Министерство цифрового развития, связи и массовых коммуникаций РФ
Уральский технический институт связи и информатики (филиал) ФГБОУ ВО
"Сибирский государственный университет телекоммуникаций и информатики" в г. Екатеринбурге
(УрТИСИ СибГУТИ)

КАФЕДРА
ИСТ

ОТЧЕТ

По дисциплине «Сетевое программирование»
Практическое занятие № 3
«Работа с авторизацией и аутентификацией»

Выполнил: студент гр. ПЕ-12б
Камков Д.А.

Проверил: Ст.преп.,
Бурумбаев Д.И.

Ассистент:

Екатеринбург, 2024

1 Цель работы:

1.1 Научиться работать с регистрацией, авторизацией и аутентификацией;

1.2 Закрепить знания по теме «Аутентификация и авторизация пользователей в клиент-серверных приложениях».

2 Перечень оборудования:

22.1 Персональный компьютер;

2.2 Postman;

2.3 node.js;

2.4 Visual Studio Code..

3.Ход работы:

3.1 Для реализации системы регистрации, авторизации и аутентификации необходимо выполнить ряд действий. Первым делом необходимо настроить базу данных. Для примера, можно использовать базу данных MySQL при помощи XAMPP, WAMP или другого ПО. Внутри необходимо создать базу данных familia_login. Затем внутри этой базы данных нужно создать таблицу с именем users при помощи SQL-запроса, который представлен в листинге 1.

Листинг 1 – SQL-запрос для создания таблицы

```
CREATE TABLE users ( id int(10) unsigned NOT NULL AUTO_INCREMENT, name varchar(50) COLLATE utf8mb4_unicode_ci NOT NULL, email varchar(50) COLLATE utf8mb4_unicode_ci NOT NULL, password varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL, PRIMARY KEY (id), UNIQUE KEY email (email) ) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

Далее в качестве примера представлена реализация при помощи node.js и JavaScript. Для начало необходимо установить NPM при помощи команды:

Листинг 2 – Инициализация NPM

```
npm i express ejs express-session express-validator bcryptjs mysql2
```

А также изменить созданный файл package.json

Листинг 3 – Package.json

```
{
  "dependencies": {
    "axois": "^0.0.1-security",
    "bcryptjs": "^2.4.3",
    "cookie-session": "^2.1.0",
    "ejs": "^3.1.10",
    "express": "^4.19.2",
```

```
    "express-session": "^1.18.0",
    "express-validator": "^7.0.1",
    "mysql2": "^3.9.4",
    "nodemon": "^3.1.0",
    "passport": "^0.7.0",
    "passport-github2": "^0.1.12",
    "passport-google-oauth2": "^0.2.0"
  },
  "name": "rol",
  "version": "1.0.0",
  "main": "index.js",
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "dev": "nodemon index.js"
  },
  "author": "",
  "license": "ISC",
  "description": ""
}
```

Далее необходимо создать структуру для реализации системы как показано на рисунке 1.

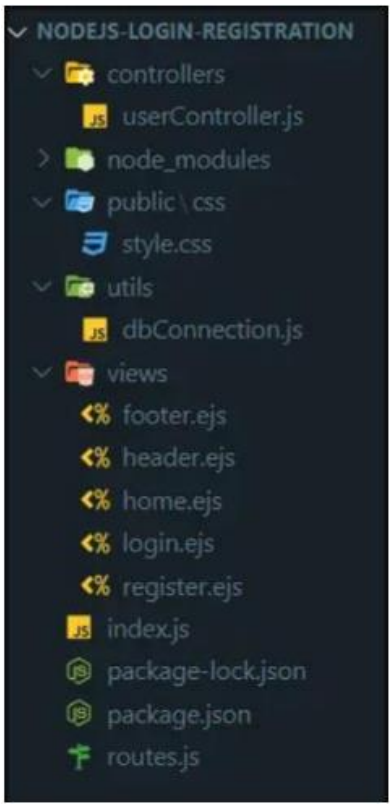

Рисунок 1 – Структура проекта

Далее для подключения базы данных к проекту необходимо в файле dbConnection.js необходимо прописать код, представленный в листинге 4.

Листинг 4 – Файл dbConnectrion.js

```
const mysql = require('mysql2');
const dbConnection = mysql.createPool({
   host: 'localhost',
   user: 'root',
   password: 'root',
   database: 'kamkov_login'
});
module.exports = dbConnection.promise();
```

Далее необходимо наполнить каталог views. Описанные далее файлы будут находиться в этой папке. Содержимое файлов представлено в листингах 5-10.

Листинг 5 – Код в файле header.ejs

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8"/>
   <meta http-equiv="X-UA-Compatible" content="IE=edge"/>

   <meta
         name="viewport"
         content="width=device-width, initial-
scale=1.0"
   />

   <title>
      <%= (typeof (title) !== 'undefined') ? title : 'Node JS Login and Registration
System - W3jar' %>
   </title>
   <link rel="preconnect" href="https://fonts.gstatic.com"/>
   <link
href="https://fonts.googleapis.com/css2?family=Ubuntu:wght@400;700&display=
swap"
         rel="stylesheet"
   />
   <link rel="stylesheet" href="./style.css"/>
</head>
<body></body>
</html>
```

Листинг 6 – Код в файле footer.ejs

```
</body>
</html>
```

Листинг 7 – Код в файле login.ejs

```
<%- include('header',{ title:'Login' }); -%>
<div class="container">
   <h1>Login</h1>
   <form action="" method="POST">
      <label for="user_email">Email</label>
      <input
            type="email"
            class="input"
            name="_email"
            id="user_email"
            placeholder="Enter your email"
      />
      <label for="user_pass">Password</label>
      <input
            type="password"
            class="input"
            name="_password"
            id="user_pass"
            placeholder="Enter new password"
      />
      <% if(typeof error !== 'undefined'){ %>
         <div class="err-msg"><%= error %></div>
      <% } %>
      <input type="submit" value="Login">
      <div class="link"><a href="./auth/google">Google</a></div>
      <div class="link"><a href="./auth/github">GitHub</a></div>

   </form>
</div>
<div class="link"><a href="./signup">Sign Up</a></div>
</form>
</div>
<%- include('footer'); -%>
```

Листинг 8 – Код в файле register.ejs

```
<%- include('header', { title:'Signup' }); -%>
<div class="container">
   <h1>Sign Up</h1>
   <form action="" method="POST">
      <label for="user_name">Name</label>
      <input
            type="text"
            class="input"
            name="_name"
            id="user_name"
            placeholder="Enter your name"
      />
      <label for="user_email">Email</label>
      <input
            type="email"
            class="input"
            name="_email"
            id="user_email"
            placeholder="Enter your email"
      />
      <label for="user_pass">Password</label>
      <input
            type="password"
            class="input"
            name="_password"
            id="user_pass"
            placeholder="Enter new password"
      />
      <% if(typeof msg != 'undefined'){ %>
         <div class="success-msg"><%= msg %></div>
      <% } if(typeof error !== 'undefined'){ %>
         <div class="err-msg"><%= error %></div>
      <% } %>
      <input type="submit" value="Sign Up"/>
      <div class="link"><a href="./login">Login</a></div>
   </form>
</div>
<%- include('footer'); -%>
```

Листинг 9 – Код в файле home.ejs

```
<%- include('header'); -%>
<div class="container">
    <div class="profile">
        <div class="img">👨</div>
        <h2><%= user.name %></h2>
        <span><%= user.email %></span>
        <a href="/logout">Log Out</a>
    </div>
</div>
<%- include('footer'); -%>
```

