



Backend Development, 5cu

Study diary

Ivan Lubnin

Contents

[1 Week exercises 3](#_Toc181042553)

[1.1 Written answers to questions 3](#_Toc181042554)

[1.2 The first weekly exercises here, relevant code, explanations and screenshots (the same applies in following week exercises). 3](#_Toc181042555)

[2 Week excercises 4](#_Toc181042556)

[3 Week exercises 5](#_Toc181042557)

[4 Week exercises 6](#_Toc181042558)

[5 Week exercises 7](#_Toc181042559)

[Sources used with exercises 8](#_Toc181042560)

**Important general note for all weekly exercises: List all external sources you’ve used to implement the task (github, tutorialspoint, other projects, network sites, blogs etc..)**

# Week exercises

## Written answers to questions

**VSCode**

VSCode offers key features that enhance backend development. It includes an integrated debugger,

allowing step-by-step code inspection, which is critical for identifying and fixing bugs. VSCode

also supports a wide range of extensions—such as REST Client, Prettier, and ESLint—that streamline

backend workflows. Additionally, the built-in terminal lets developers run commands and scripts directly

within the editor, making it convenient to manage Node.js servers and other backend tools without

switching applications.

**Node.js and Express**

Node.js is a runtime environment for running JavaScript outside the browser, enabling the creation

of server-side applications. Unlike traditional web servers, Node.js uses an asynchronous, non-blocking

model, which makes it efficient for handling multiple requests. Express, a web framework for Node.js,

\simplifies server development by providing an intuitive way to manage routes, use middleware,

and handle HTTP requests. This is much easier than using Node's built-in HTTP module, which lacks

these features.

**Extensions:**

- REST Client: The REST Client extension in VSCode allows developers to test API endpoints directly from

the editor. This is helpful for testing routes without needing external tools.

- Prettier: Prettier enforces consistent code formatting, making code more readable. It can be configured

in VSCode settings or via a .prettierrc file, helping teams maintain a uniform style.

-ESLint: ESLint catches common coding mistakes, such as undefined variables or incorrect syntax. It also

enforces code style rules, which reduces errors and improves readability.

**Using Postman:**

Postman is widely used for API testing, allowing developers to make requests, inspect responses,

and manage multiple API environments. Compared to the REST Client in VSCode, Postman offers a more

visual and user-friendly interface with added features, such as automated testing and API documentation

tools. However, REST Client is lighter and integrates directly into VSCode, which some developers find

convenient for simpler tasks.

## The first weekly exercises here, relevant code, explanations and screenshots (the same applies in following week exercises).

const express = require('express');

const app = express();

const PORT = 3000;

app.use(express.json());

const movies =

[

    { id: 1, title: "Inception", director: "Christopher Nolan", year: 2010 },

    { id: 2, title: "The Matrix", director: "The Wachowskis", year: 1999 },

    { id: 3, title: "Parasite", director: "Bong Joon-ho", year: 2019 }

];

app.get('/', (req, res) =>

{

    let movieList = '<h1>Movie List</h1><ul>';

    movies.forEach(movie =>

    {

        movieList += `<li>${movie.title} (${movie.year}) - Directed by ${movie.director}</li>`;

    });

    movieList += '</ul>';

    res.send(movieList);

});

app.get('/movies', (req, res) =>

{

    res.json(movies);

});

app.post('/movies', (req, res) =>

{

    const newMovie =

    {

        id: movies.length + 1,

        title: req.body.title,

        director: req.body.director,

        year: req.body.year

    };

    movies.push(newMovie);

    res.status(201).json(newMovie);

});

app.get('/movies/:id', (req, res) =>

{

    const movie = movies.find(m => m.id === parseInt(req.params.id));

    if (!movie) return res.status(404).send('Movie not found');

    res.json(movie);

});

app.delete('/movies/:id', (req, res) => {

    const movieId = parseInt(req.params.id);

    const index = movies.findIndex(movie => movie.id === movieId);

    if (index !== -1) {

        movies.splice(index, 1);

        res.status(204).send();

    } else {

        res.status(404).send('Movie not found');

    }

});

app.listen(PORT, () =>

{

    console.log(`Server is running on http://localhost:${PORT}`);

});

