Period-2 Node.js, Express + JavaScript Backend Testing, NoSQL, MongoDB and Mongoose

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

# **Explain and Reflect**

## **Why would you consider a Scripting Language as JavaScript as your Backend Platform?**

* Its easy and quick to develop a network application.
* You stay in the same coding language both in the frontend and backend
* There isn’t a lot of code required, to get a working app.

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

## **Pros**

* You can make a good and responsive network application rather fast.
* Allows the use of data streaming, web sockets and fast file uploads.
* It's easy to set up a REST-API with express.
* It is efficient at handling thousands of concurrent requests (For example - a chat application).
* It is very simple to implement server middleware, that will be executed between all requests.

## **Cons**

* Java is good at handling CPU heavy tasks, Node.JS + Express is not. Because Node is, despite its asynchronous event model, by nature single threaded. When you launch a Node process, you are running a single process with a single thread on a single core.
* Java integrates well with relational databases like MySQL. Node.JS + Express does not, and take use of “Nosql” databases.
* Java as oppposed to Node.JS + Express is a strictly typed language which provides a certain security.
* 500 errors in Node.JS and Express will crash the entire application, Java will not.

## **Node.js uses a Single Threaded Non-blocking strategy to handle asynchronous task. Explain strategies to implement a Node.js based server architecture that still could take advantage of a multi-core Server.**

## **Solution 1 (built it)**

Node.js does not come with mutlithreading out of the box, since it's single threaded, but it is possible to build and program it yourself. We just haven't done that in class.

## **Solution 2 (Multiple servers)**

For scaling throughout on a webservice, you should run multiple Node.js servers on one or more machine/es, one per core and split request traffic between them. This provides excellent CPU-affinity and will scale throughout nearly linearly with core count. You could also put a load balancer in front of it. The load balancer will balance the load of incoming requests, thus achieving a multicore solution

## **Explain briefly how 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**

To ensure my application will work I would use nodemon at least for development, nodemon is a utility that restarts the server or the application if it crashes or after something.

Another strategy is to use a process manager. When using a process manager, the process manager manages the starting of the application. You no longer start the application yourself, but instead instruct a process manager to do it for you. These process managers can be configured to automatically restart the application on crashes.

[Express on project managers](https://expressjs.com/en/advanced/pm.html) Useful process managers:

* Restart the app automatically if it crashes.
* Gain insights into runtime performance and resource consumption.
* Modify settings dynamically to improve performance.
* Control clustering.

Some popular process managers are:

* [forever](https://github.com/foreverjs/forever)
* [pm2](http://pm2.keymetrics.io/)
* [StrongLoop Process Manager](http://strong-pm.io/)

These can all be used.

## **Ensure that you Node-process restarts after a server (Ubuntu) restart**

Read above.

## **Ensure that you can take advantage of a multi-core system**

Use the web api's SetTimeout or similar to delegate tasks to the browsers multi-threading capabilities. Or one could use the cluster module for node, which is probably the best solution.

## **Ensure that you can run “many” node-applications on a single droplet on the same port (80)**

You could configure a load balancer for this purpose. Nginx could also be used as a reverse proxy.

## **Explain the difference between “Debug outputs” and application logging. What’s wrong with console.log(..) statements in our backend-code.**

The problem with using console.log is that the output cannot easily be disabled when deployed to a production environment. Since console.log is a blocking call, the impact on the performance of the application will suffer.

The debug package exposes a function that can be used to print debugging messages.

const a = require('debug')('a') // Creates a debug function with the name a

const b = require('debug')('b') // Creates a debug function with the name b

const c = require('debug')('c') // Creates a debug function with the name c

a('Printed by a')

b('Printed by b')

c('Printed by c')

These messages can easily be enabled or disabled based on the DEBUG environment variable.

* when DEBUG=\*, all debug statements are printed.
* when DEBUG=a, only the a debug statements are printed.
* when DEBUG=\*,-a, all debug statements except a are printed.
* when DEBUG=a,b, only a and b debug statements are printed.

