# JS Flow 2

## Contents

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

- It’s a very popular language, where you can use the same language in both frontend and backend

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

- It is easy to understand and quick to develop, as it does not require much code
- Easy to avoid non-blocking code.
- Requires less servers to obtain same amount of requests as other languages, it is because Node.js makes use of a single process to handle the multiple request

## Explain briefly how to deploy a Node/Express application including how to solve the following deployment problems

- Set up a server, let nginx point to the build folder, allow pm2 to run as background service

### Ensure that you Node-process restarts after a (potential) exception that closed the application

- Pm2 restarts the app

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

- pm2 command “startup” kan så starte pm2 ved start af serveren
- The pm2 command *startup* can start the pm2 at start of the server

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

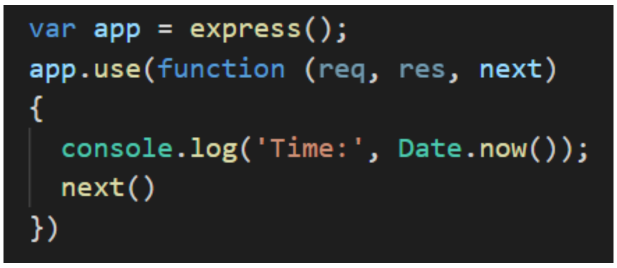
- Run the different servers on multiple different local port, adjust the configuration of the server in nginx

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

- Debug output is when you log out the console, for example *console.log()* while with the application logging has a plugin which handles the logging of what one will have in the files

- *Console.log()* is blocking and will make the code slower

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

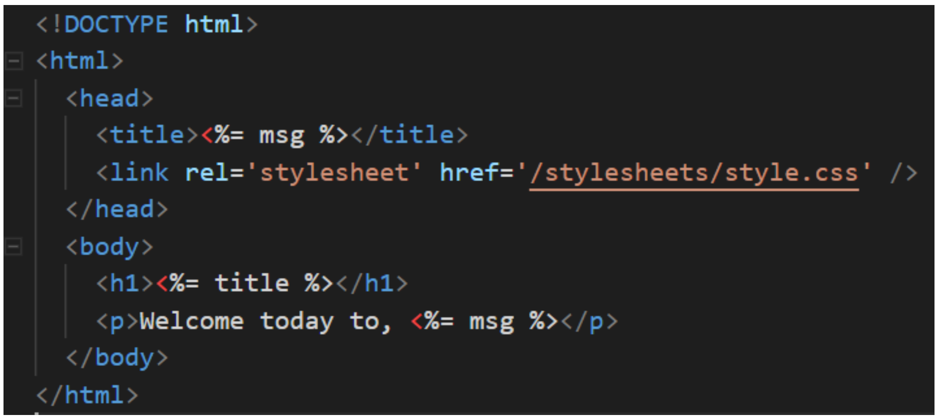
- Middleware functions are functions which has access to the request object, the response object and the next middleware function in the programs req-res cycle
- 

```
var app = express();
app.use(function (req, res, next)
{
    console.log('Time:', Date.now());
    next()
})
```

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

- Javascript template engines virker som jsp som vi kender det fra 2. semester. Det gør at vi kan skrive javascript kode direkte i html siderne. I stedet for .html filer vil vi have f.eks .ejs eller .jsp
- The JS template engine works as we know from the 2nd semester as *jsp*, *jsp* means that we can write javascript code directly in the html pages, and instead of the *.html* we will have *.jsp*

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

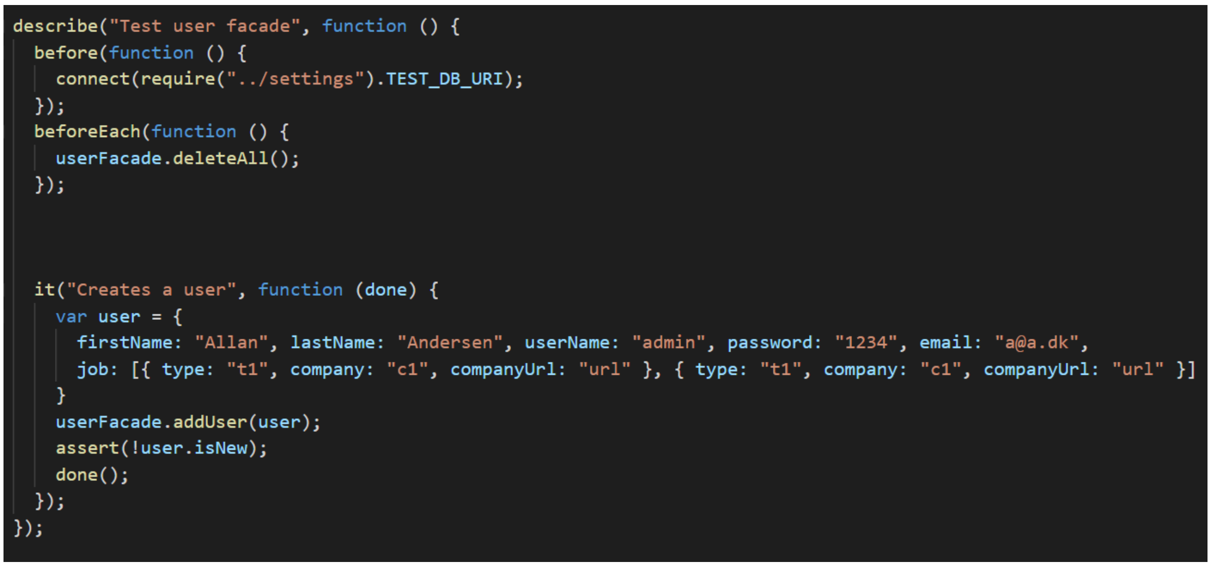
- 

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= msg %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome today to, <%= msg %></p>
  </body>
</html>
```

## 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.

- You can run servers locally and set up some Mocha test up against a REST-Api, where you fetch data from the endpoint and test what the return value is

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

- 

```
describe("Test user facade", function () {
  before(function () {
    connect(require("../settings").TEST_DB_URI);
  });
  beforeEach(function () {
    userFacade.deleteAll();
  });



  it("Creates a user", function (done) {
    var user = {
      firstName: "Allan", lastName: "Andersen", userName: "admin", password: "1234", email: "a@a.dk",
      job: [{ type: "t1", company: "c1", companyUrl: "url" }, { type: "t1", company: "c1", companyUrl: "url" }]
    }
    userFacade.addUser(user);
    assert(!user.isNew);
    done();
  });
});
```

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

- NoSQL describes a database type where the data is not placed in rows and colons, the structure does not need to be determined beforehand, it is determined by its content with following fieldnames

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

### Pros

- Better scalability
- Change can be made without stopping the database

### Cons

- Poor cross platform support

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

1. One: favor embedding unless there is a compelling reason not to
2. Two: needing to access an object on its own is a compelling reason not to embed it
3. Three: 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. Four: Don’t be afraid of application-level joins: if you index correctly and use the projection specifier (as shown in part 2) then application-level joins are barely more expensive than server-side joins in a relational database.
5. Five: 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. Six: 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.