A screen shot of a movie list

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

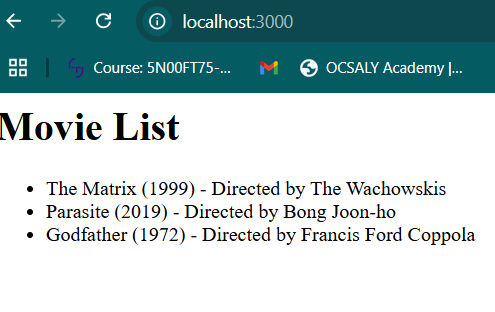
Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated



A screen shot of a computer program

Description automatically generated

# Week excercises

## Weekly exercises here, relevant code, explanations and screenshots

const express = require('express');

const app = express();

const port = 3004;

const morgan = require('morgan');

app.use(express.json());

app.use(morgan('dev'));

app.listen(port, () =>

{

    console.log(`Server running at http://localhost:${port}/`);

});

const movies =

[

    { id: 1, title: "Inception", director: "Christopher Nolan", year: 2010 },

    { id: 2, title: "The Matrix", director: "The Wachowskis", year: 1999 },

    { id: 3, title: "Parasite", director: "Bong Joon-ho", year: 2019 },

    { id: 4, title: "Tenet", director: "Christopher Nolan", year: 2020 }

];

app.get('/', (req, res) =>

{

    let myhtml = '<h1>Awesome movies!</h1><ul>';

    movies.forEach(movie =>

    {

        myhtml += `<li>Movie: ${movie.title} was directed by ${movie.director} in ${movie.year}</li>`;

    });

    myhtml += '</ul>';

    res.send(myhtml);

});

app.get('/movies', (req, res) =>

{

    const { title, director, year } = req.query;

    let filteredMovies = [...movies];

    console.log("The request came with the parameters:", req.query);

    console.log("Array of films:", filteredMovies);

    if (title)

    {

        filteredMovies = filteredMovies.filter (movie =>

            movie.title.toLowerCase().includes(title.toLowerCase())

        );

        console.log("After filtration by title:", filteredMovies);

    }

    if (director)

    {

        filteredMovies = filteredMovies.filter (movie =>

            movie.director.toLowerCase().includes(director.toLowerCase())

        );

        console.log("After filtration by director:", filteredMovies);

    }

    if (year)

    {

        const yearNum = parseInt(year);

        if (isNaN(yearNum))

        {

            return res.status(400).send('Invalid year parameter');

        }

        filteredMovies = filteredMovies.filter(movie => movie.year === yearNum);

        console.log("After filtration by year:", filteredMovies);

    }

    res.json(filteredMovies);

});

app.get('/movies/:id', (req, res) =>

{

    const movie = movies.find(movie => movie.id === parseInt(req.params.id));

    if (movie)

    {

        res.json(movie);

    }

    else

    {

        res.status(404).send('No movie found');

    }

});

app.post('/movies', (req, res) =>

{

    const { title, director, year } = req.body;

    if (!title || !director || !year || isNaN(year) || year < 1888)

    {

        return res.status(400).send({error:'Invalid data'});

    }

    const newId = movies.length > 0 ? Math.max(...movies.map(m => m.id)) + 1 : 1;

    const newMovie = { id: newId, title, director, year };

    movies.push(newMovie);

    res.status(201).json(newMovie);

});

app.put('/movies/:id', (req, res) =>

{

    const movie = movies.find(movie => movie.id === parseInt(req.params.id));

    if (!movie)

    {

        return res.status(404).send('No movie found');

    }

    const { title, director, year } = req.body;

    if (!title || !director || !year || isNaN(year) || year < 1888)

    {

        return res.status(400).send({error:'Invalid data'});

    }

    movie.title = title;

    movie.director = director;

    movie.year = year;

    res.status(200).json(movie);

});

app.delete('/movies/:id', (req, res) =>

{

    const movieId = parseInt(req.params.id);

    const index = movies.findIndex(movie => movie.id === movieId);

    if (index === -1)

    {

        return res.status(404).send('No movie found');

    }

    movies.splice(index, 1);

    res.sendStatus(204);

});

app.use((req, res) =>

{

    res.status(404).send('Route not found');

});

A screenshot of a movie

Description automatically generated

A screenshot of a computer program

Description automatically generated

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**Exercise 2: Validation and Proper HTTP status codes**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**Exercise 3: Querry parameters**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**Exercise 4: Testing with JSON Client and Logging with Morgan**

**A screenshot of a computer program

Description automatically generated**

# Week exercises

## Written answers to questions

#1

1. Data Structure and Storage

SQL

Data is stored in structured tables with predefined schemas (rows and columns). Each column has a

specified datatype, and changes to the schema can be complex.