Names can also be enabled based on a regex like syntax.

const a = require('debug')('name:a')

const b = require('debug')('name:b')

* when DEBUG=name:a, a debug statements are printed.
* when DEBUG=name:\*, all debug statements starting with name are printed.

## **Demonstrate a system using application logging and “coloured” debug statements.**

Logging using Winston:

const app = express()

const winston = require('winston')

const consoleTransport = new winston.transports.Console()

const myWinstonOptions = {

transports: [consoleTransport]

}

const logger = new winston.createLogger(myWinstonOptions)

function logRequest(req, res, next) {

logger.info(req.url)

next()

}

app.use(logRequest)

function logError(err, req, res, next) {

logger.error(err)

next()

}

app.use(logError)

### **Colored debug statements:**

The debug module has a great namespace feature that allows you to enable or disable debug functions in groups. It is very simple–you separate namespaces by colons, like this:

debug('app:meta')('config loaded')

debug('app:database')('querying db...');

debug('app:database')('got results!', results);

Enable debug functions in Node by passing the process name via the DEBUG environment variable. The following would enable the database debug function but not meta:

$ DEBUG='app:database' node app.js

To enable both, list both names, separated by commas:

$ DEBUG='app:database,app:meta' node app.js

Alternately, use the asterisk wildcard character (\*) to enable any debugger in that namespace. For example, the following enables any debug function whose name starts with “app:":

$ DEBUG='app:\*' node app.js

You can get as granular as you want with debug namespaces…

debug('myapp:thirdparty:identica:auth')('success!');

debug('myapp:thirdparty:twitter:auth')('success!');

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

Testing a REST API is much easlier in JavaScript than in many other languages like Java. When using languages like Java, web services often need to be deployed to a dedicated server. Using JavaScript we can programmatically start our express server using the listen method. We can then use the node-fetch to make requests to the REST API.

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

* <https://expressjs.com/en/resources/middleware.html>
* <https://expressjs.com/en/guide/using-middleware.html>

A middleware with no mount path will be executed every time the app recieves a request

var app = express()

app.use(function (req, res, next) {

console.log('Time:', Date.now())

next()

})

## **Explain, using relevant examples, how to implement sessions and the legal implications of doing this.**

var cookieSession = require("cookie-session");

app.use(

cookieSession({

name: "session",

secret: "I\_should\_never\_be\_visible\_in\_code",

// Cookie Options

maxAge: 30 \* 60 \* 1000 // 30 minutes

})

);

GDPR (30) Natural persons may be associated with online identifiers provided by their devices, applications, tools and protocols, such as internet protocol addresses, cookie identifiers or other identifiers such as radio frequency identification tags.

This may leave traces which, in particular when combined with unique identifiers and other information received by the servers, may be used to create profiles of the natural persons and identify them. In short: when cookies can identify an individual via their device, it is considered personal data.

## **Compare the express strategy toward (server side) templating with the one you used with Java on second semester.**

Both JSP and EJS uses tags to embed Java and JavaScript respectively in HTML code

JSP tags:

* <%! Variable declaration or method definition %>
* <%= Java valid expression %>
* <% Pure Java code %>

EJS tags:

* <% 'Scriptlet' tag, for control-flow, no output
* <%\_ ‘Whitespace Slurping’ Scriptlet tag, strips all whitespace before it
* <%= Outputs the value into the template (HTML escaped)
* <%- Outputs the unescaped value into the template
* <%# Comment tag, no execution, no output
* <%% Outputs a literal '<%'
* %> Plain ending tag
* -%> Trim-mode ('newline slurp') tag, trims following newline
* \_%> ‘Whitespace Slurping’ ending tag, removes all whitespace after it

## **Demonstrate a simple Server Side Rendering example using a technology of your own choice (pug, EJS, ..).**

[API for REST end points](https://github.com/grem848/mini-project-fullstackjs2019/blob/master/routes/api.js)

This creates endpoints on the server, revealing users and locationbologs.

