rgba(220,220,220,0.2)",
          strokeColor: "rgba(220,220,220,1)",
          pointColor: "rgba(220,220,220,1)",
          pointStrokeColor: "#fff",
          data: {}
        }
      ]
    },
    latestLabel = startingData.labels[0];

var myLiveChart = new Chart(ctx).Line(startingData, {
  responsive: true,
  animationEasing: 'easeOutBounce',
  animationSteps: 15,
  scaleGridLineColor: 'lightgray'
});
```

Obrázok 38: Inicializácia hodnôt pre Chart.js.

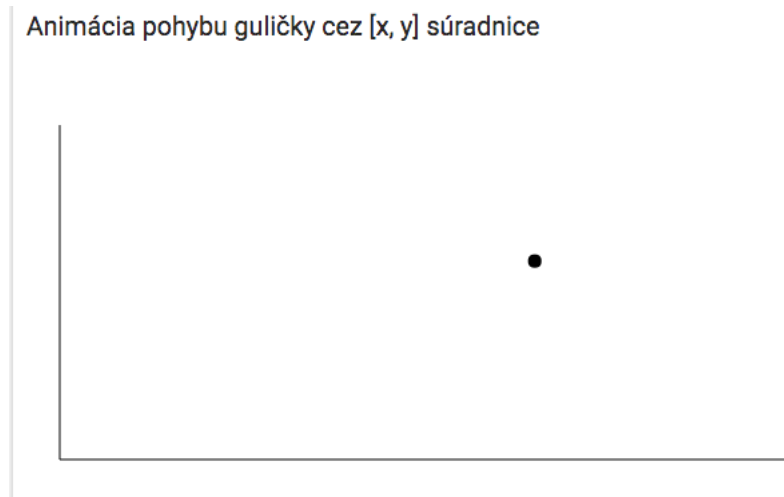
V sekcii 4.5 sme si hovorili o tom, čo musíme spraviť, aby nám generovalo údaje. Po presmerovaní na stránku /matlab treba počkať kým sa spustí Matlab zo simuláciou a následne sa začnú generovať realtime dáta do prehliadača. Táto časť by mohla byť zrýchlená tým, že sa Matlab bude nachádzať na výkonnom serveri. Na obrázku 39 je využitá knižnica Chart.js pre zobrazenie grafov.



Obrázok 39: Graf vykresľujúci závislosti [x, y] v šikmom vrhu.

4.5.2 Animácia pomocou html canvas

Vykreslené dáta na obrázku 40 sú zhodné s dátami na grafe, len je to spôsobom animácie guličky. Táto animácia bola vytvorená pomocou html canvas technológie.



Obrázok 40: Animácia vykresľujúca závislosti [x, y] v šikmom vrhu.

4.5.3 Tabuľka dát

Ako posledná časť sledovania dát je tabuľka, ktorej údaje sa vkladajú do zoznamu rovnako realtime ako aj pri grafe a animácii pred ním.



























Tabuľka dát			
Čas	Os x	Os y	Rýchlosť y
0	0	0	75.18
0.2	5.47	14.84	73.21
0.4	10.94	29.29	71.25
0.6	16.42	43.34	69.29
0.8	21.89	57	67.33
1	27.36	70.27	65.37
1.2	32.83	83.15	63.4
1.4	38.31	95.63	61.44
1.6	43.78	107.72	59.48
1.8	49.25	119.42	57.52
2	54.72	130.73	55.56
2.2	60.2	141.65	53.59
2.4	65.67	152.17	51.63
2.6	71.14	162.3	49.67
2.8	76.61	172.04	47.71
3	82.08	181.38	45.75

Obrázok 41: Tabuľka dát času, x, y a smer rýchlosti vy.

