Digital Learning Hub

Develop your digital skills

The mission of the Digital Learning Hub is to help reduce the digital skills gap in Luxembourg. We do so by offering in-person, affordable courses in various fields of IT and related sectors.

Databases using SQL databases (MySQL) and NoSQL databases (MongoDB)

Instructor : Vanessa Al Daham
Course Language: English

Welcome

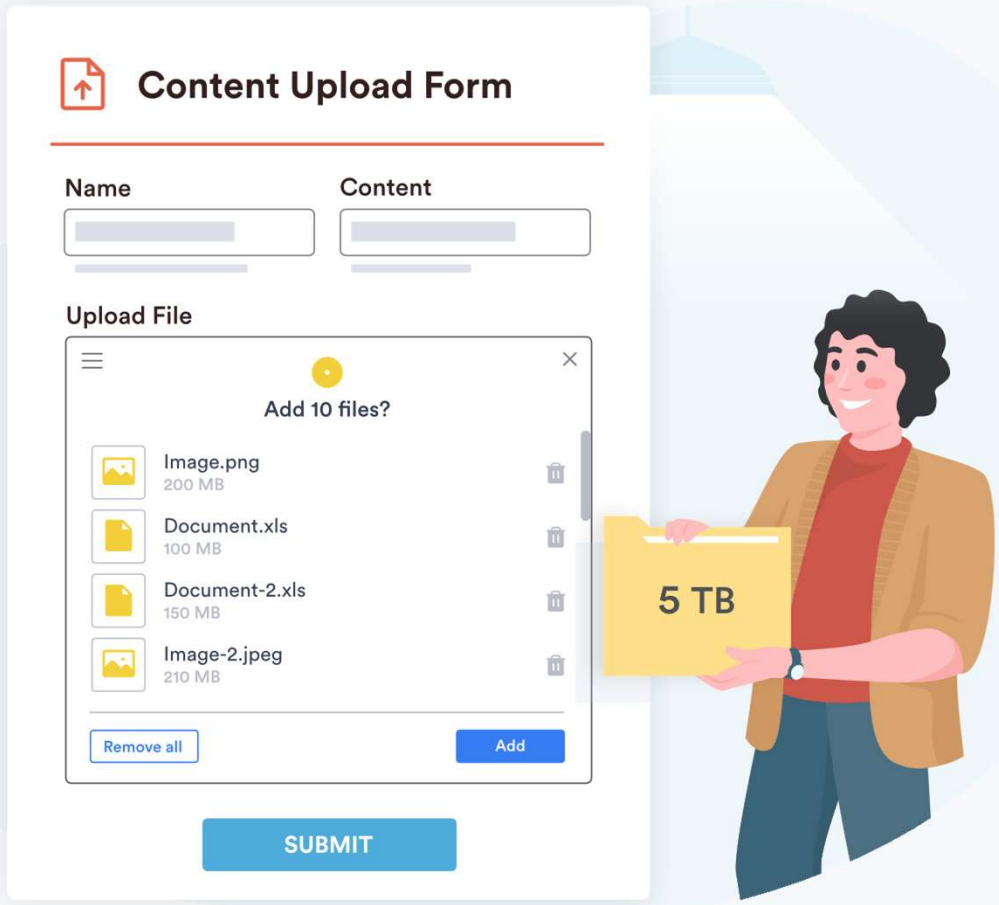
Let’s get to know each other

What’s your name?

Where do you work? Do you have any experience in it?

What are your expectations of this course?

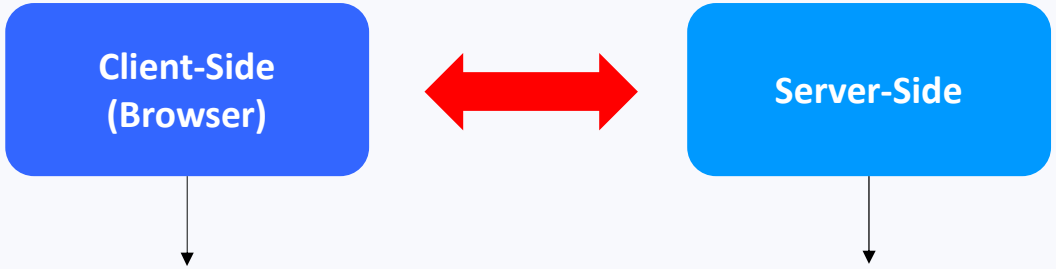
Intermediate Course



Digital Learning Hub

Advanced Backend

Adding Files Upload to a Website



- User needs to be able to select (and maybe preview) a file
- Submitted form should include file + other data

- Incoming file should be extracted (just like the other data)
- File has to be stored + (possibly) served

Adding a file Picker to a Form ->

Advanced Backend

Adding a File Picker to a Form

Download the 01-Starting Project – Materials to get Started.

1. Run the following command to initialize your backend: **npm init**
2. Install the packages needed for this mini-project:
   >> npm install express
   >> npm install ejs
   >> npm install mongodb
   >> npm install nodemon - -save-dev
   Update the package.json in **"scripts"**: **"start": "nodemon app.js"**

- **To add a file picker we do so as the following:**
  <input type="file"> - Add this element in the div(form-control) after the label element
  Give that element an **id** and **name attribute** with the value "**image**"
  Give that input a **required** attribute
  Give that input the **accept attribute** to only accept image files as the following: **accept=".jpg,.jpeg,.png" || "image/jpg,image/png"**

- For the **form element** to be able to tell the browser how this data will be packed into the request body when the browser will send the request to the server, this is on us to tell the browser how that process should be done by using the following:
  enctype="multipart/form-data" – used when we have a file picker in a form

Parsing Incoming File Uploads With The multer Package ->

Advanced Backend

Parsing Incoming File Uploads With The multer Package

For dealing with files in Nodejs we have many packages we could use, but one is popular and easy to use which is <<multer>>
You can search for it as : multer npm
You can visit the website too: https://www.npmjs.com/package/multer

- Use the following command in your Project terminal to install this package:
  npm install --save multer
- Now you can restart your server with: **npm start**
- In the routes folder, in the user.js file add the following:

This is a function that expect a file to be attached, this is why we need to apply the multer middleware to. Therefore this Middleware should not be applied to all of the routes, but only to this one, update the user.js file as follows:

```
const multer = require('multer');
const upload = multer({ });

router.post('/profiles', upload.single('image') ,function(req,res){
        const uploadedImageFile = req.file;
        const userData = req.body.;
        res.redirect('/')
});
```

How to Store files on the backend ->

Advanced Backend

How to store Files on the backend

- **Storing a file in a database is not ideal**, because Databases are not optimized for file storage and retrieval. **Files should only be stored on our File System (Hard Drive)** and we only **store the path to that file in the Database**

➢ Create a new folder **<<images>>** in the project directory
➢ Update the multer function as follows:

```
const upload = multer({ dest: 'images'}); --Now test
```

➢ **You will notice that with this previous update the files are being saved in the <<images>>** folder but in a weird format and you cannot really see the image! Therefore the solution is the following:

```
const storageConfig = multer.diskStorage({
        destination: function(req,file,cb){
                cb(null, 'images');
        },
        filename: function(req,file,cb){
                cb(null, Date.now()+ '-'+ file.originalname);
        },

});

const upload = multer({
        storage: storageConfig
});
```

Storing file name in Database ->

Advanced Backend

Storing file name in Database

- In the user.js file we need to import the database file as follows:

```
const db = require('../data/database');
```

- And update the previous function as follows:

```
router.post('/profiles', upload.single('image') ,async function(req,res){
    const uploadedImageFile = req.file;
    const userData = req.body.;
    await db
      .getDb()
      .collection('users')
      .insertOne({
        name: userData.username,
        imagePath: uploadedImageFile.path,

      });
    res.redirect('/')
});
```

Serving Uploaded files to website Visitors ->

Advanced Backend

Serving uploaded files to website visitors

- We need to update the function that handles the get request to the **/** route as follows:

```
router.get('/', async function(req,res){
    const uploadedImageFile = req.file;
    const users = await db
      .getDb()
      .collection('users')
      .find().toArray();
    res.render('profiles',{users:users});
});
```

- Update by yourself the **profiles.ejs** file to add a loop for in the unordered list element to display a list of users

- In the **app.js** add the following:

```
app.use('/images',express.static('images'));
```

Adding an image Preview Feature ->

Advanced Backend

Adding an image preview Feature

- When we select the image to upload it we want to listen to the file clicker to update an image preview. In the **public** folder create a **scripts** folder and inside of it a **file-preview.js** file

- **Add the path for that script in the new-user.ejs file and add the defer attribute to it. Create an image element in the image section with an id = "image-preview"**
- In the file-preview.js add the following:

```
const filePickerElement = document.getElementById('image');
const imagePreviewElement = document.getElementById('image-preview');

function show-preview(){
        const files = filePickerElement.files;
        if(!files || files.length === 0){return;}
        const pickedFile = files[0];

}
filePickerElement.addEventListener('change', show-preview ) ;
imagePreviewElement.src = URL .createObjectURL(pickedFile);
imagePreviewElement.style.display = 'block';
```

Why would we need AJAX ->

AJAX

AngularJS AJAX - $http

Advanced Backend

What is AJAX and why would we need it?

- AJAX is: **A**synchronous **J**avaScript **A**nd **X**ML
- AJAX represents a general idea: **Sending HTTP requests via JavaScript**

- **To realize this we have Two Built-In Browser-Side JavaScript Features**

XML Request **fetch()**

1. We have previously sent many HTTP requests **without AJAX:**
   - **Through URLs :** send a GET request to a URL
   - **Clicking a link :** Send a GET request to URL
   - **Submit a form :** Send a GET oR POST request to URL

This method of use always leads to a new page being loaded

2. Sending HTTP requests **with AJAX:**
   - Send a HTTP request via browser-side JS
   - Handle response in the same script code

This method gives us full control over the browser behavior and can prevent loading a new page

What is AJAX ->

Advanced Backend

What is AJAX

- **XMLHttpRequest :** an Object built-into browser-side JavaScript that contains utility methods for sending HTTP requests via JavaScript – - **Originally developed to send XML data - – Commonly used in a form of a third package library <<axios>>**

- **XML:** Extensible Markup Language – - **It is a data format for formatting / Structuring text data in a machine readable way – XML is no longer used nowadays, it has been replaced by JSON**

Note: HTML is based on XML, but HTML is standardized and XML is not.

- **Fetch() :** an object built-into browser-side JavaScript that contains utility methods for sending HTTP requests via JavaScript – - **uses modern JS features such as promises -- - it is an alternative to the XMLHttpRequest object and libraries**

The starting project ->

Advanced Backend

The starting Project

- **Download the folder 01-StartUp – Material to get started**
- In the terminal, start by:

  >> npm init
  >> npm install express
  >> npm install ejs
  >> npm install mongodb
  >> npm install nodemon - -save-dev
  Update the package.json in **"scripts"**: **"start": "nodemon app.js"**

- Try to start the project with: **nodemon app.js**

Sending and Handling a Get AJAX request ->

Advanced Backend

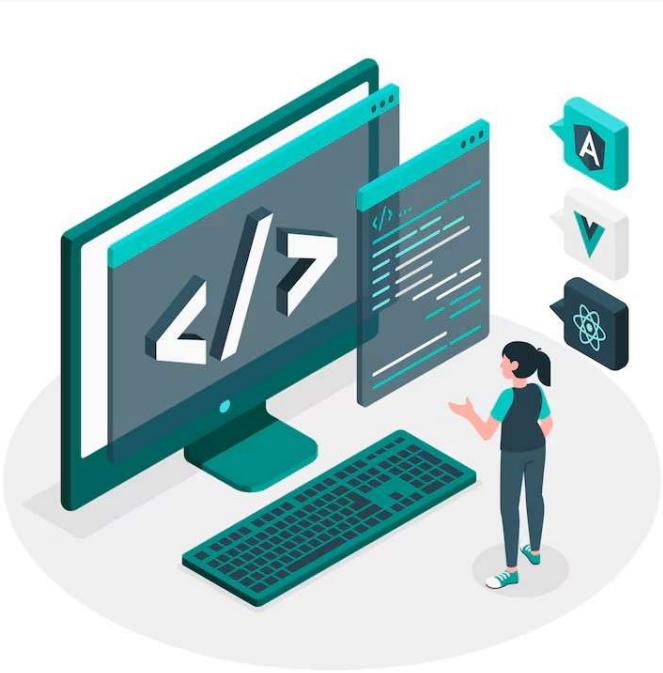
Sending and Handling a Get AJAX request

- We will start **First by writing a browser-side JS code:**
  - **>> Create a scripts folder in the public folder**
  - **>> Create a comments.js file in the scripts folder**

- The post-detail.ejs file contains the **Load Comments** button:
  - **>> In the head section add the script src path to the comments.js script file with the defer**
  - **>> Give it an id="load-comments-btn"**
  - **>> Add to it an data-postid=<%= post._id %>**

- In the comments.js file add the following:

```
const loadCommentsBtn = document.getElementById('load-comments-btn');

async function fetchCommentsForPost(event){
    const postId = loadCommentsBtn.dataset .postid;
    const response = await fetch(`/post/${postId}/comments`) ;
    const data = await response.json();
}

loadCommentsBtn.addEventListener('click', fetchCommentsForPost ) ;
```



- In blog.js get the get request route we have **/post/:id/comments** the function should return the following:

```
res .json(comments);
```

Updating the DOM based on the response ->

Advanced Backend

Updating the DOM based on the response

We want to update the page once we want to load the comments to display them on the page.

- In the post-detail.ejs file in the **comments** section:
  >> Remove all the EJS code, and only keep the p and the button elements

- Update the comments.js file as follows:

```
const commentsSectionElemenrt = document.getElementById('comments');

function createCommentsList(comments){
    const commentListElement = document.createElement('ol');
    for(const comment of comments){
        const commentElement = document.createElement('li');
        commentElement.innerHTML = `
          //copy the comment-item.ejs html file content and put the values between
          ${}`;
          commentListElement.appendChild(commentElement);
    }
    return commentListElement ;
}
```

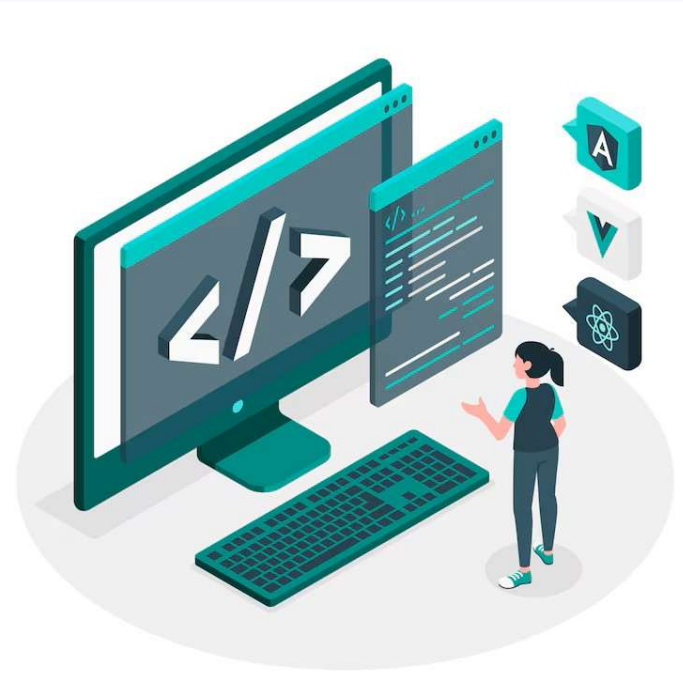
Updating the DOM based on the response ->

Advanced Backend

Updating the DOM based on the response

- Update the function fetchCommentsForPost as follows:

```
const commentListElement =  createCommentsList(data);
commentsSectionElemenrt .innerHTML = '';
commentsSectionElemenrt.appendChild(commentListElement);
```



Preparing the POST request data ->

Advanced Backend

Preparing the POST request data

We will still use AJAX to make it so that when we add a comment the page does not reloads entirely and we lose the view of all the loaded comments.

