

Express - MongoDB

Express.JS

Avantages d'Express.js

1. Rend le développement d'applications web Node.js rapide et facile.
2. Facile à configurer et à personnaliser.
3. Vous permet de définir les routes de votre application en fonction des méthodes HTTP et des URL.
4. Comprend divers modules middleware que vous pouvez utiliser pour effectuer des tâches supplémentaires sur la demande et la réponse.
5. Facile à intégrer avec différents moteurs de modèles comme Jade, Vash, EJS, etc.
6. Permet de définir un intergiciel de gestion des erreurs.
7. Permet de servir facilement les fichiers statiques et les ressources de votre application.
8. Permet de créer un serveur API REST.
9. Facilité de connexion avec des bases de données telles que MongoDB, Redis, MySQL.

Express.JS

Vous pouvez installer express.js en utilisant npm. La commande suivante installera la dernière version d'express.js globalement sur votre machine afin que chaque application Node.js sur votre machine puisse l'utiliser.

```
npm install -g express
```

La commande suivante installera la dernière version d'express.js dans le dossier de votre projet.

```
C:\MyNodeJSApp> npm install express --save
```

Comme vous le savez, --save mettra à jour le fichier package.json en spécifiant la dépendance d'express.js.

Express.js Web Application

Dans cette section, vous apprendrez à créer une application Web à l'aide d'Express.js.

Express.js offre un moyen simple de créer un serveur Web et de rendre des pages HTML pour différentes requêtes HTTP en configurant des routes pour votre application.

Express.js Web Application

Tout d'abord, importez le module Express.js et créez le serveur web comme indiqué ci-dessous.

app.js: Express.js Web Server

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

// define routes here..