NoSQL

Data is stored in a more flexible format like key-value pairs, documents, wide-columns, or graphs.

Schemas are not strictly enforced, making it easier to handle unstructured or semi-structured data.

2. Relationships

SQL

Handles relationships using foreign keys and joins. It is designed for complex relational queries

and ensures data integrity.

NoSQL

Relationships are typically not enforced, but some NoSQL databases (e.g., graph databases)

are specifically designed for highly connected data. Relationships are often handled at the

application level.

3. Scalability

SQL

Scales vertically by upgrading hardware (e.g., more CPU or RAM). Horizontal scaling (distributing

across multiple servers) is possible but more difficult.

NoSQL

Scales horizontally by distributing data across multiple servers or nodes, making it more

suitable for handling large volumes of data.

4. Use Cases

SQL

Best for applications that require ACID (Atomicity, Consistency, Isolation, Durability) compliance,

like banking, financial systems, and inventory management.

NoSQL

Ideal for applications with large, rapidly changing, or unstructured data, such as real-time

analytics, content management, and IoT applications.

#2

SQL Database Example: PostgreSQL

Primary Features:

- Open-source, relational database system.

- ACID-compliant with advanced querying capabilities.

- Supports JSON data for semi-structured data needs.

Use Cases:

- Enterprise-level applications, data warehouses, and analytics.

- Used by companies like Uber and Instagram.

NoSQL Database Example: MongoDB

Primary Features:

- Document-oriented database storing data in JSON-like documents.

- Schema flexibility allows rapid application development.

- Supports horizontal scaling and sharding for big data.

Use Cases:

- Real-time analytics, content management systems, and mobile applications.

- Used by companies like eBay and Shutterfly.

#3

SQL databases use structured tables with predefined schemas, making them ideal for applications with

structured and relational data, such as financial systems or inventory management. They handle

relationships through foreign keys and joins and typically scale vertically by upgrading hardware. In

contrast, NoSQL databases use flexible formats like documents or key-value pairs, which are better suited

for unstructured or rapidly changing data, such as real-time analytics or content management. They scale

horizontally across multiple servers and handle relationships at the application level. Examples include

PostgreSQL (SQL) for structured data and MongoDB (NoSQL) for flexible, document-based storage.

**Steps to Set Up MongoBD atlas:**

Account Creation:

- Visited MongoDB Atlas.

- Signed up using my email

2. Cluster Creation:

Logged into the MongoDB Atlas dashboard.

Clicked on "Build a Cluster" and selected the Free Tier option.

* Chose AWS as the cloud provider and selected region.
* Named the cluster TAMK2025 and waited for the cluster to de-ploy.
* Database and Collection Creation:
* Navigated to Database -> Collections.
* Clicked "Add My Own Data".

- Created a database named moviesDB and added collection “movies”.

4. Inserting Data:

- Opened the movies collection.

- Manually added three movie documents using the Insert Docu-ment option.

5. Connecting to the Database:

- Go to Database -> Connect and selected "Connect your application".

- Chose “node.js” as driver and copied connection string.

- Updated the connection string with database username and password.

**Connect the App to MongoDB**

To replace local JSON storage with a MongoDB database, I did the following updates:

* Installed the MongoDB Node.js driver using:

npm install mongodb

* Connected to MongoDB Atlas using a connection string:

const uri = "mongodb+srv://<username>:<password>@cluster1.mongodb.net/?retryWrites=true&w=majority&appName=Cluster1";

* Established a connection with the moviesDB database and movies collection:

const database = client.db("moviesDB");

moviesCollection = database.collection("movies");

* Ensured the server starts only after a successful connection to MongoDB.

