Period-2 Node, Express with TypeScript, JavaScript Backend Testing, MongoDB (and Geo-location)

Note: This description is too big for a single exam-question. It will be divided up into separate questions for the exam

**Explain** Pros & Cons in using Node.js + Express to implement your Backend compared to a strategy using, for example, Java/JAX-RS/Tomcat

*(NodesJS is a runtime environment which runs javascript in the V8 engine)*

*(NodeJS performs best in real time  applications with intense I/O which require speed and scalability. Ex: social networks, gaming apps, live chats or stock exchange software)*

***Keywords for nodeJS - speed, fast, scalable, data intensive, real time***

**Answer:**

Pros:

* Same language in both frontend and backend
* Use of JSON
* NodeJS has a non-blocking event queue so no need for dealing with threads
* NodeJS is faster than Java, JAX-RS and Tomcat
* Newer technology compared to Java
* Scalability
* NPM (node package manager)
* Reuse code in frontend and backend by wrapping it into modules and creating a new level of abstraction

Cons:

* NodeJS is single threaded so a CPU heavy operation can slow down the whole server (application)
* When managing heavy computations(number crunching - various calculations) it is a disadvantage to use a single threaded technology like NodeJS since it can block the event loop and result in delay for application users
* Many packages to choose from so it can be hard to implement the “right” package
* Unhandled error will shut down the server
* Compared to Java, backend programming with JavaScript has been around for a shorter time so there is less knowledge in this area compared to Java.
* Callback hell (hard to maintain the code but we make use of promises ;))

**Explain** the difference between *Debug outputs* and *ApplicationLogging*. What’s wrong with console.log(..) statements in our backend code?

**Answer:**

console.log is blocking which means it “stops” the server when used in production. Never use console.log as a debugging tool it is more professional to use debug() or a logging tool like morgan.

Debug (npm i debug) is used when we are in development environment and want to log request to the command line. Works like console.log – however console.log runs synchronize and is not the professional way of doing debugging. To be able to fully debug remember to attach a flag in visual studio code – view: command pallete: attach debug: flag. Also make use of breakpoints to better see what is going on at a given point in the code.

Console.log() does not always give the whole picture and should NOT be shown in production environment. If used keep it simple and in development environment.

console.log is blocking which means it “stops” the server when used in production. Never use console.log as a debugging tool it is more professional to use debug() or a logging tool like morgan.

ApplicationLogging is used to log request to a log file or a database while separating the logs into different categories, such as http request (npm install morgan) or errors that occur in the application

ApplicationLogging is used to log request to a log file or a database while separating the logs into different categories, such as http request (npm install morgan) or errors that occur in the application while running.

* Debug output:
  + Makes it possible to stop the code with breakpoints and physically see all the steps in the line of code as you go.

There are multiple functions like, stopping, skipping, forward, backwards etc.

Debugging this way makes it easier to see what is actually going on, due to the visuals and you can see information about variables, call stack etc.

You can also use an “Application like” feature called log points which is used for creating a point similar to a breakpoint, but a logpoint doesn’t break the code but logs a message instead.

Debug output is very developer friendly because of the interactive element of debugging your code.

* ApplicationLogging:
  + If you’re using ApplicationLogging you have to create a log file containing information about events occuring in the application. They can include errors, warnings and information about events.

The log file will contain a lot of information about the application and can be difficult to read and understand.

**Demonstrate** a system using application logging and environment controlled debug statements.

**Answer:**

*(Also see app.ts for a few logger example)*

Using winston/Morgan we have created a logger ( /src/middleware/logger.ts )  which logs to 2 files located in a logs folder ( /logs/combined.log & /logs/error.log ).

* Winston is a logging library.
* Morgan is a middleware function logger.

Environment controlled debug statements are set up in the .env file ( DEBUG=www,app,setup-friend-testdata,friend-routes ) and then used in [www.ts](http://www.ts) ( /src/bin/www.ts ).

**Explain**, using relevant examples, concepts related to testing a REST-API using Node/JavaScript/Typescript + relevant packages

**Answer:**

For testing REST-API in Node/javaScript/typeScript we use different libraries:  
- Mocha is a testing framework that provides functions that are executed according in a specific order, and that logs their results to the terminal window.  
- Chai is an assertion library that is often used alongside Mocha. It provides functions and methods that help you compare the output of a certain test with its expected value.  
- SuperTest is an HTTP assertions library that allows you to test your Node.js HTTP servers.  
- Nock is an HTTP server mocking and expectations library for Node.js.