var server = app.listen(5000, function () {
  console.log('Node server is running..');
});
```

Express.js Web Application

Dans l'exemple ci-dessus, nous avons importé le module Express.js en utilisant la fonction `require()`.

Le module `express` renvoie une fonction.

Cette fonction renvoie un objet qui peut être utilisé pour configurer l'application Express (`app` dans l'exemple ci-dessus).

L'objet `app` comprend des méthodes pour acheminer les demandes HTTP, configurer le middleware, rendre les vues HTML et enregistrer un moteur de modèle.

La fonction `app.listen()` crée le serveur web Node.js à l'hôte et au port spécifiés.

Elle est identique à la méthode `http.Server.listen()` de Node. Exécutez l'exemple ci-dessus en utilisant la commande `node app.js` et faites pointer votre navigateur sur `http://localhost:5000`. Il affichera `Cannot GET /` car nous n'avons pas encore configuré de routes.

Configure Routes

Utilisez l'objet `app` pour définir les différentes routes de votre application. L'objet `app` comprend les méthodes `get()`, `post()`, `put()` et `delete()` pour définir les routes pour les requêtes HTTP GET, POST, PUT et DELETE respectivement.

L'exemple suivant montre la configuration des routes pour les demandes HTTP.

Configure Routes

Utilisez l'objet `app` pour définir les différentes routes de votre application. L'objet `app` comprend les méthodes `get()`, `post()`, `put()` et `delete()` pour définir les routes pour les requêtes HTTP GET, POST, PUT et DELETE respectivement.

L'exemple suivant montre la configuration des routes pour les demandes HTTP.

Configure Routes

Example: Configure Routes in Express.js

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

app.get('/', function (req, res) {
  res.send('<html><body><h1>Hello World</h1></body></html>');
});

app.post('/submit-data', function (req, res) {
  res.send('POST Request');
});

app.put('/update-data', function (req, res) {
  res.send('PUT Request');
});

app.delete('/delete-data', function (req, res) {
  res.send('DELETE Request');
});

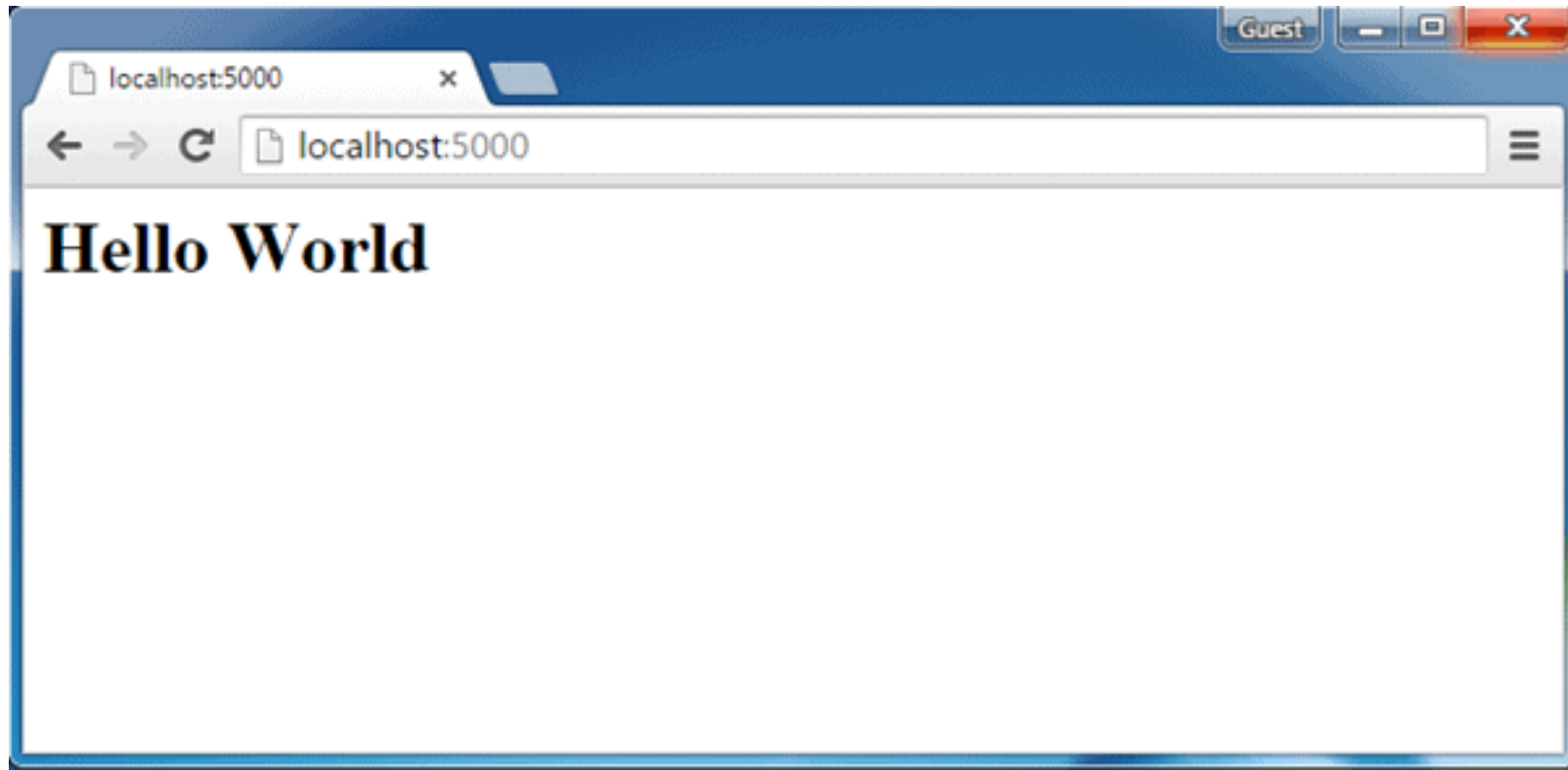
var server = app.listen(5000, function () {
  console.log('Node server is running..');
});
```

Configure Routes

Dans l'exemple ci-dessus, les méthodes `app.get()`, `app.post()`, `app.put()` et `app.delete()` définissent des routes pour HTTP GET, POST, PUT, DELETE respectivement. Le premier paramètre est le chemin d'une route qui commencera après l'URL de base. La fonction de rappel comprend la demande et l'objet de réponse qui sera exécuté sur chaque demande.

Exécutez l'exemple ci-dessus en utilisant la commande `node server.js`, et faites pointer votre navigateur sur `http://localhost:5000` et vous verrez le résultat suivant.