How the Backend Interacts with the Database

The backend now directly communicates with MongoDB instead of using a local JSON file. The key interactions include:

* Fetching Movies (GET /movies):  
  Retrieves all movies from the database using “find().toArray()”

const movies = await moviesCollection.find().toArray();

* Retrieving a Single Movie by ID (GET /movies/:id):  
  Fetches a specific movie using its \_id.

const movie = await moviesCollection.findOne({ \_id: new ObjectId(id) });

* Adding a New Movie (POST /movies):  
  Inserts a new document into the collection.

const result = await moviesCollection.insertOne(newMovie);

* Updating a Movie (PUT /movies/:id):  
  Modifies an existing document based on \_id.

const result = await moviesCollection.updateOne(

{ \_id: new ObjectId(id) },

{ $set: updatedMovie });

* Deleting a Movie (DELETE /movies/:id):  
  Removes a movie by its \_id.

const result = await moviesCollection.deleteOne({ \_id: new ObjectId(id) })

## The first weekly exercises here, relevant code, explanations and screenshots

**Exercise 2: Setting Up MongoDB Atlas**

**A screenshot of a computer

Description automatically generated**

**Exercise 3: Updating the Backend to Use MongoDB**

const { MongoClient, ServerApiVersion, ObjectId } = require('mongodb');

const express = require('express');

const app = express();

const port = 3004;

const uri = "mongodb+srv://lubninivan05:3cPT3gSM0WA88qHe@cluster1.yp6zm.mongodb.net/?retryWrites=true&w=majority&appName=Cluster1";

const client = new MongoClient(uri,

{

    serverApi:

    {

        version: ServerApiVersion.v1,

        strict: true,

        deprecationErrors: true,

    }

});

let moviesCollection;

async function run()

{

    try

    {

        await client.connect();

        const database = client.db("moviesDB");

        moviesCollection = database.collection("movies");

        console.log("Connected to MongoDB!");

        app.listen(port, () =>

        {

            console.log(`Server running at http://localhost:${port}/`);

        });

    }

    catch (error)

    {

        console.error("Error:", error);

        process.exit(1);

    }

}

app.use(express.json());

app.get('/movies', async (req, res) =>

{

    try

    {

        const movies = await moviesCollection.find().toArray();

        res.json(movies);

    }

    catch (error)

    {

        res.status(500).json({ error: "Error" });

    }

});

app.get('/movies/:id', async (req, res) =>

{

    const { id } = req.params;

    try

    {

        const movie = await moviesCollection.findOne({ \_id: new ObjectId(id) });

        if (movie)

        {

            res.json(movie);

        }

        else

        {

            res.status(404).json({ error: "Movie not found" });

        }

    } catch (error)

    {

        res.status(500).json({ error: "Error" });

    }

});

app.post('/movies', async (req, res) =>

{

    const newMovie = req.body;

    try

    {

        const result = await moviesCollection.insertOne(newMovie);

        res.status(201).json({ message: "Movie added", insertedId: result.insertedId });

    }

    catch (error)

    {

        res.status(500).json({ error: "Error adding movie" });

    }

});

app.put('/movies/:id', async (req, res) =>

{

    const { id } = req.params;

    const updatedMovie = req.body;

    try

    {

        const result = await moviesCollection.updateOne(

            { \_id: new ObjectId(id) },

            { $set: updatedMovie }

        );

        if (result.matchedCount > 0)

        {

            res.json({ message: "Movie updated" });

        }

        else

        {

            res.status(404).json({ error: "Movie not found" });

        }

    }

    catch (error)

    {

        res.status(500).json({ error: "Error" });

    }

});

app.delete('/movies/:id', async (req, res) =>

{

    const { id } = req.params;

    try

    {

        const result = await moviesCollection.deleteOne({ \_id: new ObjectId(id) });

        if (result.deletedCount > 0)

        {

            res.json({ message: "Movie deleted" });

        }

        else

        {

            res.status(404).json({ error: "Movie not found" });

        }

    }

    catch (error)

    {

        res.status(500).json({ error: "Error" });

    }

});

run().catch(console.dir);

A screen shot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A computer code with red text

Description automatically generated

# Week exercises

## Written answers to questions

**Why Modularization is needed:**

Modularization is crucial in backend development for several reasons:

1. Maintainability: It keeps the code organized, making it easier to manage, update, and fix bugs.
2. Reusability: Allows you to reuse modules across different parts of your app or even other projects.
3. Scalability: As your app grows, you can scale modules independently without disrupting the entire system.
4. Testability: Modules can be tested in isolation, ensuring better quality and easier debugging.
5. Collaboration: Enables multiple developers to work on different modules simultaneously, speeding up development.
6. Flexibility: Makes it easier to update, replace, or integrate new services without affecting the whole system.