- In the post-detail.ejs file in the **form** section that handles the comment:
  >> Remove the action and the method attributes

- Update the comments.js file as follows:

```
const commentsFormElemenrt = document.querySelector('#comments-form form');
const commentsTitleElemenrt = document.getElementById('title');
const commentsTextElemenrt = document.getElementById('text');

commentsFormElement.addEventListener('submit', saveComment ) ;

function saveComment(event){
        event.preventDefault();
        const enteredTitle = commentsTitleElemenrt.value;
        const enteredText = commentsTextElemenrt.value;
}
```

Sending and Handling a POST AJAX request ->

Advanced Backend

Sending and Handling a POST AJAX Request

- Update the function saveComment as follows:

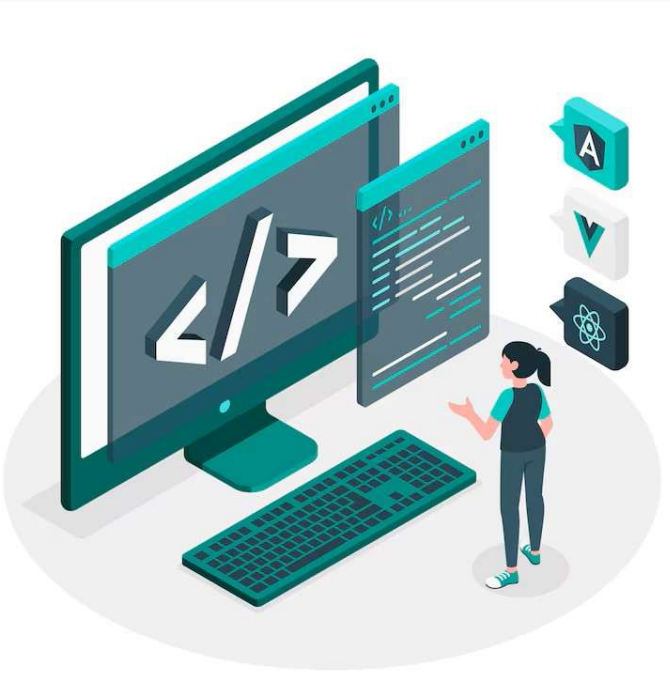
```
const postId = loadCommentsBtn.dataset .postid;
const comment = { title : enteredTitle , text : enteredText };


 fetch(`/post/${postId}/comments`,{
        method: 'POST',
        body: JSON.stringify(comment),
        headers: {'Content-Type': 'application/json'},
});
```

- In the app.js file add the following:

```
app.use(express.json())
```

- In the blog.js file add the following for the post request for the .post/:id/comment URL:

```
res.json({message: 'comment added'})'
```



Improving the User Experience->

Advanced Backend

Improving User Experience

We now make the recent added coments show up directly after being added.

- Update the function saveComment as follows:
  - >> Add the async keyword before the function
  - >> Add the await keyword before the fetch request
  - >> Store the fetch promise in a const response variable
  - >> call the fetchCommentsForPost function at the bottom of it

- Update the function fetchCommentsForPost as follows:
  - >> Add a condition to check if the data exist and it has a length greater than 0
  - >> move the rest of the code in this function to be wrapped by the previous condition
  - >> Otherwise if the condition is not met set the following: commentsSectionElemenrt.firstElementChild .textContent = “We could not find any comments”

Handling Errors – Server-Side Technical ->

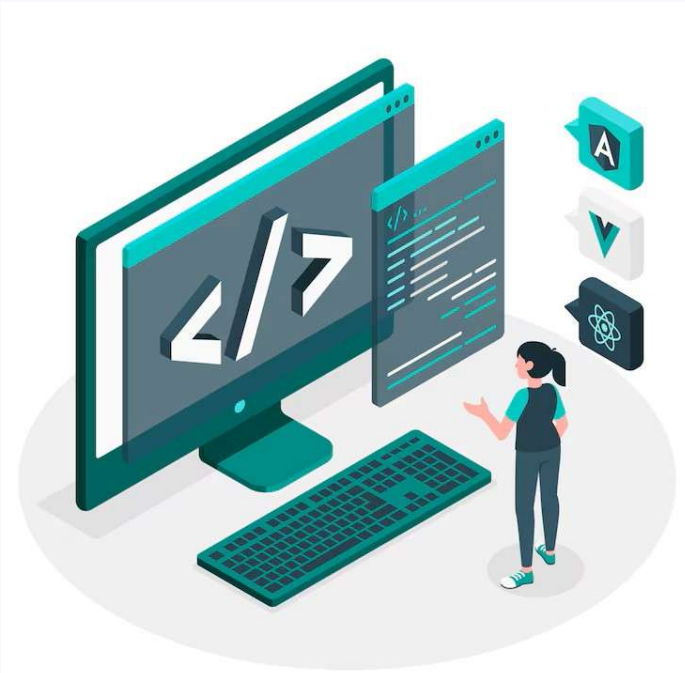
Advanced Backend

Handling Errors Server-Side Technical

We want to handle the case when adding a comment was unsuccessful

- Update the saveComment function as follows:
  - >> Before calling the fetchCommentsForPost check for the response.ok status, if is true you call that function
  - >> Otherwise we will display an error message using the alert
  - >> Move the response variable and the previous condition entirely in a try block and the error message to the request failure in catch block

- Update the fetchCommentsForPost function as follows:
  - >> Check for the response.ok status, if it is not ok send and alert <<catching comments failed>> and return
  - >> Now move the entire function body in a try and in a catch error block an alert <<Getting comments failed>



More HTTP Methods ->

Advanced Backend

More HTTP Methods

Default Browser Methods

GET

- Fetch some data
- Enter a URL
- click a link
- Form with method **GET**

POST

- Store some data
- Method **POST**
- Form with method POST

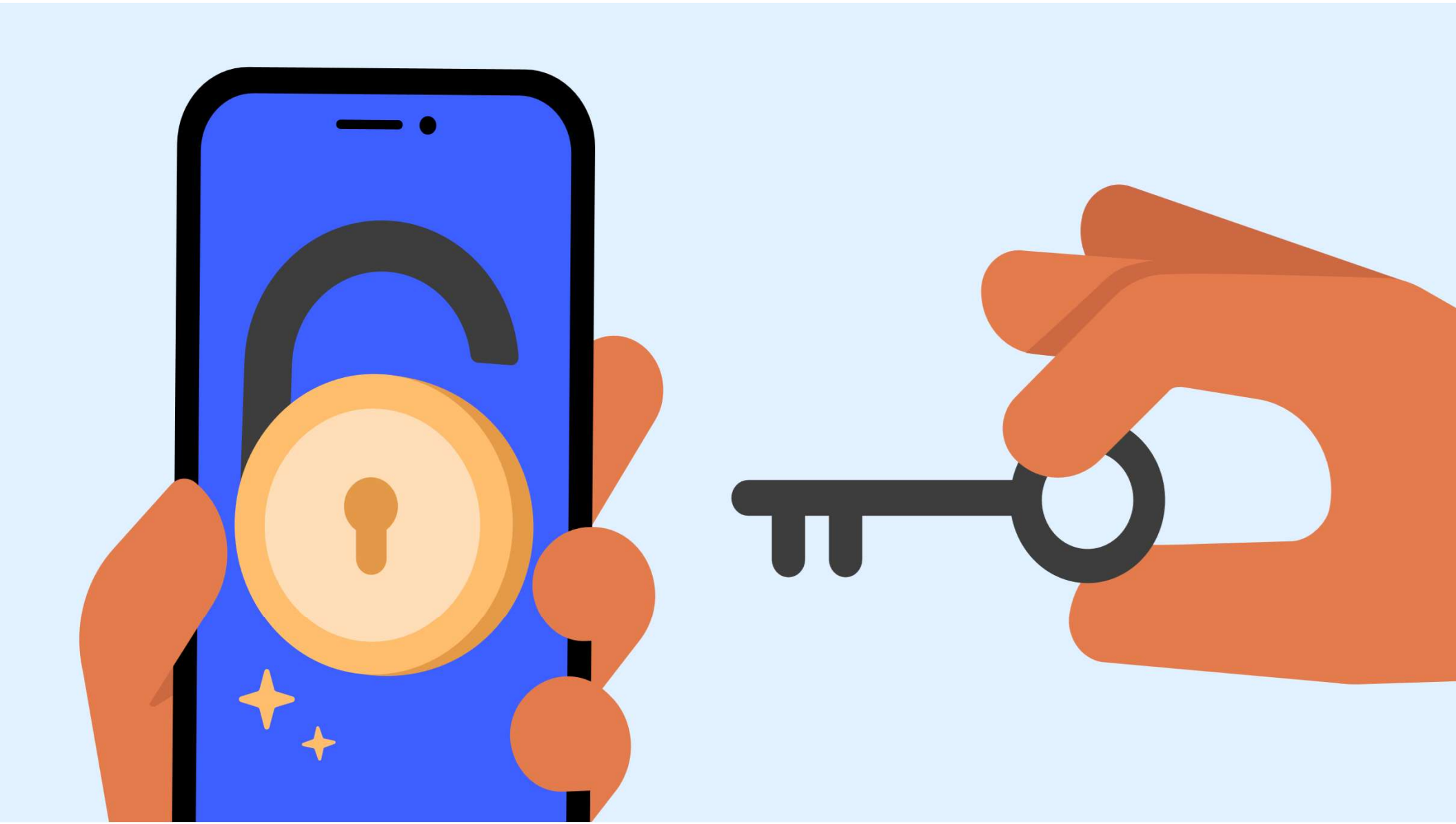
Available via AJAV/ JS driven HTTP requests

PUT Replace / update some data
Method **PUT**

PATCH update some data
Method **PATCH**

DELETE Delete some data
Method **DELETE**

What is Authentication and why we would add it ->



Advanced Backend

What is Authentication and we would add it

Usually on websites, there are certain pages that should only be accessible by authenticated users
Such as:

- Personal Profile on a Social network site
- Your shopping cart and order history in an online shop
- The administration area of your own blog website

To create Authentication for our websites we need the following steps:

- Create a sign up four our users to create accounts(email + password)
- Creating a Login functionality (enter email+ password)
- Establish user Authentication to allow access to protected pages

The start up project

- Download the **01-starting-project – Materials** folder to start with this mini project

Adding a Sign up Functionality ->

Advanced Backend

Adding a Sign up Functionality

- This functionality is represented by the **post request** for the **/signup** URL. Update it as the following:
  - We will retrieve the information from the form:

```
const userData = req.body;
const email = userData.email;
const confirmEmail = userData["confirm-email"];
const password = userData.password;
```

  - Prepare the data we want to send to the database:

```
const user ={
        email: email,
        password: password
};
```

  - We will insert these data into the database:

```
await db.getDb()collection('users').insertOne(user);
```

  - Redirect the user to the login page:

```
res.redirect('/login');
```

Hashing Passwords ->

Advanced Backend

Hashing Passwords

Not Hashing the password and storing it in its raw format is in our Database is a big security flaw !!

- **Hashing password:** It means to convert a string to a **non-decodable** different string
  Hashed values cannot be reverted, decoded, or transformed back into the original value

- **To hash a password we need an algorithm**
  - A famous package that does that for us is **bcrypt.js**
  - **Install is by running : npm install bcryptjs**

- **Import bcryptjs:**

```
const bcrypt = require('bcryptjs');
```

- **Hash the password:**

```
const hashedPassword = await bcrypt.hash(password, 12);
```

- **Update the user object to store the hashedPassword instead of the password**

Adding user Login Functionality ->

Advanced Backend

Adding user Login Functionality

- This functionality is represented by the **post request** for the **/login** URL. Update it as the following:
  - **Start by retrieving the needed data for confirmation:**

```
const userData = req.body;
const email = userData.email;
const password = userData.password;
```

  - Retrieve the matched data from the database:

```
const existingUser = await db.getDb()collection('users').findOne({email:email});
```

  - **Check if the user does not exists:**

```
If(! existingUser){
        console.log('could not log in!')
        return res.redirect('/login');
}
```

  - **Checking if the user exists:**

```
const passwordsAreEqual = await bcrypt.compare(password, existingUser.password);
If(! passwordsAreEqual){
        console.log('could not log in! – passwords does not match')
        return res.redirect('/login');
}
```

  - **If the previous checks failed means the user is authenticated:**

```
res.redirect('/admin');
```

Validating Sign up Information ->

Advanced Backend

Validating Sign up Information

We will make sure that the email is not empty, does not already exist in my database

- Check if the entered email and the confirm email entered are correct data:

```
If(!email || !confirmEmail || !password || password.trim().length < 6
|| email !== confirmEmail || !email.includes('@')){
        return res.redirect('/signup');
}
```

- Check if the user already exists:

```
const existingUser = await db.getDb()collection('users').findOne({email:email});
If(existingUser){
        return res.redirect('/signup');
}
```

Introduction to Sessions and Cookies ->

Advanced Backend

Introduction to Sessions and Cookies

For the backend every incoming request is similar which does not help the server to tell if a user should be granted access or not for that an **<<entry ticket>>** **[Session]** must be saved on the server and handed out to the visitor.
Every session will have its unique ID.

- **Session :** Server-side – We used the express-session packages – npm install express-session
- **Session Cookies :** Client-side (used to track the user) – We use the cookie-parser package npm install

- Go to app.js to set up the session feature:

```
const session = require('express-session');
app.use(session({
    secret : 'extremely-secret',
    resave : false,
    saveUninitialized : false,
    store : sessionStore
}));
```

Introduction to Sessions and Cookies ->

Advanced Backend

Introduction to Sessions and Cookies

- Install and use the compatible session store :

```
>> npm install connect-mongodb-session
const mongodbStore = require('connect-mongodb-session'); // in app.js
const MongoDBStore = mongodbStore(sesstion);
const sessionStore = new  MongoDBStore({
    uri : 'mongodb://localhost:27017',
    databaseName : 'auth-demo',
    collection : 'session'
});
```

Storing Authentication Data in Sessions ->

Advanced Backend

Storing Authentication Data in Sessions

We will use now the session for granting access to the users.