## **Explain, using relevant examples, your strategy for implementing a REST-API with Node/Express and show how you can "test" all the four CRUD operations programmatically using, for example, the Request package.**

<https://vegibit.com/mongoose-crud-tutorial/>

<https://github.com/judre/Miniproject>

## **Explain, using relevant examples, about testing JavaScript code, relevant packages (Mocha etc.) and how to test asynchronous code.**

<https://github.com/judre/Miniproject>

## **Explain, using relevant examples, different ways to mock out databases, HTTP-request etc.**

We can use nock to mock a website

const nock = require('nock');

describe("loadWiki()", function() {

before(function() {

//the website to be mocked

nock("https://en.wikipedia.org")

//the HTTP method and the path

.get("/wiki/Abraham\_Lincoln")

//the response the mocked website should send

.reply(200, "Mock Abraham Lincoln Page");

});

it("Load Abraham Lincoln's wikipedia page", function(done) {

tools.loadWiki({ first: "Abraham", last: "Lincoln"}, function(html) {

expect(html).to.equal("Mock Abraham Lincoln Page");

done();

});

});

});

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

* <https://expressjs.com/en/advanced/best-practice-performance.html>
* <https://expressjs.com/en/advanced/best-practice-security.html>
* <https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/deployment>
* <https://medium.freecodecamp.org/how-to-write-a-production-ready-node-and-express-app-f214f0b17d8c>
* <https://www.codementor.io/mattgoldspink/nodejs-best-practices-du1086jja>

# **NoSQL, MongoDB and Mongoose**

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

### **What is NoSQL?**

NoSQL encompasses a wide variety of different database technologies that were developed in response to the demands presented in building modern applications:

* Developers are working with applications that create massive volumes of new, rapidly changing data types — structured, semi-structured, unstructured and polymorphic data.
* Long gone is the twelve-to-eighteen month waterfall development cycle. Now small teams work in agile sprints, iterating quickly and pushing code every week or two, some even multiple times every day.
* Applications that once served a finite audience are now delivered as services that must be always-on, accessible from many different devices and scaled globally to millions of users.
* Organizations are now turning to scale-out architectures using open source software, commodity servers and cloud computing instead of large monolithic servers and storage infrastructure.

Relational databases were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of the commodity storage and processing power available today.

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

## **Pros of NoSQL**

1. Flexible Scalability Unlike rational database management model that is difficult to scale out when it come to commodity clusters NoSQL models make use of new nodes which makes them transparent for expansion. The model is designed to be used even with low cost hardwares. In this current world where outward scalability is replacing upwards scalability, NoSQL models are the better option.
2. Stores Massive Amounts Of Data Given the fact that transaction rates are rising due to recognition, huge volumes of data need to be stored. While rational models have grown to meet this need it is illogical to use such models to store such large volumes of data. However these volumes can easily be handled by NoSQL models
3. Database Maintenance The best rational models need the service of an expert to design, install and maintain. However, NoSQL models need much less expert management as it already has auto repair and data distribution capabilities, fewer administration and turning requirements as well as simplified data designs.
4. Economical Rational models require expensive proprietary servers and storage systems whereas NoSQL models are easy and cheap to install. This means that more data can be processed and stored at a very minimal cost.

## **Cons of NoSQL**

1. Not Mature Rational models have been around for some time now compared to NoSQL models and as a result they have grown to be more functional and stable systems over the years.
2. Less Support Every business should be reassured that in case a key function in their database system fails, they will have unlimited competent support any time. All rational model vendors have gone the extra mile to provide this assurance and made it sure that their support is available 24 hours which is not a step yet guaranteed by NoSQL vendors.
3. Business Analytics And Intelligence NoSQL models were created because of the modern-day web 2.0 web applications in mind. And because of this, most NoSQL features are focused to meeting these demands ignoring the demands of apps made without these characteristics hence end up offering fewer analytic features for normal web apps.

