Express Exercises

Problem-1 Understanding middleware

In this first exercise, we will start with an empty project. For the exercises that follow, we will use an Express "start project".

1) Create a new "empty project" and add a JavaScript file app.js.

2) Import Express: `npm install express`

3) Add the following code to create and start an Express instance:

```
var express = require("express");
var app = express();

//add your content here

app.listen(3000,function(){
  console.log("Server started, listening on: "+3000);
})
```

4) Don't take this as a real-life example, but simply as an example to see the middleware chain in action. We will add our own middleware as sketches below:

```
app.use("/api/calculator/:operation/:n1/:n2", function(req,res,next){
})
```

This will interact on all requests made to URLs like `/api/calculator/`**`add/10/10`** where the last three values are parameters (Observe the colon to indicate a parameter, these values can be obtained via `req.params.xxxx`, when the body-parser middleware is included)

5) Later in the app.js file, add a route-handler, which must handle all requests on the form: /api/calculator/*

```
app.get("/api/calculator/*",function(req,res){

})
```

6) Add the necessary code in the custom middleware (what you did in part-4) to build an object like:

```
var calculatorRequest = {
  operation: req.params.operation,
  n1: Number(req.params.n1),
  n2: Number(req.params.n2)
}
```

7) Put this object on the request object. Don’t forget to call `next();`

8) Show that you can access this object from the route-handler, and make a response according to the value of operation (just return the result as a plain string).

Hints: In order to read parameters you must include the body-parser module and initialize it like:

```
var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({extended: false}));
```

Problem-2 Server Side Templating with Pug (Jade), EJS, or Handlebars

You might feel that the following is a bit unfair, since a decision like this probably should be based on more experience. But use the slides, your little experience from the exercises done so far, and Google "jade templating", "ejs templating" and "handlebar templating" and see if you can find enough information to pick one of the three as your preference. If you have time, you could do the exercise with all three strategies to get a better feeling about what they each offer.

When done; create a new Node.js Express app and in the wizard pick the template language selected above.

1) Create a new folder ***model*** and add a JavaScript file ***jokes.js*** in the folder

Add a jokes array, with a few jokes to file as sketched below:

```
var jokes = [
  "A day without sunshine is like, night.",
  "At what age is it appropriate to tell my dog that he's adopted?",
  "I intend to live forever, or die trying"
];
```

Provide the file with a "façade" object as sketched below, and implement the missing methods:

```
module.exports = {
  allJokes : jokes,
  getRandomJoke : _getRandomJoke,
  addJoke : _addJoke
}
```

This file simulates a database façade.
Use on of the links below as reference for the following questions:

- https://pugjs.org/api/getting-started.html
- http://webapplog.com/handlebars/
- http://www.embeddedjs.com/

Replace xxx below with the e xtension used by your selected templating language (jade, hbs or ejs)

2) Add a new page (rendered by your template strategy) that should show a random joke

- Update *index.xxx* to provide a link to this page
- Add the routeHandler ("/joke") to index.js and follow the pattern from the existing handler to see how data can be passed to the new "template-page".
- Provide the new page with a link back to the "main" page

3) Add a new page (rendered by your template strategy) that should show all jokes (in any way you like)

- Update *index.xxx* to provide a link to this page and provide the new page with a link back to main.
- Add the routeHandler ("`/jokes`") to index.js and follow the pattern from the existing handlers to see how data can be passed to the new "template-page".

4) Add a new page (rendered by your template strategy) that should allow us to create new jokes

- Update *index.xxx* to provide a link to this page and provide the new page with a link back to main.
- Add a new routeHandler ("`/addjoke`") to index.js and follow the pattern from the existing handlers to see how to pass in data to the new "template-page".
- Add a new routeHandler ("`/storejoke`") to index.js. Let the new page submit its form data to this URL, fetch the new joke from the request object and store it in the "database".
  End this handler with a redirect back to `/addjoke`, so the user can continue to add jokes.

Problem-3 Express Sessions

In this part we will continue with the code provided by Problem 2.

We will provide the page with a very simple login page. On this, users has to provide a name (no password), before they are allowed to see any of the pages provide in problem-2.

There is absolutely no security involved in this problem, the main task is to see how sessions are created and maintained with Node and Express, and provide yet another middleware example.