- The **/login post route** should be update as the following before we redirect to the **/admin**:

```
req.session.user = {
      id : existingUser._id.toString(),,
     email : existingUser.email
};
 req.session.isAuthenticated = true;
 req.session.save(function(){
      return res.redirect('/admin');
});
```

The session will be automatically saved in the database , express-session is doing that by default

Using Sessions and Cookies for controlling Access ->

Advanced Backend

Using Sessions and Cookies for controlling Access

➢ The **/admin get route** should be updated as follows:

- Check if the request session is not Authenticated:

```
If(! req.session.isAuthenticated ){
        return res.status(401).render('401');
}
```

- In the app.js update the use of the session to add the cookie duration as follows:

```
app.use(session({
      secret :  'extremely-secret',
      resave : false,
      saveUninitialized : false,
      store : sessionStore,
      cookie : { //optional
            maxAge : 30 * 24 * 60 * 60 *100, //expires after 30 days
      }
}));
```

Adding the logout Functionality ->

Advanced Backend

Adding the logout Functionality

- The **/logout post route** set the user in the session to null:
  ```
  req.session.user = null;
  req.session.isAuthenticated = false;
  ```

- Redirect the user back to the initial page:
  ```
  res.redirect('/');
  ```

➢ Let's have a closer look at cookies

Diving deeper into sessions ->

Advanced Backend

Diving Deeper into Sessions

- When a user wants to sign up and there is a error in the data provided we want to redirect the user to the same page but keep their inputs so they could just modify them instead of re-typing everything. **In the post request for the .signup route**

- Update the validation condition to create a temporary session as follows:

```
If(!email || !confirmEmail || !password || password.trim().length < 6
|| email !== confirmEmail || !email.includes('@')){

        req.session.inputData{
            hasError : true,
            message : 'invalid input – Please check your data',
            email : email,
            confirmEmail : confirmEmail,
            password : password,
        };
         req.session.save(function(){
            res.redirect('/signup');
        });
        return ;
    }
```

Dive deeper into sessions ->

Advanced Backend

Dive Deeper into Sessions

- The **/signup get route** do as follows:

```
let sessionInputData = req.session.inputData ;
If(! sessionInputData){
   sessionInputData = {
      hasError : false,
      email : '',
      confirmEmail : '',
      password : '',
   };
}
 req.session.inputData = null ;
res.render('signup', {inputData : sessionInputData});
```

- **Go to the signup.ejs file and do the modification by yourself**

Authorization vs Authentication ->

Advanced Backend

Authentication vs Authorization

- **Authentication:**
  - Signup + Login with credentials
  - Authenticated || logged in user may then access restricted resources
- **Authorization:**
  - Even if Authenticated users may not be allowed to access everything on the website, such as: not all logged in users may access your online shop order history

- To implement **Authorization:**
  >> copy the admin.ejs file and rename it to profile.ejs and tweek the page content a l ittle

- Only administrators should be able to access the admin page, go on the demo.js:
  >> Copy the admin route and make the URL /profile
  >> render the profile template

- In the headers.ejs add a link to it in the ul as a list item
- Add manually in your databse an admin user as follows:

```
db.users.updateOne({_id: ObjectId("6838e304f4ab328e99914044")}, {$set:{isAdmin: true}})
```

Authorization vs Authentication ->

Advanced Backend

Authorization vs Authentication

- In the get request for the **/admin**, add the following:

```
const user = await db.getDb().collection('users').findOne({_id : req.session.user.id});
if(!user || !user.isAdmin){
        res.status(403). render('403');
}
```

- **Copy the 401.ejs and rename it to 403.ejs and say: Not Authorized! Instead of Authenticated**

Improving the website ->

Advanced Backend

Improving the Website

- **To improve:**
  - Signup with an already existing user
  - To not show up the logout button if the user is not logged in
  - Showing the admin and profile option only if we are able to visit them
  - To only show the signup and login only if the user is not yet authenticated

- **For POST signup:**

  >> If the user already exists we want to show the error:

```
if(existingUser){
   req.session.inputData{ . . . }
   req.session.save(function(){
      res.redirect('/signup');
    });
  return;
}
```

Improving the website ->

Advanced Backend

Improving the website

- For POST login:

```
if(!existingUser){
    req.session.inputData{ . . . } >> Take out the confirm email
    req.session.save(function(){
      res.redirect('/signup');
    });
  return;
}
//update this condition
if(!passwordsAreEqual){
    req.session.inputData{ . . . }
    req.session.save(function(){
      res.redirect('/signup');
    });
  return;
}
```

Improving the website ->

Advanced Backend

Improving the Website

- For GET login:
  - >> Copy into it the code we had for the GET signup
  - >> Take out the confirm email

- For the login.ejs :
  - >> Copy into it the ejs code we had for the checking we have errors same as we did in the signup.ejs
  - >> Initialize the fields with the right values

Writing Custom Middlewares & Using res.locals ->

Advanced Backend

Writing Custom Middlewares & Using res.locals

- Cleaning up the navigation bar:

  >> in app.js we will create our own middleware that set up some data that will be exposed to all templates automatically

  >> After the session write the middleware as follows:

```
app.use(async function(req,res,next){
    const user = req.session.user;
    const isAuth = req.session.isAuthenticated;

    if(!user || !isAuth){
        return next();
    }
    const userDoc = await db.getDb().collection('users').findOne({_id: user.id});
    const isAdmin = userDoc.isAdmin;

    res.locals.isAuth = isAuth ; >> setting global variables
    res.locals. isAdmin = isAdmin ;
    next();
});
```

Writing Custom Middlewares & Using res.locals ->

Advanced Backend

Writing Custom Middlewares & Using res.locals

- For header.ejs :
  - >> wrap the logout and profile list items by the following:
    ```
    <% if(locals.isAuth){ %>
    <%  } %>
    ```
  - >> wrap the admin list item by the following:
    ```
    <% if(locals.isAdmin){ %>
    <%  } %>
    ```
  - >> wrap the logig and signuo list items by the following:
    ```
    <% if(! locals. isAuth){ %>
    <%  } %>
    ```
- For GET admin:
  - >> change the if condition to: !res.locals.isAuth;
  - >> Remove the variable user, no need to fetch a user anymore
  - >> change the second if condition to: !res.locals.isAdmin and add a return

Writing Custom Middlewares & Using res.locals ->

Advanced Backend

Writing Custom Middlewares & Using res.locals

- **For the GET profile:**
  >> change the if condition to: !res.locals.isAuth;

Introduction Model View Controller pattern ->

Advanced Backend

Introduction to MVC Pattern

Model

Contains logic for interacting with your data and data storage

Defines functions/ methods storing data in database or fetching data from a database

View

Contains logic for presenting data/content to the user

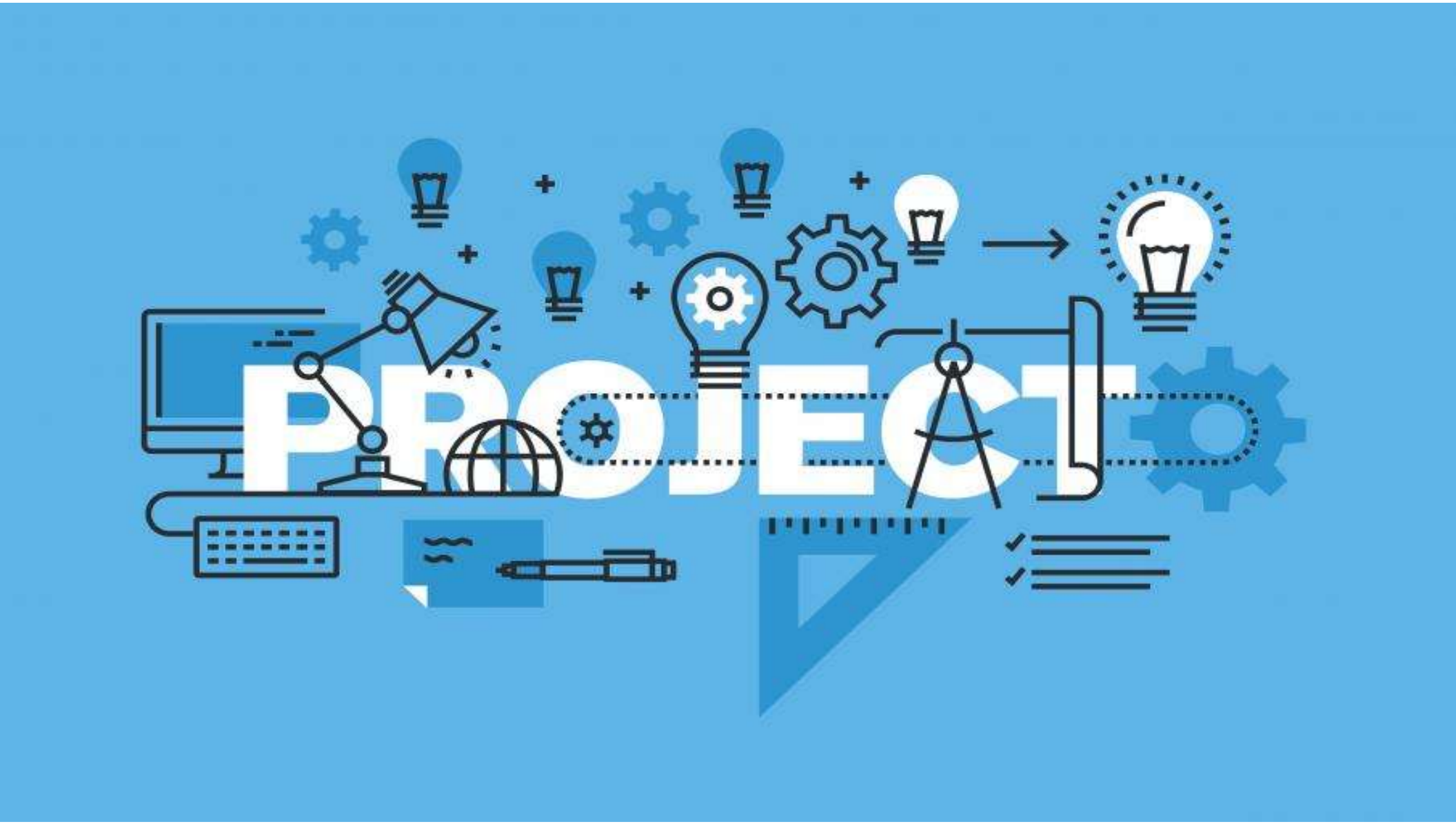
These are the templates + logic (example: EJS tags)

Controller

Connects Model with views

Typically connected to our Routes

Creating our First Model ->

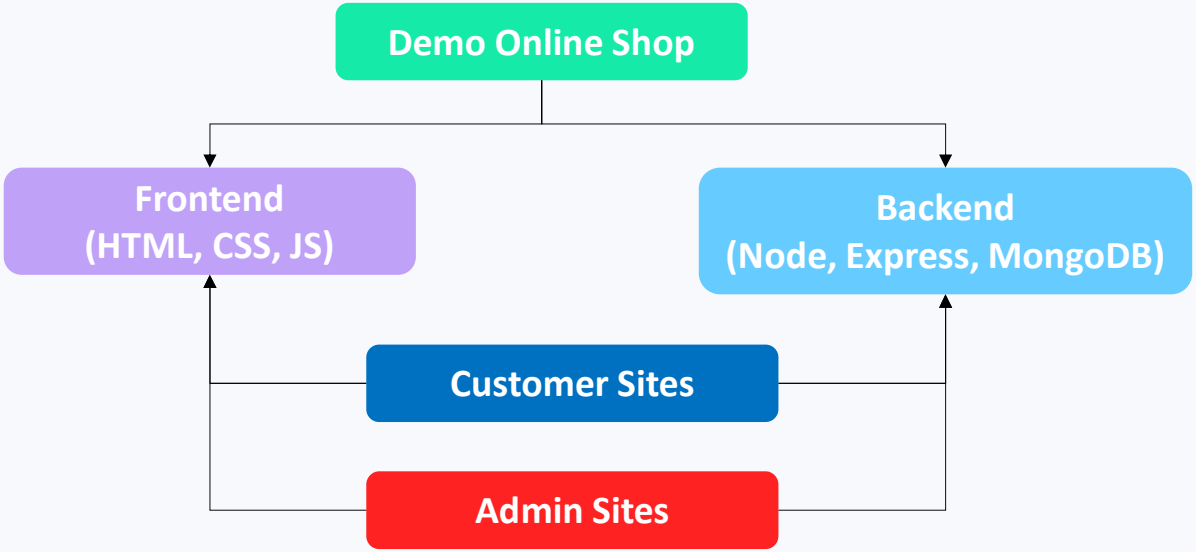

PROJECT

Advanced Backend

Project Introduction: Project Plan

In this course section we will build the project from all the knowledge we have gained previously.

- ❑ **Our Project is a Demo Online Shop.**



Key Pages [views] in the project ->

Advanced Backend

Key Pages [views] in the project

Customer

- Signup
- Login

- All products
- Product Details
- Shopping Cart

- All Orders for customer

Admin

- Signup
- Login

- Dashboard

- All Product
- New Product
- Update Product

- All orders

Data Entities [Models] ->

Advanced Backend

Data Entities [Models]

- **User:** Email, password, isAdmin, name, address.
- **Products:** Name, summary, price, image, description.
- **Cart:** Items, total price, number of items.
- **Order:** User data, products/cart data, date, status.

Project Setup ->

Advanced Backend

Project Setup

We can create our project feature by feature.

1. **We will start by creating the authentication:**
   - Views that belong to the authentication **– signup and login**
   - Controllers related to authentication
   - Models related to authentication **- User**
   - Add the CSS code

- ❑ **Start with an empty folder and open it in VSCode**
- ❑ **Open the terminal:**
  - **> npm init -y**
  - **> npm install express**
  - **> npm install –save-dev nodemon**
- ❑ **Create the app.js file:**
  - **> start by importing express package**
  - **> create the app object**
  - **> Make the app object listen on the port 3000**

Creating Folders, files and Set the first route ->

Advanced Backend

Creating Folders, files and Set the first route

We will set up the project structure

- **Create 3 folders :** controllers, models, views

- **In the views folder create the following folders:**
  - **auth folder** [will hold all authentication related views]
  - **admin** [will hold all administration related views]
  - **products** [will hold all product related views]
  - **cart** [will hold all cart/order related views]
  - **includes**
  - **customer:** includes, products, cart and auth should be moved in it.