## The first weekly exercises here, relevant code, explanations and screenshots

**Exercise 1:**

**A screen shot of a computer

Description automatically generated**

**Exercise 2: Create validation for your project**

const Joi = require('joi');

const Movie = require('../models/movie');

const movieSchema = Joi.object({

    title: Joi.string().min(1).required(),

    director: Joi.string().min(1).required(),

    year: Joi.number().integer().min(1900).max(2100).required(),

});

const createMovie = async (req, res) => {

    try

    {

        const { error } = movieSchema.validate(req.body);

        if (error)

        {

            return res.status(400).json({ error: error.details[0].message });

        }

        const movie = new Movie(req.body);

        const savedMovie = await movie.save();

        res.status(201).json(savedMovie);

    }

    catch (error)

    {

        res.status(500).json({ error: 'Failed to create movie' });

    }

};

module.exports =  createMovie;

**Exercise 3: Authentication**

Authentication rout

const express = require('express');

const { login } = require('../controllers/authController');

const router = express.Router();

// Login route

router.post('/login', login);

module.exports = router;

file “authenticate”

const jwt = require('jsonwebtoken');

const authenticate = (req, res, next) =>

    {

    // extract the Authorization header

    const authHeader = req.headers.authorization;

    // ceck if the Authorization header is present

    if (!authHeader || !authHeader.startsWith('Bearer '))

    {

        return res.status(401).json({ error: 'Authorization token required' });

    }

    // extract the token

    const token = authHeader.split(' ')[1];

    try

    {

        // verify the token

        const decoded = jwt.verify(token, process.env.JWT\_SECRET);

        // attach the decoded token to the request object for later use

        req.user = decoded; // `decoded` contains data like { id, role }

        next(); // proceed to the next middleware or route handler

    }

    catch (error)

    {

        res.status(401).json({ error: 'Invalid or expired token' });

    }

};

module.exports = authenticate;

file “movie controller”

//this defines all routes to get to the student data

const express = require('express');

//import controllers

const

{

    getAllMovies,

    getMovieById,

    createMovie,

    updateMovie,

    deleteMovie,

} = require('../controllers/movieController.js')

const authenticate = require('../middlewares/authenticate');

const validateMovie = require('../middlewares/validateMovie')

const router = express.Router();

//define top level router and pass to controller

router.get('/', getAllMovies);

router.get('/:id', getMovieById);

router.post('/', authenticate, validateMovie, createMovie);

router.put('/:id', authenticate, validateMovie, updateMovie);

router.delete('/:id', authenticate, deleteMovie);

// export this controller to server.js

module.exports = router;

**A screen shot of a computer

Description automatically generated**

# Week exercises

## Written answers to questions

**Exercise 2**

What is real-time communication in the context of web applications? Provide examples.

Real-time communication in web applications refers to instant, two-way communication between clients and servers, where data is exchanged immediately without delays. Examples include chat applications, live notifications, and online multiplayer games. REST is not suitable for real-time because it is request/response-based and doesn't support continuous, persistent connections, making it inefficient for scenarios that require immediate data updates.

Why doesn't a regular REST work in these scenarios?

Compare WebSocket and MQTT: What are their main characteristics? What are typical use cases for each? Why is WebSocket suitable for a real-time chat application?

Main Characteristics:

* WebSocket: Full-duplex communication over a single, persistent connection; bidirectional.
* MQTT: Lightweight publish/subscribe messaging protocol, designed for low-bandwidth, high-latency, or unreliable networks.

Typical Use Cases:

* WebSocket: Real-time chat applications, live sports scores, collaborative tools.
* MQTT: IoT devices, remote sensors, or low-power devices in connected environments.

xWhy is WebSocket suitable for a real-time chat application?

WebSocket is ideal for real-time chat because it maintains an open, persistent connection between the client and server, allowing for instant message delivery and reduced latency, making it more efficient than repeated HTTP requests.

**Exercise 3**

Question 1: What is HTTPS?