Листинг 10 – Код в файле routes.js

```
const router = require("express").Router();
const passport = require("passport");
const {body} = require("express-validator");
const {
    homePage,
    register,
    registerPage,
    login,
    loginPage,
} = require("./controllers/userController");

const ifNotLoggedin = (req, res, next) => {
    console.log("NotLogged");
    if (!req.session.userID) {
        return res.redirect("/login");
    }
    next();
};
const ifLoggedin = (req, res, next) => {
    if (req.session.userID) {
        return res.redirect("/");
    }
    next();
};

//GOGLE AUTH
const dbConnection = require("./utils/dbConnection");

function isLoggedIn(req, res, next) {
    req.user ? next() : res.sendStatus(401);
```

```
}

router.get("/auth/google", passport.authenticate("google"), (req, res) =>
   res.send(200)
);

router.get("/auth/google/callback", (req, res) => {
   passport.authenticate("google", {
      successRedirect: "/auth-success",
      failureRedirect: "/auth-error",
   })(req, res);
});

router.get("/auth-success", isLoggedIn, async (req, res) => {
   console.log("Auth-suc");
   console.log("reqsess" + req.session);
   const [row] = await dbConnection.execute(
      "SELECT * FROM `users`WHERE `email`=?",
      [req.user.email]
   );
   req.session.userID = row[0].id;
   res.redirect("/");
});

router.get("/auth-error", (req, res) => {
   res.send("Something went wrong");
});

//Github auth
router.get("/auth/github", passport.authenticate("github"), (req, res) => {
   console.log("git");
   res.send(200);
});

router.get("/auth/github/callback", (req, res) => {
   passport.authenticate("github", {
      successRedirect: "/git/auth-success",
      failureRedirect: "/git/auth-failure",
   })(req, res);
});

router.get("/git/auth-success", isLoggedIn, async (req, res) => {
   console.log("Auth-suc");
   console.log("reqsess" + req.session);
   const [row] = await dbConnection.execute(
```

```
        "SELECT * FROM `users`WHERE `email`=?",
        [req.user.username]
    );
    req.session.userID = row[0].id;
    res.redirect("/");
});

router.get("/git/auth-failure", (req, res) => {
    res.send("Something went wrong");
});

//LOGIN
router.get("/", ifNotLoggedin, homePage);
router.get("/login", ifLoggedin, loginPage);
router.post(
    "/login",
    ifLoggedin,
    [
        body("_email", "Invalid email address")
            .notEmpty()
            .escape()
            .trim()
            .isEmail(),
        body("_password", "The Password must be of minimum 4characters length")
            .notEmpty()
            .trim()
            .isLength({min: 4}),
    ],
    login
);

//REGISTER
router.get("/signup", ifLoggedin, registerPage);
router.post(
    "/signup",
    ifLoggedin,
    [
        body("_name", "The name must be of minimum 3 characterslength")
            .notEmpty()
            .escape()
            .trim()
            .isLength({min: 3}),
        body("_email", "Invalid email address")
            .notEmpty()
            .escape()
```

```
        .trim()
        .isEmail(),
    body("_password", "The Password must be of minimum 4characters length")
        .notEmpty()
        .trim()
        .isLength({min: 4}),
  ],
  register
);

//LOGOUT
router.get("/logout", (req, res, next) => {
  req.session.destroy((err) => {
    next(err);
  });
  res.redirect("/login");
});
module.exports = router;
```

Далее необходимо в корневом каталоге приложения создать папку controllers и внутри этой папки создать файл userController.js, содержание которого представлено в листинге 11.

Листинг 11 – Код в файле userController.ejs

```
const {validationResult} = require("express-validator");
const bcrypt = require("bcryptjs");
const dbConnection = require("../utils/dbConnection");

// Home Page
exports.homePage = async (req, res, next) => {
  const [row] = await dbConnection.execute(
    "SELECT * FROM `users`WHERE `id`=?",
    [req.session.userID]
  );
  if (row.length !== 1) {
    return res.redirect("/logout");
  }
  res.render("home", {
    user: row[0],
  });
};
// Register Page
exports.registerPage = (req, res, next) => {
  res.render("register");
};
```

```
// User Registration
exports.register = async (req, res, next) => {
    console.log("reg");
    const errors = validationResult(req);
    const {body} = req;
    if (!errors.isEmpty()) {
        return res.render("register", {
            error: errors.array()[0].msg,
        });
    }
    try {
        const [row] = await dbConnection.execute(
            "SELECT * FROM `users` WHERE `email`=?",
            [body._email]
        );
        if (row.length >= 1) {
            return res.render("register", {
                error: "This email already in use.",
            });
        }
        const hashPass = await bcrypt.hash(body._password, 12);
        const [rows] = await dbConnection.execute(
            "INSERT INTO `users`(`name`,`email`,`password`)VALUES(?,?,?)",
            [body._name, body._email, hashPass]
        );
        if (rows.affectedRows !== 1) {
            return res.render("register", {
                error: "Your registration has failed.",
            });
        }
        res.render("register", {
            msg: "You have successfully registered.",
        });
    } catch (e) {
        next(e);
    }
};
// Login Page
exports.loginPage = (req, res, next) => {
    res.render("login");
};
// Login User
exports.login = async (req, res, next) => {
    const errors = validationResult(req);
    const {body} = req;
```

```
    if (!errors.isEmpty()) {
        return res.render("login", {
            error: errors.array()[0].msg,
        });
    }
    try {
        console.log("try");
        const [row] = await dbConnection.execute(
            "SELECT * FROM`users` WHERE `email`=?",
            [body._email]
        );
        if (row.length != 1) {
            return res.render("login", {
                error: "Invalid email address.",
            });
        }
        const checkPass = await bcrypt.compare(body._password, row[0].password);
        if (checkPass === true) {
            req.session.userID = row[0].id;
            return res.redirect("/");
        }
        res.render("login", {
            error: "Invalid Password.",
        });
    } catch (e) {
        next(e);
    }
};
```

Для добавления стилистики нашего приложения необходимо добавить файл style.css внутри общей папки приложения. Код представлен в листинге 12.

Листинг 12 – Код в файле style.css

```
*,
*::before,
*::after {
    box-sizing: border-box;
}

html {
    -webkit-text-size-adjust: 100%;
    -webkit-tap-highlight-color: rgba(0, 0, 0, 0);
    font-size: 16px;
}
```

```
body {
    background-color: #f7f7f7;
    font-family: "Ubuntu", sans-serif;
    margin: 0;
    padding: 0;
    color: #222222;
    overflow-x: hidden;
    overflow-wrap: break-word;
    -moz-osx-font-smoothing: grayscale;
    -webkit-font-smoothing: antialiased;
    padding: 50px;
}

.container {
    background-color: white;
    max-width: 450px;
    margin: 0 auto;
    padding: 40px;
    box-shadow: 0 1rem 3rem rgba(0, 0, 0, 0.175);
    border-radius: 3px;
}

.container h1 {
    margin: 0 0 40px 0;
    text-align: center;
}

input,
button {
    font-family: "Ubuntu", sans-serif;
    outline: none;
    font-size: 1rem;
}

.input {
    padding: 10px;
    width: 100%;
    margin-bottom: 10px;
    border: 1px solid #bbbbbb;
    border-radius: 3px;
}

.input:hover {
    border-color: #999999;
```

```
}

.input:focus {
    border-color: #0d6efd;
}

[type="submit"] {
    background: #0d6efd;
    color: white;
    border: 1px solid rgba(0, 0, 0, 0.175);
    border-radius: 3px;
    padding: 12px 0;
    cursor: pointer;
    box-shadow: 0 0.125rem 0.25rem rgba(0, 0, 0, 0.075);
    margin-top: 5px;
    font-weight: bold;
    width: 100%;
}

[type="submit"]:hover {
    box-shadow: 0 0.5rem 1rem rgba(0, 0, 0, 0.15);
}

label {
    font-weight: bold;
}

.link {
    margin-top: 10px;
    text-align: center;
}

.link a {
    color: #0d6efd;
}

.success-msg,
.err-msg {
    color: #dc3545;
    border: 1px solid #dc3545;
    padding: 10px;
    border-radius: 3px;
}

.success-msg {
```

```
    color: #ffffff;
    background-color: #20c997;
    border-color: rgba(0, 0, 0, 0.1);
}

.profile {
    text-align: center;
}

.profile .img {
    font-size: 50px;
}

.profile h2 {
    margin-bottom: 3px;
    text-transform: capitalize;
}

.profile span {
    display: block;
    margin-bottom: 20px;
    color: #999999;
}

.profile a {
    display: inline-block;
    padding: 10px 20px;
    text-decoration: none;
    border: 1px solid #dc3545;
    color: #dc3545;
    border-radius: 3px;
}

.profile a:hover {
    border-color: rgba(0, 0, 0, 0.1);
    background-color: #dc3545;
    color: #ffffff;
}
```