- **Create a routes folder, in side it create an auth.routes.js file**
- **In the auth.routes.js file:**
  - **> Import express, and the authController (will be created next)**
  - **> create a router object**
  - **> create a get route: /signup , /login** [serving the page will be handled in the controller]
  - **> export the router object to be available for use in other files**

Creating Folders, files and Set the first route ->

Advanced Backend

Creating Folders, files and Set the first route

- In the controllers folder create the following folder:
  - auth.controller.js file

- In the auth.controller.js file:
  > create a getSignup and a getLogin functions
  > export the an object of pointers towards the created functions

- In the app.js:
  > Import the authRoutes and use it as a middleware

Creating the first EJS views ->

Advanced Backend

Creating the First EJS views

- **In the terminal:**
  > npm install ejs

- **In the app.js file:**
  > set the view engine to ejs
  > Set the option to tell express where to find the views with the path to the views + Import the built in path package

- **In the views/auth file create:**
  > signup.ejs
  > login.ejs

- **In the views/includes file create:**
  > head.ejs – Doctype, html and head opening tags, the title of the page should be dynamic
  > header.ejs - we need a header with a div that takes a link logo that redirect us to the homepage / and a nav element.
  > footer.ejs – holds the body and html element tags closing

Creating Folders, files and Set the first route ->

Advanced Backend

Creating the First EJS views

- **The signup.ejs file will include:**
  - **In the form :**
    - in a p add a label and input of type email
    - in a p add a label and input of type email, to confirm the email
    - in a p add a label and input of type password of minimal length 6 characters
    - Add a horizontal line
    - in a p add a label and input of type text for full name
    - in a p add a label and input of type text for Street
    - in a p add a label and input of type text for Postal code of minimal length 4 and max length of 4
    - in a p add a label and input of type text for city
    - A button : Create Account with a styling class **<<btn>>**
    - in a p add a link with href **<</login>>** to Login instead with an id **<<switch-form>>**

Creating Base CSS style ->

Advanced Backend

Creating Base CSS Style

- Go on google fonts to get the html code related to the Montserrat font and insert it in the head.ejs

- Create a public folder:
  - > create in it a styles folder >> create in it a base.css , forms.css and auth.css style files
  - > in the app.js use the express static middleware to make the public folder accessible to all visitors
  - > in head.ejs add a link to your base.css file

- In the signup.ejs file:
  - > Before the head closing add a link to forms.css and the auth.css files

Creating Base CSS style ->

Advanced Backend

Creating Base CSS Style

- **The base.css file will include:**
  - **For all the elements on the page:**
    - **> box-sizing : border-box;** controls what the width of the element will be the border will be counted into the width
  - **For html:**
    - **> font-family: 'Montserrat', 'sans-serif';**
    - **> Set color variables: [these are the values for properties we will use]**

```
--color-gray-50: rgb(243, 236, 230);
--color-gray-100: rgb(207, 201, 195);
--color-gray-300: rgb(99, 92, 86);
--color-gray-400: rgb(70, 65, 60);
--color-gray-500: rgb(37, 34, 31);
--color-gray-600: rgb(32, 29, 26);
--color-gray-700: rgb(31, 26, 22);
```

Creating Base CSS style ->

Advanced Backend

Creating Base CSS Style

- In the base.css file include:
  > For the html:

```
--color-primary-50: rgb(253, 224, 200);
--color-primary-100: rgb(253, 214, 183);
--color-primary-200: rgb(250, 191, 143);
--color-primary-400: rgb(223, 159, 41);
--color-primary-500: rgb(212, 136, 14);
--color-primary-700: rgb(212, 120, 14);
--color-primary-200-contrast: rgb(100, 46, 2);
--color-primary-500-contrast: white;

--color-error-100: rgb(255, 192, 180);
--color-error-500: rgb(199, 51, 15);

--color-primary-500-bg: rgb(63, 60, 58);
```

Creating Base CSS style ->

Advanced Backend

Creating Base CSS Style

- In the base.css file include:
  > For the html:

```
--space-1: 0.25rem;
--space-2: 0.5rem;
--space-4: 1rem;
--space-6: 1.5rem;
--space-8: 2rem;

--border-radius-small: 4px;
--border-radius-medium: 6px;

--shadow-medium: 0 2px 8px rgba(0, 0, 0, 0.2);
```

Creating Base CSS style ->

Advanced Backend

Creating Base CSS Style

- In the base.css file include:

> For the body:

```
background-color : var(--color-gray-500);
color : var(--color-gray-100);
margin: 0;
```

> For the main:

```
width :90%;
max-width : 50rem;
margin: 0 auto;
```

> For the ul and ol:

```
list-style: none;
 margin: 0;
padding:0;
```

> For the a:

```
text-decoration: none;
color: var(--color-primary--400);
```

Creating Base CSS style ->

Advanced Backend

Creating Base CSS Style

- In the base.css file include:
  - > For the class btn:

```
cursor: pointer;
font:inherit;
padding:var(--space-2)  var(--space-6);
background-color: var(--color-primary-500);
color:  var(--color-primary-500-contrast);
border 1px solid var(--color-primary-500);
border-radius: var(--border-radius-small);
```

  - > For the class btn , hover and active pseudo selectors:

```
background-color: var(--color-primary-700);
border-color: var(--color-primary-700);
```

Creating Auth CSS style ->

Advanced Backend

Creating Auth CSS Style

- In the auth.css file include:
  - > For the h1:

```
text-align : center;
color : var(--color-gray-300);
```

  - > For the form:

```
max-width : 25rem;
margin : var(--space-8) auto;
padding : var(--space-4);
background-color: var(--color-gray-600);
border-radius:var(--border-radius-medium);
text-align : center;
```

  - > For the form link:

```
color : var(--color-primary-200);
```

  - > For the form link, hover and active pseudo classes:

```
color : var(--color-primary-400);
```

Creating Forms CSS style ->

Advanced Backend

Creating Forms CSS Style

- In the forms.css file include:
  - > For the form horizontal line:

```
border-color: var(--color-primary-200);
margin: var(--space-4)  var(--space-6);
```

  - > For the labels:

```
color: var(--color-gray-100);
display: block;
margin-bottom: var(--space-2);
```

  - > For the inputs and textAreas:

```
font:inherit;
padding: var(--space-2);
border-radius:var(--border-radius-small);
border: none;
width:90%;
```

Adding MongoDB database server & connection ->

Advanced Backend

Adding MongoDB database server & connection

- Open the terminal : > npm install mongodb

- Create a data folder that holds a database.js file that will hold the logic to connect to MongoDB as the following:
  > import mongodb
  > Create a mongodb client object
  > Create an async function connectToDatabase to connect the client to the mongodb://localhost:27017. Store the promise result in a client object, the function will store a db connection
  > create getDb function, if we do not have a database yet we throw an error object otherwise we will return the database
  > export the created functions

- In the auth.controller.js file include:
  > create a signup function:

- In the auth.routes.js file add the post signup route:

```
router.post('/signup', authController.signup);
```

- In the signup.ejs form add the method and action

Adding MongoDB database server & connection ->

Advanced Backend

Adding MongoDB database server & connection

- In the app.js file:
  > Import the database object we create in the previous slide
  > Call the connectToDatabase function before listening to the port as the following:

```
db.connectToDatabase()
  .then(function(){
        app.listen(3000);
  })
  .catch(function(){
        console.log('Connection to the database!');
        console.log(error);
  });
```

Adding User signup ->

Advanced Backend

Adding User Signup

- Create a user.model.js file in the models folder :
  - > Create a class User
  - > Create a constructor that takes:
    email, password, fullname, ***street, postalCode and city [optional]***
  - > Import the database
  - > Create an async method signup to get the database connection and insert each user to a users collection
  - > Keep in mind we store the passwords in our database in a hashed way to make sure it is encrypted using the bcryptjs package [npm install bcryptjs] and import it
  - > Export that model

- In app.js:
  - > Use the urlencode middleware that express offers and set the extended key to false

- In the auth.controller.js file:
  - > Import the User model
  - > In the signup function create a new user object and call the Users's signup method, and redirect to the loging page.

- For the login.ejs file: copy the signup.ejs file content but only get ride of the email and password confirmation and the personal details section.

Adding CSRF Protection ->

Advanced Backend

Adding CSRF Protection

We need to update our signup process, to add validation, CSRF security and error handling.

> open the terminal: npm install csurf
> Import the csurf package in the app.js and activate it as a middleware
> Create a middlewares folder and create in it a csrf-token.js file

- In the csrf-token.js file:
  > create a function addCsrfToken as the following:

```
function addCsrfToken(req,res,next){
        res.locals.csrfToken = req.csrfToken();
        next();
}
```

  > export the created function

> Import csrf-token.js file in the app.js and use this customized middleware after declaring the use of the csrf token
> In signup.ejs file: create a hidden input with a name _csrf and give it the value generated by our customized middleware – add it to the loging form too.

Implementing Error with the Error Handling Middleware ->

Advanced Backend

Implementing Error with the Error Handling Middleware

- In the middlewares folder create an error-handler.js file and create the following function:

```
function handleErrors(error,req,res,next){
        console.log(error);
        res.status(500).render('shared/500');
}
```

  > export the function

- In the views folder create a shared folder where we create the 500.ejs file. Copy into it the signup structure page but omit the form. Set the title to 'An error occurred'. Remove the styles links. Move the includes folder into the shared folder and update the includes path in login and signup files

- In app.js:
  > import the error-handler middleware
  > Use this middleware at the bottom

Adding and Configuring Sessions ->

Advanced Backend

Adding and Configuring Sessions

Whenever we are dealing with CSRF we should always add sessions.

> open the terminal: npm install express-session connect-mongodb-session
> Create a config folder and add a session.js file, it will include the following:

- Import the connect-mongo-session and the express-session packages
- Create a functions createSessionStore and createSessionConfig like the following:

```
function createSessionStore(){
  const MongoDBStore =
              mongoDbStore(exprssSession);

  const store = new MongoDBStore({
    uri:’ mongodb://localhost:27017',
    databaseName: 'online-shop',
    collection: 'session'
  });

  return store;
}
```

```
function createSessionConfig(){
  return {
    secret:'super-secret',
    resave:false,
    saveUninitialized: false,
    store: createSessionStore(),
    cookie:{
      maxAge: 2*24*60*60*1000,
    }
  };
}
```

Adding and Configuring Sessions ->

Advanced Backend

Adding and Configuring Sessions

- In the app.js file:
  - > import the session.js file
  - > import the express-session package
  - > Create a session configuration object
  - > Use the express-session before the csrf use because the csrf needs a session to exist before it is used. It must take the session configuration object as argument

Implement authentication & User Login ->

Advanced Backend

Implement authentication & User Login

- Create a util folder, create in it an authentication.js file

- In the an authentication.js :
  > create an function createUserSession
  --> This function will get the request object, the user and an action to be executed once the session is updated
  --> Access the request session to store the userId we have in our DB
  --> Save the session and pass to it the action
  --> Export this function

- In the auth.controller.js:
  > import the authentication.js
  --> use the createUserSession function and pass to it the request object, the existinguser object and an anonymous function like the following:

```
authUtil.createUserSession(req, existingUser, function () {
        res.redirect('/');
});
```

- In the customer folder in the product create an all-products.ejs:
  > copy into is the signup.ejs file content, remove the form and the styles. Modify the titles

Implement authentication & User Login ->

Advanced Backend

Implement authentication & User Login

- In the user.model file add the following functions:
  - > create a getUserWithSameEmail function that retrieves all users that has a matching email to the one entered by the user
  - > create a hasMatchingPassword that takes the hashed password as argument to compare it with the password entered by the user using the compare function from the bcrypt package

- In the routes folder create a base.routes.js and a products.routes.js file:
  - > copy the auth.routes.js content in it, remove the routes and the authController
  - > in the products.routes.js Create a get route /products that renders to the all-products.ejs page do the same in the base.routes.js to be redirected into the /products

- In the app.js:
  - > import the products and base routes
  - > Use the products and base routes

Finishing Authentication & Checking The Auth Status ->

Advanced Backend

Finishing Authentication & Checking The Auth Status

- In the an middleware folder create check-auth.js file
  - > create an function checkAuthStatus
    - --> Store the userId from the session object
    - --> If the userId was not present in the session object, we will return the next action
    - --> store the userId in the locals and an isAuth variable
    - --> At the end we proceed to the next action
    - --> Export the created function

- In the app.js:
  - > import the check-auth.js
    - --> use the middleware after the csrf middelware

- In the header.ejs file:
  - > in the nav element create a ul element
  - > in the ul create:
    - - - a li that have a button Logout that should appear if a user is authenticated

- In the auth.routes.js:
  - > create a post route /login that triggers the login function in the auth.controller

Adding Logout Functionality ->

Advanced Backend

Adding Logout Functionality

- In the header.ejs file :
  > wrap the login button in the li in a form, with a post method that triggers the /logout link. Add a hidden input for the csrf protection.

- In the util folder authentication.js:
  > Create the destroyUserAuthSession function
  > Set the session uid to null

- In the auth.controller.js:
  > Create the logout function
  > use the destroyUserAuthSession
  > redirect the user to the login page

- In the auth.routes:
  > create a post route /logout

Handling Errors In Asynchronous Code ->

Advanced Backend

Handling Errors In Asynchronous Code

- In the auth.controller file:
  > in the signup function
  --> wrap the await command line in a try-catch and add the next argument to benefit from the express error handle middleware

Do the same for all the function where there is a possibility for the code to raise errors

Adding User input Validation->

Advanced Backend

Adding User input validation

- In the auth.controller.ejs file we want to validate the user input for the signup function:
  - > Create a validation.js file in the util folder
  - > Create a function isEmpty as the following:

```
function isEmpty(value){
    return !value || value ===‘’;
}
```

  - > Create a function userCredentialsValidation as the following:

```
function userCredentialsValidation(email,password){
    return(
        email && email.include('@') &&
        password && password.trim().length >=6
    );
}
```

  - > Create a userDetailsValidation function that takes all the user details as arguments the validation check will be as the following:

```
return ( userCredentialsValidation(email,password) &&
    !isEmpty(name) && !isEmpty(street)
    && !isEmpty(postal)&& !isEmpty(city)
);
```

Handling Errors In Asynchronous Code ->

Advanced Backend

Adding User input validation

> In validation.js add the following:

```
function emailConfirmed(email,confirmEmail){
   return (email === confirmEmail);
}
```

> Export these functions

> Import these function in the auth.controller and use It where needed in a condition to see which od these 2 function will result in failure

> In the user.model add the following async function:

```
async existsAlready(){
    const existingUser = await this.getUserWithSameEmail();
    if(existingUser){
       return true;
    }
    return false;
 }
```

And use it in the signup function in the auth.controller file

Flashing Errors & Input Data Onto Sessions ->

Advanced Backend

Flashing Errors & Input Data Onto Sessions

- In the auth.controller.ejs file we want to validate the user input for the signup function:
  - > Create a validation.js file in the util folder
  - > Create a function isEmpty as the following:

```
function isEmpty(value){
    return !value || value ===‘’;
}
```

  - > Create a function userCredentialsValidation as the following:

```
function userCredentialsValidation(email,password){
    return(
        email && email.include('@') &&
        password && password.trim().length >=6
    );
}
```

  - > Create a userDetailsValidation function that takes all the user details as arguments the validation check will be as the following:

```
return ( userCredentialsValidation(email,password) &&
    !isEmpty(name) && !isEmpty(street)
    && !isEmpty(postal)&& !isEmpty(city)
);
```

Displaying Error Messages & Saving User Input ->

Advanced Backend

Displaying Error Messages & Saving User Input

- In the **util** folder create a **session-flash.js** file:
  - > Create a function **flasheDataToSession** that takes the request, the data and the action as augments. Create a **flashedData** variable into the session object to store the data in it save the updates made to the session and move on to the next action to be taken
  - > Create a function **getSessionData** that retrieves the data we have in store in the session and afterwards make sure you empty the stored data from our **flashedData** variable. This function takes the request as argument and returns the fetched data
  - > Export these 2 function