When you read tests written in Mocha, you’ll see regular use of the keywords `describe` and `it`. These keywords, provided by Mocha, provide structure to the tests by batching them into test suites and test cases. Se example below.

```javascript

describe("Verify the getAllFriends method", () => {

        it("It should get two friends", async () => {

            const allFriends = await facade.getAllFriends();

            expect(allFriends.length).to.equal(2)

        })

    })  
```

More examples for testing with Mocha and Chai here:

<https://github.com/cph-pk/Fullstack-startcode/blob/day4/test/friendFacadeTest.ts>

Here is an example testing an endpoint using SuperTest with authentication.

```javascript  
describe("While attempting to get all users", function () {

it("it should get three users when authenticated", async () => {

   const response = await request

     .get('/api/friends/all')

     .auth("pp@b.dk", "secret")

   expect(response.body.length).to.equal(3)

})

  })  
```

More examples for testing with Mocha, Chai and SuperTest her:

<https://github.com/cph-pk/Fullstack-startcode/blob/day4/test/friendEndpointsTest.ts>

Here is an example testing with Nock here:

```javascript

describe("While attempting to get Donald", async function () {

it("Should provide age '61'", async function () {

     nock('https://api.agify.io')

         .get("/?name=donald")

         .reply(200, {

             "name": "donald",

             "age": 61,

             "count": 10771

         })

     const result = await request.get("/nameinfo/donald")

     expect(result.body.age).to.be.equal(61);

})

})  
```

More examples for testing with Nock here:

<https://github.com/cph-pk/Fullstack-startcode/blob/day4/playground/what/whattodoTest.ts>

<https://github.com/cph-pk/Fullstack-startcode/blob/day4/playground/infoForNameServerTest.ts>

**Explain** a setup for Express/Node/Test/Mongo-DB/GraphQL development with Typescript, and how it handles "secret values",  debug, debug-outputs, application logging and testing.

**Answer:**

By using this setup we can do full stack development and be sure to do typecheck with Typescript.

The main **benefit of Typescript** is that it offers the ability to add static types to your Javascript code. Javascript is a dynamically typed language, which is one of its best or one of its worst features, depending on who you ask.

In package.json we make use of “npm test” with the following script “test”: “mocha --exit -r ts-node/register test/\*\*/\*.ts”. This way we can make use of the test framework mocha in our files in the test folder.

Secret values can be seen in our .env file (not uploaded to github since it is in the .gitignore file). The values are used to set port, mongodb connection, database name, debug and where it is used, production and development mode and the option to remove authentication.

debug (npm install debug) is used in playground/debug-tester.ts and [www.ts](http://www.ts). debug output can also be obtained with the use of the framework morgan.

For logger take a look in bin/[www.ts](http://www.ts). Logger is used for logging in production while debug is used for debugging in development (do not use console.log).

Testing examples can be found in the folder test. The frameworks used are explained in the question above.

**Explain** a setup for Express/Node/Test/Mongo-DB/GraphQL development with Typescript. Focus on how it uses Mongo-DB (how secret values are handled, how connections (production or test) are passed on to relevant places in code, and if use, how authentication and authorization is handled

**Answer:**

Look at the answer above.

MongoDB: to connect to our database we have a connection string in the .env file. In the folder config there is a dbConnector.ts file where there is a class to connect to mongodb which is used in production and development. There is also another class in this file for testing - this class makes use of a database which is stored in memory (useful and fast since there is no need to connect to the live database all the time).

When we start up the server in [www.ts](http://www.ts) there is also made a connection to the database. Here we make use of the global object app to set the database with a key and value pair. This is then used in  routes/friendRoutesAuth.ts.

In facade/friendFacade.ts we make use of the mongodb framework to be able to make a collection (table) in our database.

The inMemory database is used in the files test/friendEndPointsTest.ts and test/friendFacadeTest.ts.

Authentication is done with the web browsers basic authentication. **Basic authentication** is a simple **authentication** scheme built into the HTTP protocol. The client sends HTTP requests with the **Authorization** header that contains the word **Basic** word followed by a space and a base64-encoded string username:password.

We do this in middleware/basic-auth.ts where a header is set with the response ‘WWW-Authenticate’, ‘Basic realm=”example”’.