Последним действием является оформление главного файла index.js, представленного в листинге 13.Последним действием является оформление главного файла index.js, представленного в листинге 13.

Листинг 13 – Код в файле index.js

```
const express = require("express");
const session = require("express-session");
const path = require("path");
const routes = require("./routes");
const app = express();

app.set("views", path.join(__dirname, "/views"));
app.set("view engine", "ejs");
app.use(express.urlencoded({extended: false}));
app.use(
   session({
      name: "session",
      secret: "my_secret",
      resave: false,
      saveUninitialized: true,
      cookie: {
         maxAge: 3600 * 1000, // 1hr
      },
   })
);


const passport = require("passport");
require("./passport");
app.use(passport.initialize());
app.use(passport.session());

app.use(express.static(path.join(__dirname, "public")));
app.use(routes);
app.use((err, req, res, next) => {
   // console.log(err);
   return res.send("Internal Server Error");
});
app.listen(4000, () => console.log("Server is running on port 4000"));
```

3.2 Далее в качестве индивидуального задания 1 необходимо проверить работоспособность полученного приложения. Результаты занести скриншотами из базы данных в отчет

3.3 В качестве индивидуального задания 2 необходимо интегрировать дополнительный способ авторизации через Google и Github.

Для авторизации через Google необходимо инициализировать дополнительные модули через команду, представленную в листинге 14.

Листинг 14 – Установка дополнительных модулей

```
npm install express passport passport-google-oauth2 cookie-session
```

Далее необходимо добавить в корневую структуру добавить файл passport.js, содержащий код, представленный в листинге 15.

Листинг 15 – Код файла passport.js

```
const passport = require("passport");
const bcrypt = require("bcryptjs");

const GoogleStrategy = require("passport-google-oauth2").Strategy;
const GitHubStrategy = require("passport-github2").Strategy;

const dbConnection = require("./utils/dbConnection");

passport.use(
   new GoogleStrategy(
      {
         clientID:"667426422941-
4gcg5sqtjos8jbagaktb2ti4bvp077tp.apps.googleusercontent.com",
         clientSecret: "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",
         callbackURL: "http://localhost:4000/auth/google/callback",
         scope: ["email", "profile"],
      },
      async function (accessToken, refreshToken, profile, done) {
         const [row] = await dbConnection.execute(
            "SELECT * FROM `users`WHERE `email`=?",
            [profile.email]
         );
         console.log(row[0]);
         console.log(row == null);
         if (row[0] == undefined) {
            await dbConnection.execute(
               "INSERT INTO `users`(`name`,`email`,`password`) VALUES(?,?,?)",
               [
                  profile.displayName,
                  profile.email,
                  await bcrypt.hash(accessToken, 12),
               ]
            );
         }
         done(null, profile);
      }
   )
);
```

```
passport.use(
   new GitHubStrategy(
      {
         clientID: "9e4d3572dbc07c66e466",
         clientSecret: "6aaf104f992ceeec6425a04effe9815098cb9817",
         callbackURL: "http://localhost:4000/auth/github/callback",
         scope: ["email", "profile"],
      },
      async function (accessToken, refreshToken, profile, done) {
         const [row] = await dbConnection.execute(
            "SELECT * FROM `users`WHERE `email`=?",
            [profile.username]
         );

         if (row[0] == undefined) {
            await dbConnection.execute(
               "INSERT INTO `users`(`name`,`email`,`password`) VALUES(?,?,?)",
               [
                  profile.displayName? profile.displayName:"none",
                  profile.username,
                  await bcrypt.hash(accessToken, 12),
               ]
            );
         }
         done(null, profile);
      }
   )
);

passport.serializeUser((user, done) => {
   done(null, user);
});
passport.deserializeUser(function (user, done) {
   done(null, user);
});
```

Для получения данных с платформы Google необходимо:

1) Перейти на сайт https://console.cloud.google.com/ и зайти в раздел API’s & Services

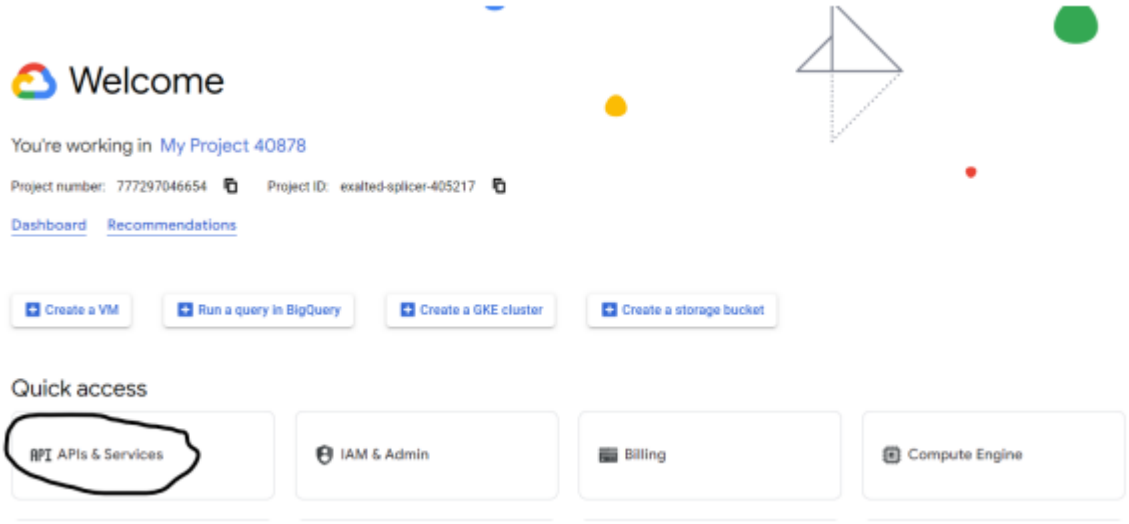

Рисунок 2 – Главная страница Google Cloud

Далее необходимо зайти во вкладку Credentials, а затем Create Credentials и выбрать Create OAuth client ID. Далее необходимо выбрать тип приложения “Web application” (при условии разработки веб-приложения). В ячейке «Authorized redirect URIs» нажать вкладку «Add URL» и добавить две ссылки, как показано на рисунке 3 и нажать кнопку «Create».