- In the **auth.controller.js** file:
  - > Import the **session-flash.js** file
  - > Update the **getSignup** function. Create a **sessionData** variable where you use the **getSessionData** to the data from the session. If the **sessionData** is undefined set all the data fields to empty strings. The function should render the signup page with the **inputData**
  - > Update the **signup** function to that we store all the user inputs and after checking for validation, if the validation fails, we use the **flasheDataToSession** to send the error message with the data entered by user so that it wont be lost. If the user already exists send a proper error message to the user and keep the user's data in the form
  - > Update the **getLogin** so that if the **sessionData** was empty the data fields should also be empty otherwise we keep that inserted data

Displaying Error Messages & Saving User Input ->

Advanced Backend

Displaying Error Messages & Saving User Input

- In the auth.controller.js file:
  - > Update the login so that you create a sessionErrorData object, that hold the error message, the email and password values. If the user does not exists store the input values on the session and pass on the error message, do the same if the password is incorrect

- In the signup.ejs and login.ejs files:
  - > Update the files so that you retrieve the values for the input fields from the session and create a place to display the error messages

- Update the base.css file and add these:

```
.alert {
  border-radius: var(--border-radius-small);
  background-color: var(--color-error-100);
  color: var(--color-error-500);
  padding: var(--space-4);
}
```

```
.alert h2 {
  font-size: 1rem;
  margin: var(--space-2) 0;
  text-transform: uppercase;
}

.alert p {
  margin: var(--space-2) 0;
}
```

Admin Authorization and Protected Navigation ->

Advanced Backend

Admin Authorization and Protected Navigation

We will now work on the administration area where admins can add new products etc…

- To make a user as Administrator we will add an **isAdmin** key manually in our database
  - > Using the mongoshell, and turn one of the existing user into an Admin as the following:

```
db.users.updateOne({_id: ObjectId('6859175a6c6ddf3e905666f1')},
{$set: {isAdmin: true}})
```

- In the **authenticaion.js utility**:
  - > Update the createUserSession function to store the isAdmin in the session
- In the **check-auth.js middleware**:
  - > Extract the isAdmin result from the session to the locals object
- In the included **hedear.ejs** file we will work on the navigation:
  - > For none administrators options : shop and Cart**[only if we are not admins]**, orders**[only if the user is authenticated and we are not the admins]**
  - > For the Administrators, the options are: Manage products, Manage orders
  - > The signup an login options **[should be visible if we are not logged in]**

Setting up base navigation Style ->

Advanced Backend

Setting up Base navigation Style

- In the header.ejs file:
  - >> add an **id** for the header <<**main-header**>>
  - >> add an **id** for the logo div <<**logo**>>
  - >> add a button in the bottom after the closing nav tag with an **id** <<**mobile-menu-btn**>>.
    This button should include 3 spans
  - >> Create a aside element to mobile menu itself with an **id** <<**mobile-menu**>> **[it will include the same previous nav]**
  - >> Create a nav-items.ejs where you take the ul entirely and you paste them into that file.
  - >> Now include it in the header.ejs where needed. Give the **ul** a **class** of <<**nav-items**>>
- Download the navigation.css file and put it in the styles public folder
- In the head.ejs link the created navigation style

Frontend JavaScript For Toggling The Mobile Menu ->

Advanced Backend

Frontend JavaScript For Toggling The Mobile Menu

- Create a **scripts** folder in the **public** folder
  - > create a **mobile.js** file
- In the **mobile.js** file:
  - > create a variable **mobileMenuBtnElement**, to get the menu button by its id
  - > create a vatiable **mobileMenuElement**, to get the mobile menu by its id
  - > create a function **toggleMobileMenu** :
    - >> we will show the **mobileMenuElement** when the button is clicked
  - > Add an event listener for the menu button to trigger the created function when clicked
- In **head.js** file : > link the script file created.

Adding Product Admin Pages & Forms ->

Advanced Backend

Adding Product Admin Pages & Forms

- In the **controllers** folder create an **admin.controller.js** file
  - >> Create a function **getProducts** to show the admin products page. To render the **all-products** page
  - >> Create a function **getNewProduct** that will display for us the page to add a new Product. To render the **new-product** page
  - >> Create a function **createNewProduct** to submit a new product

- In the **routes** folder create an **admin.routes.js** file:
  - >> import express and create a router and the **admin.controller.js**
  - >> Create a get route **/products**, use the **getProducts** function here
  - >> Create a get route **/products/new** , use the **getNewProduct** function here
  - >> In **app.js** you can import these routes and use them with a prefix **'/admin'**

- In **view/admin** folder add a **products** subfolder:
  - >> Create an **all-products.ejs** file
  - >> Create an **new-product.ejs** file

Adding Product Admin Pages & Forms ->

Advanced Backend

Adding Product Admin Pages & Forms

- In the **all-products.js** file:
  - > Copy into it the customer **all-products.ejs** file content
  - > Add In the main 2 sections, one that have a link that takes us to the a new product page and another to display the list of products
- You can download the **all-products.ejs** and **new-product.ejs** files and integrate them in your project.
- Download the updated **base.css** file

Adding The Image Upload Functionality ->

Advanced Backend

Adding The Image Upload Functionality

- In the new-product.ejs file:
  - >> The action should be: **/admin/products?_csrf=<%= locals.csrfToken %>**
    set the method: **POST**, and set the enctype to: **multipart/form-data**
  - >> Open the terminal: npm install - -save multer and npm install uuid

- Create an image-upload.js middleware file:
  - >> import the **multer** and the **uuid** packages
  - >> execute the multer function in a constant **upload**.
    It take a configuration object:

```
{
  storage: multer.diskStorage({
        destination: 'product-data/images', [create these folders]
        filename: function(req,file,cb){
              cb(null,uuid()+ '-'+ file.originalname)
            }
    })
}
```

  - >> use the **single** method to extract a single file from the request. It takes the name value of the element tag which in our case is: image. Store the results in a **configuredMulterMiddelware.** Export this variable
  - >> In app.js you can import these routes and use them with a prefix **'/admin'**

Adding The Image Upload Functionality ->

Advanced Backend

Adding Product Admin Pages & Forms

- In the admin.routes.js file:
  - > Import the **image middelware**
  - > Create a post route for **/products.** Its should use the **image middleware** and the createNewProduct function we created in the admin.controller

- In the admin.controller.js files:
  - > For the createNewProduct function:
    - >> For the moment only redirect to the **/admin/products**

Adding a Product Model & Storing Products In The Database ->

Advanced Backend

Adding a Product Model & Storing Products In The Database

- In the **models** folder create a **product.model.js** files:
  - > Create a class **Product**:
    - >> The constructor that takes one argument: **productData** which will be an object that you extract from all the needed details (title,summary,price,description,imageName,imagePath,imageUrl, id[if it exists])
    - >> Import the **database.js** file
    - >> Create an **asyn save** method that stores all the data in the object argument and inserts it into the database collection **<<products>>**

- In the **admin.controller.js** file:
  - > Import the Product model.
  - > In the **createNewProduct** function, create a new **Product** object that takes the request body fields and the request filename stored in an image key
  - > Turn the function into an async function and add the **next** argument to it
  - > call the product **save** method to store the product details in the database, wrapped around with a try and catch

Fetching & Outputting Product Items ->

Advanced Backend

Fetching & Outputting Product Items

- In the admin folder:
  - > create an includes folder
    - >> create in it a product-item.ejs file
    - >> Create an article element with a class <<**product-item**>>, that takes an image, a title and div element that will contain 2 buttons with a class <<**btn btn-alt**>>
    - >> A button to view and edit a product that will be a link looking like a button
    - >> A delete button to delete a product

- In the all-products.ejs file:
  - > In the list of products section
    - >> add a list item with an id <<**products-grid**>>
    - >> Include in it the product-item.ejs **[This should be in a for loop]**

- In the product.model.js file:
  - > Create a static async method findAll, where you retrieve all existing products in the database and store the result in a **products** variable that will be returned. With the help of the map function all the products should be converted into objects

Fetching & Outputting Product Items ->

Advanced Backend

Fetching & Outputting Product Items

- In the **admin.controller.js** file:
  - > Turn the **getProducts** into an async function, add the **next** argument and use the findAll method to fetch all existing products in a **try-catch** block
  - > Pass on the fetch result to the template

- In the **all-products.ejs** file:
  - > loop in the products received by the **getProducts** method and pass on the product key with its value to the **product-item.ejs** file

- In the **product-item.ejs** file:
  - > retrieve the **title**, image **src**, the **link** for editing with the **id**

- In the **app.js** file:
  - > we want to serve the **product-data** folder statically with a filter **/products/assets**

Styling Product Items ->

Advanced Backend

Styling Product items

- Download the products.css style file and integrate it in the **public/styles** folder
- In the all-products.ejs file:
  > Link the products.css stylesheet
- In the product-item.ejs file:
  > Move the title and the div that has 2 buttons into a div with class <<**product-item-content**>> and give the buttons div a class <<**product-item-actions**>>

Adding the Product Details Page ->

Advanced Backend

Adding the Product Details Page

We need to be able to press on the View & Edit button so that we can see the new-product Page implemented with the details related to the product so that we can update and save.

- In the admin/products/includes folder:
  - > Create a product-form.ejs file and copy into it the form we have in the new-prodcuts.ejs file and modify the form action to the following:
    <%= submitPath %>?_csrf=<%= locals.csrf_token %>
  - > In the new-product.ejs include that product-form.ejs, and pass to it the submitPath value which is **'/admin/products'**

- In the admin/products folder:
  - > create a new update-product.ejs file and copy into it the new-prodcuts.ejs file content, change the title, and the submitPath should be the following:
    '/admin/products/' + product.id , {product : product }

- In the admin.routes.js file:
  - > Create the get and post routes for the same url: **/products/:id**

- In the admin.controller.js file:
  - > Create 2 functions: getUpdateProduct and updateProduct. Add these 2 functions to the previously created routes

Adding the Product Details Page ->

Advanced Backend

Adding the Product Details Page

- In the **product.model.js** file :
  - > Import mongodb
  - > Create a **static async findById** method that takes a **productId** argument.
    Construct that argument with the **ObjectId** mehod
    Find the product from the database according to the id match with the pre-constructed id. Store the result in a product variable .
    Check if we did not got the **product, then** set a code **property** to 404 and **throw** the error and return it.
  - > At the end return a new product model with the data retrieved from the database

- In the **admin.controller.js** file:
  - > For the **getUpdateProduct**, turn it into an **async** function, add the **next** argument to it and call the **findById** method. If we found a product with the matched id we should render **'admin/products/update-product'**

- Update the **product-form.ejs** file:
  - > pre-fill the input field with the values related to the product

Updating Products as Administrator ->

Advanced Backend

Updating Products as Administrator

- In the **admin.controller.js** file:
  - > Update the empty **updateProduct** function
  - >> Turn into an async function with 3 arguments: req, res, and next
  - >> Create a new Product model, where all the input fields will be retrieved from the request **body** object, and the **id** retrieved from the request **params** object
  - >> Check if we have a file sent with the request:
    - using the created product object use the **replaceImage** method

- In the **product.model.js** file:
  - > In the save method, add the following:

```
if(this.id){
  const productId = new mongodb.ObjectId(this.id);
  await db.getDb().collection('products').updateOne({_id:productId},{
  $set: productData
 });
}else{
 await db.getDb().collection('products').insertOne(productData);
}
```

  - > Take out the **imagePath** and **imageUrl** from the constructor and place them in a new **updateImageData** method and call it in the constructor.
  - > Create a new method **replaceImage** as the following:

```
replaceImage(newImage){
  this.image = newImage;
 this.updateImageData();
}
```

Updating Products as Administrator ->

Advanced Backend

Updating Products as Administrator

- In the product-form.ejs file:
  - > for the image element tag add the following:

```
<% if(imageRequired){ %>
        required
<% } %>
```

- In the new-product.ejs file:
  - > Pass the imageReuired key with a true value alongside the input keys that have empty values.

- In the update-product.ejs file:
  - > do the same for the imageRequired key with a false value

- In the product-model.js file:
  - > In the save method
  - > Before we start updating our data, check if the image does not exists, we do not want to update the image with undefined, we will use the delete keyword to delete the image key from the product data

Updating Products as Administrator ->

Advanced Backend

Updating Products as Administrator

- In the **admin.controller.js** file:
  - > in the **updateProduct**, you can use the **save** method from the product-model.js to update your product details, wrap it in a **try-catch** block, in case of error we will use the **next** and **return**. After that redirect to **/admin/products**

- **In your routes folder:**
  - > Add the **imageUploadMiddelware** to the post route **/products/:id**

Adding A File Upload Preview ->

Advanced Backend

Adding a File Upload Preview

We will improve our UI a little bit, by creating a small image preview.

- In the product-form.ejs file:
  - > Create a div with an id <<**image-upload-control**>>
  - Move the image input into it.
  - Below that create an image element

- Update the forms.css style and add the following:

```
#image-upload-control{
   display:flex;
   align-items: center;
}

#image-upload-control input{
   max-width:15rem;
}
```

```
#image-upload-control img{
   display:none;
   width:4rem;
   height: 4rem;
   object-fit: cover;
   border-radius: var(--border-radius-small);
}
```

Adding A File Upload Preview ->

Advanced Backend

Adding a File Upload Preview

- **In the scripts folder:**

  > create a image-preview.js file that includes the following:

```
const imagePickerElement = document.querySelector('#image-upload-control input’);
const imagePreviewElement = document.querySelector('#image-upload-control img');

function uploadImagePreview(){
   const files = imagePickerElement.files;
    if(!files || files.length ===0){
      imagePreviewElement.style.display='none';
      return;
    }
    const pickedFile = files[0];
    imagePreviewElement.src = URL.createObjectURL(pickedFile);
    imagePreviewElement.style.display = 'block’;
}

imagePickerElement.addEventListener('change', uploadImagePreview);
```

Adding A File Upload Preview ->

Advanced Backend

Adding a File Upload Preview

- In the **new-product.ejs** file:
  > import the script we created previously

- In the **update-product.ejs** file:
  > import the script we created previously

Adding the Delete Product functionality ->

Advanced Backend

Adding the Delete Product functionality

- **In the product.model.js file:**
  > create a remove method that will return the deleted product from the database

- **In the admin.controller.js file:**
  > Create an async function deleteProduct that takes 3 arguments: req,res and next as the following:

```
async function deleteProduct(req,res,next) {
  let product;
  try{
        product =  await Product.findById(req.params.id);
        await product.remove();
  }catch(error){
        return next(error);
  }
  res.json({ message: 'Deleted Product!' });
}
```

- In admin.routes.js file add the delete route for the **/products/:id**

Adding the Delete Product Functionality ->

Advanced Backend

Adding the Delete Product functionality

- **In the public/scripts folder:**
  > create a **product.management.js** file that includes:

Using Ajax Frontend JS Requests & Updating The DOM ->

Advanced Backend

Using Ajax Frontend JS Requests & Updating The DOM

- In the **scripts** folder:
  > create a file **product-management.js** – that will help us with deleting products
  Download the file and integrate it with your project.