*(authentication can be skipped by enabling the environment variable SKIP\_AUTHENTICATION=”random value”)*

Authorization is done with the help of the framework basic-auth (npm i basic-auth). We first make use of it in routes/friendRoutesAuth.ts  where we import the Module and save it in a variable (authMiddleware). Basic-auth.ts makes use of its function auth() to auth a request. We also verify a user by using the verify function in friendFacade *(remember to mention that we compare the encrypted password with the encrypted password in mongodb by using bcrypts compare function)*.

### auth(req)

Get the basic auth credentials from the given request. The Authorization header is parsed and if the header is invalid, undefined is returned, otherwise an object with name and pass properties.

**Explain**, preferably using an example, how you have deployed your node/Express applications, and which of the Express Production best practices you have followed.

**Explain** possible steps to deploy many node/Express servers on the same droplet, how to deploy the code and how to ensure servers will continue to operate, even after a droplet restart.

**Explain**, your chosen strategy to deploy a Node/Express application including how to solve the following deployment problems:

* Ensure that you Node-process restarts after a (potential) exception that closed the application
* Ensure that you Node-process restarts after a server (Ubuntu) restart
* Ensure that you can run “many” node-applications on a single droplet on the same port (80)

**Explain**, using relevant examples, the Express concept; middleware.

Express is a routing and middleware framework and middleware functions are functions that access to the request object(req), the response object(res) and the next() function in the applications request-response cycle. The next function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.

If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.

Example: /src/middleware/basic-auth.ts

**Explain**, conceptually and preferably also with some code, how middleware can be used to handle problems like logging, authentication, cors and more.

Express is a series of middleware functions which have access to the request and response object and are able to pass the middleware function down the line to the next middleware function.

Examples:

/src/middleware/basic-auth.ts

/src/middleware/myCors.ts

/src/middleware/simpleLogger.ts

**Explain, using relevant examples, your strategy for implementing a REST-API with Node/Express  + TypeScript** and demonstrate how you have tested the API.

**Answer: STILL MISSING ANSWERS**

See flow 2 : Day\_1 : friends project

See Postman under : 4 Semester Express API v1 for examples on how to test my rest endpoints.

**Explain**, using relevant examples, how to test JavaScript/Typescript Backend Code, relevant packages (Mocha, Chai etc.) and how to test asynchronous code.

Examples:

/src/test/friendEndPointsTest.ts & /src/test/friendFacadeTest.ts

/playground/infoForNameServer.ts & /playground/infoForNameServerTest.ts

NoSQL and MongoDB

**Explain**, generally, what is meant by a NoSQL database.

**Answer:**

NoSQL is a document database. Data or records are stored as documents and use json(javascript object notation) like syntax.

NoSQL is a database form that is not as strict as SQL (Structured query language). That means that you do not need to have a schema(schemaless) that tells you what kind of data you can input into the tables. Tables = Collections, Rows = documents. NoSQL is there for more flexible and can be better for performance. Besides that, you do not need relations, while you can do that a side product of that is that you can get duplicate data.

NoSQL stands for “not only SQL”, and is a database system that doesn't follow the standard relational database models like MySQL.

NoSQL databases do not follow all the rules of a relational database —specifically, it does use a traditional row/column/table database design and does not use structured query language (SQL) to query data.

* In a relational database, a book record is often disassembled (or “normalized”) and stored in separate tables, and relationships are defined by primary and foreign key constraints. In this example, the **Books** table has columns for **ISBN**, **Book Title**, and **Edition Number**, the **Authors** table has columns for **AuthorID** and **Author Name**, and finally the **Author-ISBN** table has columns for **AuthorID** and **ISBN**. The relational model is designed to enable the database to enforce referential integrity between tables in the database, normalized to reduce the redundancy, and generally optimized for storage.
* In a NoSQL database, a book record is usually stored as a [JSON](http://json.org/) document. For each book, the item, **ISBN**, **Book Title**, **Edition Number**, **Author Name**, and **AuthorID** are stored as attributes in a single document. In this model, data is optimized for intuitive development and horizontal scalability.

***Explain*** Pros & Cons in using a NoSQL database like MongoDB as your data store, compared to a traditional Relational SQL Database like MySQL.

**Answer:**

In big businesses you typically use both solutions. Each solution has its strength. SQL uses schemas (is a disadvantage if you want to be flexible with your data but it is an advantage if you want to have a predictable layout). Relations can also be a huge advantage if you change data often and therefor only need to change the value in one place and not several places like with NoSQL. The negative things is that is you have many reads that can be worse for the performance compared with the NoSQL world where you already have all the data merged in one collection.