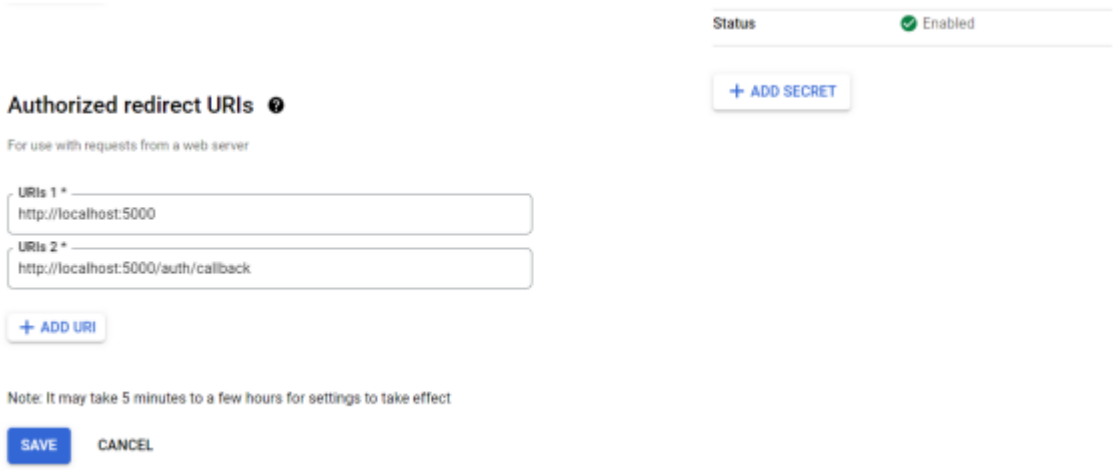

Рисунок 3 – Добавление ссылок

После этого появится ID и Secret, как показано на рисунке 4, которые необходимо добавить в код.

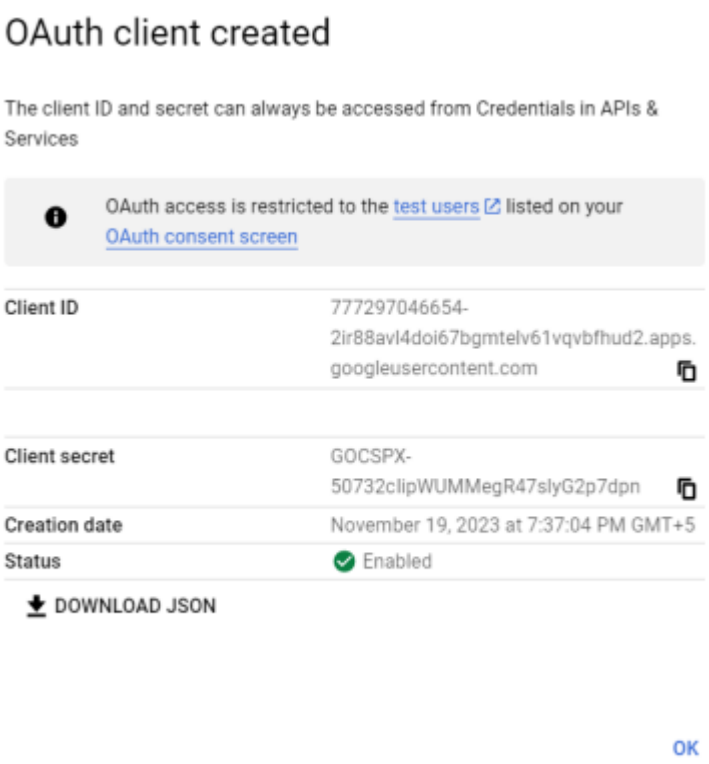

Рисунок 4 – Полученные результаты аутентификации

Авторизацию через github необходимо изучить самостоятельно и интегрировать в ваше решение.

3.4 Результат выполненной работы:

