

СЕМИНАРСКА РАБОТА

Безбедност при комуникација во реално време

СОДРЖИНА

1. BOBED	3
2. NODEJS.....	4
3. SSL.....	6
4. EXPRESS.....	8
5. ANGULARJS.....	9
6. POSTGRESQL.....	15
7. SOCKET.IO.....	22
8. MONGODB.....	25
9. ЗАКЛУЧОК.....	33
10. ЛИТЕРАТУРА.....	34

ВОВЕД

Во овој семинарски труд ќе биде објаснет принципот на работа на систем за комуникација во реално време. Технологии кои се користат за реализација се: NodeJS (Express, Socket.io frameworks) - серверска страна, AngularJS - клиентска страна, PostgreSQL и MongoDB - backend. Овој сет од технологии се користи кога имаме работа со голем број на клиенти и имаме размена на голем број на податоци. Принципот на работа е едноставен: клиентот се најавува со своето корисничко име и лозинка и доколку има други клиенти може да праќа пораки. Притоа се опфатени некои принципи како проверка на дупликат на корисничко име при креирање на нов профил, токен базирана автентикација при најава, креирање на self-signed certificate и креирање на https помеѓу клиентот и серверот, SSL помеѓу серверот и PostgreSQL и SSL помеѓу серверот и MongoDB.

1. NODEJS

NodeJS претставува софтверска платформа за скалабилни серверски и мрежни апликации. NodeJS апликациите се пишувани во JavaScript, и може да бидат извршени во NodeJS околина на Mac OS X, Windows и Linux без промени. Овие апликации се дизајнирани за максимизирање на продуктивноста и ефикасноста користејќи неблокирачки I/O и асинхрони настани. NodeJS апликациите се извршуваат како една нишка, иако NodeJS користи повеќе нишки за податочни и мрежни настани. Затоа NodeJS често се користи за апликации кои се извршуваат во реално време.

1.1 Инсталација NodeJS и NPM

Принципот на инсталација на NodeJS зависи од оперативниот систем. Овде ќе биде прикажан начинот на инсталација на Ubuntu 14.04. Исто така покрај NodeJS потребно е инсталирање на NPM(Node Package Manager) кој подоцна ќе ни послужи за преземање на потребните пакети (модули). За инсталација користиме:

- `sudo apt-get install nodejs`
- `sudo apt-get install npm`

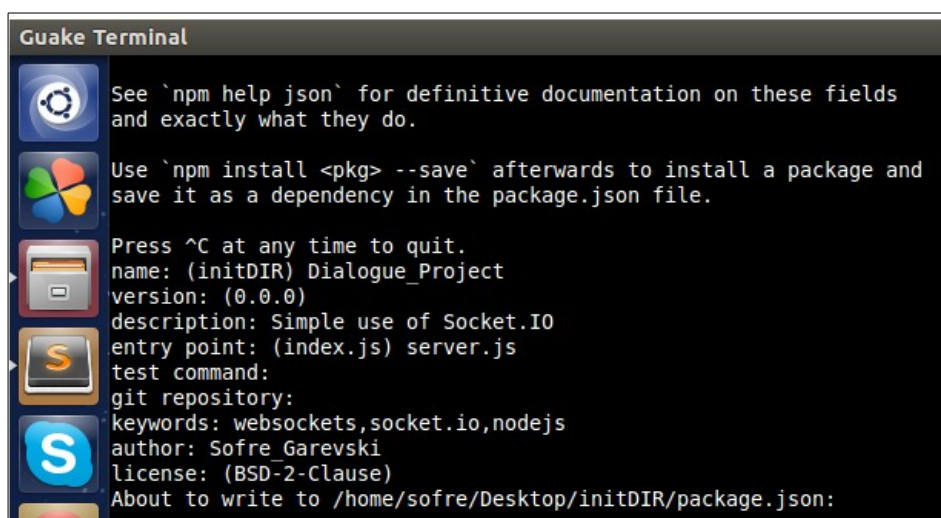
Доколку го инсталираме на Windows се користи волшебник кој може да биде превземен од официјалната страна, а NPM ќе биде автоматски вклучен.

1.2 Иницијализација

Откако ќе инсталираме NodeJS и NPM може да иницијализираме сопствен модул (пакет). Иницијализацијата се прави со следната команда:

- `sudo npm init`

По извршување на оваа команда внесуваме податоци за самиот модул како автор, верзија, клучни зборови итн. Ова е прикажано на Слика 1.2.1.



Слика 1.2.1 Иницијализација на модул

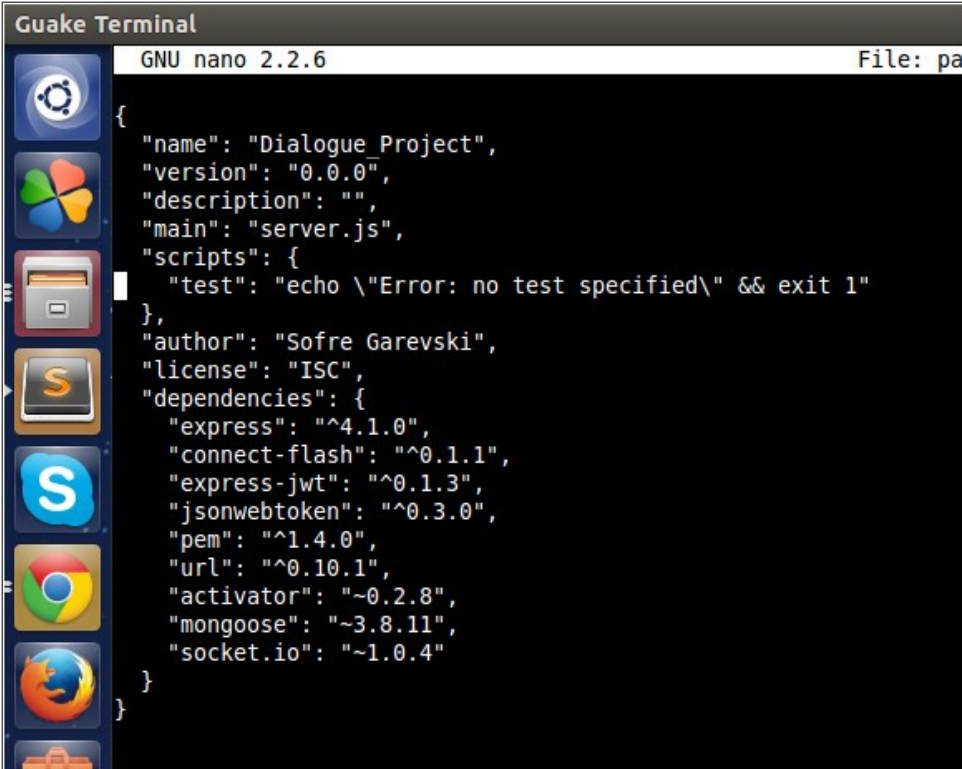
Резултатот од оваа команда и внесените параметри е package.json JSON фајл кој ги содржи внесените параметри.

1.3 Инсталација на модули (пакети)

Нашата апликација има зависност од повеќе модули и тоа: Express, Connect, HTTPS, FS(File Stream), PG(PostgreSQL), Mongoose и Socket.IO. Постојат два начини на инсталација на модулите.

1. Со додавање на сите имиња на модулите во *dependencies* во package.json и верзија на модулот. Потоа со извршување на командата *sudo npm install* модулите се преземаат и се креира директориум *node_modules*.
2. Со извршување на командата *sudo npm install module_name —save*. Во нашиот случај би имале: *sudo npm install express —save* итн.

Сите модули може да бидат најдени на www.npmjs.org. Во нашиот случај ги имаме express, connect, https, fs, pg, mongoose, socket.io како модули. По инсталацијата на модулите package.json би изгледал како на Слика 1.3.1.



```
GNU nano 2.2.6 File: package.json
{
  "name": "Dialogue Project",
  "version": "0.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Sofre Garevski",
  "license": "ISC",
  "dependencies": {
    "express": "^4.1.0",
    "connect-flash": "^0.1.1",
    "express-jwt": "^0.1.3",
    "jsonwebtoken": "^0.3.0",
    "pem": "^1.4.0",
    "url": "^0.10.1",
    "activator": "~0.2.8",
    "mongoose": "~3.8.11",
    "socket.io": "~1.0.4"
  }
}
```

Слика 1.3.1 Package.json после инсталација на потребните пакети

2. SSL

Нормалниот веб сообраќај се праќа неенкриптиран преку интернет. Тоа значи дека секој кој има пристап до правилните алатки може да го прислушува целиот сообраќај. Очигледно, ова може да води до проблеми, особено каде безбедноста и приватноста се неопходни. Secure Socket Layer се користи за енкриптирање на податочниот тек помеѓу веб серверот и веб клиентот (пребарувачот).