* What is HTTPS and how is it different from HTTP? HTTPS (Hypertext Transfer Protocol Secure) is an extension of HTTP that uses encryption to secure communication between the web browser and the server. The key difference is that HTTPS uses SSL/TLS to encrypt data, while HTTP transmits data in plaintext.
* Why is HTTPS important for web applications? HTTPS ensures secure communication, protecting against data interception and tampering. It also helps maintain privacy by encrypting sensitive data (e.g., login credentials), ensures data integrity, and builds trust with users by showing a secure connection (padlock in the browser).
* How does HTTPS work? HTTPS works by encrypting the data between the client and the server using SSL/TLS protocols. This process involves certificates issued by trusted Certificate Authorities (CAs). The server presents a certificate to the client, verifying its identity, and then establishes an encrypted session to protect data during transit.

Question 2: Steps for Enabling HTTPS in Real Applications

* Obtain an SSL/TLS Certificate: SSL/TLS certificates can be obtained from trusted Certificate Authorities (CAs) like Let's Encrypt, Comodo, or DigiCert. Some CAs offer free certificates (e.g., Let's Encrypt), while others may charge for more advanced features.
* Update the Server to Use HTTPS: To enable HTTPS, the server must be configured with the SSL/TLS certificate and the corresponding private key. This typically involves setting up the server (e.g., Apache, Nginx) to listen on port 443, configure the certificate files, and redirect HTTP traffic to HTTPS.
* Browser Trust and Certificate Authorities: Browsers trust certificates issued by recognized Certificate Authorities (CAs). When a browser connects to a website, it checks the server's certificate against a list of trusted CAs. If the certificate is valid, the browser establishes a secure connection; otherwise, it shows a warning.

--------------------------------------------------------------------------------------------------------

1. What are the limitations of self-signed certificates, and why shouldn’t they be used in production?

* Limitations of self-signed certificates:
  + No Trust from Browsers: Self-signed certificates are not signed by a trusted Certificate Authority (CA), so browsers cannot verify the authenticity of the server.
  + Security Risks: Since anyone can generate a self-signed certificate, attackers could potentially use one to impersonate a legitimate server, leading to security vulnerabilities like man-in-the-middle attacks.
  + Lack of Verification: Self-signed certificates do not undergo the vetting process that CAs provide, which could lead to an untrusted or insecure connection.

Why they shouldn't be used in production: In production environments, security is critical, and self-signed certificates do not provide the necessary trust and verification. They can lead to trust issues with users and expose sensitive data to potential attacks.

2. How did Postman react to the self-signed certificate, and why?

When using a self-signed certificate, Postman will show a warning or error, indicating that the certificate is not trusted. It may block requests or give an option to bypass the warning. It happens, because Postman detects that the certificate is not signed by a trusted Certificate Authority (CA), so it can't verify the authenticity of the server. This is a security measure to prevent potential malicious attacks.

**3.** How would you proceed to obtain a valid SSL/TLS certificate for production, and why is it important?

* How to obtain a valid SSL/TLS certificate for production:
  + Choose a trusted Certificate Authority (CA): For a valid SSL/TLS certificate, you should obtain it from a trusted CA such as Let's Encrypt (free), DigiCert, or Comodo.
  + Complete domain validation: The CA will verify your ownership of the domain through methods like email validation or DNS record setup.
  + Install the certificate on the server: After obtaining the certificate, install it on your web server (e.g., Apache, Nginx) and configure the server to use HTTPS.

This is important because a valid SSL/TLS certificate provides trust, ensuring secure data transmission between the client and server. It prevents unauthorized access, man-in-the-middle attacks, and protects sensitive user data like passwords and payment details. A valid certificate also improves SEO rankings and avoids warnings or security issues in browsers.

## The first weekly exercises here, relevant code, explanations and screenshots

**Exercise 1**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer screen

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**Exercise 2**

require('dotenv').config(); // Load evironment variables

const express = require('express');

const morgan = require('morgan');

const https = require('https'); // Импорт модуля HTTPS

const WebSocket = require('ws');

const selfsigned = require('selfsigned');// you can generate your sertificates for enabling https in your app

const connectToDatabase = require('./config/db');

const moviesRoutes = require('./routes/movies');

const authRoutes = require('./routes/auth');

const pems = selfsigned.generate(null, { days: 365 });

const privateKey = pems.private; // Исправленный доступ к ключу

const certificate = pems.cert; // Исправленный доступ к сертификату

//our task: change the server, so it uses https instead of http

//encryption needs certificates and signing but on the dev phase we can use self-signing to generate cetificates