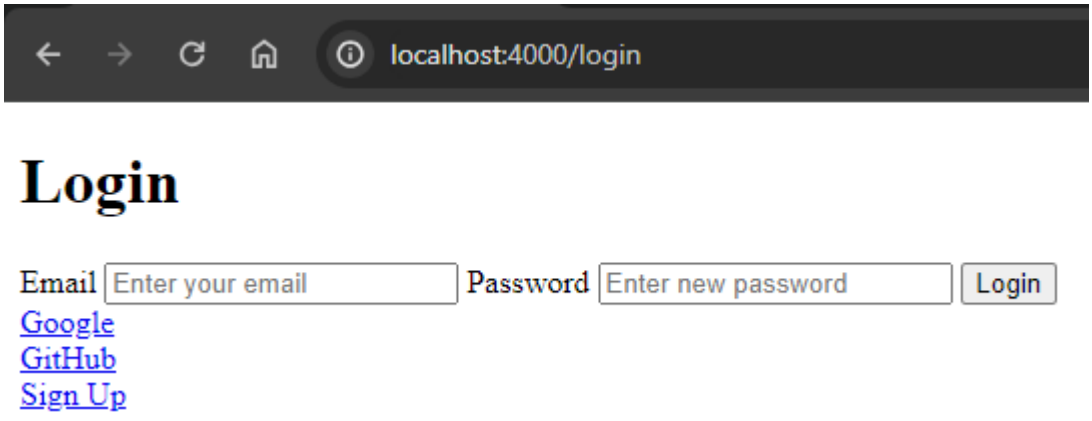

Рисунок 5 – страница входа

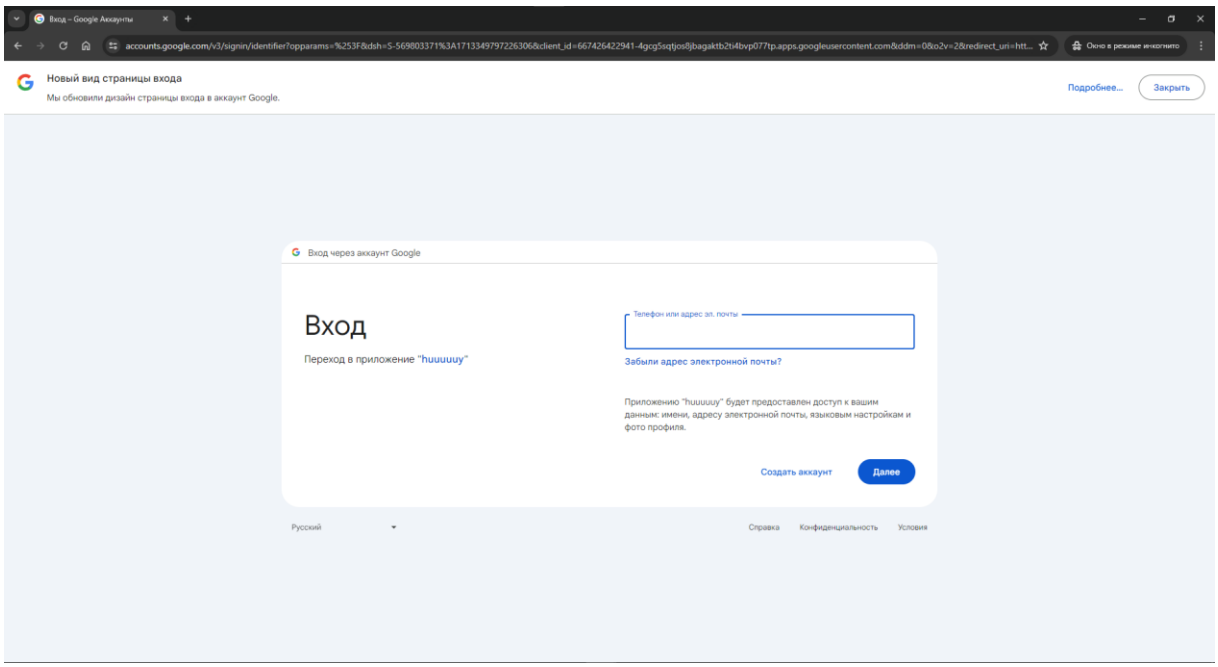

Рисунок 6 – Страница входа через Google

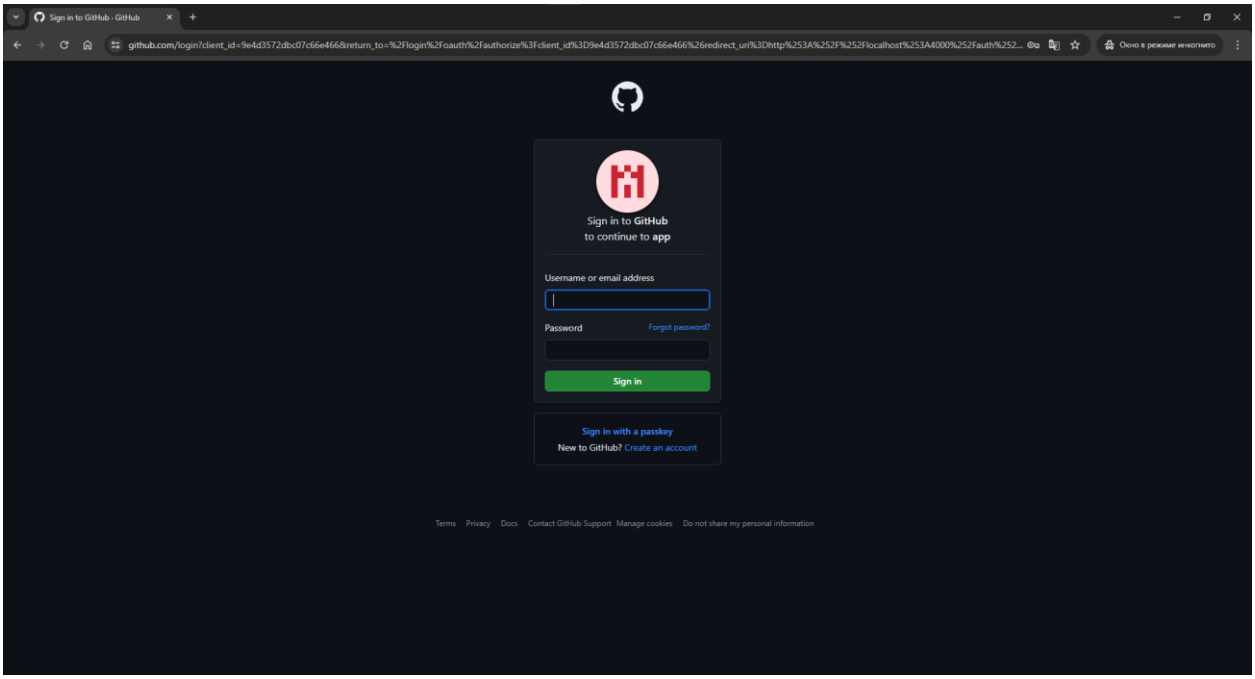

Рисунок 7 – Страница входа через GitHub

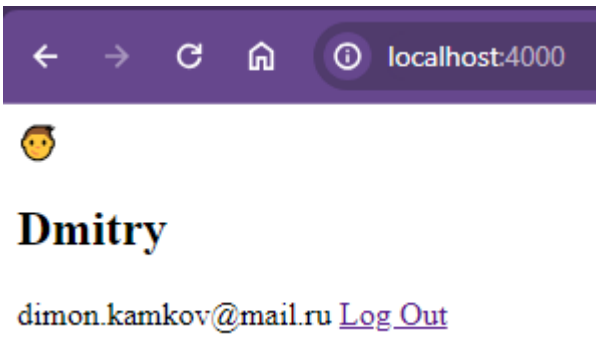

Рисунок 8 – Домашняя страница входа

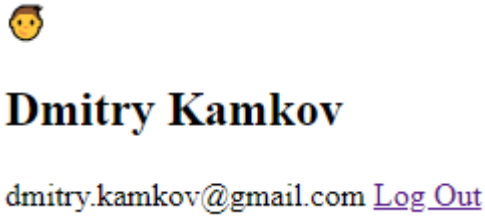

Рисунок 9 – Домашняя страница входа через Google

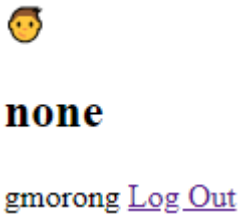

Рисунок 10 – Домашняя страница входа через GitHub

4. Контрольные вопросы:

4.1 Какова роль аутентификации и авторизации в клиент-серверных приложениях?

Аутентификация и авторизация являются критическими компонентами в приложениях клиента-сервера для обеспечения безопасности и контроля доступа к ресурсам.

Аутентификация проверяет личность пользователей или клиентов, пытающихся получить доступ к серверу. Это гарантирует, что пользователь - это то, кем он утверждает. Этот процесс обычно включает в себя предоставление учетных данных, таких как имена пользователей, пароли, биометрические данные или криптографические ключи. После аутентификации, сервер может доверять личности клиента.

Авторизация, с другой стороны, определяет, какие действия разрешено аутентифицированному пользователю выполнять на сервере. Это включает в себя предоставление или отказ в разрешениях на основе личности пользователя и ресурсов, к которым они пытаются получить доступ. Политики авторизации часто определяются на основе ролей, групп или конкретных разрешений, назначенных пользователям.

Вместе аутентификация и авторизация помогают обеспечить соблюдение политик безопасности, предотвратить несанкционированный доступ к конфиденциальной информации или функциональности, а также защищать целостность и конфиденциальность связи с клиентом-сервером. Без надлежащей механизмов аутентификации и авторизации приложения клиентских серверов были бы уязвимы для несанкционированного доступа, утечки данных и других угроз безопасности

4.2 Какие методы аутентификации можно использовать в клиент-серверных приложениях?

Приложения клиентского сервера могут использовать различные методы аутентификации для обеспечения безопасного доступа. Вот некоторые общие:

4.2.1. Имя пользователя и пароль: это самая основная форма аутентификации, в которой пользователь предоставляет имя пользователя и пароль, которые подтверждаются в отношении сохраненных учетных данных на сервере.

4.2.2. Аутентификация на основе токенов: В аутентификации на основе токенов сервер выдает токен клиенту после успешной аутентификации. Этот токен затем используется для последующих запросов вместо отправки имени пользователя и пароля с каждым запросом. Примеры включают JSON Web Tokens (JWT) и токены OAuth.

4.2.3. Сертификаты: Сертификаты могут использоваться для аутентификации как клиента, так и сервера в взаимно аутентифицированном соединении SSL/TLS.Каждая сторона представляет сертификат, подписанный доверенным органом сертификата (CA), чтобы доказать свою личность.

4.2.4. Биометрическая аутентификация: Биометрическая аутентификация использует уникальные биологические характеристики, такие как отпечатки пальцев, шаблоны радужной оболочки или распознавание лица, чтобы проверить идентичность пользователя.

4.2.5. Многофакторная аутентификация (MFA): MFA требует, чтобы пользователи предоставляли две или более формы аутентификации перед предоставлением доступа. Это может включать в себя комбинацию того, что пользователь знает (пароль), что -то, что есть у пользователя (токен или смарт -карта) или чего -то, что пользователь (биометрические данные).

4.2.6. Single Sign-On (SSO): SSO позволяет пользователям аутентифицировать один раз и получить доступ к нескольким связанным системам, не будучи предназначенным для входа в систему в каждом из них. Он обычно используется в корпоративных средах, где пользователям нужен доступ к нескольким приложениям.

4.2.7. LDAP (Протокол доступа к легким каталогам): LDAP может использоваться для централизованной служб аутентификации и авторизации. Приложения клиентского сервера могут аутентифицировать пользователей по сравнению с сервером каталогов LDAP.