SSL користи како што се нарекува асиметрична криптографија, често нарекувана криптографија со јавен клуч (PKI). Со криптографијата со јавен клуч, два клуча се креирани, еден јавен и еден приватен (таен). Се што е енкриптирано со еден од клучевите може да биде декриптиран со неговиот соодветен клуч. Така ако порака или податочен тек се енкриптирани со серверскиот приватен(таен) клуч, може да биде декриптиран само со користење на соодветниот јавен клуч, осигурувајќи се дека сигурно податоците пристигнале од серверот.

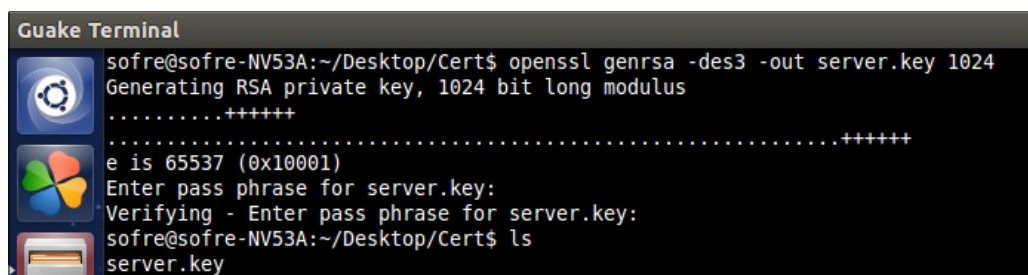
Ако SSL иницијализира криптографија со јавен клуч за енкрипција на податочниот тек кој патува преку мрежата, зошто е неопходен сертификат? Техничкиот одговор на ова прашање е дека сертификат не е неопходен — податоците се безбедни и неможат лесно да бидат декриптирани од трето лице. Како и да е, сертификатите имаат значајна улога во комуникацискиот процес. Сертификатот, потпишан од верен Сертификациски Акторитет (CA), тврди дека носителот на сертификатот е навистина тој што се претставува. Без верен потпишан сертификат, податоците може да бидат енкриптирани, но лицето со кое се разменуваат да не е тоа што мислиме. Без сертификати, присвојувачките напади ќе бидат многу по зачестени.

2.1 Генерирање на приватен клуч

Openssl претставува алатка која се користи за генерирање на RSA приватен клуч и CSR (Certificate Signing Request). Исто така може да се користи за генерирање на self-signed сертификати кои може да бидат користени за тестирачки намени или внатрешна употреба.

Првиот чекор е да креираме RSA приватен клуч. Овој клуч е 1024 битен RSA клуч кој е енкриптиран со Triple-DES и зачуван во PEM формат така што ќе биде читлив како ASCII текст. Командата за генерирање на приватен клуч е:

- `openssl genrsa -des3 -out server.key 1024`

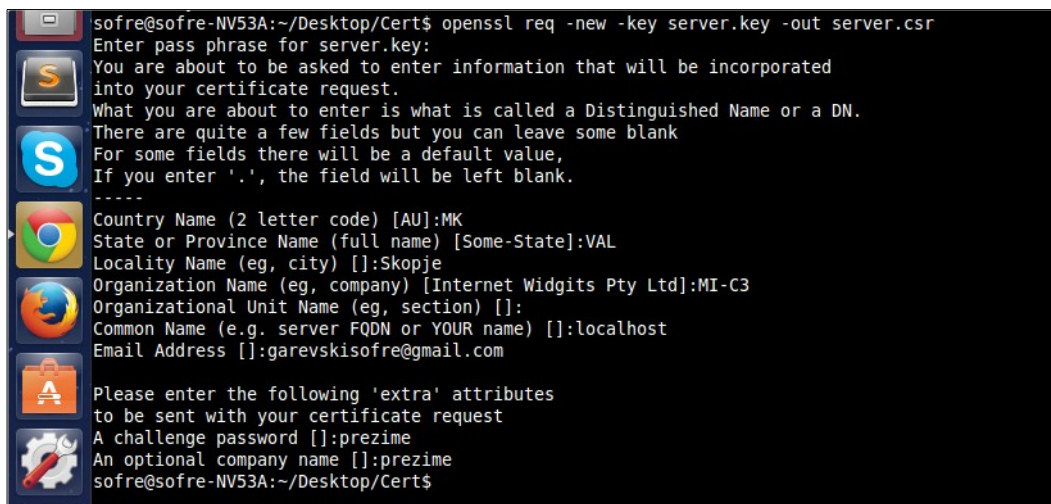


Слика 2.1.1 Генерирање на приватен клуч

2.2 Генерирање на CSR (Certificate Signing Request)

После генерирање на приватен клуч, CSR може да биде генериран. CSR сертификатот после може да биде искористен на два начина. Да биде предаден на Сертификациски Авторитет (CA) како Thawte или Verisign кои ќе го верифицираат идентитетот на побарувачот и ќе издадат потпишан сертификат. Втората опција е своерачно да го потпишеме CSR фајлот, кое подоцна и ќе го направиме. За генерирање на CSR:

- `openssl req -new -key server.key -out server.csr`



Слика 2.2.1 Генерирање на CSR

За време на генерирањето на CSR ќе биде потребно внесување на додатни информации за X.509 сертификатот. Едно од прашањата ќе биде "Common Name (e.g. server FQDN or YOUR name)" важно е ова поле да биде пополнето со целосното име на серверот кој треба да биде заштитен од SSL. Ако вебсајтот кој треба да биде заштитен е <https://public.akadia.com> потребно е да внесеме public.akadia.com.

2.3 Бришење на лозинката од клучот

Еден неочекуван ефект од заштитениот приватен клуч е тоа што Apache или било кој друг сервер ќе праша за лозинка секојпат кога веб серверот е стартуван. Нормално ова не е неопходно бидејќи потребно е да има некој кој ќе внесува лозинка секојпат после рестартирање или паѓање на серверот. Можно е да се избрише Triple-DES енкрипцијата од клучот, така што ќе нема потреба од повторно пишување на лозинка. За бришење на лозинката од клучот:

- `cp server.key server.key.org`
- `openssl rsa -in server.key.org -out server.key`

Новокреираниот `server.key` нема повеќе лозинка.

2.4 Генерирање на Self-Signed сертификат

За генерирање на self-signed сертификат се користи наредбата:

- `openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt`

3. EXPRESS

Express претставува најпопуларен framework на NodeJS, слично како што Rails или Sinatra се за Ruby. Претходно креираме пар од приватен клуч (*server.key*) и self-signed сертификат (*server.crt*). Следно потребно е да направиме сервер кој кога некој клиент ќе му пристапи ќе му ја прикаже нашата апликација. Express користи повеќе "template engines" за приказ на содржина како: asyncEJS, bake, bind.js, Blade, bliss.js, blue, CoffeeKup, CoffeeMugg итн. Ние нема да користиме никој од нив туку за клиентска страна ќе користиме AngularJS кој претставува популарна JavaScript работна рамка која подоцна ќе биде објаснета. Целта е да ги одделиме серверската од клиентската страна, така што серверот ќе биде само посредник помеѓу клиентот и базата. За почеток ќе креираме фолдер *angular* и внатре ќе ја креираме *index.html* — страната која ќе треба да му се прикаже на клиентот. Исто така ќе креираме фајл *server.js* со следниот код:

```

1  var express = require('express'),
2      fs = require('fs'),
3      connect = require('connect'),
4      https = require('https');
5
6  var app = express();
7
8  var options = {
9      key: fs.readFileSync('server.key'),
10     cert: fs.readFileSync('server.crt')
11  }
12
13  app.use('/', express.static(__dirname + '/angular'));
14  app.set('env', 'production');
15  app.use(connect.bodyParser());
16
17  var server = https.createServer(options, app).listen(3000, function () {
18     console.log('Server listens on port 3000');
19  });

```

Слика 3.1 Иницијализирање на сервер

Во првите четири линии се референцираме кон потребните модули. FileStream (fs) и HTTPS се модули кои доаѓаат заедно со инсталацијата на NodeJS односно нема потреба да ги бараме со NPM. Connect и Express ги инсталираме со NPM. Потоа иницијализираме express апликација и синхронно ги вчитуваме *server.key* и *server.crt*. Така на линија број 13 бараме од серверот доколку некој пристапи на нашата страна да му го даде како одговор се она што се наоѓа во *angular* директориумот. Односно:

- `app.use()` - овозможува користење на некој middleware¹
- `express.static()` - овозможува монтирање на датотека од фајл системот на одредена патека
- `__dirname` — ја дава патеката во која се наоѓа фајлот во кој се користи `__dirname`

1 Middleware – функција која прима три параметри: *request*, *response*, *next*. Кога некој ќе пристапи на дадена патека, се извршува. Слично на сервлети кај Јава.

