

# Back-end

MongoDB and NodeJS

# Backend Definition

- Back-end is the part of web-development that does not interact with the user
- Consists of 3 parts: Server, application, and database.
- Allows for HTML pages to show different updated data
- API's

# What to Download?

Starting with back-end, we are going to need to download 2 programs:

1. NodeJS: <https://nodejs.org/en/> and choose the LTS version. NodeJS is the language that we're going to be using for our back-end and the installation is necessary to implement all the necessary commands in terminal. NodeJS also includes NPM!
2. MongoDB:

# NodeJS Basics

- Create a file called app.js (<anything\_you\_want>.js)
- Node's syntax is just like JavaScript's. You can write if statements, loops, create objects...
- In order to compile something with NodeJS, we use terminal. Navigate using terminal to the file where app.js is saved and type "node app.js" to compile and run it. We just compiled JavaScript!

# Hello World Example

```
for (let i = 0; i < 10; i++)  
{  
  console.log("Hello World!");  
}
```

This will print Hello World to your console 10 times

# Using NPM

- NPM is a package manager for JavaScript. This means that NPM is a place where anyone can access countless libraries and frameworks.
- We can look for specific packages in the NPM website:  
<https://www.npmjs.com/>
- For back-end, we will mainly be using three packages: express, ejs, and body-parser.
- Using NPM is easy! Just navigate using terminal to the folder where your file is at and type “npm install <package\_name>” and npm will do all the work.
- It is normal to have several warning when installing packages.
- To install express, use: “npm install express”.
- To install ejs, use: “npm install ejs”.
- To install body-parser, use “”npm install body-parser

# Example of adding our two packages

```
[giovannis-mbp:npmTutorial Giovanni$ npm install express ]
npm WARN saveError ENOENT: no such file or directory, open '/Users/Giovanni/Google Drive/KU/Programming KU/Web Page/Info Sessions/Backend/npmTutorial/package.json'
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN enoent ENOENT: no such file or directory, open '/Users/Giovanni/Google Drive/KU/Programming KU/Web Page/Info Sessions/Backend/npmTutorial/package.json'
npm WARN npmTutorial No description
npm WARN npmTutorial No repository field.
npm WARN npmTutorial No README data
npm WARN npmTutorial No license field.

+ express@4.16.3
added 49 packages in 2.562s
[giovannis-mbp:npmTutorial Giovanni$ npm install ejs ]
[giovannis-mbp:npmTutorial Giovanni$ npm install ejs ]
npm WARN saveError ENOENT: no such file or directory, open '/Users/Giovanni/Google Drive/KU/Programming KU/Web Page/Info Sessions/Backend/npmTutorial/package.json'
npm WARN enoent ENOENT: no such file or directory, open '/Users/Giovanni/Google Drive/KU/Programming KU/Web Page/Info Sessions/Backend/npmTutorial/package.json'
npm WARN npmTutorial No description
npm WARN npmTutorial No repository field.
npm WARN npmTutorial No README data
npm WARN npmTutorial No license field.

+ ejs@2.5.7
added 1 package in 1.081s
```

# Including Express

- Express is a very handy resource that allows us to easily handle calls to web pages.
- In order to implement it to our program, we use the *require* keyword.
- With this keyword, we can specify which package we wish to use. However, since this is JavaScript, we want to assign this require to a variable.
- So the code that we want to add to our app.js file is:

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

- To use express we need to write it in function form so we write:

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



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

# Hosting a local server using express and NodeJS

- Now that express has been included on our project, we can finally use it!
- An application is not very impressive if it is stuck in terminal, so let's host our own local server to bring it online
- This is done with the following code:


```
app.listen("8080");
```

- This is essentially telling our program to host our code in port 8080:  
localhost:8080

- Try to run app.js on terminal now. You'll see that the flashing cursor changes:

---

```
[giovannis-mbp:MoreExpress Giovanni$ node app.js
```



- This means that our application is now online! Go to your browser and type the following url: localhost:8080
- If everything is working correctly then you should get a page with text like "Cannot GET /".
- Type ctrl-c inside terminal to stop hosting localhost (we have to do this every time we want to make a change)
- Let's now fix this cannot GET error.

# GET Requests

- GET requests are one of the two main HTTP requests, the other being POST.
- GET's work by requesting data by a specified resource.
- When we type our localhost:8080 url, we are making a get request to our local server to give us whatever is in the root of localhost. But we haven't written any code that specifies what to show, so we get a Cannot GET "/" message, "/" meaning our root.
- To fix this, we can link a .ejs file so that it opens when a user makes a GET request to our localhost root.

# GET Request Code

- Inside app.js, write the following code after the express include:

```
app.get("/", function(req, res){  
  res.render("home.ejs");  
});
```

- There's a lot to discuss in these three lines of code so let's dive right in.
- We previously defined app as express() from the line "var app = express()". App.get takes in two parameters: a path (in this case we are using root) and a function. This function in turn takes two parameters that we call req and res. These stand for request and response respectively

```
app.get("/", function(req, res){  
  res.render("home.ejs");  
});
```

- We don't need to worry about req just yet, however, we can see that res is being used.
- Responde basically tells our GET what to show. In this case, we are showing whatever is inside the file called home.ejs.
- If we try to run this code and open localhost:8080, we will get an error. Why? We haven't yet created the home.ejs file!

# .ejs Files

- EJS stands for embedded JavaScript and is basically HTML that can have JavaScript in it.
- The reason why we use ejs files instead of simple html files is because html files are static in nature: Meaning they can't change.
- Webpages usually display different information based on user input and whatever is in the database, so we usually don't want a static HTML page.
- We can write normal html code here, or we can have a mixture of html and JavaScript. For now, we are going to treat it as an HTML file.