Any businesses looking to implement NoSQL model needs to do it with caution, remembering the above mentioned pros and cons they posse in contrast to their rational opposites.

## **Explain reasons to add a layer like Mongoose, on top on of a schema-less database like MongoDB**

### **Reasons to use Mongoose**

Mongoose is an object document modeling (ODM) layer that sits on top of Node's MongoDB driver. If your coming from SQL, it's similar to an ORM for a relational database. While it's not required to use Mongoose with the Mongo, here are four reasons why using Mongoose with MongoDB is generally a good idea.

1. Schemas MongoDB is a denormalized NoSQL database. This makes it inherently schema-less as documents have varying sets of fields with different data types. While this provides your data model with flexibility as it evolves over time, it can be difficult to cope with coming from a SQL background. Mongoose defines a schema for your data models so your documents follow a specific structure with pre-defined data types.
2. Validation Mongoose has built in validation for schema definitions. This saves you from writing a bunch of validation code that you have to otherwise write with the MongoDB driver. By simply including things like required:true in your schema definitions, Mongoose provides out-of-the-box validations for your collections (including data types).
3. Instance Methods Mongoose provides optional pre and post save operations for data models. This makes it easy to define hooks and custom functionality on successful reads/writes etc. You can also define custom methods that act on a particular instance (or document). While you can achieve similar functionality with the native MongoDB driver, Mongoose makes it easier to define and organize such methods within your schema definition.
4. Returning results Mongoose makes returning updated documents or query results easier. A prime example can be found with update queries. While the native driver returns an object with a success flag and the number of documents modified, Mongoose returns the updated object itself so you can easily work with the results.

### **Reasons not to use Mongoose?**

There are few reasons not to use Mongoose with MongoDB (especially if you are just getting started). For more advanced queries, it can be argued that Mongoose makes things more difficult and can slow performance. Advocates of the native MongoDB driver also argue that bringing ODM to a denormalized design entirely defeats the purpose of a NoSQL database.

### **Conclusion**

Despite the arguments against using Mongoose, it remains one of the most popular ODM tools for Mongo. If you are coming from a SQL background then using Mongoose will make the transition into a NoSQL environment much easier. It will also save you time writing your own validations and instance methods and is highly recommended for smaller DBs and basic Mongo operations.

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

This topic will be introduced in period-3

## **Explain, using your own code examples, how you have used some of MongoDB's "special" indexes like TTL and 2dsphere**

These two topics will be introduced in period-3

## **Demonstrate, using a REST-API you have designed, how to perform all CRUD operations on a MongoDB**

<https://vegibit.com/mongoose-crud-tutorial/>

<https://github.com/judre/Miniproject>

### 

## **Explain the “6 Rules of Thumb: Your Guide Through the Rainbow” as to how and when you would use normalization vs. denormalization.**

<https://keon.io/mongodb-schema-design/> <https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-1>

Here are some “rules of thumb” to guide you through these indenumberable (but not infinite) choices

1. favor embedding unless there is a compelling reason not to
2. needing to access an object on its own is a compelling reason not to embed it
3. Arrays should not grow without bound. If there are more than a couple of hundred documents on the “many” side, don’t embed them; if there are more than a few thousand documents on the “many” side, don’t use an array of ObjectID references. High-cardinality arrays are a compelling reason not to embed.
4. Don’t be afraid of application-level joins: if you index correctly and use the projection specifier then application-level joins are barely more expensive than server-side joins in a relational database.
5. Consider the write/read ratio when denormalizing. A field that will mostly be read and only seldom updated is a good candidate for denormalization: if you denormalize a field that is updated frequently then the extra work of finding and updating all the instances is likely to overwhelm the savings that you get from denormalizing.
6. As always with MongoDB, how you model your data depends – entirely – on your particular application’s data access patterns. You want to structure your data to match the ways that your application queries and updates it.