Configure Routes



Handle POST Request

Ici, vous apprendrez à gérer les requêtes HTTP POST et à récupérer les données du formulaire soumis.

Tout d'abord, créez le fichier `Index.html` dans le dossier racine de votre application et écrivez-y le code HTML suivant.

Handle POST Request

Example: Configure Routes in Express.js

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta charset="utf-8" />
  <title></title>
</head>
<body>
  <form action="/submit-student-data" method="post">
    First Name: <input name="firstName" type="text" /> <br />
    Last Name: <input name="lastName" type="text" /> <br />
    <input type="submit" />
  </form>
</body>
</html>
```

Handle POST Request

Pour traiter les requêtes HTTP POST dans Express.js version 4 et supérieure, vous devez installer le module middleware appelé body-parser. L'intergiciel faisait auparavant partie d'Express.js, mais vous devez désormais l'installer séparément.

Ce module body-parser analyse les données codées JSON, buffer, string et url soumises à l'aide d'une requête HTTP POST. Installez body-parser en utilisant NPM comme indiqué ci-dessous.

```
npm install body-parser --save
```

Maintenant, importez body-parser et récupérez les données de la requête POST comme indiqué ci-dessous.

Handle POST Request

app.js: Handle POST Route in Express.js

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

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

app.get('/', function (req, res) {
  res.sendFile('index.html');
});

app.post('/submit-student-data', function (req, res) {
  var name = req.body.firstName + ' ' + req.body.lastName;

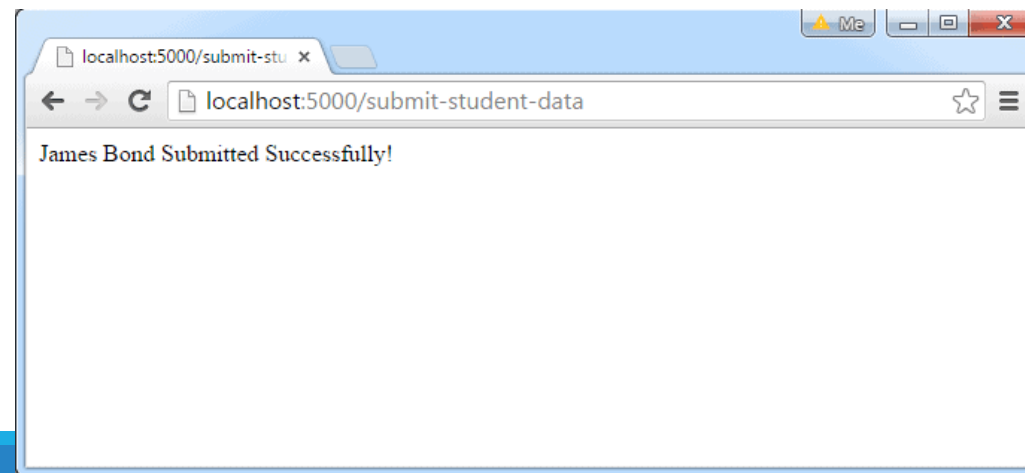
  res.send(name + ' Submitted Successfully!');
});

var server = app.listen(5000, function () {
  console.log('Node server is running..');
});
```

Handle POST Request

Dans l'exemple ci-dessus, les données POST sont accessibles à l'aide de `req.body`. Le `req.body` est un objet qui comprend des propriétés pour chaque formulaire soumis. `Index.html` contient les types de saisie `firstName` et `lastName`, vous pouvez donc y accéder en utilisant `req.body.firstName` et `req.body.lastName`.

Maintenant, exécutez l'exemple ci-dessus en utilisant la commande `node server.js`, faites pointer votre navigateur sur `http://localhost:5000` et voyez le résultat suivant.



Handle POST Request

Exercice

Créer un formulaire de contact :

Nom, prénom, email et message

La page suivante devra afficher le message suivant :

Bonjour [Nom] [prénom],

Merci de nous avoir contacter.

Nous reviendrons vers vous dans les plus bref délai à cette adresse : [email]



Mongodb

DÉCOUVREZ COMMENT ACCÉDER À LA BASE DE DONNÉES
DOCUMENTAIRE MONGODB À L'AIDE DE NODE.JS DANS CETTE
SECTION.