Scaling: Horizontal -> add more servers. Can be hard if not impossible with SQL because data can not be split into more services. Vertical -> add more power to the existing server. The downside is the power has a limit when only using one server. That is the restrictions of a SQL database because horizontal scaling is very hard of near impossible and vertical has its limits. With NoSQL you can easily scale horizontally due to the way data is stored. Its way simpler because you have no relations and data is standalone.

If you have many collections and you update that data a lot NoSQL might not be the way to go since you need to update many places. Collections are data where you have the data you typically need, so you do not need to merge(join) tables together. If you have many reads NoSQL world can be better for performance.

There is no clear winner it depends on your needs and if you get so big that it becomes a topic you would probably use both. SQL if you need a schema, use relations, if you work with data that changes frequently and is used in different places of your application. NoSQL is good if you want to read a lot maybe write a little but not necessary update a lot of places in the application. If you use a few places heavily in the applications and others not so much and want to be able to fetch that data fast. If scaling is important (ex: horizontal) then NoSQL might be the way to go.

End the end chose the right tool for the job you want to get done!

**Pros**

* **Performance:** NoSQL databases can often outperform SQL databases, simply because all information is stored inside one database, whereas with SQL you might have to query data across multiple tables.
* **Scalability:** the only way to scale up an SQL database, is to upgrade CPU and ram. With NoSQL you can simply shard the database, and divide it into several servers.

**Cons**

* **Not mature:** While NoSQL has been around since the 70s, it only started being popular in the 2000s, meaning that SQL has had much more time and investment, and thus is in a more advanced state compared to NoSQL. Less help is available online, and experts are rarer.

**Explain** about indexes in MongoDB, how to create them, and *demonstrate* how you have used them.

**Answer:**

only place use so far is in utils/makeTestFriends.ts in line 13 -> friendsCollection.createIndex({ email: 1 }, { unique: true })

An index supports a query when the index contains all the fields scanned by the query.

MongoDB uses multikey indexes to index the content stored in arrays. If you index a field that holds an array value, MongoDB creates separate index entries for every element of the array. These multikey indexes allow queries to select documents that contain arrays by matching on element or elements of the arrays.

Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform a collection scan, i.e. scan every document in a collection, to select those documents that match the query statement. If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.

Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field. The ordering of the index entries supports efficient equality matches and range-based query operations. In addition, MongoDB can return sorted results by using the ordering in the index.

**Explain**, *using your own code* examples, how you have used some of MongoDB's "special" indexes like *TTL* and *2dsphere and perhaps also the Unique Index.*

***Demonstrate*,** using your own code samples, how to perform all CRUD operations on a MongoDB

**Answer:**Link to code:  
<https://github.com/cph-pk/Fullstack-startcode/blob/day4/src/facades/friendFacade.ts><https://github.com/cph-pk/Fullstack-startcode/blob/day4/src/routes/FriendRoutesAuth.ts>

**Demonstrate** how you have set up sample data for your application testing

**Answer:**

```javascript

// Before test setup DB to MongoDB and create a new in memory collection  
// New instance of facade

before(async function () {

     const client = await InMemoryDbConnector.connect();

     const db = client.db();

     friendCollection = db.collection("friends");

     facade = new FriendFacade(db)

})

// Before each test create a hashed password with bcrypt  
// Delete your collection and insert new

beforeEach(async () => {

     const hashedPW = await bcryptjs.hash("secret", 4)

     await friendCollection.deleteMany({})

     await friendCollection.insertMany([

         { firstName: "Pop", lastName: "Corn", email: "pc@b.dk", password: hashedPW, role: "user" },

         { firstName: "Pip", lastName: "Fugl", email: "pf@b.dk", password: hashedPW, role: "user" }

     ])

})  
``` **Explain** the purpose of mocha, chai, supertest and nock, which you should have used for your testing

**Answer:**

Unit testing ensures that all code meets quality standards before it's deployed. This ensures a reliable engineering environment where quality is paramount. Over the course of the product development life cycle, unit testing saves time and money, and helps developers write better code, more efficiently.