- In the **product-item.ejs** file:
  > for the delete button add the following attributes:
  data-productid = "<%=product.id %>"
  data-csrf = "<%=locals.csrfToken %>"

- In **all-products.ejs** file:
  > Import the **product-management** script

Various Fixes & Proper Route Protection ->

Advanced Backend

Various Fixes & Proper Route Protection

- Update the error-handler.js file as the following:

```
function handleErrors(error, req, res, next) {
  console.log(error);
  if(error.code === 404){
    return res.status(404).render('shared/404');
  }
  res.status(500).render('shared/500');
}
```

- Duplicate the 505.ejs file and turn it into a 404.ejs file

- In the middleware folder create a new protect-routes.js file that will include the following:

```
function protectRoutes(req,res,next){
  if(!res.locals.isAuth){
    return res.redirect('/401');
  }

  if(req.path.startsWith('/admin') && !res.locals.isAdmin){
    return res.redirect('/403');
  }
  next();
}
```

Various Fixes & Proper Route Protection ->

Advanced Backend

Various Fixes & Proper Route Protection

- In the **app.js** file:
  > Import that **protect-routes** middleware, and use it before the **adminRoutes**

- Create the **401.ejs** and the **403.ejs** files and define the routes for these pages in **base.routes.js**

Outputting Products For Customers ->

Advanced Backend

Outputting Products For Customers

- In the **controllers** folder create a new file **products.controller.js** :
  - > Import the Product model
  - > Create an async **getAllProducts** that retrieves all products found in our database using the **findAll** method and passes the result to the **all-products.ejs** page
  - > Export the created function

- In **products.routes.js** file:
  - > Import the **getAllProducts** function to use it on the get route

- Download the **all-products.ejs** file and integrate it in the project **customer/products** folder
- Delete the **product-item.ejs and download the new one** into the **shared/includes** folder and fix the path in the **all-products.ejs** file and do the same update in the **customer/products/all-products** file

Outputting Product Details ->

Advanced Backend

Outputting Product Details

- In the customer/products folder:
  - > Download the a product-details.ejs file and integrate it in the project
  - > Download the updated product.css and integrate it in your project

- In the product.controller.js file:
  - > Create an async getProductDetails function that takes the 3 arguments. The function is as the following:

```
async function getProductDetails(req,res,next){
 try{
   const product = await Product.findById(req.params.id);
   res.render('customer/products/product-details',{product:product});
 }catch(error){
   next(error);
  }
}
```

- In product.routes.js file:
  - > Create a new get route **/products/:id,** and connect the getProductDetails function to it

Adding a Cart Model ->

Advanced Backend

Adding a Cart Model

- In the **models** folder create a new file **cart.model.js** :

  > Create a **Cart** class with its constructor that takes an **items argument** with a **default of empty array** as the following:

```
constructor(items = [], : totalQuantity = 0, totalPrice=0){
        this.items = items;
        this. totalQuantity = totalQuantity;
        this. totalPrice = totalPrice;
}
```

  > Create an **addItem** method that takes the **product** as an **argument that will be** stored on the session. This method will **push** each **product into** the **items array.** Make sure we don't store the same product twice as the following:

Adding a Cart Model ->

Advanced Backend

Adding a Cart Model

> addItem method:

```
const cartItem={
                product:product,
                quantity : 1,
                 totalPrice : product.price
};
for(let i = 0 ; i < this.items.length; i++){
                const item = this.items[i];
                 if(item.product.id === product.id){
                    cartItem.quantity = +item.quantity +1;
                    cartItem.totalPrice = item.totalPrice + product.price;
                    this.items[i] = cartItem;
                    this. totalQuantity ++ ;
                    this. totalPrice += product.price;
                    return;
                }
}
this.items.push(product);
this. totalQuantity ++ ;
this. totalPrice += product.price;
```

Adding a Cart Model ->

Advanced Backend

Adding a Cart Model

- In the **controller** folder create a new file **cart.controller.js** :
  > Create a function **addCartItem** that take the **request** and **response** as arguments as the following:

```
res.locals.cart.addItem();
```

- In the **middleware** folder create a new file **cart.js**:
  > Import the cart model
  > Create a function **initializeCart** that takes the 3 arguments as the following:

```
function  initializeCart(req,res,next){
   let cart;

   if(!req.session.cart){
      cart = new Cart();
   }else{
      let sessionCart = req.session.cart  ;
      cart = new Cart(sessionCart.items, sessionCart.totalQuantity , sessionCart.totalPrice);
   }
   res.locals.cart = cart;
   next();
}
```

- In the **app.js** import the **cart middleware** and use it.

Working on the shopping cart logic ->

Advanced Backend

Working on the shopping Cart Logic

- In the cart.controller.js file
  > Turn the addCartItem into an async function and update the function as the following:

```
let product;
try{
        product = await Product.findById(req.body.productId);
}catch(error){
        next(error);
        return;
}
const cart = res.locals.cart;
cart.addItem(product);
req.session.cart = cart;

res.status(201).json({
        message: 'Cart Updated!',
        newTotalItems: cart.totalQuantity
});
```

Working on the hopping Cart Logic ->

Advanced Backend

Working on the shopping Cart Logic

- Download the updated base.css file and integrate it in your project:
- In the nav-items.ejs file:
  > Add the following for the Cart link:

```
<span class="badge"><%= locals.cart.totalQuantity %></span>
```

Adding Cart item via AJAX Requests ->

Advanced Backend

Adding Cart item via AJAX Requests

- In routes:
  - > create a cart.routes.js file
  - > Import **express** and the **cart controller**
  - > Create the express router
  - > Create a **post** route **/items** that uses the addCartItem function
- In the scripts folder:
  - > Download the cart-management.js file and integrate it in the project
- In the app.js file:
  - > Use the json middleware that is built in into express

Adding a Cart Page ->

Advanced Backend

Adding a Cart Page

- In views/customer/cart **folder**:
  - > Download the cart.ejs file and integrate it your project
  - > Download the **cart.css** file and integrate it in your project
  - > Adjust the page title, remove the product style and include a the cart.css style

- In views/customer/cart **folder**:
  - > Create an includes folder and download the cart-item.ejs file and integrate it in the project

- In the cart.controller.js file:
  - > Create a getCart function, that takes the request and response. It should render the cart.ejs template

- In the cart.routes.js file:
  - > Create a get route **/** and point to the getCart function

Updating Cart Items ->

Advanced Backend

Updating Cart Items

- In the cart.model.js file:
  - > create a **method** updateItem that takes the **productId** and **newQuantity** as arguments
  - > **copy** the **addItem** method's **loop** into the updateItem method
  - > After **Checking** for the **productId** do the following check for the **newQuantity** if it is greater than 0 and update the loop body as the following:

```
const cartItem = {... item};

const quantityChange = newQuantity - item.quantity;
cartItem.quantity = newQuantity;
cartItem.totalPrice = newQuantity * product.price;
this.items[i] = cartItem;

this.totalQuantity = this.totalQuantity + quantityChange;
this.totalPrice += quantityChange * product.price;
return {updatedItemPrice: cartItem.totalPrice };
```

  - > Check as well if the **productId** matches the **item.product.id** and if the **newQuantity** id <= 0. that covers the following:

```
this.items.slice(i,1);
this.totalQuantity = this.totalQuantity - item.quantity;
this.totalPrice -= item.totalPrice;
return {updatedItemPrice: 0};
```

Updating Cart Items ->

Advanced Backend

Updating Cart Items

- In the cart.controller.js file:
  - > Create an updateCartItem function that takes the request and response as arguments
  - > Get the **crat** through the **locals** object from the response
  - > In this function use the updateItem, and give it the information needed from the request body
  - > save the updated cart on the session
  - > Send back a JSON response as the following:

```
res.json({
        message:'Item Added',
        updatedCartData: {
            newTotalQuantity : cart.totalQuantity,
            newTotalPrice : cart.totalPrice,
            updatedItemPrice:updatedItemData.updatedItemPrice
        }
});
```

- In the cart.route.js file:
  - > Create a patch route **/items** and use the updateCartItem function

Updating Carts Via AJAX Requests ->

Advanced Backend

Updating Carts Via AJAX Requests

- In the script folder:
  > create a cart-item-management.js file that includes the following:

```
const cartItemUpdateFormElements = document.querySelectorAll(".cart-item-management");

async function updateCartItem(event){
  event.preventDefault();

  const form = event.target;

  const productId = form.dataset.productId;
  const csrfToken = form.dataset.csrf;
  const quantity = form.firstElementChild.value;
  let response;
```

Updating Cart Via AJAX Requests ->

Advanced Backend

Updating Cart Via AJAX Requests

```
try{
        response = await fetch('/cart/item',{
            method : 'PATCH',
            body : JSON.stringify({
                productId : productId,
                quantity : quantity,
                _csrf: csrfToken
            }),
            headers:{
                'Content-Type' : 'application/json'
            }
        });
}catch(error){
        alert('Something went wrong');
        return;
}
```

Updating Carts Via AJAX Requests ->

Advanced Backend

Updating Carts Via AJAX Requests

```
if(!response.ok){
      alert('Something went wrong');
      return;
   }
   const responseData = await response.json();
}

for( const formElement of cartItemUpdateFormElements){
   formElement.addEventListener('submit',updateCartItem);
}
```

- In cart.ejs file:
  - > include the cart-item-management.js file
  - > Turn the div that contains the button into a form in the cart-item.ejs and add to it:
  - data-productid = "<%= item.product.id %>"
  - data-csrf = "<%= locals.csrfToken %>"

Updating The DOM after the Cart item Updates ->

Advanced Backend

Updating the DOM after the Cart item Updates

- In the cart-item.ejs:
  - > Add a **span** with a class <<cart-item-price>>to the paragraph displaying the price
- In the cart.ejs:
  - > Add a **span** with an id<<cart-item-price>>to the paragraph displaying the total price
- In the cart-item-management.js:
  - > Add the following in the top of the file:

```
const cartTotalPriceElement = document.getElementById('cart-total-price');
const cartBadge = document.querySelector('.nav-items .badge');
```

> Add the following after getting the responseData:

```
if(responseData.updatedCartData.updatedItemPrice === 0){
   form.parentElement.parentElement.remove();
}else{
   const cartItemTotalPriceElement =
   form.parentElement.querySelector('.cart-item-price');

  cartItemTotalPriceElement.textContent =
  responseData.updatedCartData.updatedItemPrice.toFixed(2);
}

cartTotalPriceElement.textContent =
responseData.updatedCartData.newTotalPrice.toFixed(2);

cartBadge.textContent = responseData.updatedCartData.newTotalQuantity;
```

Adding the Order Controller & Basic Order Model ->

Advanced Backend

Adding the Order Controller & Basic Order Model

- In cart.ejs file:
  > Show the buy products button on a condition if the user is logged in as the following:

```
<% if(locals.isAuth) { %>
    <form action="/orders" method="POST">
        <input type="hidden" name="_csrf" value="<%= locals.csrfToken %>">
        <button class="btn">Buy Products</button>
    </form>
<% }else{ %>
    <p id="cart-total-fallback">Log in to proceed and purchase the items.</p>
<% } %>
```

- In the controllers folder:
  > create an orders.controller.js file

- In the routes folder:
  > create an orders.routes.js file

- In the models folder:
  > create an order.model.js file

Adding the Order Controller & Basic Order Model ->

Advanced Backend

Adding the Order Controller & Basic Order Model

- In the orders.routes.js:
  - > Include the following:
    - >> Import express
    - >> the orders.controller.js
    - >> Create a router
    - >> create a **post** route for the **/** ***[and require it in app.js and use it with the /orders prefix],*** that triggers an action **addOrder**
    - >> create a **get** route for **/ that triggers the getOrders function**
- In the orders.model.js file:
  - > Include the following:

```
class Order{
  constructor(cart,userData,status='pending',date,orderId){
    this.productData = cart;
    this.userData = userData;
    this.status = status;
    this.date = new Date(date);
    if(this.date){
      this.formattedDate = this.date.toLocaleDateString('en-US',{
        weekday : 'short',
        day : 'numeric',
        month : 'long',
        year : 'numeric'
      });
    }
    this.id = orderId
```

Saving Order in the Database ->

Advanced Backend

Save Orders in the Database

- In the orders.model.js file:

```
save(){
    if(this.id){
        const orderId = new mongodb.ObjectId(this.id);
        return db
                .getDb()
                .collection('orders')
                .updateOne({ _id: orderId }, { $set: { status: this.status }}
        );
    }else{
        const orderDocument = {
            userData : this.userData,
            productData : this.productData,
            date: new Date(),
            status: this.status
        };
        return db.getDb().collection('orders').insertOne(orderDocument);
    }
}
```

Saving Order in the Database ->

Advanced Backend

Save Orders in the Database

- In user.model.js file:
  > Add the following method:

```
static findById(userId) {
  const uid = new mongodb.ObjectId(userId);

  return db
   .getDb()
   .collection('users')
   .findOne({ _id: uid }, { projection: { password: 0} });
 }
```

Saving Order in the Database ->

Advanced Backend

Saving Order in the Database

- In the orders.controller.js file:
  - > Import the order and user Models
  - > Create the following addOrders function:

```
async function addOrder(req,res, next){
   const cart = res.locals.cart;
   let userDocument;
   try{
      userDocument = await User.findById(res.locals.uid);
   }catch(error){
     return next(error);
   }

   const order = new Order(cart,userDocument);
   try{
      await order.save();
   }catch(error){
      next(error);
      return;
   }
    req.session.cart = null;
   res.redirect('/orders');
}
```

Saving Order in the Database ->

Advanced Backend

Save Orders in the Database

- In the customer folder:
  - > Create a subfolder orders
  - > download the all-orders.ejs file and integrate it here

- In orders.controller.js file:
  - > Create the following function:

```
function getOrders(req, res) {
  res.render('customer/orders/all-orders');
}
```

Displaying Order (For Customers & Administrators) ->

Advanced Backend

Displaying Order (For Customers & Administrators)

- In the order.model.js file:

  > Create the following **static** methods transformOrderDocument and transformOrderDocuments as the following:

```
static transformOrderDocument(orderDoc) {
   return new Order(
     orderDoc.productData,
     orderDoc.userData,
     orderDoc.status,
     orderDoc.date,
     orderDoc._id
   );
}

static transformOrderDocuments(orderDocs) {
    return orderDocs.map(this.transformOrderDocument);
}
```

Displaying Order (For Customers & Administrators) ->

Advanced Backend

Displaying Order (For Customers & Administrators)

- In the order.model.js folder:
  > Create a **static async** findAll , findAllForUser and findById methods as the following:

```
static async findAll() {
    const orders = await db.getDb().collection('orders').find().sort({ _id: -1 }.toArray();
    return this.transformOrderDocuments(orders)
}

static async findAllForUser(userId) {
   const uid = new mongodb.ObjectId(userId);
   const orders = await db.getDb().collection('orders'.find({ 'userData._id': uid }).sort({ _id: -1 })
    .toArray();

   return this.transformOrderDocuments(orders);
 }

static async findById(orderId) {
   const order = await db.getDb().collection('orders')
     .findOne({ _id: new mongodb.ObjectId(orderId) });

   return this.transformOrderDocument(order);
 }
```

Displaying Order (For Customers & Administrators) ->

Advanced Backend

Displaying Order (For Customers & Administrators)

- In the order.model.js file:

> Add the following logic to the

```
static transformOrderDocument(orderDoc) {
   return new Order(
     orderDoc.productData,
     orderDoc.userData,
     orderDoc.status,
     orderDoc.date,
     orderDoc._id
   );
}

static transformOrderDocuments(orderDocs) {
    return orderDocs.map(this.transformOrderDocument);
}
```