4.5.4 Zobrazenie existujúcich simulácií

V tomto systéme nejde len o real-time vykreslenie dát, ale aj o neskoršie zobrazenie a spracovanie. Po prejdení na stránku zo simuláciami, vidíme všetky záznamy simulácií pre aktuálne prihláseného užívateľa - obrázok 42.

Zoznam všetkých simulácií

Užívateľ	Dátum	Model	Zobraziť	Zmazať
xstark	09.04.2016 11:39	projectile		
xstark	09.04.2016 11:42	projectile		
xstark	13.04.2016 09:22	projectile		
xstark	13.04.2016 09:23	projectile		
xstark	13.04.2016 09:23	projectile		
xstark	14.04.2016 02:00	projectile		
xstark	14.04.2016 02:00	projectile		
xstark	14.04.2016 02:00	projectile		
xstark	14.04.2016 11:41	projectile		
xstark	14.04.2016 11:46	projectile		
xstark	14.04.2016 11:52	projectile		
xstark	14.04.2016 11:53	projectile		
xstark	14.04.2016 11:54	projectile		

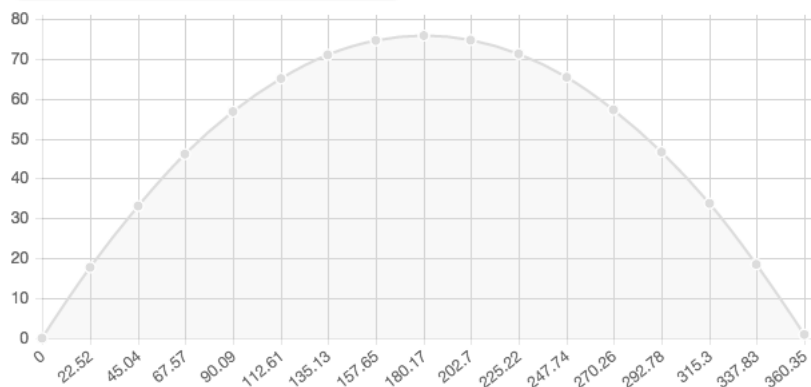
Obrázok 42: Zoznam uložených simulácií pre prihláseného užívateľa.

Graf vykreslujúci [x, y] súradnice

Vzorka dát: 29

ZOBRAZIŤ ULOŽENÉ ÚDAJE REALTIME

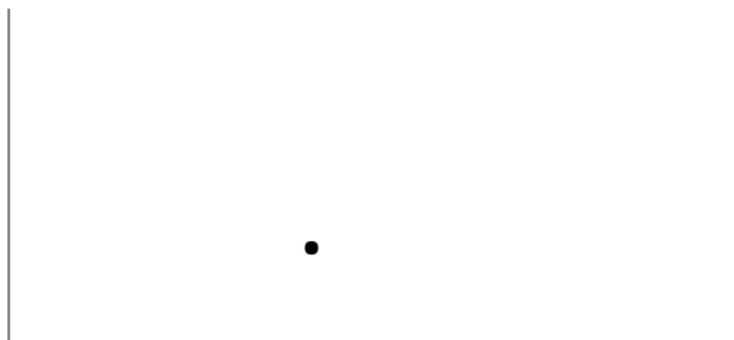
ZOBRAZIŤ GRAF ZO VŠETKÝMI ÚDAJMI



Obrázok 43: Graf pre vybranú vzorku dát.

Animácia pohybu guľičky cez [x, y] súradnice

ZOBRAZIŤ ÚDAJE V ANIMÁCII



Obrázok 44: Animácia pre vybranú vzorku dát.

Na webovej stránke sa ešte nachádza pod animáciou sa nachádza rovnaká tabuľka ako v obrázku č. 41 s rozdielom, že v tejto tabuľke sú kompletne údaje a je možné ich exportovať do CSV formátu.

Záver

Diplomová práca nám ozrejmla pojmy ako virtuálne laboratórium, aké môže mať komponenty a jeho využitie v súčasnej dobe.