# Create home.ejs

- Create a new folder in the same level as your app.js. Call this folder “views”.
- Inside views, create a file called home.ejs.
- Express will automatically look for a folder called views so any file we add inside the views folder will be found by express.
- Now, open home.ejs and type a simple h1 such as:

```
<h1>Welcome to the Home Page!</h1>
```

Don't forget the  
HTML template!

- Save the changes and run app.js again. Open localhost:8080 and if everything was done properly you will see your h1 printed on the screen.
- If you get an error about not finding home.ejs, that's because the views folder name or the file name were misspelled or you forgot to install the ejs package



# JavaScript in .ejs Files

- As mentioned before, ejs files are special because they allow us to mix html and JavaScript code. Let's try this!
- Let's right a for loop that prints a paragraph 5 times. The code is shown below

```
<% for(let i = 0; i < 5; i++){ %>  
  <p>Hello!</p>  
<% } %>
```

- This is a simple for loop. Every time we use JavaScript in an ejs file, it has to be of the form `<% code %>`. However, we don't want our HTML code inside these brackets, so we have to constantly open and close these `<% %>`'s for every line of JavaScript we add.

# Creating more GET Requests Code

- Let's add some more get requests that are outside root.
- For example, let's write some code that responds to a request made to `localhost:8080/myRequest`:

```
app.get("/myRequest", function(req, res){  
  res.render("myRequest.ejs");  
});
```

- Now let's create the `myRequest.ejs` file (again, create it inside the views folder).
- For mine, I added an `h1` that says "This is the myRequest file!"
- Now, when we run **`localhost:8080/myRequest`** we will see the `h1` we wrote.

# How to Handle Requests to Pages that don't Exist?

- When typing random things into an url, we may sometimes find pages that say something like ERROR 404: Page not found.
- These are custom made pages created to handle GET requests to pages that do not exist.
- We can do this by typing an asterisk (\*) as our first render parameter:

```
app.get("*", function(req, res){  
  res.render("notFound.ejs");  
});
```

- Now you can create the notFound.ejs file and write your very own 404 error page!
- **IMPORTANT:** Write the "\*" get after all your other requests. If you don't, notFound.ejs will always be executed

# Adding css to .ejs Files

- Our .ejs files are looking a bit bland, so let's learn how to add css to them
- We first want to create a folder where all of our css files will be. Let's name it css.
- Now let's create a file called style.css inside our css folder and add whatever css we want to it.
- Go to your app.js file and include this folder:

```
app.use(express.static("css"));
```

- Finally, we can include our css in any .ejs file just like we would include it in an html file. The only difference is that we must add a / before the file name:

- Finally, we can include our css in any .ejs file just like we would include it in an html file. The only difference is that we must add a / before the file name:
- Inside home.ejs add the following code in your header:

```
<link rel="stylesheet" href="/style.css">
```

- Now you can have css on your .ejs files!

# POST Requests

- POST requests, unlike GET requests, submit data when they are called. Think of a POST request as submitting an online form. Once you click submit, your information is taken and something is done with it. That's what the POST request does.
- Its syntax is very similar to that of a GET request. For example, let's write the following POST request code:

```
app.post("/addInfo", function(req, res){  
  res.render("addInfo.ejs");  
});
```

# Requesting a POST

- Requesting a POST is not as easy as typing its name in the url. We need to write some HTML code that generates a POST request. As mentioned earlier, a form is a great way to do just that.
- Inside our myRequest.ejs file, create a form:

```
<form action="/addInfo" method="POST">  
  <input type="text" name="myName" placeholder="name">  
  <button>Submit</button>  
</form>
```

- Now, when we go to localhost:8080/myRequest we will see a form. Once we click submit, it will take us to our addInfo.ejs file (our POST file)!
- Note that name="myName" will be what we use to identify what the user typed

# Get Information from a POST

- This is where we are using the data-parser package.
- Let's include body-parser in our app.js file and tell express to use it (write this just after your express include):

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

- Don't worry too much about the whole app.use line. It is one of the things that you tend to copy-paste on most of your applications.
- Now let's print our name in the POST file.



# Passing in POST Parameters

- Edit our previous app.post to grab the input from the user:

```
app.post("/addInfo", function(req, res){  
  let newName = req.body.myName;  
  res.render("addInfo.ejs", {newName: newName});  
});
```

- Here, newName is whatever our user gave us as an input.
- The {newName : newName} part means that we are grabbing our newName variable that we just defined and setting it equal to a variable called newName that we use inside the .ejs file. These two newNames can have different variable names but people usually use the same one.

- Now, inside our addInfo.ejs, we print the user input:

```
<h3>Hello, <%= newName %></h3>
```

- Note the equal sign inside the JavaScript brackets. This is used to get the value inside our variable.
- If everything was done correctly, you should receive a welcome after posting your name.

# Databases

- A database is a method of storing information and access this stored information.
- Think of them as tables in excel.
- 2 types:
  - Relational: SQL
  - Non-relational: Mongo

# Install MongoDB

- <https://docs.mongodb.com/manual/administration/install-community/>
- For Mac:
  - download mongoDB from here: <https://www.mongodb.com/download-center#community>
  - In terminal, type `mkdir -p /data`
  - Next, in terminal, type `mkdir -p /data/db`
  - Then, in terminal, type `sudo chmod -R go+w /data/db` This may prompt you for your admin password.
  - Now go to your mongoDB bin folder through terminal and type `./mongod`
  - MongoDB should now be running!
  - You can stop it any time by using `ctrl-c`.

# Using MongoDB

- In order to use MongoDB, we must first install and initialize Mongoose.