Displaying Order (For Customers & Administrators) ->

Advanced Backend

Displaying Order (For Customers & Administrators)

- Update the order.controller.js folder:
  > update the getOrders function as the following:

```
async function getOrders(req, res) {
  try {
    const orders = await Order.findAllForUser(res.locals.uid);
    res.render('customer/orders/all-orders', {
      orders: orders,
    });
  } catch (error) {
    next(error);
  }
}
```

> Download the updates admin.routes.js and the admin.controller.js files
> Download the updates all-order.ejs file
> Download the order-item.ejs and order-list.ejs files and integrate them in the shared/includes folder.
> Download the admin-orders.ejs file and integrate it in an admin/orders folder
> Download the order-management.js file and integrate it in the script folder

Keep Cart items Updated ->

Advanced Backend

Keep Cart Items updated

- Download the updates **cart.controller.js** and the **cart.model.js** files and integrate it in the project
- Download the **update-cart-prices.js** middleware and integrate it in the project
- Download the updated **order-management.js** file and integrate it in the project
- Download the **order.css** file
- Download the updated **app.js** file
- Download the **not-found.js** middleware file and integrate it in the project

Working with Services and APIs ->

Advanced Backend

Notes

- The three method to share data between the UI and server:

  > **URL parameters** (ex: req.params.id)
  This is a part of the URL path defined in the route as **:paramName**, it is always a string
  Use case: Dynamic routes, resource identifiers

  > **Query String** (ex: req.query)
  This is the part that appears after the **?** In the URL, it is always a string and never used for sensitive data
  Use case: Optional filters, search parameters, pagination

  > **Request Body** (ex: req.body)
  This is always used with the submit form, it turns the name attribute values into keys used to store values, it requires a **POST method** and the **express.urlencode() middleware**
  Use case: Form submissions, creating/updating data

Note ->

Advanced Backend

Notes

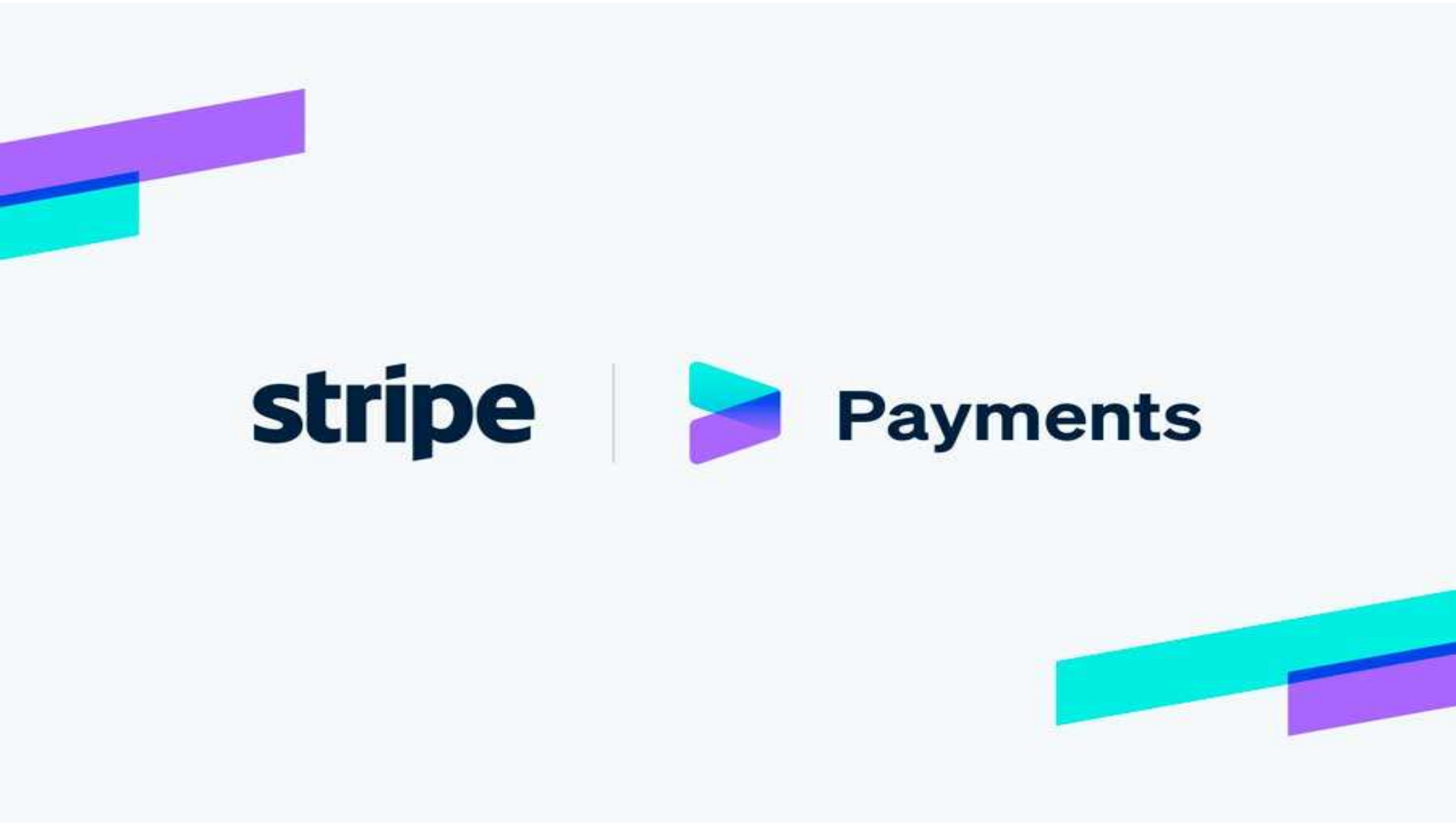
- Case Senario :
  - > **cart.controller :**
    This part manages the application's **data**, **logic**, and **rules(oftently in a form of class with methods and variables)**.
    In Express, these typically contain the schemas and methods for interacting with a database (like Mongoose or Sequelize models).

  - > **Views:**
    This part represents the presentation layer and is responsible for displaying the data to the user.
    These folders usually contain template files (like .ejs, .hbs, or .pug files).

  - > **Controller:** This part acts as the interface between the Model and View.
    It receives input from the user (via routes), processes it (by interacting with the Model), and updates the View.

  - * **Other common patterns**: MVP, Layered pattern (Medium to large web apps where logic is complex), Repository/Service Pattern, Featured Based pattern

Working with Services and APIs ->


stripe
Payments

Advanced Backend

What Are Services & APIs ?

- **Services:**
  - > Are just Services that we can use in our projects/Websites to enhance them.
  - > Such as:
    - **Packages** like : ***Axios*** and ***Express*** **[A package is actually a service]**
    - **Google Maps API**
    - **Google Analytics**
    - **Stripe Payments**
- **APIs :**
  - > All Services, provides some sort of end point that we can access through our code (URLs to which we can send requests that gives us responses in a JSON format)
  - > You can learn how to use an API by reading its documentation
  - > The collection of actions and end points is what we call APIs

As web developer we cannot build all features on our own, for that we use paid or free third-party services/APIs to add certain features to your app, or we can build a service/API that can be consumed by others

Introduction to Stripe ->

Advanced Backend

Introduction to Stripe

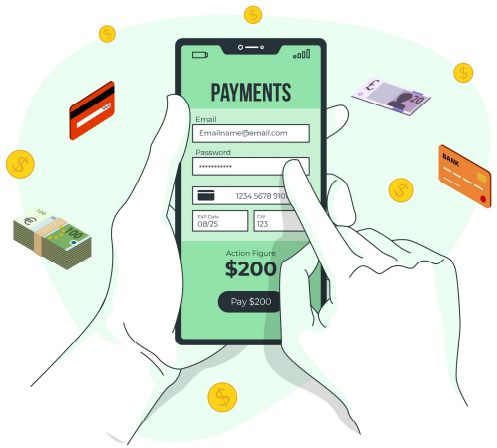
- Visit the following link to check the documentation: https://stripe.com/
- Go on **Developers > Get Started > Set Up Stripe > checkout quickstart**
- Visit this link to see how you can integrate this service with your code: https://docs.stripe.com/checkout/quickstart

- **To use Stripe:**
  - **You will need to create an account**
  - **You will need to install the stripe package : npm install - -save stripe**
  - **We will write the code to send the right data to the stripe server through the stripe package**

- **Checkout this documentation too :**
  https://docs.stripe.com/payments/accept-a-payment?integration=checkout

Creating a Stripe account ->

Advanced Backend



Creating a Stripe Account

- Click on create account
- Fill in your data
- Verify your email

- Afterwards you should see yourself on your dashboard, you should be on testing mode.
- Select **Developer** option on the bottom left and choose **API Keys** to find the API keys you will need **for sending requests**

Setting Up Stripe API Requests ->

Advanced Backend

Setting up Stripe API requests

- The best place to add stripe in our project is in the orders.controllers.js file in our controllers folder
- In the addOrder function, after storing the order and clearing the cart, we will make the real charge by adding the following:

  const stripe = require('stripe') ('Your secret API key')
- Copy the session code from the documentation first
- Update few lines of the code by the following updates:

  > add the following key: payment_method_types:['card'],

  > Update the value of the **name** key of the **product_data** key to the following:

  > Update the value of the **unit_amount** key of the **product_data** key to the following:

  > In the views/cutomer/orders folder create 2 files : success.ejs and failure.ejs, copy the all-orders.ejs content into these files and remove the style CSS and update the title of each page

  > In the orders.routes.js in the routes folder add these 2 routes as the following:

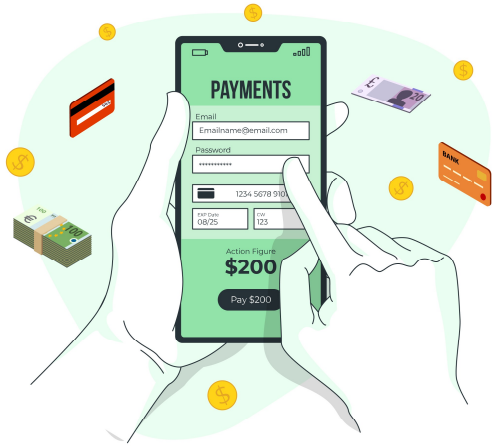
  router.get('/success', ordersController.getSuccess);

  and router.get('/failure' , ordersController.getFailure);

  > In the orders.controller.js file add the following:

```
function getSuccess(req,res){
    res.render('customer/orders/success');
}
```

```
function getFailure(req,res){
    res.render('customer/orders/failure');
}
```



Setting Up Stripe API Requests ->

Advanced Backend

Setting up Stripe API requests

- Update few lines of the code by the following updates:
  - > Update the **success_url** value key to the following:
    'http://localhost:3000/orders/success'
  - > Update the **cancel_url** value key to the following:
    'http://localhost:3000/orders/failure'
  - > Update the **line_items** key to the following:

```
cart.items.map(function(item) {
        return
        {
                price_data:
                {
                currency: 'usd',
                        product_data:
                        {
                        name: item.product.title,
                },
                unit_amount: +item.product.price.toFixed(2) * 100,
                },
                quantity: item.quantity,
        }
}),
```

Testing Stripe ->

Advanced Backend

Testing Stripe

- While testing Stripe payment use the following as a guide for dummy data related to visa Card:
  https://docs.stripe.com/payments/accept-a-payment
- **Fill in the other text inputs from random Dummy data that pops up into your mind**
- Click on the **Pay** button to pay
- If your payment was successful you see the success page you have created if not you should see the failure page.
- You can visit the Transactions from your dashboard to see you have gained and the balances to see how much net amount you have gained.

Update features for the Online Shop ->

Advanced Backend

Update features for the Online Shop

- Let’s remove the buy button if no items exist on the cart!
- Go to **views**/**customer**/**cart**/**cart.ejs** and update the condition to the following:

```
<% if(locals.isAuth && locals.cart.totalQuantity > 0) {%>
```

Deploying Websites ->

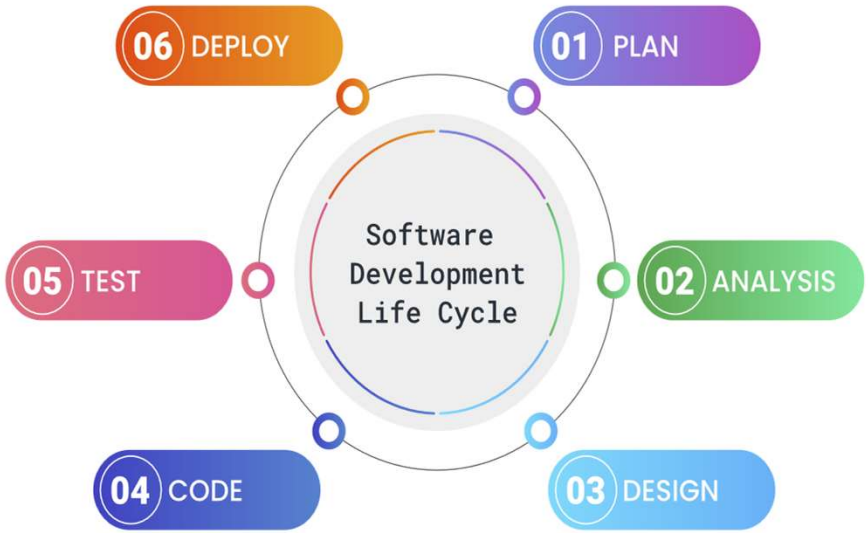

DEPLOYING
BUILDING FOR PRODUCTION
WEB APPLICATIONS

Advanced Backend

Deploying Websites

- First your website should be **Hosted** on another machine
  > To do so we must **Deploy** our website on a remote machine that is exposed to the world, because our machines **does not accept external HTTP requests**

- **Hosting providers** sell pre-configured remote machines and services.
- Choosing a Hosting provider depends on which type of website you are building : **Static** or **Dynamic** website



Static

- Only HTML, CSS & browser side JS
- No server-rendered templates, no server-side code
- No database server

Static Hosting Provider

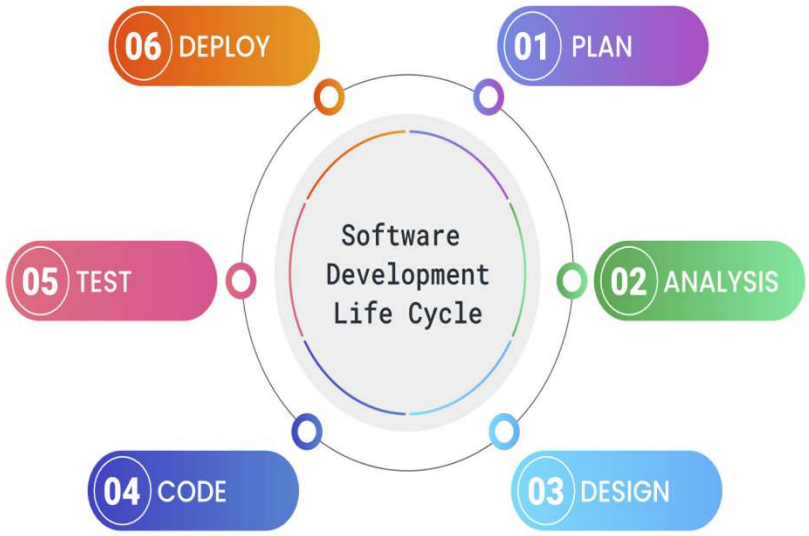
Dynamic

- Frontend & Backend
- Includes server-side code / server side templates
- Might require a database server