Выбор метода аутентификации зависит от таких факторов, как требования к безопасности, удобство использования, масштабируемость и конкретные характеристики архитектуры клиента-сервер.

4.3 Как обеспечить безопасность процесса аутентификации в клиент-серверных приложениях?

Обеспечение безопасности процесса аутентификации в приложениях-сервере клиента имеет решающее значение для защиты конфиденциальных данных и предотвращения несанкционированного доступа. Вот несколько лучших практик для повышения безопасности:

4.3.1. Используйте безопасные протоколы: Используйте протоколы, такие как HTTPS или TLS/SSL для безопасной связи между клиентом и сервером. Эти протоколы шифруют данные во время передачи, предотвращая подслушивание и вмешивание.

4.3.2. Строительные политики пароля: обеспечение сильных политик паролей, включая требования к минимальной длине, сложности (сочетание букв, чисел и символов) и периодические изменения пароля. Поощряйте использование менеджеров паролей для надежного создания и хранения сложных паролей.

4.3.3. Многофакторная аутентификация (MFA): Реализуйте MFA, чтобы добавить дополнительный уровень безопасности за пределами паролей. Это может включать методы, такие как коды SMS, биометрическая аутентификация (отпечаток пальца, распознавание лица) или аппаратные жетоны.

4.3.4. Безопасное хранение учетных данных: хранить учетные данные пользователя надежно, используя алгоритмы хэширования (например, BCRYPT или Argon2) с уникальной солью для каждого пользователя. Избегайте хранения паролей в открытом виде или использования алгоритмов слабых хеширования (например, MD5 или SHA-1).

4.3.5. Управление сеансами: реализовать безопасные методы управления сеансами, чтобы гарантировать, что токены сеанса надежно передаются и хранятся. Используйте методы, такие как срок годности, отзыв токенов и безопасные атрибуты cookie (например, Httponly, Secure, Samesite), чтобы предотвратить угон сеанса и атаки CSRF.

4.3.6. Контроль доступа на основе ролей (RBAC): Реализуйте RBAC для обеспечения соблюдения принципа наименьшей привилегии, предоставляя пользователям только разрешения, необходимые для выполнения их задач. Это уменьшает потенциальное влияние скомпрометированных полномочий.

4.3.7. Заголовки безопасности: Используйте заголовки безопасности, такие как Политика безопасности контента (CSP), x-контент-тип-опции, x-frame-options и защита X-XSS для смягчения общих рисков безопасности веб-приложений, таких как Cross-Сайт сценариев (xss) и щелчок.

4.3.8. Регулярные аудиты безопасности и тестирование на проникновение: Проводите регулярные аудиты безопасности и тестирование на проникновение для выявления уязвимостей в процессе аутентификации и быстро их решить. Это помогает гарантировать, что система остается устойчивой против развивающихся угроз безопасности.

4.3.9. Обработка ошибок: реализовать надлежащие механизмы обработки ошибок, чтобы избежать утечки конфиденциальной информации, которая может быть использована злоумышленниками. Предоставьте общие сообщения об ошибках пользователям и войдите подробную информацию об ошибках для администраторов.

4.3.10. Образование в области безопасности и обучение: Обучать пользователей и администраторов о лучших практиках безопасности, включая важность сильных паролей, фишинговую осведомленность и риски обмена учетными данными или нажатия по подозрительным ссылкам.

Следуя этой лучшей практике, вы можете значительно повысить безопасность процесса аутентификации в приложениях-серверах клиента и снизить риск несанкционированного доступа и нарушений данных.

4.4 Какие технологии и протоколы используются для обеспечения безопасности при аутентификации в клиент-серверных приложениях?

Безопасность в аутентификации для клиентских приложений зависит от различных технологий и протоколов, чтобы обеспечить целостность, конфиденциальность и подлинность связи. Вот несколько ключевых:

4.4.1. Транспортный уровень безопасности (TLS): Ранее известный как SSL, TLS обеспечивает шифрование и аутентификацию по сети. Это гарантирует, что данные, передаваемые между клиентом и сервером, зашифрованы и не могут быть легко перехвачены или подделаны вредоносными субъектами.

4.4.2. Инфраструктура общедоступного ключа (PKI): PKI-это система, используемая для управления цифровыми сертификатами и государственно-частными парами ключей. Это обеспечивает безопасную связь, проверяя подлинность объектов, участвующих в процессе связи.

4.4.3. Цифровые сертификаты: Они выдаются сертификатными органами (CAS) и служат электронными полномочиями, которые проверяют личность отдельных лиц, серверов или организаций. Они используются в TLS для установления безопасных соединений и аутентификации сервера с клиентом.

4.4.4. Аутентификация задачи-ответа: Этот метод включает в себя сервер, бросая вызов клиенту с запросом на аутентификацию, и клиент отвечает соответствующими учетными данными или криптографическим доказательством личности.

4.4.5. OAuth (Open Authorization): OAuth является открытым стандартом для доступа к делегированию, обычно используемому для разрешения между службами. Это позволяет клиентскому приложению получать доступ к ресурсам от имени пользователя после того, как пользователь предоставляет разрешение.

4.4.6. JSON Web Tokens (JWT): JWT-это компактное, безопасное для URL средства представления претензий, которые должны быть переданы между двумя сторонами. Он обычно используется для аутентификации и авторизации в веб -приложениях.

4.4.7. OAuth 2.0 и OpenID Connect: это широко принятые стандарты для аутентификации и авторизации. OAuth 2.0 обеспечивает делегированный доступ, в то время как OpenID Connect Builds поверх OAuth 2.0 для предоставления услуг аутентификации.

4.4.8. Многофакторная аутентификация (MFA): MFA добавляет дополнительный уровень безопасности, требуя, чтобы пользователи предоставили две или более формы аутентификации перед предоставлением доступа. Это может включать в себя то, что они знают (пароль), то, что у них есть (смартфон) или что -то, что они есть (биометрические).

4.4.9. Secure Sockets Layer (SSL): SSL - это более старая технология, которая в значительной степени была заменена TLS, но все еще используется. Он обеспечивает безопасную связь между клиентами и серверами через Интернет.

Внедряя комбинацию этих технологий и протоколов, клиентские приложения могут установить безопасные механизмы аутентификации для защиты конфиденциальных данных и обеспечения конфиденциальности пользователей.

4.5 Что такое многофакторная аутентификация и каковы ее преимущества в клиент-серверных приложениях?

Многофакторная аутентификация (MFA) - это система безопасности, которая требует нескольких форм идентификации для проверки идентификации пользователя перед предоставлением доступа к системе или приложению. Эти факторы обычно делятся на три категории:

1. Факторы знаний: то, что знает пользователь, например, пароль или PIN -код.

2. Факторы владения: то, что есть у пользователя, например, смартфон, токен или смарт -карта.

3. Факторы присутствия: что-то, присущее пользователю, например, отпечаток пальца, сканирование радужной оболочки или распознавание голоса.

В приложениях клиентских серверов реализация MFA предлагает несколько преимуществ:

1. Усовершенствованная безопасность: требуя нескольких факторов для аутентификации, MFA снижает вероятность несанкционированного доступа, даже если один фактор (такой как пароль) поставлен под угрозу.

2. Смягчение рисков, связанных с паролем: Пароли подвержены украденным, угадающимся или сфотографированным.MFA снижает эти риски, добавляя дополнительные уровни безопасности за пределами пароля.

3. Соответствие нормативным требованиям: Во многих отраслях и юрисдикциях требуется сильные меры аутентификации для защиты конфиденциальных данных.MFA помогает организациям соблюдать эти правила.