//we ll need http server - required for integrating WebSocket with express

const http = require('http');

const {initializeWebSocket} = require('./wsConnections.js');

const app = express();

const http\_port = process.env.HTTP\_PORT;

const https\_port = process.env.HTTPS\_PORT;

//checking certificates before starting up server

if (!privateKey || !certificate)

{

    console.error("Error: no sertificates generated");

    process.exit(1); //stop the process, if keys are not generated

}

else

{

    console.log("Sertificates loaded succesfully");

    console.log("Private Key:", privateKey.substring(0, 30) + "...");

    console.log("Certificate:", certificate.substring(0, 30) + "...");

    const server = https.createServer({ key: privateKey, cert: certificate }, app);

}

//we need a http server for the websocket

const server = https.createServer({key: privateKey, cert: certificate}, app);

//middleware

app.use(morgan('dev'));

app.use(express.json());

//general route

app.get('/', (req, res) =>

{

    res.send("You are in Movie management system")

});

//routes for the whole app

app.use('/movies', moviesRoutes);

app.use('/auth', authRoutes);

//start the server

const startServer = async() =>

{

    //connect the database, then star listening in port

    await connectToDatabase();

    server.listen(https\_port, () =>

    {

        console.log("Server running on port 3443");

    });

};

//initializwe the websocket and start CRUD server

initializeWebSocket( server );

startServer();

// catch-all route for undefined routes

app.use((req, res) =>

{

    res.status(404).send('404 Not Found: The requested resource does not exist.');

});

**A screenshot of a computer program

Description automatically generated**

# Week exercises

## Written answers to questions

## The first weekly exercises here, relevant code, explanations and screenshots

My theme is Event planner, managing a list of events with titles, dates, and locations.

Code of file “server” (full code you can see in github, link is below)

require('dotenv').config(); // Load evironment variables

const express = require('express');

const morgan = require('morgan');

const https = require('https'); // Импорт модуля HTTPS

const WebSocket = require('ws');

const selfsigned = require('selfsigned');// you can generate your sertificates for enabling https in your app

const connectToDatabase = require('./config/db');

const itemsRoutes = require('./routes/items');

const authRoutes = require('./routes/auth');

const pems = selfsigned.generate(null, { days: 365 });

const privateKey = pems.private; // Исправленный доступ к ключу

const certificate = pems.cert; // Исправленный доступ к сертификату

//our task: change the server, so it uses https instead of http

//encryption needs certificates and signing but on the dev phase we can use self-signing to generate cetificates

//we ll need http server - required for integrating WebSocket with express

const http = require('http');

const {initializeWebSocket} = require('./wsConnections.js');

const app = express();

const http\_port = process.env.HTTP\_PORT;

const https\_port = process.env.HTTPS\_PORT;

//checking certificates before starting up server

if (!privateKey || !certificate)

{

    console.error("Error: no sertificates generated");

    process.exit(1); //stop the process, if keys are not generated

}

else

{

    console.log("Sertificates loaded succesfully");

    console.log("Private Key:", privateKey.substring(0, 30) + "...");

    console.log("Certificate:", certificate.substring(0, 30) + "...");

    const server = https.createServer({ key: privateKey, cert: certificate }, app);

}

//we need a http server for the websocket

const server = https.createServer({key: privateKey, cert: certificate}, app);

//middleware

app.use(morgan('dev'));

app.use(express.json());

//general route

app.get('/', (req, res) =>

{

    res.send("You are in Item management system")

});

//routes for the whole app

app.use('/items', itemsRoutes);

app.use('/auth', authRoutes);

//start the server

const startServer = async() =>

{

    //connect the database, then star listening in port

    await connectToDatabase();

    server.listen(https\_port, () =>

    {

        console.log("Server running on port 3443");

    });

};

//initializwe the websocket and start CRUD server

initializeWebSocket( server );

startServer();

// catch-all route for undefined routes

app.use((req, res) =>

{

    res.status(404).send('404 Not Found: The requested resource does not exist.');

});

Testing: A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

Trying to post as “regular” user A screenshot of a computer

Description automatically generated

Sources used with exercises