Dynamic Hosting Provider: NodeJs / PHP…

Hosting Database servers ->

Advanced Backend



Hosting Database Servers

- **Mongodb atlas** is a full managed cloud database that runs on remote machines provided by the MongoDB team

- **Deployment preparation Steps:**

  > Test your website (all features and options on all browsers) and prepare the code for deployment

  > Double check if your websites uses specific features in the Frontend code that might not be supported by all browsers

  > Optimize your website for Search Engine Optimization by adding important meta data that helps google understand and present your website

  > Improve the performance by shrinking the frontend assets (JS, images, CSS code)

Testing and code Preparation ->

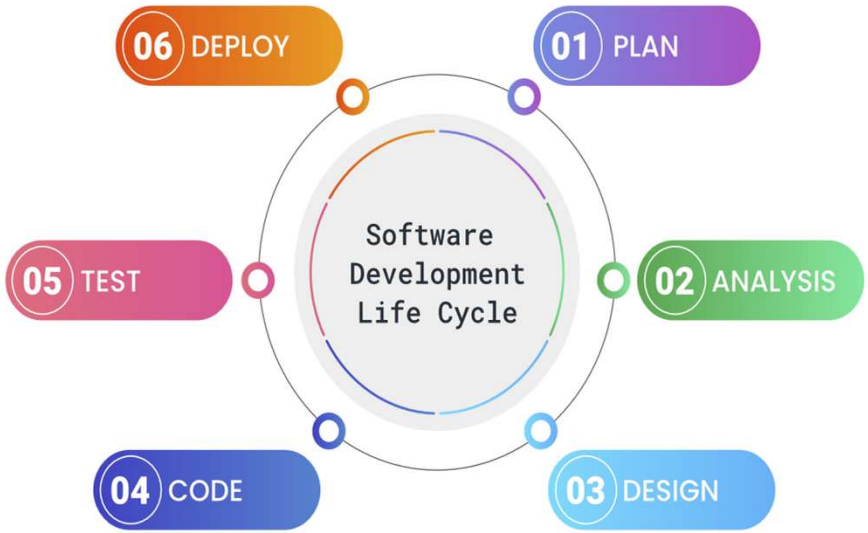
Advanced Backend

Testing and code Preparation

- There are 2 manners of testing your website
  > Manually
  > Automated Testing **[Advanced] :** You write code that test your website automatically

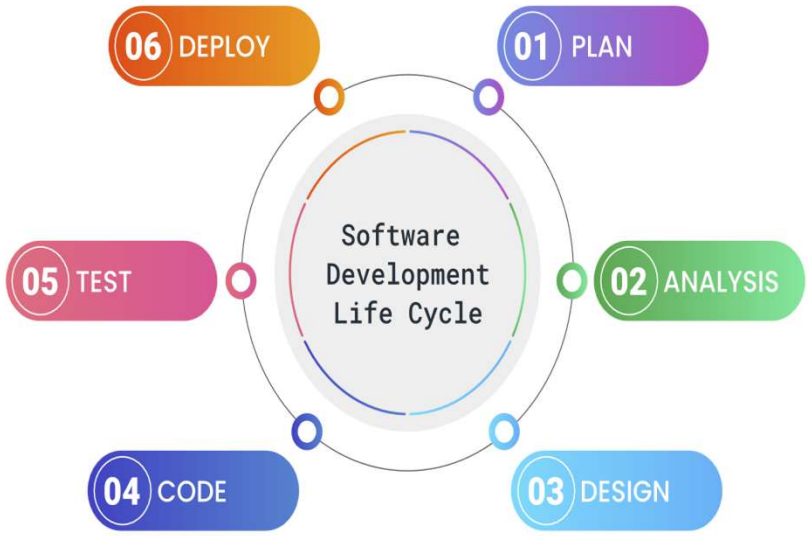
- **Enviroment variables:**
  > This is a features that exists in all programming languages.
  It is a variable used where a value is needed which is a special type of variable that could be set on the remote machine that is hosting our code and it will be injected into the running website code once it start running on that remote machine.
  The variable enviroment will be available in our code
  Such as:
  > **Check how it is done in the database.js file**

- If the users of your website should be able to upload data, use a **cloud storage provider** such as **cloudinary : https://cloudinary.com/**



Evaluating CROSS-Browser support ->

Advanced Backend



Evaluating Cross-Browser Support

- Nowadays most CSS properties and JS features are supported by all browser
- In case you are doubting a certain CSS property or JS feature you can check it out on the following website: https://caniuse.com/

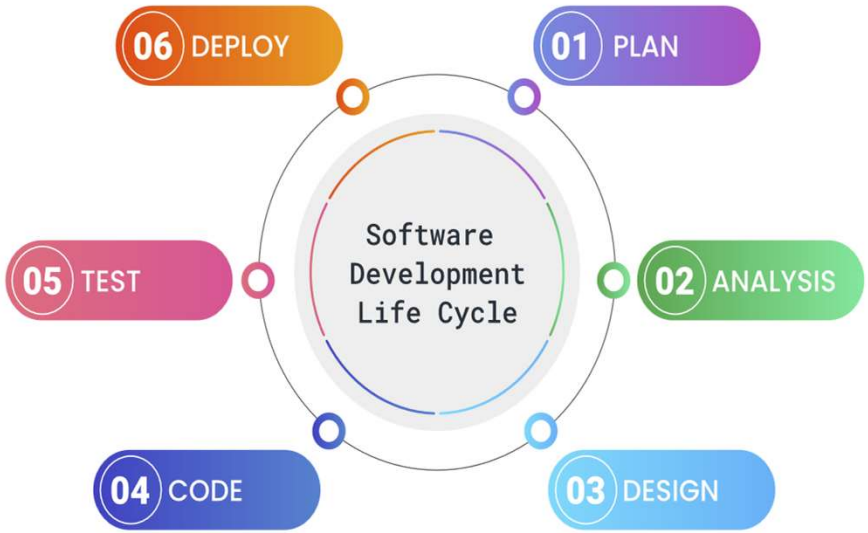
Search Engine Optimization ->

Advanced Backend

Search Engine Optimization

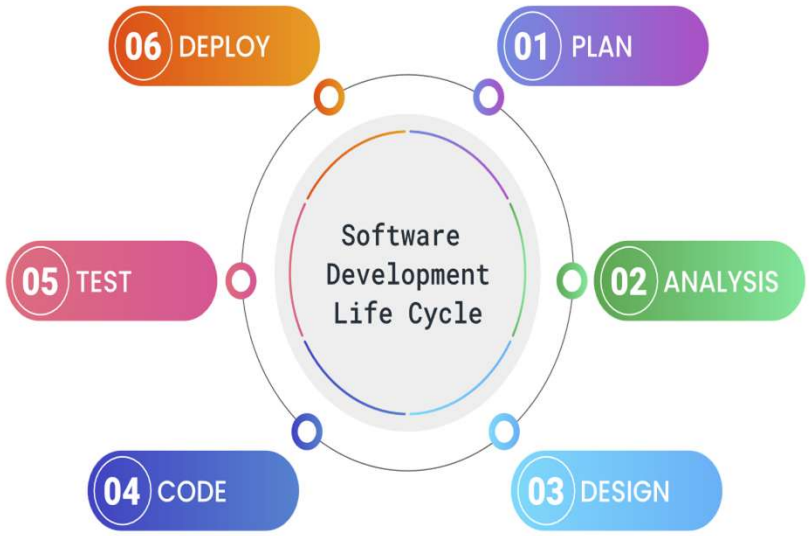
- Focus on High quality website by adding the following:

```
<meta
        name = “description”
        content =“description that will appear in search results”
>
```



Improving Performance and Shrinking assets ->

Advanced Backend



Improving performance and Shrinking assets

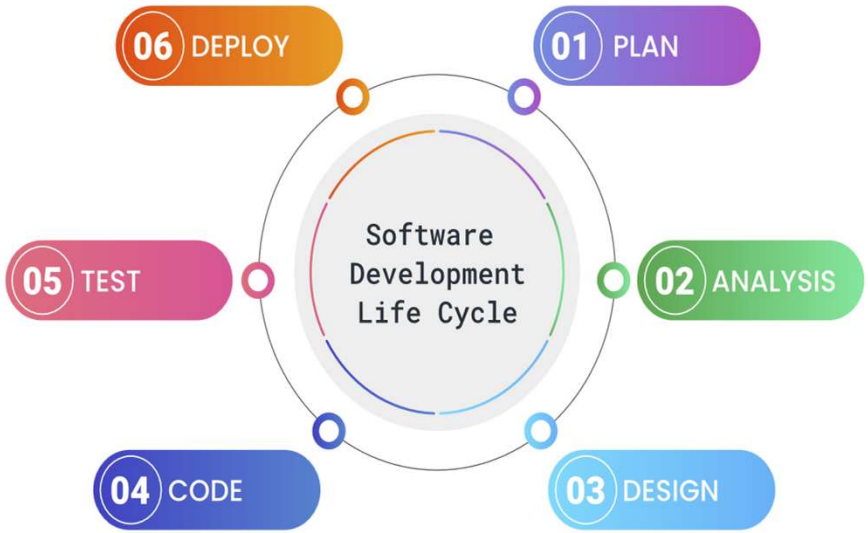
- We should ensure that our website visitors will not need to download large files or large image files
- You can optimize your raw JS code by using : **https://minify-js.com/**
- Optimize your images by using photoshop or other images editing tools to shrink the images dimensions, reducing the image quality according to how it is used on our website.
- You can use **cloudinary** to upload your website images there

Static Website Deployment ->

Advanced Backend

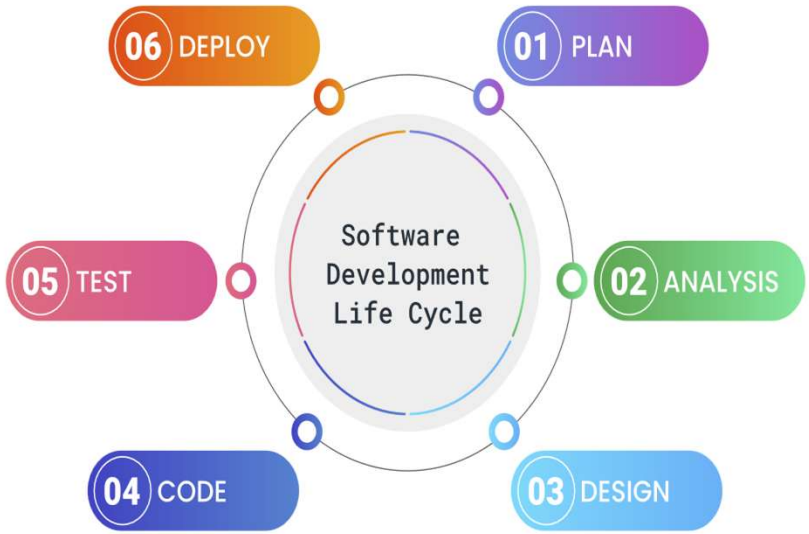
Static Website Deployment

- You can use **Netlify** to deploy your Static website by dropping the website folders



Dynamic Website Deployment ->

Advanced Backend



Dynamic Website Deployment

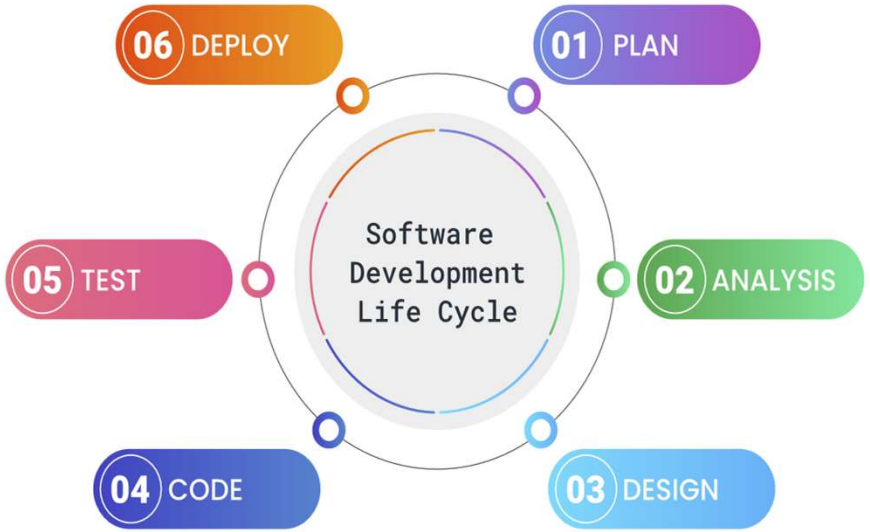
- We will use a Nodejs Hosting provider << **Heroku**>>
  https://www.heroku.com/
- For a Guide go on : **Developers** > **Languages** > select **NodeJs**
- **Download Heroku CLI** according to your OS
- Create your **account** on Heroku by **signing up**
- In your VSCode project directory open a terminal and run the following:
  >> **heroku login**
  >> **press any key**
  Login into Heroku
- Make sure you have **Git** installed and **create a git repository for your project**

Deploying a MongoDB database with Atlas ->

Advanced Backend

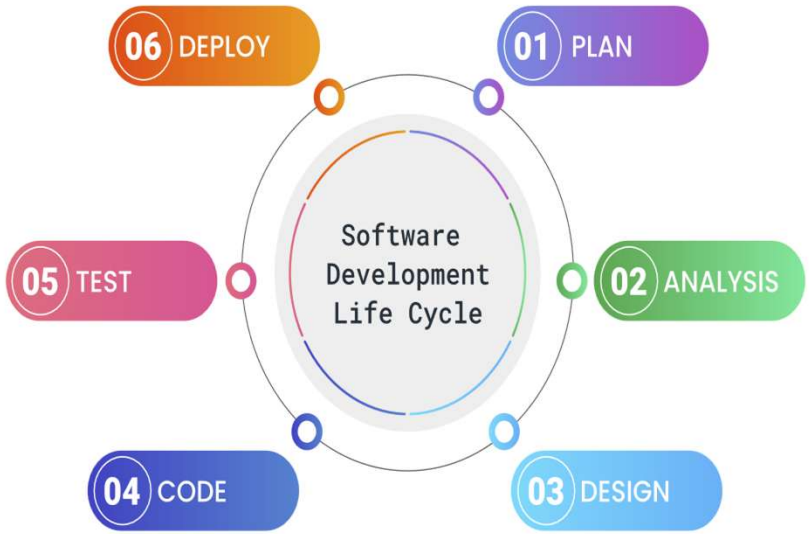
Deploying a MongoDB database with Atlas

- //content



Finishing Dynamic Website Deployment->

Advanced Backend



Finishing Dynamic Website Deployment

- //content

Web Services and Building Custom REST APIs ->

Course : Feedback

Please Scan the QR code to leave us your Feedback.