Installation de mongodb

Afin d'accéder à la base de données MongoDB, nous devons installer les pilotes MongoDB. Pour installer les pilotes MongoDB natifs à l'aide de NPM, ouvrez une invite de commande et écrivez la commande suivante pour installer le pilote MongoDB dans votre application.

```
npm install mongodb --save
```

Il faut ensuite créer une base de données mongo.

Deux solutions s'offrent à vous, l'installation d'une instance de mongodb sur votre machine, ou d'utiliser mongodb Atlas qui met à disposition des bases gratuitement.

<https://account.mongodb.com/account/login>

Installation de mongodb

```
const url =  
"mongodb+srv://frednad:123test@cluster0.4vjgd.mongodb.net/Formu?retryWrites=true&w=ma  
jority"
```

```
mongoose.connect(url, {  
  useNewUrlParser:true,  
  useUnifiedTopology: true  
}).then(console.log("MongoDB connected"))  
.catch(err => console.log(err))
```

Installation de mongodb

Créons un dossier models

Créons un fichier Contact.js

```
const mongoose = require('mongoose');  
const formSchema = mongoose.Schema({  
  name : { type: String},  
  last : { type: String},  
  email : { type: String},  
});  
module.exports = mongoose.model('Form', formSchema)
```

Installation de mongodb

```
var Form = require('./models/Contact');
app.post("/submit-data-form", function(req, res){
  const Data = new Form({
    lastname: req.body.lastname,
    firstname: req.body.firstname,
    email: req.body.email,
    message: req.body.message
  });
  Data.save().then(()=>{
    res.redirect('/')
  }).catch(err=>console.log(err))
});
```

Installation de mongodb

```
app.get("/", function (req, res){  
  
  Form.find().then(data=>{  
    console.log(data);  
  }).catch(err=> console.log(err))  
  
})
```

EJS

<% Embedded JavaScript %>

Templating: Node, Express, EJS

Les vues

EJS

Les vues

`npm install ejs`

Créer un dossier views et un fichier Home.ejs

Le contenu du fichier va être de l'html basique. Juste il y aura par moment des éléments que nous allons rajouter pour combler l'affichage déjà existant.

Aller ensuite dans le fichier app.js précédemment créé et ajouter la ligne :

```
app.set('view engine', 'ejs');
```

Il nous reste plus qu'à afficher cette page, il suffit d'ajouter

```
res.render('Home');
```

Les vues

Affichons maintenant les données.

Dans app.js :

```
app.get("/", function (req, res){  
  Form.find().then(data=>{  
    res.render('Home', {data:data});  
  }).catch(err=> console.log(err))  
})
```

Les vues

Affichons maintenant les données.

Dans Home.ejs :

```
<%= data %>
```

On peut voir que les data apparaissent sur la page Home.

On peut maintenant passer aux choses sérieuse.

NB : Il faut installer l'extension EJS language support

Les vues

Affichons maintenant les données.

Dans Home.ejs :

```
<% data.forEach((form)=>{%>
  <h3>
    <%= form.lastname %>
  </h3>
<%}) %>
```

Les vues

Editons les données

```
app.get('/form/:id', (req, res) => {  
  Form.findOne({  
    _id: req.params.id  
  }).then(data => {  
    res.render('Edit', { data: data });  
  })  
  .catch(err => console.log(err));  
})
```

Les vues

Editons les données

Créons une vue Edit pour l'éditions des données

Et mettrons :

```
<%= data %>
```

Les vues

Les méthodes put et delete n'existent pas en html. Il suffit de rajouter le package

```
$ npm install method-override
```

```
const methodOverride = require('method-override')
```

```
app.use(methodOverride('_method'));
```

Les vues - update

```
<form action="/form/edit/<%= data._id%>?_method=PUT" method="post">
```

```
  <input type="hidden" name="_method" value="PUT">
```

```
    <label for="prenom">Prénom</label>
```

```
    <input type="text" id="prenom" name="firstname" value="<%= data.firstname%>">
```

```
  <br>
```

```
[...]
```

```
</form>
```


Les vues - update

```
const Data = {  
  prenom : req.body.prenom,  
  nom : req.body.nom,  
  age: req.body.age,  
  email: req.body.email,  
  message: req.body.message  
};
```