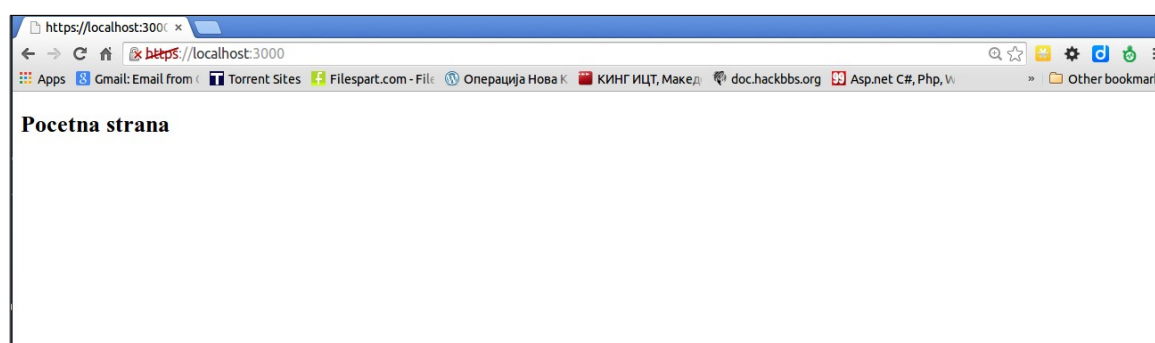
- `app.set()` - сетира node променлива со некоја вредност. Во нашиот случај ја сетира `NODE_ENV` (Environment mode) променливата со "production" вредност. Почетната вредност на оваа променлива е "development".
- `Connect.bodyParser()` - подоцна ќе ни послужи за да ги земаме вредностите на параметрите од барањата (requests)

Потоа со `https` модулот креираме сервер за дадената `express` апликација и претходните опции и со методот `listen` го подесуваме да слуша на порта 3000.

Кај `index.html` се испишува само пораката "Pocetna strana". Со извршување на наредбата:

- `sudo nodejs server.js` или на Windows `node server.js`

се прикажува следното:



Слика 3.2 Приказ на `index.js`

4. ANGULARJS

AngularJS претставува слободна(open-source) платформа развиена од Google и заедницата, која помага со креирање на single-page апликации, one-page веб апликации кои само бараат HTML, CSS и JavaScript на клиентска страна. Негова цел е да ги збогати веб апликациите со MVC(Model-View-Controller) способноста. Бидејќи се работи за single-page апликација, значи дека целото сценарио ќе се одвива во постоечката `index.html`. Во нашата апликација потребно е кога клиентот ќе пристапи да се појави форма за креирање на нов профил, но исто така потребно е постоечки корисник да може да се најави со корисничко име и лозинка. По внесувањето на корисничко име и лозинка, се енкриптираат податоците на клиентска страна и како такви се праќаат до серверот. Серверот треба да провери дали постои таков корисник и доколку постои да врати токен со одредено времетраење, во спротивно треба да врати грешка. За постигнување на ова потребно е да ги вклучиме следните датотеки:

- `CryptoJS` — библиотека со криптографски алгоритми имплементирана во JavaScript. Ќе ни послужи за енкрипција на корисничките податоци.
- `angular.min.js` — фајл во кој е сместена самата angular платформа
- `angular.route.min.js` — фајл за помош со рутирање во самата апликација
- `angular.resource.min.js` — библиотека која овозможува интеракција со RESTful серверски податочни извори

- `sanitize.min.js` — библиотека која помага за рендерирање на `html`-от и двонасочното податочно поврзување (`two way data binding`). Се имплементира `ngSanitize` сервисот при иницијализација на контролер
- `styles.css` — ќе ги содржи стиловите за апликацијата
- `main.js` — ќе го содржи *main* контролерот за апликацијата
- `routes.js` — ќе ги содржи патеките за апликацијата
- `userService.js` — ќе ги содржи функциите за работа со корисници (најава, креирање на профил, одјава, верификација)
- `authInterceptor.js` — ќе го содржи сервисот за автентикација на корисниците
- `dataTransfer.js` — ќе содржи сервиси за пренос на корисничките податоци и историја на пораките

`Styles.css`, `main.js`, `routes.js`, `userService.js`, `authInterceptor.js` и `dataTransfer.js` се датотеки кои допрва ќе ги направиме и објасниме, додека претходните се готови библиотеки. Така `index.html` ќе биде:

```

1 <html ng-app="app" ng-controller="main">
2   <head>
3     <title>Dialogue</title>
4     <!-- Styles files -->
5     <link rel="stylesheet" type="text/css" href="styles/stylesheets/styles.css">
6   </head>
7   <body>
8     <div class="container" ng-view>
9
10    </div>
11    <!-- CryptoJS -->
12    <script type="text/javascript" src="scripts/tripleledes.js"></script>
13    <!-- Angular libs -->
14    <script type="text/javascript" src="scripts/angular.min.js"></script>
15    <script type="text/javascript" src="scripts/angular-route.min.js"></script>
16    <script type="text/javascript" src="scripts/angular-resource.min.js"></script>
17    <script type="text/javascript" src="scripts/angular-sanitize.min.js"></script>
18    <!-- Controllers -->
19    <script type="text/javascript" src="scripts/controllers/main.js"></script>
20    <!-- Routes -->
21    <script type="text/javascript" src="scripts/routes.js"></script>
22    <!-- Services -->
23    <script type="text/javascript" src="scripts/factory/userService.js"></script>
24    <script type="text/javascript" src="scripts/factory/authInterceptor.js"></script>
25    <script type="text/javascript" src="scripts/factory/dataTransfer.js"></script>
26
27  </body>
28 </html>

```

Слика 4.1 Имплементирање и иницијализација на ангулар проект

Како што може да се види од сликата најпрво ќе треба да направиме модул *app* и контролер *main*. Сите темплејти кои подоцна ќе ги креираме ќе се вчитуваат во *div* контејнерот кој е потребно да го има атрибутот *ng-view*. Во самиот *angular* директориум ќе имаме *scripts* директориум и во него ќе бидат сместени основните библиотеки и два директориуми *controllers* и *factory*. Во *controllers* ќе биде сместен *main* контролерот додека во *factory* ќе бидат сместени сите сервиси. *Main* контролерот е иницијализиран и прикажан на слика 4.2.

```

1  var app = angular.module('app', ['ngRoute', 'ngResource', 'ngSanitize']);
2
3  app.controller('main', function($scope){
4    |
5  });

```

Слика 4.2 Иницијализација на main контролерот

Следно што доаѓа се два темплејти кои ќе ги сместиме во *templates* директориумот (angular/templates). *Main.html* ќе биде темплејтот кој ќе го користам за login страната додека *home.html* ќе биде темплејтот за почетната страна. За премин од една на друга форма потребно е да имаме дефинирано патеки. Патеките ќе ги дефинираме во *route.js*.

```

1  app.config(function ($routeProvider,$locationProvider,$httpProvider) {
2
3      $routeProvider
4          .when('/',{
5              templateUrl:'templates/main.html',
6              controller:'main',
7              access : {allowAnonymous : true}
8          })
9          .when('/home',{
10             templateUrl:'templates/home.html',
11             controller:'main',
12             access : {allowAnonymous : false}
13          });
14
15      $httpProvider.interceptors.push('authInterceptor');
16  });

```

Слика 4.3 Дефиниција на патеки

Во нашиот случај имаме само две патеки и во *access* кажуваме која од нив ќе биде слободна. *HttpProvider.interceptors* претставува низа која служи за чување на сервиси кои имаат работа со справување со грешки, автентикација или било каква потреба на асинхроно пре-процесирање на барања или пост-процесирање на одговори (responses). *Interceptor* (пресретнувач) претставува сервис кој се повикува и враќа пресретнувач (interceptor). Има два вида на пресретнувачи:

- request — пресретнувачот се повикува со *http config* објект
- response — пресретнувачот се повикува со *http response* објект.