4. Гибкость и удобство использования: MFA может быть реализован с использованием различных факторов, позволяя организациям выбирать комбинацию, которая наилучшим образом соответствует их потребностям в безопасности и предпочтениях пользователей. Эта гибкость может улучшить общий опыт пользователя, предлагая удобные, но безопасные варианты.

5. Адаптивность к развивающимся угрозам: По мере развития киберугрозов MFA обеспечивает динамический механизм защиты. Новые факторы аутентификации могут быть добавлены или существующие могут быть изменены для реагирования на новые риски безопасности.

В целом, мультифакторная аутентификация значительно повышает положение безопасности приложений клиента-сервера, добавляя слои проверки за пределы только паролей, тем самым снижая риск несанкционированного доступа и утечки данных.

4.6 Какие основные принципы должны соблюдаться при осуществлении процесса авторизации в клиент-серверных приложениях?

Реализация процесса авторизации в приложениях-сервере клиента включает в себя несколько фундаментальных принципов для обеспечения безопасности и функциональности:

1. Аутентификация: Проверьте идентификацию пользователя или клиента, подключенного к серверу. Обычно это включает в себя такие учетные данные, как имена пользователей, пароли, ключи API или токены.

2. Авторизация: Определите, к каким действиям или ресурсам допускается аутентифицированный пользователь или клиент. Это часто основано на ролях, разрешениях или списках контроля доступа (ACL).

3. Безопасная связь: Убедитесь, что связь между клиентом и сервером зашифрована с использованием протоколов, таких как HTTPS или SSL/TLS, чтобы предотвратить подслушивание и подделка.

4. Управление сеансом: безопасно управлять сеансами пользователей для поддержания государственных взаимодействий между клиентом и сервером. Это включает в себя обработку токенов сеанса, время истечения срока действия и безопасное хранение данных сеанса.

5. Наименее привилегия: Следуйте принципу наименьшей привилегии, предоставляя пользователям или клиентам только разрешения, необходимые для выполнения их предполагаемых действий, и ничего более.

6. Входная проверка: проверить и дезинфицировать все входные данные от клиентов, чтобы предотвратить общие уязвимости безопасности, такие как инъекционные атаки (например, инъекция SQL, XSS).

7. Это помогает отслеживать шаблоны доступа и обнаружить подозрительную активность.

8. Обработка ошибок: Реализуйте правильные механизмы обработки ошибок, чтобы предоставить значимые сообщения об ошибках клиентам без раскрытия конфиденциальной информации, которая может использоваться для атак.

9. Многофакторная аутентификация (MFA): рассмотрите возможность реализации дополнительных уровней безопасности, таких как MFA, для улучшения процесса аутентификации, требуя от пользователей предоставлять несколько форм проверки.

10. Регулярные аудиты безопасности: Проводите регулярные аудиты и оценки безопасности для выявления уязвимостей, оценки рисков и обеспечения соответствия стандартам безопасности и передовой практики.

Придерживаясь этих принципов, разработчики могут создавать надежные и безопасные процессы авторизации в приложениях клиентских серверов, защищая конфиденциальные данные и обеспечивая надежный пользовательский опыт.

4.7 Как обеспечить защиту от несанкционированного доступа к ресурсам в клиент-серверных приложениях?

Обеспечение приложений клиентского сервера включает в себя реализацию нескольких мер для предотвращения несанкционированного доступа к ресурсам.Вот список некоторых общих стратегий:

1. Аутентификация: Требуйте, чтобы пользователи были аутентифицированы перед доступом к ресурсам. Это может быть достигнуто с помощью комбинаций имени пользователя/паролей, биометрической аутентификации или аутентификации на основе токенов, таких как OAuth.

2. Авторизация: После проверки подлинности пользователи должны иметь доступ только к ресурсам, которые они уполномочены использовать. Реализовать контроль доступа на основе ролей (RBAC) или контроль доступа на основе атрибутов (ABAC) для управления разрешениями на основе ролей пользователей или конкретных атрибутов.

3. Шифрование: Зашифрованные конфиденциальные данные, передаваемые между клиентом и сервером, для предотвращения перехвата несанкционированными сторонами. Используйте протоколы, такие как HTTPS/TLS, чтобы обеспечить конфиденциальность и целостность данных.

4. Входная проверка: Проверьте и дезинфицируйте весь ввод пользователя на стороне клиента, чтобы предотвратить инъекционные атаки, такие как инъекция SQL или сценарии поперечного сайта (XSS).Кроме того, подтвердите ввод на стороне сервера, чтобы обеспечить целостность данных.

5. Управление сеансами: реализовать безопасные методы управления сеансами для предотвращения захвата сеанса или фиксационных атак. Используйте безопасные файлы cookie, регенерируйте идентификаторы сеансов после входа в систему и принудительные тайм -ауты сеанса.

6. Брандмауэры и системы обнаружения вторжений (IDS): развернуть брандмауэры для мониторинга и управления входящим и исходящим сетевым трафиком. Идентификаторы могут обнаружить и реагировать на подозрительную деятельность, помогая смягчить потенциальные нарушения безопасности.

7. Многофакторная аутентификация (MFA): Требовать, чтобы пользователи предоставляли несколько форм аутентификации, такие как пароль и одноразовый код, отправленный на их мобильное устройство, чтобы добавить дополнительный уровень безопасности.

8. Заголовки безопасности: Используйте заголовки безопасности, такие как Политика безопасности контента (CSP), строгому транспортному обеспечению (HSTS) и X-контента, чтобы повысить безопасность веб-приложений и защиту от общих уязвимостей.

9. Регулярные аудиты безопасности: Проводите регулярные аудиты безопасности и тестирование на проникновение для выявления уязвимостей и слабостей в приложении. Обратно решать любые проблемы, чтобы поддерживать безопасную среду.

10. Принцип наименьшей привилегии: Следуйте принципу наименьшей привилегии, предоставляя пользователям только разрешения, необходимые для выполнения их задач. Ограничение доступа уменьшает потенциальное влияние нарушения безопасности.

Внедряя эти меры, вы можете значительно снизить риск несанкционированного доступа к ресурсам в приложениях клиентских серверов. Кроме того, оставаясь информированным о возникающих угрозах безопасности и передовой практике, имеет решающее значение для поддержания надежной позиции безопасности.

4.8 Каким образом можно реализовать сессионную аутентификацию в клиент-серверных приложениях?

Аутентификация сеанса в приложениях клиентских серверов обычно включает в себя несколько шагов для обеспечения безопасной связи между клиентом и сервером при сохранении идентификации пользователя по нескольким запросам. Вот общий обзор того, как можно реализовать аутентификацию сеанса:

1. Пользовательская аутентификация:

- Когда пользователь входит в приложение, их учетные данные (такие как имя пользователя и пароль) подтверждаются сервером.
- После успешной аутентификации сервер генерирует уникальный идентификатор сеанса (идентификатор сеанса) для пользователя.

2. Управление сеансом:

- Сервер хранит идентификатор сеанса вместе с любыми соответствующими пользовательскими данными (например, идентификатором пользователя, роли, разрешениями) в хранилище данных (например, база данных, кэш).
- Идентификатор сеанса обычно хранится в сессии cookie на стороне клиента или возвращается к клиенту в ответе.

3. Запросы клиента:

- С каждым последующим запросом клиент отправляет идентификатор сеанса на сервер, обычно в заголовке HTTP или в качестве параметра в полезной нагрузке запроса.
- Сервер получает идентификатор сеанса с входящего запроса.