Cieľom tejto práce bolo naštudovanie problematiku virtuálnych laboratórií, vytvoriť stručnú analýzu existujúcich riešení a vytvoriť komplexnú aplikáciu, ktorá sa mala skladať z viacerých častí. Začali sme tvorbou a úpravou referenčnej simulácie šikmého vrhu pre Simulink podľa potrieb. Ďalej bolo treba zabezpečiť prenos dát medzi Matlabom a Node.js, čo sme vyriešili vďaka RESTful komunikácií. Údaje bolo potrebné posielat ďalej do webového prehliadača. Táto časť sa vyriešila použitím JavaScript knižnice Socket.io čo je obdoba websocketov s rozšírenou podporou aj pre staršie prehliadače. Na strane webu sa využil aktuálne populárny frontend JavaScript framework Angular.js. Medzi dôležité súčasti implementácie nebolo len zobrazovať realtime údaje v grafoch, ale ich aj ukladať pre neskoršie spracovanie. Údaje sa podarilo ukladať do dokumentovej databáze MongoDB. Ciele, ktoré sme si stanovili, sme aj splnili.

Za konečným výsledkom je vidieť mnoho práce. Síce súčasné riešenie nie je možné nasadiť do reálnej prevádzky bez istých úprav a integrácií, ale poslúži ako solídny základ, na ktorom je možné stavať a využiť ho minimálne v priestoroch FEI STU na simuláciu systému alebo na zber dát z reálneho zariadenia.

Tu sa práca ešte nekončí a je možné rozšíriť StarkLab ďalšou zaujímavou funkcionalitou. Ako napríklad zovšeobecnenie rozhrania pre komunikáciu. Vytvorenie rozhrania aj pre iný software ako Matlab pre jednoduchšie pripojenie výpočtov, alebo simulácie. Nasadenie Matlabu na samostatný server s dostupnou doménou. Nahrávanie simulácií a Matlab súborov cez webové rozhranie. Možností ako ho vylepšiť je množstvo. V prípade, že bude niekto pokračovať v tejto téme, môže sa nimi inšpirovať.

Zoznam použitej literatúry

- [1] BORKA, T. Aplikácia typu klient-server pre virtuálne laboratórium s využitím platformy .net. Master's thesis, FEI STU, 2012. FEI-5406-30519.
- [2] EXPRESS.JS. Express routing, 2016. <http://expressjs.com/en/starter/basic-routing.html>.
- [3] EXPRESS.JS. Writing middleware for use in express apps, 2016. <http://expressjs.com/en/guide/writing-middleware.html>.
- [4] FARKAŠ, R. Vzdialené laboratórium riadenia reálnych systémov využívajúce technológie matlab a java. Master's thesis, FEI STU, 2011. FEI-5384-16514.
- [5] GOOGLE. Angularjs, 2016. <https://en.wikipedia.org/wiki/AngularJS>.
- [6] GOOGLE. What is angular?, 2016. <https://docs.angularjs.org/guide/introduction>.
- [7] HATHERLY, Paul. *The Virtual Laboratory and interactive screen experiments*. s. 1. <https://web.phys.ksu.edu/icpe/publications/teach2/Hatherly.pdf>.
- [8] KUNDRÁT, M. Aplikacia pre virtualne laboratorium vyuzivajuca java matlab interface. Master's thesis, FEI STU, 2014. FEI-5384-47855.
- [9] SANTANA I.; FERRE M.; IZAGUIRRE E.; ARACIL R.; HERNÁNDEZ L. *Remote Laboratories for Education and Research Purposes in Automatic Control Systems*. IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, VOL. 9, NO. 1, FEBRUARY 2013. s. 3. <http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6140966>.
- [10] MATHWORKS. Matlab - the language of technical computing, 2016. <http://www.mathworks.com/products/matlab/>.
- [11] MATHWORKS. Matlab com integration, 2016. http://www.mathworks.com/help/matlab/matlab_external/introducing-matlab-com-integration.html.