Request претставува функција која врши промена на *config* објектот кој се праќа до серверот. Во нашиот случај доколку постои *token* додаваме хедер за авторизација. Целта е да ги заштитиме некои од патеките од ненајавен корисник. *Response* функцијата враќа *res* објект или промис од *res* променливата. Ова е покажано на Слика 4.5. За најавување, одјавување, креирање на нов корисник и верифицирање на корисник треба да имаме посебни функции. Ове функции ќе ги сместиме во посебен сервис *UserService.js*. Исто така ќе треба да направиме функции за повикување на истите од *main.js* контролерот. За да ограничиме кој корисник на која патека има пристап потребно е во *main.js* да го додадеме кодот од Слика 4.4.

```

$scope.$on('$routeChangeStart', function(e, next, current) {
  if (next.access !== undefined && !next.access.allowAnonymous
    && !$window.sessionStorage.token) {
    $location.path('/');
  } else if (next.access !== undefined && !next.access.allowAnonymous
    && $window.sessionStorage.token) {
    Auth.verify();
  } else if (next.access !== undefined && next.access.allowAnonymous
    && $window.sessionStorage.token) {
    if ($route.current)
      $location.path($route.current.$$route.originalPath);
  }
});

```

Слика 4.4 Рестрикција на патеките за најавени и ненајавени корисници

```

1 app.factory('authInterceptor',function ($rootScope, $q, $window) {
2   return {
3     request: function(config){
4       config.headers = config.headers || {};
5       if($window.sessionStorage.token)
6       {
7         config.headers.Authorization = 'Bearer ' + $window.sessionStorage.token;
8       }
9       return config;
10    },
11    response:function(res){
12      return res || $q.when(res);
13    }
14  };
15 });
16

```

Слика 4.5 Request и response функциите

```

1 app.factory('Auth',function ($http,$window,$location) {
2   return {
3     login : function(user){
4       var encUser = CryptoJS.TripleDES.encrypt(user.username, "Fraza za user");
5       var encPass = CryptoJS.TripleDES.encrypt(user.password, "Fraza za password");
6
7       $http.post('/login',{username:encUser,password:encPass})
8         .success(function(data){
9           $window.sessionStorage.token = data.token;
10          $location.path("/home");
11        })
12        .error(function(data){
13          alert('Message from server:\n'+data.message);
14          delete $window.sessionStorage.token;
15          console.log('Error',data.message);
16        });
17    },
18    logout : function () {
19      delete $window.sessionStorage.token;
20      $location.path('/');
21    }
22  };
23 });
24

```

Слика 4.6 Auth сервисот со login и logout функции

При логирање ги енкриптираме корисничкото име и лозинка со TripleDES криптирачки алгоритам. Комуникацијата помеѓу клиентот и серверот е енкриптирана со https, и не е неопходно да имаме енкрипција, но ова е само пример за енкриптирање на клиентска страна. Потоа правиме POST барање до серверот. Доколку одговорот е потврден ја сетираме сесиската променлива *token* и го водиме клиентот на самата *home* страна. Доколку серверот врати 401-

Unauthorize потребно е да му прикажеме порака на клиентот. За одјава на клиентот потребно е да ја избришеме сесиската променлива *token*.

За креирање на корисник ќе ја користиме функцијата *createUser* дел од *Auth* сервисот. Ќе ги енкриптираме сите податоци за новиот корисник, а потоа со http POST барање ќе ги испратиме до сервер. Доколку акцијата е успешна го зачувуваме генерираниот токен. Доколку се направи грешка ја прикажуваме грешката на клиентот.

Функцијата *verify* ја користиме за проверка на постоечкиот токен, дали е валиден и дали алоцираното време на корисникот е истрошено. *CreateUser* и *verify* се прикажани на Слика 4.7.

```

1  createUser : function (user) {
2      //console.log('CLIENT',user);
3      var encFname = CryptoJS.TripleDES.encrypt(user.firstname, "Fraza za firstname");
4      var encLname = CryptoJS.TripleDES.encrypt(user.lastname, "Fraza za lastname");
5      var encUser = CryptoJS.TripleDES.encrypt(user.username, "Fraza za user");
6      var encPass = CryptoJS.TripleDES.encrypt(user.password, "Fraza za password");
7      var encGender = CryptoJS.TripleDES.encrypt(user.sex, "Fraza za gender");
8
9      $.http.post('/newUser',{firstname:encFname,lastname:encLname,
10         username:encUser,password:encPass,gender:encGender})
11
12         .success(function(data){
13             //console.log('DATA',data);
14             $window.sessionStorage.token = data.token;
15             $location.path("/home");
16         })
17         .error(function(data){
18             alert('Message from server:\n'+data.message);
19             delete $window.sessionStorage.token;
20             console.log('Error',data.message);
21         })
22     },
23     verify:function(){
24         $.http.post('/verify',{token:$window.sessionStorage.token}).error(function(response){
25             delete $window.sessionStorage.token;
26             $location.path('/');
27         });
28     }
29 }

```

Слика 4.7 Креирање на корисник и верификација

Откако ќе се креира нов корисник или ќе се логира постоечки корисник потребно е да ги земеме корисничките податоци од сервер. За таа цел ќе креираме сервис *userData* во *dataTransfer.js* датотеката. Тоа е прикажано на Слика 4.8.

```

1  app.factory('userData',function ($resource) {
2      return $resource('/userData');
3  });
4

```

Слика 4.8 userData сервис

Досега ги испишавме сервисите кои треба да ги задоволуваат основните функции за работа на самиот систем. Исто така потребно е истите да ги имплементираме и во самиот *main* контролер.

```

3 app.controller('main', function($scope,Auth,userData,$window){
4   window.scope = $scope;
5   $scope.uniqueUsername = false;
6
7   $scope.login = function(user) {
8     Auth.login(user);
9   }
10
11   $scope.logout = function() {
12     Auth.logout();
13   }
14
15   $scope.createUser = function(user) {
16     Auth.createUser(user);
17   }
18
19   if ($window.sessionStorage.token) {
20     userData.get({
21       token: $window.sessionStorage.token
22     }, function(data) {
23       $scope.user = data.doc;
24     });
25   }
26 });

```

Слика 4.9 Основните функционалности во main контролерот

При креирање на нов корисник потребна е валидација на полињата и унифицираност на самиот nickname. Затоа во *main* контролерот се креирани две функции *checkFields* и *checkNickname* соодветно. *checkFields* враќа true или false додека *checkNickname* прави повик до самата база и ја сетира променливата *uniqueUsername*. Ова е прикажано на Слика 4.10.

```

1 $scope.checkFields = function(user) {
2   if (user) {
3     var firstname = /[a-zA-Z]/g.exec(user.firstname || '');
4     var lastname = /[a-zA-Z]/g.exec(user.lastname || '');
5     var username = /[a-zA-Z0-9]/g.exec(user.username || '');
6     var gender = user.sex;
7     //console.log("PASS:",user.password,"PASS2",user.password2);
8     if (gender)
9       gender == 'true' ? true : false;
10    return (username && firstname && $scope.uniqueUsername && lastname &&
11      gender && user.password != undefined && user.password2 != undefined &&
12      user.password2 != "" && user.password2 != "" && (user.password === user.password2));
13    || false;
14  }
15  return false;
16 };
17
18 $scope.checkNickname = function(username) {
19   $http.post('/checkNickname', {
20     nickname: username
21   }).success(function(res) {
22     $scope.uniqueUsername = true;
23   }).error(function(data) {
24     $scope.uniqueUsername = false;
25     alert(data.message);
26   });
27 }

```

Слика 4.10 Функции за валидација на полињата при креирање на нов корисник и проверка на унифициран nickname

Следно што доаѓа е да овозможиме акциите за најава, креирање на корисник, проверка на nickname и одјава да ги изведеме на страна на база. Исто така ќе треба да ја обезбедиме комуникацијата помеѓу серверот на базата и самиот клиент — серверот креиран во nodejs.

5.POSTGRESQL

PostgreSQL претставува релационен тип на податочен управувачки систем (ORDBMS). Развиен е од PostgreSQL Global Development Group и е слободен (open source). Токму поради тие карактеристики ќе го инсталираме и имплементираме во нашиот проект. За инсталација треба да ги извршиме следните чекори:

- `sudo apt-get update`
- `sudo apt-get install postgresql postgresql-contrib`
- `sudo apt-get install pgadmin3`

Кај Windows PostgreSQL може да се преземе од официјалната страна, а инсталацијата е преку волшебник. Исто така и PgAdminIII ќе биде вклучен во инсталацијата. Следно потребно е да се креира база, но претходно треба да се креира податочен корисник (admin). За креирање на корисник:

- `sudo su - postgres`
- `psql` или `psql -U postgres -W postgres`
- `CREATE USER sofre WITH PASSWORD 'garevski';`
- `ALTER USER sofre SUPERUSER CREATEDB;`
- `CREATE DATABASE dialogue OWNER sofre;`

Со овие неколку команди се најавивме како корисник 'postgres', креиравме корисник `sofre` му доделивме привилегии на суперкорисник, и креиравме база `dialogue`. Следно што треба да направиме е да креираме табела за корисници.