Before you start make sure you understand the topic http sessions (2- 3. semester stuff, so you should do):

- http://en.wikipedia.org/wiki/Session_%28computer_science%29
- http://js2016.azurewebsites.net/express1/express1.html#12

Before we begin you should also figure out whether, what we are going do is legal, without an explicit accept from users, since sessions (out of the box) are implemented using Cookies.

Use these links to answer that question:

- http://ec.europa.eu/ipg/basics/legal/cookies/index_en.htm
- https://www.cookielaw.org/faq/

1)

We need a new piece of middleware for this problem:

In the terminal, do a: `npm install express-session`

Include the module in your *app.js* file`: var session = require("express-session");`

Initialize the module:

```
app.use(session({secret:'secret_3162735',saveUninitialized:true, resave: true}));
```

As ALWAYS, when using an external package the API-description for that package should be your reference: https://www.npmjs.com/package/express-session

2) Add a custom middleware handler before you start handling routes as sketched below:

```
app.use(function (req, res, next) {
  ...
});
```

This will be activated for all requests that comes into the server (why?). Add code to do the following:

Check if a *userName* has been added to the user session

    If a *userName* is found, the user is already "logged in" so just call: `return next();` (does what ?)

If Not check whether a *username* can be found in a form parameter (that is, we came from the login page, which we will create in step-3)

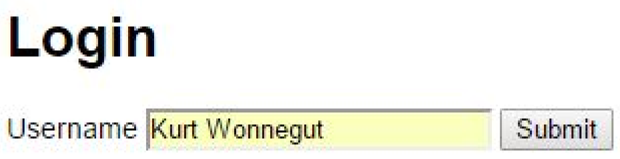
If a *userName* was found, store it in the session object in a property *userName* and call:

`return res.redirect("/")` to let the browser call back, this time with the *userName* in the session

If no *userName* was found, forward the page to a login page as sketched below:

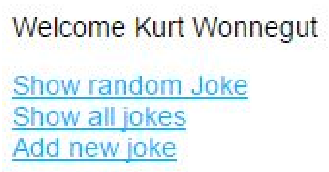
```
req.url = "/login"; //We will create this page in the next step
return next();
```

3) Create a new page (rendered by your selected template strategy) which should allow users to enter their name



4) Verify that it is impossible to see any of the pages (apart from the Login-page) before a user name is provided.

5) Change the main page (*index.xxx*) to show the users name as sketched below.



6) Use Chromes developer tools to monitor and explain the default way server-sessions are implemented by Node/Express using cookies.

7)

In this step we will add more information to the session object.

Change the three route handlers implemented in problem 2:

- /Joke
- /jokes
- /storejoke

so that each, add a counter variable (jokeCount, jokesCount and storeJokeCount) to the session object, and increment the counter for each invocation of a route.

8)

Now we have started to track user behaviour, using our sessions. Check, using the same links as in the start, whether we can do that without user accept.

- http://ec.europa.eu/ipg/basics/legal/cookies/index_en.htm
- https://www.cookielaw.org/faq/

9)

If the answer to the question above was (it is ;-) that this requires accept from users, change the program to obey the EU-rules. You or your company can be seriously fined if you don't obey these rules.

You can use one the templates supplied here for your solution:
http://ec.europa.eu/ipg/basics/legal/cookies/index_en.htm#section_4

This is generally something being taken very seriously and new related rules concerning our "personal data" are being rolled out right now.

Problem-4 Getting RESTless

Even when we have selected to generate our pages on the server, there could be many reasons to expose functionality via a REST-API (which ?).

1) Implement a simple REST-API, as sketched below, on top on our simple memory joke-model.

- `GET:   api/joke/random`
- `GET:   api/jokes`
- `POST:  api/joke`

Note: Since we by now is requiring that all requests have a session with a username property, you should add a check at the top of the middleware that checks for userName and just call `return next()`, if `res.url` starts with "/api/"

2) Test the API

Initially you can use Postman, but with Node it's actually much easier just to create the requests programmatically.

I suggest you use the request package for this (remember npm include –save request), which will let you perform a Post request as easy as:

```
var request = require("request");

var options = {
  url: "http://localhost:3000/api/joke",
  method: "POST",
  json : true,
  body : {joke : "I'm a joke"}
}
request(options,function(error,res,body){
  console.log(body.joke); //Assume the service returns the new Joke
})
```

Making a GET-request is even easier, see https://www.npmjs.com/package/request

3) Using the API from a client

Finally, just to end, where we ended last semester, design a simple page using Angular, JQuery or whatever to implement the same behaviour, rendered client side, as we did server side in part 2.