- [12] MATHWORKS. *Matlab RESTful web services*, r2015b ed. <http://www.mathworks.com/help/matlab/internet-file-access.html>, 2016.
- [13] MATHWORKS. Matlab weboptions, 2016. <http://www.mathworks.com/help/matlab/ref/webopt>

- [14] MATHWORKS. Matlab webread, 2016. <http://www.mathworks.com/help/matlab/ref/webread.h>
- [15] MATHWORKS. Start matlab program from mac terminal, 2016. <http://www.mathworks.com/help/matlab/ref/matlabmac.html>.
- [16] MICROSOFT. What is com? <https://www.microsoft.com/com/default.mspx>.
- [17] MONGODB. What is mongodb, 2016. <https://en.wikipedia.org/wiki/MongoDB>.
- [18] NODEJS. History of nodejs, 2016. <https://en.wikipedia.org/wiki/Node.js>.
- [19] OLAKARA, A. R. Nodejs architecture, 2015. <http://abdelraoof.com/blog/2015/10/19/introduction-to-nodejs/>.
- [20] OLAKARA, A. R. Understanding nodejs event loop, 2015. <http://abdelraoof.com/blog/2015/10/28/understanding-nodejs-event-loop/>.
- [21] REJNKOVÁ, P. Diagram případů užití, 2009. http://uml.czweb.org/pripad_uziti.htm.
- [22] S., G. L. B. Current trends in remote laboratories. *IEEE Transactions on Industrial Electronics* (december 2009). <http://ieeexplore.ieee.org/xpls/icp.jsp?arnumber=5280206>.
- [23] TSONEV, Krasimir. *Node.js By Example*. PACKT PUBLISHING.
- [24] TUTORIALSPPOINT. What is nodejs, 2015. http://www.tutorialspoint.com/nodejs/nodejs_introd
- [25] TUTORIALSPPOINT. Angularjs concepts, 2016. http://www.tutorialspoint.com/angularjs/angularjs_overview.htm.
- [26] TUTORIALSPPOINT. Express overview, 2016. http://www.tutorialspoint.com/nodejs/nodejs_exp
- [27] TUTORIALSPPOINT. Node.js - request object properties, 2016. http://www.tutorialspoint.com/nodejs/nodejs_request_object.htm.
- [28] TUTORIALSPPOINT. Node.js - response object properties, 2016. http://www.tutorialspoint.com/nodejs/nodejs_response_object.htm.
- [29] VAQQAS, M. Restful web services: A tutorial, 2014. <http://www.drdobbs.com/web-development/restful-web-services-a-tutorial/240169069>.

- [30] VARGA, S. Virtuálne laboratórium riadenia dynamických systémov. Master's thesis, FEI STU, 2015. FEI-5396-5865.
- [31] vlab.co.in. *Philosophy of Virtual Laboratories*. <http://vlab.co.in/>.
- [32] ČERVENÝ, T. Mobilný klient pre účely virtuálneho laboratória využívajúci web rozhranie. Master's thesis, FEI STU, 2014. FEI-5384-29988.

Prílohy

A	Štruktúra elektronického nosiča	II
---	---	----

A Štruktúra elektronického nosiča

```
\
\Bakalarska_praca.pdf
\FEIk_Identuty.xpi
\FEIkIdentity
\FEIkIdentity\chrome.manifest
\FEIkIdentity\install.rdf
\FEIkIdentity\content
\FEIkIdentity\content \function.js
\FEIkIdentity\content \options.xul
\FEIkIdentity\content \overlay.xul
\FEIkIdentity\content \window.js
\FEIkIdentity\content \window.xul
\FEIkIdentity\defaults
\FEIkIdentity\defaults\preferences
\FEIkIdentity\defaults\preferences \prefs.js
\FEIkIdentity\locale
\FEIkIdentity\locale \sk-SK
\FEIkIdentity\locale \sk-SK\options.dtd
\FEIkIdentity\locale \sk-SK\window.dtd
\FEIkIdentity\skin
```