- `CREATE TYPE friend_record AS (id_user INTEGER, username NAME, first_name NAME, last_name NAME, sex BOOLEAN, imageurl TEXT);`
- `CREATE TABLE users (id_user serial NOT NULL, username NAME, password TEXT, first_name NAME, last_name NAME, sex BOOLEAN, imageurl TEXT, friends friend_record[]);`

Со првата команда креиравме нов тип на податок наречен *friend_record*. Следно што доаѓа е да креираме функции за земање на постоечки корисник (`getUser`), функција за креирање на корисник (`createUser`) и функција за проверка на nickname (`checknickname`). За земање на постоечки корисник:

```
CREATE OR REPLACE FUNCTION getuser(NAME, TEXT)
  RETURNS user_record AS
$BODY$
DECLARE
  result_record user_record;
BEGIN
  SELECT id_user, username, first_name, last_name, sex, imageurl
    INTO result_record
  FROM users
  WHERE "username" = $1 AND "password" = $2;

  RETURN result_record;
END
$BODY$
LANGUAGE plpgsql
```


Слика 5.1 Креирање на функција getuser

За креирање на нов корисник имаме:

```
CREATE OR REPLACE FUNCTION createuser(NAME, NAME, NAME, TEXT, BOOLEAN)
  RETURNS user_record AS
$BODY$
DECLARE
  result_record user_record;
BEGIN
  INSERT INTO users (first_name, last_name, username, "password", "sex", "imageurl")
  VALUES($1,$2,$3,$4,$5,'./templates/images/user.png');

  result_record.first_name = $1;
  result_record.last_name = $2;
  result_record.username = $3;
  result_record.sex = $5;
  result_record.imageurl = './templates/images/user.png';

  RETURN result_record;
END
$BODY$
LANGUAGE plpgsql
```

Слика 5.2 Креирање на нов корисник

За проверка на nickname имаме:

```
CREATE OR REPLACE FUNCTION checknickname(NAME)
  RETURNS friend_record AS
$BODY$
DECLARE
  result_record friend_record;
BEGIN
  SELECT id_user, username, first_name, last_name, sex, imageurl, friends
  INTO result_record
  FROM users
  WHERE "username" = $1;

  RETURN result_record;
END
$BODY$
LANGUAGE plpgsql
```

Слика 5.3 Проверка на nickname

Следно што треба да направиме е да овозможиме SSL конекција со PostgreSQL серверот. За тоа потребно е да ги искористиме како CA претходно генерираните серверски клуч (server.key) и сертификат(server.crt). Ќе генерираме клиентски клуч (client.key) и сертификат (client.crt) каде CA (Certification Authority) ќе ни биде server.crt. За таа цел ги испишуваме следните команди:

- openssl genrsa -des3 -out client.key 1024
- openssl req -new -key client.key -out client.csr
- cp client.key client.key.org
- openssl rsa -in client.key.org -out client.key
- openssl x509 -req -CAcreateserial -in client.csr -CA server.crt -CAkey server.key -out client.crt

- `cp client.crt /etc/ssl/certs/client.crt`
- `cp client.key /etc/ssl/private/client.key`
- `cp server.crt /etc/ca-certificates/server.crt`
- `cd /etc/ssl/private`
- `sudo chmod 740 client.key`
- `sudo chown postgres client.key`
- `sudo chgrp postgres server.key`

Со претходните команди освен што креираваме клиентски клуч и сертификат, ги копираме на соодветните локации и ги промениваме пермисиите на *client.key* датотеката. Следно што треба да направиме е промени во *postgres.conf* датотеката и *pg_hba.conf* датотеката. За *postgres.conf* кај линијата *# - Security and Authentication* треба да внесеме:

- `ssl = on`
- `ssl_cert_file = '/etc/ssl/certs/client.crt'`
- `ssl_key_file = '/etc/ssl/private/client.key'`
- `ssl_ca_file = '/etc/ca-certificates/server.crt'`

За *pg_hba.conf* треба да ги внесиме следните линии:

- `hostssl all dialogue 172.0.0.1/32 cert clientcert=1`
- `hostnossl all all 0.0.0.0/0 reject`

Со последните две линии кажуваме дека доколку имаме клиент со IP:172.0.0.1/32 сака да се конектира со базата *dialogue* да му побара сертификат, сите останати клиенти кои немаат сертификат да бидат одбиени. По извршените промени треба да го рестартираме Postgres серверот со командата:

- `cd /etc/init.d`
- `sudo ./postgresql restart`

Следно што треба да направиме е да го поврземе Postgres драјверот во *server.js* со самиот Postgres сервер.

```

1  var connStr = "localhost://sofre:garevski@localhost/dialogue?ssl=true";
2
3  pg.defaults.poolSize = 100;
4
5  pg.connect(connStr, function(err, client, done) {
6    if (err)
7      return console.log("ERROR",err);
8    console.log('CONNECTED TO POSTGRES');
9    //login user
10   app.post('/login', login(client));
11   //check new user
12   app.post('/checkNickname', checkNickname(client));
13   //add new User
14   app.post('/newUser', newUser(client));
15 } );

```

Слика 5.4 Конекција со Postgres база

Целта е да оствариме една конекција помеѓу nodejs серверот и Postgres серверот и нејзе да ја користат сите клиенти. Максималниот број на клиенти по конекција го одредуваме со променливата *pool/Size*. Во променливата *connStr* ги внесуваме *server://user:password@server/database* со параметар *ssl=true*. Го сетираме бројот на клиенти по конекција и се конектираме. Доколку настане грешка при конекцијата, во *callback* функцијата ја испишуваме. Откако ќе се оствари конекција ќе се справиме со логирање, креирање на корисник и проверка на корисничко име. Кодот од Слика 5.4 го сместуваме во *server.js* после *options* објектот. Исто така потребно е да го побараме *pg* драјверот на почеток.

```
1 var express = require('express'),
2   fs = require('fs'),
3   pg = require('pg'),
4   connect = require('connect'),
5   https = require('https');
```

Слика 5.5 Имплементирање на pg модулот

Следно потребно е да ги напишеме *login middleware*, *checkNickname middleware* и *newUser middleware*. За повикување на функциите од база ќе направиме модул *queries.js*. Така за *queries.js* ќе имаме:

```
1 function getUser (username, password) {
2   return "select to_json(getuser('"+ username + "','"+ password +')) as doc";
3 }
4
5 function createUser (user) {
6   return "select to_json(createuser('"+ user.firstname + "','"+ user.lastname + "','"+
7     user.username + "','"+ user.password + "','"+ user.gender +')) as doc";
8 }
9
10 function checkNickname (nick) {
11   return "select to_json(checkNickname('"+nick+"')) as doc";
12 }
13
14 exports.getUser = getUser;
15 exports.createUser = createUser;
16 exports.checkNickname = checkNickname;
```

Слика 5.6 queries.js модул

Исто така треба да направиме модул за Crypto.js клиентската скрипта која ќе треба да ги декриптира пратените податоци. За Crypto.js креираме модул *cryptolib.js* и го сместуваме во истиот директориум (*middlewares*).

```
1 /*
2 CryptoJS v3.1.2
3 code.google.com/p/crypto-js
4 (c) 2009-2013 by Jeff Mott. All rights reserved.
5 code.google.com/p/crypto-js/wiki/License
6 */
7 var CryptoJS=CryptoJS||function(u,l){var d={},n=d.lib={},p=function(){};s=n.Base={extend:function(
8   q=n.WordArray=s.extend({init:function(a,c){a=this.words=a||[];this.sigBytes=c!=l?c:4*a.length},toS
9   9
10  exports.CryptoJS = CryptoJS;
```

Слика 5.7 cryptolib.js модул

На Слика 5.7 не е прикажан целиот CryptoJS алгоритам. Важно е само да се забележи дека е додадена променлива *CryptoJS* и истата најдолу е експортирана така што *Crypto* алгоритмот е претворен во модул. За попрегледна работа со декриптирањето на податоците ќе направиме модул *crypto.js*. За *crypto.js* ќе имаме:

```

1 var crypto = require('./cryptolib.js');
2
3 function decrypt(user){
4     if(user.firstname)
5         var decrFname = crypto.CryptoJS.TripleDES.decrypt(user.firstname, "Fraza za firstname");
6     if(user.lastname)
7         var decrLname = crypto.CryptoJS.TripleDES.decrypt(user.lastname, "Fraza za lastname");
8     if(user.gender)
9         var decrGender = crypto.CryptoJS.TripleDES.decrypt(user.gender, "Fraza za gender");
10
11     var decrUser = crypto.CryptoJS.TripleDES.decrypt(user.username, "Fraza za user");
12     var decrPass = crypto.CryptoJS.TripleDES.decrypt(user.password, "Fraza za password");
13
14     if(user.firstname)
15         var firstname = decrFname.toString(crypto.CryptoJS.enc.Utf8);
16     if(user.lastname)
17         var lastname = decrLname.toString(crypto.CryptoJS.enc.Utf8);
18     if(user.gender)
19         var gender = decrGender.toString(crypto.CryptoJS.enc.Utf8);
20
21     var user = decrUser.toString(crypto.CryptoJS.enc.Utf8);
22     var pass = decrPass.toString(crypto.CryptoJS.enc.Utf8);
23
24     return {firstname:firstname||null, lastname:lastname||null,gender:gender||null,
25            username:user, password:pass}
26 }
27
28 exports.decrypt = decrypt;