**Explain**, using a sufficient example, how to mock and test endpoints that relies on data fetched from external endpoints

**Answer:**

In the example below we will test an external endpoint from an external HTTPS.

```javascript

describe("While attempting to get Donald", async function () {

it("Should provide age '61'", async function () {

     nock('<https://api.agify.io>') // we add the HTTPS address here

         .get("/?name=donald") // endpoint goes here (don’t forget ‘/’)

         .reply(200, { //reply with a status 200, and the body will contain a (partial) response in JSON

             "name": "donald",

             "age": 61,

             "count": 10771

         })

     const result = await request.get("/nameinfo/donald") // here we request the endpoint

     expect(result.body.age).to.be.equal(61);  // test if ‘age’ in body is equal 61

})

})  
```

More examples here:  
<https://github.com/cph-pk/Fullstack-startcode/blob/day4/playground/what/whattodoTest.ts>

<https://github.com/cph-pk/Fullstack-startcode/blob/day4/playground/infoForNameServerTest.ts>

**Explain**, using a sufficient example a strategy for how to test a REST API. If your system includes authentication and roles explain how you test this part.

**Answer:**

We test the REST API in test/friendEndpointsTest.ts. The strategy being used is SuperTest to to endpoints, Mocha to be able to use before, after, beforeEach, afterEach and chai to be able to chain commands. Besides that we make use of an in memory database to be able to test the collection from our mongodb without making use of the facade. In other words we make a copy of the real database in memory with the API from mongodb.

import { MongoMemoryServer } from "mongodb-memory-server" ->

The package downloads a mongodb binary to your home directory and instantiates a new memory-backed MondoDB instance as needed. For each test file you can spin up a new server which means you can run them all parallel.

We authenticate by using supertest .auth method.

In routes/friendRoutesAuth.ts we do an if check to make sure only users with a role as admin can do certain actions:

```javascript

 if (USE\_AUTHENTICATION && !req.credentials.role || req.credentials.role !== "admin")

```

Her is kode in friendEndPointTest.ts:

```javascript  
describe("While verifying the get any user, given a userId (email)", function () {

it("It should allow an admin user to find Donald Duck", async () => {

   const response = await request

     .get('/api/friends/find-user/dd@b.dk')

     .auth("aa@a.dk", "secret")

   expect(response.status).to.equal(200)

   expect(response.body.firstName).to.equal("Donald")

})

it("It should not let a non-admin user find Donald Duck", async () => {

   const response = await request

     .get('/api/friends/find-user/dd@b.dk')

     .auth("pp@b.dk", "secret")

   expect(response.status).to.equal(401)

})

  })  
```

For more examples:  
<https://github.com/cph-pk/Fullstack-startcode/blob/day4/test/friendEndpointsTest.ts>

**Explain**, using a *relevant example*, a full JavaScript backend including relevant test cases to test the REST-API (not on the production database)

**Answer:**

*Sudo code*: user request -> start hit in code -> follow request around in code -> show how we test this flow of a request in the code

A request can be tested with the program postman

(request example: api/friends/[pp@b.dk](mailto:pp@b.dk))

Application starts in bin/[www.ts](http://www.ts) where we start the server and a connection to our database and set the value on a key/value pair via the global variable app ->

App.ts  (remember to talk about all the imports, especially import dotenv from "dotenv"; dotenv.config(); that loads our environment variables and error handling (all the middleware we use)) ->

friendRoutesAuth.ts (talk about authorization and authentication) ->

friendFacade.ts (query mongodb with const friend: IFriend = await this.friendCollection.findOne({ email: friendEmail });, talk about Interface, Joi validating, if time refer to the validation method in the bottom where bcrypt is used) ->

back to friendRoutesAuth.ts where we make a DTO to show the relevant data to a user.

Test case:

Look in test/friendEndpointTests.ts and go to the test case shown in the question above

```javascript

describe("While verifying the get any user, given a userId (email)", function () {

    it("It should allow an admin user to find Donald Duck", async () => {

      const response = await request

        .get('/api/friends/find-user/dd@b.dk')

        .auth("aa@a.dk", "secret")

      expect(response.status).to.equal(200)

      expect(response.body.firstName).to.equal("Donald")

    })

```

If time, talk about the two other tests about being authenticated and authorized before being able to search on other users.

In test/friendFacadeTest.ts we test the getFriend method from our facade:

```javascript

it("It should find Pop Corn", async () => {

            const findFriend = await friendCollection.findOne({ email: "pc@b.dk" })

            expect(findFriend.lastName).to.be.equal("Corn")

        })

```

If you have time to the exam talk about the @test belove (It should not find [xxx.@.b.dk](about:blank)) and you could also talk about the tests in "Verify the getVerifiedUser method" to talk about verification.