4. Проверка сеанса:

- Сервер проверяет идентификатор сеанса, чтобы убедиться, что он соответствует активному сеансу.
- Если идентификатор сеанса действителен, сервер получает связанные с ними пользовательские данные из хранилища сеанса и переходит с обработкой запроса.
- Если идентификатор сеанса недействителен или истек, сервер может ответить ошибкой аутентификации, побуждая клиента повторно.

5. СЕГЛЕДНАЯ ИСКЛЮЧЕНИЕ И ВОЗВРАЩЕНИЕ:

- Сессии, как правило, имеют конечную жизнь, чтобы снизить риски безопасности, связанные с долгоживущими сессиями.
- Сервер может обеспечить срок действия сеанса, установив срок действия истечения срока действия для каждого сеанса и периодически очистив сеансы истечения из магазина сеанса.

- Чтобы сохранить сеанс активным, клиент может отправлять запросы на сервер через регулярные промежутки времени, процесс, известный как сеанс «обновление» или «обновление».

6. Выход

- Когда пользователь выходит из приложения, сервер недействительный сеанс, связанный с идентификатором сеанса пользователя.

- Обычно это включает в себя удаление данных сеанса из сеанса - хранилища и удаление любых файлов cookie с сеанса от клиента.

7. Меры безопасности:

- Идентификаторы сеансов должны быть надежно сгенерированы, чтобы предотвратить угадание или атаки догадки или грубой силы.

- Безопасность транспортного уровня (TLS) должна использоваться для шифрования связи между клиентом и сервером, предотвращая подслушивание и подделка данных сеанса.

-Защиты для подключения к перекрестным сценариям (XSS) и подделке поперечного запроса (CSRF) должны быть реализованы для смягчения общих уязвимостей безопасности, связанных с сеансом.

Реализуя аутентификацию сеанса в соответствии с этими принципами, приложения клиента-сервер могут безопасно управлять пользовательскими сеансами и защищать конфиденциальные пользовательские данные от несанкционированного доступа.

4.9 Как обеспечить безопасность при передаче учетных данных между клиентом и сервером в клиент-серверных приложениях?

Обеспечение передачи данных бухгалтерского учета между клиентами и серверами в приложениях-клиентских приложениях имеет решающее значение для поддержания целостности данных, конфиденциальности и подлинности. Вот несколько ключевых стратегий для обеспечения безопасности:

1. Шифрование: Используйте алгоритмы шифрования, такие как TLS (безопасность транспортного уровня) или SSL (уровень безопасных сокетов) для шифрования данных во время передачи. Это гарантирует, что даже в случае перехвата данные остаются нечитаемыми для несанкционированных сторон.

2. Аутентификация: Реализуйте сильные механизмы аутентификации для проверки идентификации как клиента, так и сервера. Это может включать в себя такие методы, как взаимная аутентификация TLS, где как клиент, так и сервер аутентифицируют друг друга, используя цифровые сертификаты.

3. Контроль доступа: обеспечить строгие элементы управления доступа, чтобы гарантировать, что только авторизованные пользователи имеют доступ к данным учета. Это включает в себя реализацию контроля доступа на основе ролей (RBAC) и принципы наименее привилегии для ограничения доступа к конфиденциальной информации.

4. Целостность данных: Используйте криптографические алгоритмы хеширования, чтобы обеспечить целостность передаваемых данных .Это позволяет получателю убедиться, что данные не были подделаны во время передачи.

5. Безопасные протоколы: Используйте протоколы безопасной связи, такие как HTTPS для веб-приложений или SSH для безопасного доступа к оболочке. Эти протоколы обеспечивают безопасный канал для передачи данных по сети.

6. Брандмауэры и системы обнаружения вторжений (IDS): Развертывание брандмауэров и идентификаторов для мониторинга и фильтрации сетевого трафика, обнаружения и блокирования любых несанкционированных попыток доступа или манипулирования бухгалтерскими данными.

7. Регулярные обновления и управление исправлениями: Держите все программное обеспечение и системы в курсе новейших исправлений безопасности, чтобы защитить от известных уязвимостей и эксплойтов.

8. Шифрование данных в состоянии покоя: в дополнение к шифрованию данных во время передачи, также шифруйте конфиденциальные данные бухгалтерского учета, когда они хранятся на серверах или клиентских устройствах для защиты от несанкционированного доступа.

9. Аудит и ведение журнала: Реализуйте комплексные механизмы аудита и ведения журнала для отслеживания всех доступа и модификаций для бухгалтерских данных. Это помогает в обнаружении и исследовании любых инцидентов безопасности.

10. Обучение информированию о безопасности: Обучать клиентов и сотрудников о лучших практиках безопасности, таких как избегание фишинговых атак и использование сильных паролей, для снижения риска человеческой ошибки, ставя под угрозу безопасность данных.

Внедряя эти меры, вы можете значительно повысить безопасность бухгалтерских данных, передаваемых между клиентами и серверами в приложениях клиентских серверов.

4.10 Какие методы могут использоваться для обработки ошибок аутентификации и авторизации в клиент-серверных приложениях?

В приложениях клиента сервера обработка ошибки, связанные с аутентификацией и авторизацией, имеют решающее значение для обеспечения безопасности и пользовательского опыта. Вот некоторые методы, которые обычно используются:

1. Обработка кода ошибок: Определите коды ошибок для различных вопросов аутентификации и авторизации, таких как неверные учетные данные, токены с истекшим сроком действия или недостаточные разрешения. Сервер может вернуть эти коды ошибок клиенту, который затем обрабатывает их соответствующим образом, отображая удобные сообщения или предпринимая корректирующие действия.

2. Отзыв и освежающий токен: реализовать системы аутентификации на основе токков с механизмами для отзывания токена и освежения. Если токен скомпрометирован или истек, сервер может отклонить его и предложить клиенту запросить новый токен .Это помогает предотвратить несанкционированный доступ и сохранять безопасность.

3. Централизованная аутентификация и авторизация: Используйте централизованную аутентификацию и услуги авторизации, такие как OAuth 2.0 или OpenID Connect.Эти стандарты обеспечивают надежные механизмы для обработки ошибок аутентификации и авторизации, включая проверку токенов, срок действия токена и проверку объема.

4. Контроль доступа на основе ролей (RBAC): используйте RBAC для управления разрешениями пользователей на основе ролей. Когда возникает ошибка авторизации, сервер может сообщить клиенту, что пользователю не хватает необходимой роли или привилегий для доступа к ресурсу. Этот подход упрощает управление управлением доступом и повышает безопасность.

5. Регистрация и мониторинг: реализовать комплексные механизмы ведения журнала и мониторинга для отслеживания событий аутентификации и авторизации. Это позволяет администраторам идентифицировать и устранять ошибки, такие как обнаружение при подозрительных попытках входа в систему или несанкционированные модели доступа.

6. Ограничение и дросселирование. Ограничивая количество попыток аутентификации в течение определенного периода времени, сервер может предотвратить злоупотребление и повысить безопасность.

7. Пользовательские страницы и сообщения ошибок: Предоставьте пользовательские страницы или сообщения пользователям, когда возникают ошибки аутентификации или авторизации. Эти сообщения должны быть информативными, но ласковыми, помогая пользователям понять проблему и направлять их на то, как ее разрешить, например, сброс их пароля или обращение в поддержку.

8. Многофакторная аутентификация (MFA): Реализуйте MFA, чтобы добавить дополнительный уровень безопасности за пределами традиционной аутентификации имени пользователя и пароля. В случае ошибок аутентификации сервер может предложить пользователям выполнять дополнительные шаги проверки, такие как введение одноразового пароля, отправленного на их мобильное устройство.

Объединив эти методы, приложения клиента-сервер могут эффективно обрабатывать ошибки, связанные с аутентификацией и авторизацией, обеспечивая как безопасность, так и удобство использования для пользователей.