```

Слика 5.8 crypto.js модул

Crypto.js модулот има само една функција *decrypt* која врши декриптирање на објектот *user* кој доаѓа како влезен параметар. Целта е да не го препишуваме целиот блок на функцијата *decrypt* на сите места каде е потребно туку да го дефинираме еднаш и да го користиме секаде. Моментално *decrypt* се користи само при логирање и креирање на нов корисник. Имајќи ги *crypto.js* и *login.js* можеме да го напишеме login middleware-от.

```

1 var crypto = require('./crypto.js');
2 var jwt = require('jsonwebtoken');
3 var queries = require('./queries.js');
4
5 module.exports = function(db,done) {
6     return function(req, res, next){
7         var user = req.body.username,
8             password = req.body.password;
9         var user = crypto.decrypt({username:user,password:password});
10        db.query(queries.getUser(user.username,user.password),function(err, result){
11            if(err){
12                throw err;
13            }
14            if(result.rows[0].doc.first_name)
15            {
16                var user = result.rows[0];
17                var token = jwt.sign(user, 'secret',{expiresInMinutes:60});
18                res.json({token:token});
19            }
20            else
21            {
22                res.send(401, {message:'Wrong user or password'});
23                return;
24            }
25        });
26    }
27 }

```

Слика 5.8 login middleware

Потребно е да се имплементира и *jsonwebtoken* модулот кој ќе ни послужи за добивање на токен за внесените податоци. login модулот ги декриптира најпрво корисничкото име и лозинка и ги сместува во објект *user*. Потоа од *queries* се користи *getUser* функцијата и доколку врати податоци, ги енкриптираме со *jsonwebtoken* и ги праќаме до корисникот. Во спротивно враќаме грешка 401 со порака за погрешно корисничко име или лозинка. За *checkNickname* middleware имаме:

```

1  var queries = require('./queries.js');
2
3  module.exports = function(db,done) {
4      return function(req, res, next){
5
6          var nickname = req.body.nickname;
7          db.query(queries.checkNickname(nickname),function(err, result){
8              if(err){
9                  return console.error('Error from Query:', err);
10             }
11
12             if(result.rows[0].doc.username)
13             {
14                 res.send(409,{message:"Duplicate username"});
15                 return;
16             }
17             else
18             {
19                 res.send(200,{message:"Unique username"});
20             }
21         });
22     };
23 }
24

```

Слика 5.9 checkNickname middleware

За newUser middleware-от имаме:

```

1  var crypto = require('./crypto.js')
2  var url = require('url');
3  var jwt = require('jsonwebtoken');
4  var queries = require('./queries.js');
5
6
7  module.exports = function(db) {
8      return function(req, res, next) {
9
10         var username = req.body.username,password = req.body.password,
11             firstname = req.body.firstname,lastname = req.body.lastname,
12             gender = req.body.gender;
13
14         var user = crypto.decrypt({firstname: firstname,lastname: lastname,
15             username: username,password: password,gender: gender
16         });
17
18         db.query(queries.createUser(user), function(err, result) {
19             if (err)
20                 return console.error('Error from Query:', err);
21
22             if (result.rows[0].doc.first_name) {
23                 var user = result.rows[0];
24                 var token = jwt.sign(user, 'secret', {
25                     expiresInMinutes: 60
26                 });
27                 res.json({token: token});
28             } else {
29                 res.send(401, {message: 'Wrong user or password'});
30                 return;
31             }
32         });
33     };
34 }

```

Слика 5.10 newUser middleware

По најавата или креирањето на нов корисник потребно е да ги земеме корисничките податоци и да му ги препратиме на клиентот. Ова ќе го изведе со имплементирање на *userData* middleware на страна на серверот.


```

var url = require('url');
var jwt = require('jsonwebtoken');

module.exports = function () {
  return function(req, res, next){
    var url_parts = url.parse(req.url,true);
    var token = url_parts.query.token;

    jwt.verify(token,'secret',function(err, decoded){
      if(err){
        res.send(401,{ 'error': 'Session expired' });
      }
      else
      {
        res.send(200,decoded);
      }
    });
  };
}

```

Слика 5.11 userData middleware

Потребно е со NPM да го инсталираме *url* модулот кој ќе ни послужи за преземање на параметрите дадени во урл барањето. Потоа со *verify* функцијата го верифицираме пратениот токен, и праќаме одговор до клиентот. Со оваа функција не земаме податоци од база туку само го декодираме токенот. Слично на ова е и *verify* middleware-от кој враќа назад информација само за валиден или невалиден токен. Тоа е прикажано на Слика 5.12.

```

1  var jwt = require('jsonwebtoken');
2
3  module.exports = function () {
4    return function(req, res, next){
5      var token = req.body.token;
6      jwt.verify(token,'secret',function(err, decoded){
7        if(err){
8          res.send(401,{ 'error': 'Session expired' });
9        }
10       else
11       {
12         res.send(200,{ 'session': 'ok' });
13       }
14     });
15   }
16 }

```

Слика 5.12 verify middleware

Исто така потребно е да ги додадеме направените модули како зависности и да ги имплементираме во *server.js*.

```

38  app.use(connect.bodyParser());
39
40  //verify user
41  app.post('/verify', verify());
42  //get UserData
43  app.get('/userData', userData());

```

Слика 5.13 Имплементација на verify и userData middleware-ите

6. SOCKET.IO

Досега имплементиравме `https`, автентикација, верификација на токен и безбедносна комуникација со `Postgres` сервер. Следно што треба да направиме е кога двајца клиенти ќе се логираат на системот да може да комуницираат помеѓу себе. Ова ќе го имплементираме со `socket.io` модулот кој ќе го инсталираме со `NPM`.

- `sudo npm install socket.io --save`

Најпрво потребно е да го иницијализираме `socket.io` на клиентска и серверска страна. Кај серверот треба да се повзе со постоечкиот сервер, да слуша на конекција и да ги зачувува сокетите кои се конектирале. Кај клиентот потребно е да се поврземе со серверот и да му пратиме податоци за клиентот а назад да добиеме информација за активни конекции. Ова е прикажано на Слика 6.1.

```

65 io.sockets.on('connection',function(socket){
66
67     socket.on('set user', function(data, callback) {
68         if (!storedSockets[data.user.username]) {
69             socket.user = data.user;
70             //var nickname = data.user.username;
71             socket.emit('current user', {
72                 'username': socket.user.username,
73                 'users': getUsers(socket)
74             });
75
76             storedSockets[socket.user.username] = socket;
77
78             var users = Object.keys(storedSockets).filter(function(key) {
79                 return key != socket.user.username;
80             });
81
82             users.forEach(function(user) {
83                 storedSockets[user].emit('new user', {
84                     'currentuser': user,
85                     'newuser': socket.user.username,
86                     'users': getUsers(storedSockets[user])
87                 });
88             });
89
90         } else {
91             socket.user = data.user;
92             socket.emit('current user', {
93                 'username': socket.user.username,
94                 'users': getUsers(socket)
95             });
96         }
97     });
98 });

```

Слика 6.1 Серверски код

При конектирање на клиент се емитира `'set user'` настан кој потоа се управува на сервер. Најпрво во `socket.user` се зачувуваат податоците кои се праќаат од клиентот. Доколку сокетот не постои во `storedSockets` променливата серверот емитира настан `'current user'` и на истиот сокет му враќа објект со `username` и активни корисници. Исто така во променливата `users` ги издвојува активните конектирани клиенти, а потоа на сите им емитира настан `'new user'`. Доколку сокетот веќе постои во `storedSockets` променливата само му враќа информации за активните корисници.

Кај серверот се користи функцијата *getUsers* која како влезен параметар прима сокет а на излез ги враќа сите сокети од *storedSockets* освен параметарот.

```

52 var getUsers = function(socket) {
53     var users = [];
54     var keys = Object.keys(storedSockets).filter(function(key) {
55         return key !== socket.user.username;
56     });
57
58     keys.forEach(function(key) {
59         users.push(storedSockets[key].user);
60     });
61     //console.log('USERS', users);
62     return users;
63 };

```

Слика 6.2 getUsers функцијата

На клиентска страна потребно е најпрво да ја вклучиме socket.io скриптата во *index.html* и да креираме објект *chatHistory* во кој подоцна ќе ја чуваме историјата.

```

14 <script type="text/javascript" src="scripts/angular-resource.min.js"></script>
15 <script type="text/javascript" src="/socket.io/socket.io.js"></script>

```

Слика 6.3 Имплементирање на socket.io

Конектирањето и интеракциите со серверот ќе бидат напишани во *main.js* датотеката.

```

25 $scope.socket = io.connect('https://localhost',{secure:true,port:3000});
26
27 $scope.socket.on('connect', function() {
28
29     $scope.socket.emit('set user', {
30         user: $scope.user
31     });
32
33     $scope.socket.on('current user', function(data) {
34         $scope.user.friends = data.users;
35         data.users.forEach(function(user) {
36             if (!$scope.chatHistory[user.username]) {
37                 $scope.chatHistory[user.username] = {};
38             }
39         });
40         if (!$scope.$$phase)
41             $scope.$apply();
42     });
43
44     $scope.socket.on('new user', function(data) {
45         $scope.user.friends = data.users;
46         console.log('NEW', data);
47
48         if (!$scope.chatHistory[data.newuser]) {
49             $scope.chatHistory[data.newuser] = {};
50         }
51
52         if (!$scope.$$phase)
53             $scope.$apply();
54     });
55
56 });