Les vues - update

```
Form.updateOne({_id: req.params.id}, {$set: Data})  
  .then((result) => {  
    console.log(result);  
    res.redirect('/')  
  }).catch((err) => {  
    console.log(err);  
  });
```

Les vues - delete

Form.remove étant déprécié :

```
app.delete('/form/delete/:id', (req, res) => {  
  Form.findOneAndDelete({  
    _id: req.params.id  
  }).then(() => {  
    res.redirect('/');  
  }).catch(err => console.log(err));  
})
```

Les vues - delete

```
<form action="/contact/delete/<%= data.id %>?_method=DELETE" method="post">  
  <input type="hidden" name="_method" value="DELETE">  
  <button type="submit">Supprimer la donnée</button>  
</form>
```

Les vues

Affichons le résultat dans un tableau.

Dans Home.ejs

Donnons la possibilité aux utilisateurs de supprimer une data depuis la page Edit d'une donnée.

Les vues - EJS

Exercice MY BLOG

Créer un modèle Post.

Créer **un formulaire** pour qu'un utilisateur crée des posts sur un sujet au choix

Créer **une page** où tout les posts seront affichés (page d'accueil).

Créer **un formulaire** pour éditer un post via un id.

Créer **un formulaire** pour supprimer un post via un id

Navbar permettra de passer d'une page à une autre

Utilisation de bootstrap autorisé.

Bonus :

L'utilisateur pourra supprimer ou éditer un post depuis la page d'accueil.

Bienvenue sur la page

Voici les posts

Titre de mon super post

Sous-titre de mon post

La c'est du Lorem pour une description

Editer

Supprimer

Titre de mon deuxieme post

Sous-titre de mon post

La c'est du Lorem pour une description

Editer

Supprimer

Titre de mon troisième post

Sous-titre de mon post

La c'est du Lorem pour une description

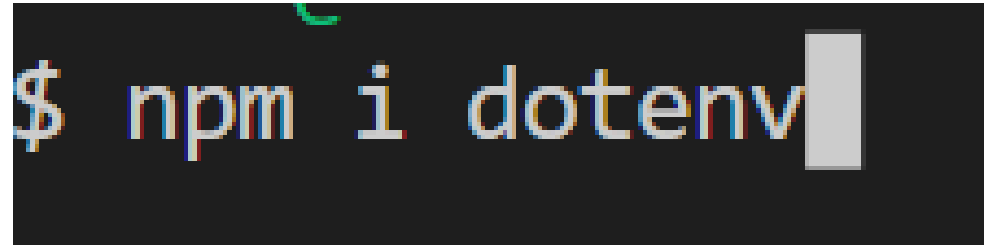
Editer

Supprimer

Variables d'environnement

Avec Express vous pouvez créer des variables d'environnement afin de sécuriser les données sensibles et ainsi éviter qu'elles soient visibles de tous. Ce mode permet de faciliter le travail en équipe.

Installons le package dotenv.

A terminal window with a dark background. The command '\$ npm i dotenv' is entered in a light blue monospace font. A white cursor is positioned at the end of the command.

```
$ npm i dotenv
```

Crée le fichier .env à la racine de votre projet, et là vous pouvez mettre vos variables d'environnement.

Commençons par mettre le lien de connexion vers mongodb atlas dans ce fichier.

Variables d'environnement

Créons une variable :

```
⚙ .env  
1 DATABASE_URL="mongodb+srv://user123:user123@cluster0.ktn6e7j.mongodb.net/?retryWrites=true&w
```

Pour faire appel a cette variable :

```
const url = process.env.DATABASE_URL
```

Variables d'environnement

```
require('dotenv').config();
```

Gestion utilisateur

AJOUTONS LA PARTIE LOGIN SUR L'APPLICATION

Model User

Nous allons avoir besoin d'un Model et 3 vues :

Model :

➤ User

3 vues :

➤ Signin

➤ Login

➤ UserPage

Model User

Créer le model User qui nous permettra de gérer les utilisateurs.

```
const mongoose = require('mongoose');  
const userSchema = mongoose.Schema({  
  username : { type: String, required : true},  
  email : { type: String, required : true},  
  password : { type: String, required : true},  
  admin : { type: Boolean}  
});  
module.exports = mongoose.model('User', userSchema)
```