```

Слика 6.4 Клиентски код

Конекцијата помеѓу клиентот и серверот е безбедна, односно ќе бидат употребени серверските сертификат и клуч за заштита на сообраќајот. На конекција се емитира 'set user' настанот. Исто како и кај серверот се слуша на 'current user' и на 'new user'

настаните. *Current user* настанот има намена да го информира сокетот кој се конектира за тековните активни сокети. Исто така секој клиент ќе чува локално историја на разговор со секој од останатите во објектот *chatHistory*. Следно што треба да направиме е да може корисникот да праќа и да прима пораки. За праќање на пораки ќе ја користиме функцијата *sendMessage* која ќе емитира настан 'new message' и исто така локално ќе ја запише пораката во *chatHistory* за соодветниот клиент. Ова е прикажано на Слика 6.5. За примање на пораките ќе треба да слушаме исто на 'new message' настанот Слика 6.6.

```

56     $scope.sendMessage = function() {
57         if ($scope.selectedUser) {
58             $scope.socket.emit('new message', {
59                 'user': $scope.user,
60                 'message': $scope.message,
61                 'to': $scope.selectedUser,
62                 'date': Date.now()
63             });
64             if (!$scope.chatHistory[$scope.selectedUser.username].messages)
65                 $scope.chatHistory[$scope.selectedUser.username].messages = [];
66             $scope.chatHistory[$scope.selectedUser.username].messages.push({
67                 'user': $scope.user,
68                 'message': $scope.message,
69                 'to': $scope.selectedUser,
70                 'date': Date.now()
71             });
72         }
73         $scope.message = "";
74     } else {
75         alert("Please select user");
76     }
77 }

```

Слика 6.5 sendMessage функција за емитирање на нова порака

```

79     $scope.socket.on('new message', function(data) {
80
81         if (!$scope.chatHistory[data.user.username]) {
82             $scope.chatHistory[data.user.username] = {};
83         }
84
85         if (!$scope.selectedUser || $scope.selectedUser.username != data.user.username)
86             $scope.chatHistory[data.user.username].seen = true;
87
88         $scope.chatHistory[data.user.username].messages.push(data);
89
90         if (!$scope.$$phase)
91             $scope.$apply();
92     });

```

Слика 6.6 Слушање на new message настан

Следно што треба да направиме е функција во *main.js* која кога корисникот ќе кликне врз некој од конектираните клиенти ќе му ја прикаже локално зачуваната историја на пораки.

```

119     $scope.selectUser = function(user) {
120         $scope.selectedUser = user;
121
122         $scope.chatHistory[$scope.selectedUser.username].seen = false;
123     }

```

Слика 6.7 selectUser функција за селектирање на клиент

7.MONGODB

MongoDB претставува нерелациона база која се карактеризира со солидни перформанси. Во нашиот систем ќе ја користиме за преземање на историја на пораките помеѓу двајца клиенти. Исто така ќе треба да овозможиме SSL при конекција. Стандардната верзија на Mongo не вклучува SSL, туку ќе треба или локално да го компајлираме Mongo или да користиме MongoDB Enterprise. Во нашиот случај ќе го компајлираме Mongo локално. Пред да започнеме со компајлирањето треба да ги овозможиме потребните предуслови:

- build-essential — Ubuntu пакет кој инсталира често користени програми потребни за правење на софтвер
- scons — Scons претставува алтернатива на "Make" алатката, која се користи од MongoDB
- git-core — систем за контрола на верзии кој ќе го користиме за преземање на MongoDB изворниот код
- libssl-dev — потребни ќе ни бидат девелоперските фајлови кои одат заедно со SSL споделената библиотека, со цел да ја додадеме оваа поддршка на MongoDB
- various boost libs — портабилни C++ изворни библиотеки

Ова можеме да го направиме со извршување на следните команди:

- aptitude update && aptitude dist-upgrade -y && reboot
- aptitude install build-essential scons
- aptitude install git-core libssl-dev
- aptitude install libboost-filesystem-dev
- aptitude install libboost-program-options-dev
- aptitude install libboost-system-dev libboost-thread-dev -y

Следно што треба да направиме е да го преземеме изворниот код за MongoDB и да одбереме која верзија да ја инсталираме. Тоа го правиме со следните команди:

- cd /usr/src
- git clone git://github.com/mongodb/mongo.git
- cd mongo
- git tag -l | grep -v rc
- git checkout r2.7.1

Потоа компајлираме со командите:

- scons -- ssl all
- scons --ssl --prefix=/usr install

По компајлирањето можно е да се појави следната грешка:

```
scons: *** [build/linux2/ssl/mongo/lame_stacktrace_test] Error 1
scons: building terminated because of errors.
```

Слика 7.1 Грешка при компајлирање

Потребно е да ја извршиме постинсталациската скрипта која го сетира сметка (корисник) но исто така прави *data* и *log* директориуми.

- `cd debian`
- `chmod +x postinst`
- `./postinst configure`

Откако ќе се конфигурира потребно е да може при рестартирање на компјутерот да се активира серверот. Ова се прави со конфигурацискиот фајл *mongodb.conf* кој ќе го добиеме со командите:

- `cp mongodb.upstart /etc/init/mongodb.conf`
- `cp mongodb.conf /etc`

Исто така треба да го копираме конфигурацискиот фајл за лог ротација. Лог ротација е процес на автоматско архивирање на лог фајлови. За да имаме контрола над MongoDB лог фајловите ја конфигурираме *logrotate* програмата за управување.

- `nano /etc/logrotate.d/mongodb-server`

Го додаваме следниот код:

```
/var/log/mongodb/*.log {
    weekly
    rotate 10
    copytruncate
    delaycompress
    compress
    notifempty
    missingok
}
```

Слика 7.2 Конфигурирање на logrotate

Доколку конфигурацискиот фајл за autostart на *mongod* процесот не успее тогаш ќе треба мануелно да го подигаме *mongod* процесот. Ова ќе биде објаснето подоцна. Следно што треба да направиме е да направиме *.pem* фајл од клучот и сертификатот на Postgres базата со следната команда:

- `cat client.key client.crt > mongodb.pem`

Ги копираме `mongodb.pem` и `server.crt` во `/etc/ssl` и правиме промени во `/etc/mongodb.conf` фајлот со командите:

- `nano /etc/mongodb.conf`
- `sslMode = requireSSL`
- `sslPEMKeyFile = /etc/ssl/mongodb.pem`
- `sslCAFile = /etc/ssl/server.crt`

За подигање на `mongod` процесот мануелно ја извршуваме следната команда:

- `mongod --sslMode requireSSL --sslPEMKeyFile /etc/ssl/mongodb.pem --sslCAFile /etc/ssl/server.crt`

За конектирање како клиент ја извршуваме следната команда:

- `mongo --ssl --sslPEMKeyFile /etc/ssl/mongodb.pem`

Следно што треба да направиме е да го имплементираме *mongoose* драјверот за работа со MongoDB преку Nodejs. Тоа го правиме со NPM, а потоа и го додаваме како зависност во `server.js` фајлот. Исто така ќе треба да ги ископираме `client.crt` и `client.key` од `/etc/ssl/certs/` и `/etc/ssl/private/` соодветно во самиот проектен директориум. Следно треба да направиме база на која ќе се конектираме со некое корисничко име и лозинка. Тоа ќе го постигнеме со следниве неколку команди:

- `mongo --ssl --sslPEMKeyFile /etc/ssl/client.pem`
- `use dialogue`
- `db.createUser({user:"sofre", pwd:"garevski", roles:[{role: "userAdmin", db:"dialogue"}]})`

Во `server.js` го имплементираме следниот код:

```

21 var optionsMongo = {
22   server:{
23     ssl: true,
24     sslKey:fs.readFileSync('client.key'),
25     sslCert:fs.readFileSync('client.crt')
26   },
27   user:'sofre',
28   password:'garevski',
29   auth:true
30 };
31
32 mongoose.connect('mongodb://localhost/dialogue',optionsMongo,function(err){
33   if(err)
34     return console.log('ERROR',err);
35   console.log('CONNECTED TO MONGO');
36 });
37
38 var dbMongo = mongoose.connection;
```

Слика 7.3 Конекција со MongoDB

Ако го стартуваме серверот ќе треба да ни испише дека успешно сме се конектирале на серверот. Следно што треба да направиме е да ги дефинираме полињата кои ќе ги содржи колекцијата во која ќе се запишуваат пораките. Покрај пораките ќе биде потребно да водиме евиденција за корисниците кои ги имаме

регистрирано. За тоа ќе дефинираме две шеми (schema) во *schemas.js* (middlewares), *messages* и *users* со следниот код:

```

1  var mongoose = require('mongoose');
2
3  var messageSchema = mongoose.Schema({
4    idSender: Number,
5    idReceiver: Number,
6    message: String,
7    date: { type: Date, default: Date.now }
8  });
9
10 var usersSchema = mongoose.Schema({
11   id_user: Number,
12   username: String,
13   password: String,
14   first_name: String,
15   last_name: String,
16   sex: Boolean,
17   imageurl: String
18 });
19
20 var messageModel = mongoose.model('message', messageSchema);
21 var userModel = mongoose.model('usersModel', usersSchema);
22
23 exports.messageSchema = messageSchema;
24 exports.messageModel = messageModel;
25
26 exports.usersSchema = usersSchema;
27 exports.usersModel = userModel;

```

Слика 7.4 Дефиниција на шеми за корисници и пораки

Следно што треба да направиме е кога ќе пристигне нова порака да ја зачува во *messages* колекцијата и да може да врати историја на пораки за двајца корисници. Затоа треба претходниот цел код за сокетите да биде вметнат во callback функцијата од Mongo. За таа цел го имаме следниот код:

```

86 dbMongo.once('open', function() {
87   var messageModel = schemas.messageModel;
88   var message = null;
89
90   app.post('/getMessageHistory', getMessageHistory(messageModel));
91
92   io.sockets.on('connection', function(socket){
93
94     socket.on('set user', function(data, callback) {

```

Слика 7.5 Callback функција

MessageModel претставува моделот генериран од *MessageSchema* шемата. *Message* ќе биде подоцна инстанца од *messageModel* моделот. Емитирањето на нова порака ќе биде во callback функцијата од методот за снимање. Тоа е прикажано на Слика 7.6 а на Слика 7.7 е даден модулот *getMessageHistory* за земање на историја на разговори за двајца клиенти.

```

127         socket.on('new message', function(data) {
128             message = new messageModel(
129                 {
130                     'idSender':data.user.id_user,
131                     'idReceiver':data.to.id_user,
132                     'message':data.message,
133                     'date':data.date
134                 });
135
136             message.save(function(err,msg){
137                 if(err)
138                     console.log('ERR SAVE:',err);
139
140                 storedSockets[data.to.username].emit('new message', data);
141             });
142         });

```

Слика 7.6 new message настан

```

1  module.exports = function(msgModel) {
2
3      return function(req, res, next) {
4
5          var idSender = req.body.idSender;
6          var idReceiver = req.body.idReceiver;
7
8          console.log('ID_SENDER:', idSender);
9          console.log('ID_RECEIVER:', idReceiver);
10
11          msgModel.find({$or:[{"idSender":idSender,"idReceiver":idReceiver},|
12              {"idSender":idReceiver,"idReceiver":idSender}]},{"_id":false},function (err,data) {
13              if(err)
14              {
15                  console.log('ERR',err);
16              }
17              console.log('DATA',data);
18              res.json(200,{"data":data});
19          });
20      }
21  }

```

Слика 7.7 getMessageHistory модул

```

166     $scope.selectUser = function(user) {
167         $scope.selectedUser = user;
168
169         if (!$scope.chatHistory[$scope.selectedUser.username].messages) {
170             $http.post('/getMessageHistory', {
171                 idSender: $scope.user.id_user,
172                 idReceiver: $scope.selectedUser.id_user
173             }).success(function(data) {
174                 console.log('HISTORY', data);
175                 $scope.chatHistory[$scope.selectedUser.username].messages =
176                     data.data.map(function(elem, index) {
177                         var result = {
178                             "id_user":elem.idReceiver == $scope.selectedUser.id_user ?
179                                 $scope.selectedUser.id_user : $scope.user.id_user,
180                             "message": elem.message,
181                             "date": elem.date,
182                             "user": {
183                                 "first_name": elem.idSender == $scope.selectedUser.id_user ?
184                                     $scope.selectedUser.first_name : $scope.user.first_name,
185                                 "last_name": elem.idSender == $scope.selectedUser.id_user ?
186                                     $scope.selectedUser.last_name : $scope.user.last_name,
187                                 "imageurl": elem.idSender == $scope.selectedUser.id_user ?
188                                     $scope.selectedUser.imageurl : $scope.user.imageurl
189                             }
190                         };
191                     });
192             return result;
193         });
194     });
195 }
196 $scope.chatHistory[$scope.selectedUser.username].seen = false;
197 }

```

Слика 7.8 selectUser функција

Потребно е да направиме промени и на клиентска страна. При селектирање на некој клиент треба да се земе историјата на разговори. Исто така треба кога ќе пристигне нова порака, а немаме историја зачувано локално, повторно да се преземе историјата од сервер. Тоа е прикажано на Слика 7.8 и Слика 7.9.

И во двете функции важно е да се направи мапирање на пораките со цел да се знае кој е испраќачот а кој примачот на пораката.

```

113     $scope.socket.on('new message', function(data) {
114         console.log('DATA', data);
115         if (!$scope.chatHistory[data.user.username]) {
116             $scope.chatHistory[data.user.username] = {};
117         }
118         if (!$scope.selectedUser || $scope.selectedUser.username != data.user.username)
119             $scope.chatHistory[data.user.username].seen = true;
120         if (!$scope.chatHistory[data.user.username].messages){
121             $http.post('/getMessageHistory', {
122                 idSender: data.user.id_user,
123                 idReceiver: data.to.id_user
124             }).success(function(dataSrv) {
125                 $scope.chatHistory[data.user.username].messages =
126                     dataSrv.data.map(function(elem, index) {
127                         var result = {
128                             "id_user": elem.idReceiver == data.to.id_user ?
129                             data.to.id_user : data.user.id_user,
130                             "message": elem.message,
131                             "date": elem.date,
132                             "user": {
133                                 "first_name": elem.idSender == data.to.id_user ?
134                                 data.to.first_name : data.user.first_name,
135                                 "last_name": elem.idSender == data.to.id_user ?
136                                 data.to.last_name : data.user.last_name,
137                                 "imageurl": elem.idSender == data.to.id_user ?
138                                 data.to.imageurl : data.user.imageurl
139                             }
140                         };
141                     });
142                     return result;
143             });
144         });
145     }
146     else{
147         $scope.chatHistory[data.user.username].messages.push(data);
148     }
149 }

```

Слика 7.9 new message настан

ЗАКЛУЧОК

Кога се работи за веб апликации преносот на податоци помеѓу два или повеќе страни се одвива незаштитено. Од клучно значење е преносот на информации да биде заштитен, посебно за информации кои се важни. За таа цел се користат криптирачки алгоритми кои даваат одредено ниво на гаранција за безбедноста на информациите. Користењето на SSL претставува најчесто користен начин за заштита не само помеѓу сервер и клиент, туку и помеѓу два сервери (апликациски сервер и сервер на кој се наоѓа базата).

ЛИТЕРАТУРА

- Smashing NodeJS – Guillermo Rauch
- MongoDB Documentation - <http://docs.mongodb.org/manual/>
- AngularJS Documentation - <https://docs.angularjs.org/api>
- <http://www.mongodb.org/about/contributors/tutorial/build-mongodb-from-source/>
- <http://www.postgresql.org/docs/9.1/static/sql-alterdatabase.html>
- <http://www.postgresql.org/docs/9.0/static/sql-alterrole.html>
- <http://www.postgresql.org/download/linux/ubuntu/>
- <http://www.kdelemme.com/2014/03/09/authentication-with-angularjs-and-a-node-js-rest-api/>
- <https://vickev.com/#!/article/authentication-in-single-page-applications-node-js-passportjs-angularjs>
- <http://djs4rce.wordpress.com/2013/08/13/understanding-angular-http-interceptors/>
- <http://www.webdeveasy.com/interceptors-in-angularjs-and-useful-examples/>
- http://www.akadia.com/services/ssh_test_certificate.html
- <http://ubuntuforums.org/showthread.php?t=2110429>
- <https://help.ubuntu.com/community/OpenSSL>
- <http://www.hacksparrow.com/mongodb-add-users-and-authenticate.html>
- <https://github.com/brianc/node-postgres/wiki/pg>
- <http://vibhorkumar.wordpress.com/2011/07/17/how-to-enable-ssl-in-postgresqlppas/>
- <https://github.com/brianc/node-postgres/wiki/pg>
- <http://stackoverflow.com/questions/12087683/postgresql-wont-start-server-key-has-group-or-world-access>
- <http://stackoverflow.com/questions/18497299/psql-fatal-connection-requires-a-valid-client-certificate>
- <http://stackoverflow.com/questions/16758396/how-do-i-edit-the-mongodb-conf-file>
- <http://gravitronic.com/compiling-mongodb-with-ssl-support-on-ubuntu-12-04-lts/>
- <http://docs.mongodb.org/manual/tutorial/configure-ssl/>
- <http://mongoosejs.com/docs/3.4.x/docs/api.html>
- <http://stackoverflow.com/questions/6599470/node-js-socket-io-with-ssl>