Model User

Créer le model User qui nous permettra de gérer les utilisateurs.

```
app.post("/api/signup", function(req, res){  
  const Data = new User({  
    username: req.body.username,  
    email: req.body.email,  
    password: req.body.password  
  })  
  Data.save().then(()=>{  
    console.log("Data saved !");  
    res.redirect('/');  
  });  
});
```

Model User

Créer la vue Signin.ejs qui permettrons aux utilisateurs de créer un compte

Ajouter un formulaire et un input pour chaque champs

```
<form action="/api/login" method="post">
  <label for="exampleInputUser" class="form-label">Username</label>
  <input type="text" name="username" class="form-control" id="exampleInputUser" aria-describedby="emailHelp">
  <label for="exampleInputEmail1" class="form-label">Email address</label>
  <input type="email" name="email" class="form-control" id="exampleInputEmail1" aria-describedby="emailHelp">
  <label for="exampleInputPassword1" class="form-label">Password</label>
  <input type="password" name="password" class="form-control" id="exampleInputPassword1">

  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Model User

Créer la route pour l'affichage de la page Signin

```
app.get("/signin", function (req, res){  
    res.render('Signin');  
})
```


Model User

On va faire la route pour la création d'un User

```
app.post("/api/signin", function(req, res){
```

```
  const Data = new User({  
    username: req.body.username,  
    email: req.body.email,  
    password: req.body.password  
  })
```

```
  Data.save().then(()=>{  
    console.log("User saved !");  
    res.redirect('/');  
  });
```

```
});
```

Model User

Créer la vue Login.ejs qui permettrons aux utilisateurs de se logger sur le site

Ajouter un formulaire et un input pour chaque champs

```
<form action="/api/login">
  <div class="mb-3">
    <label for="exampleInputEmail1" class="form-label">Email address</label>
    <input type="email" class="form-control" id="exampleInputEmail1" aria-describedby="emailHelp">
    <div id="emailHelp" class="form-text">We'll never share your email with anyone else.</div>
  </div>
  <div class="mb-3">
    <label for="exampleInputPassword1" class="form-label">Password</label>
    <input type="password" class="form-control" id="exampleInputPassword1">
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

Model User

Créer la route /login

```
app.get("/login", function (req, res){  
  res.render('Login');  
})
```

Model User

Créer la vue UserPage.ejs :

```
<body>
```

```
  <%= data %>
```

```
</body>
```

Model User

Créer la route /api/login :

Le système est assez basique la sécurité est minimal nous allons voir ensuite comment renforcer tout cela.

```
app.post("/api/login", function(req, res){
  User.findOne({
    email: req.body.email
  }).then(user =>{
    if(!user)
    {
      return res.status(404).send('No user found.');
```

```
    }
    console.log(user);
    if(user.password != req.body.password){
      return res.status(404).send('Invalid password');
```

```
    }
    res.render('UserPage', {data:user})
  }).catch(err => console.log(err));
});
```

Bcrypt.js

Bcrypt

Npm i bcrypt

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

Dans app.post("/api/signin"):

```
const Data = new User({  
  username: req.body.username,  
  email: req.body.email,  
  password: bcrypt.hashSync(req.body.password, 10)  
})
```

Bcrypt

Dans app.post("/api/login"):

```
if(!bcrypt.compareSync(req.body.password, data.password)){  
    return res.status(404).send('password not match');  
}
```


Bcrypt

Page Admin :

Hébergement Heroku

Créer un compte gratuit sur heroku

<https://www.heroku.com/>



Mini projets

Les projets

- ❖ Le projet devra comporter un CRUD
- ❖ Le sujet est libre
- ❖ L'objectif est de vous préparer à votre projet final et de vous préparer au projet react
- ❖ Le projet devra comporter un module de connexion, Bootstrap autorisé
- ❖ Attention c'est un projet **BACKEND**

Projets

- Rendu 14h
- Présentation du code et des fonctionnalités.
- 10 MIN MAX
- Explications des modules développés
- Présentation du git
- Démo du site