

I. Introduction

Au cours des précédentes semaines, nous avons réalisé la partie frontend de notre application. Maintenant il est temps de s'intéresser au backend.

Dans cette partie vous allez créer une api pour gérer les témoignages (Testimony) sur les articles de notre magasin de burgers

Les données seront stockées dans une base de données mongodb sur le cloud

II. Express.js

[Express.js](#), parfois aussi appelé « Express », est un **framework backend** [Node.js](#) minimaliste et rapide qui offre des fonctionnalités et des outils robustes pour développer des applications **backend** évolutives. Il permet de gérer aisément le routage et offre de nombreuses fonctionnalités supplémentaires à l'objet [HTTP](#) utilisé avec [node.js](#). Nous allons nous en servir pour développer des applications jouant un rôle de serveur de données pour nos applications [React.js](#)

A. Initialisation du projet

Dans le dossier du projet, nous avons précédemment créé un dossier frontend

- Placez-vous à la racine du projet
- Créez-y un dossier backend
- Lancez votre éditeur de code préféré dans le dossier backend

1. Commandes utiles

La commande **npm init -y** initialise le projet et génère la version initiale du fichier package.json qui contiendra nos packages

La commande **npm install express** ajoutera le package express à votre application.

- Exécutez les commandes

```
npm init
{
  "name": "kazaburger-backend",
  "version": "1.0.0",
  "description": "api de kazaburger",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "type": "commonjs"
}
```

Renseignez à votre convenance les informations demandées à l'exception du entrypoint à positionner à « app.js »

```
npm install express
```

- Créez un fichier **app.js** contenant le code suivant

```
//import du module
const express = require("express");

//instanciation d'express
const app = express();

//definition du port d'écoute du serveur
const port = 4000;

//définition des routes
app.get("/", (req, res) => {
  res.set("Content-Type", "text/html");
  res.send("Hello world !!");
});

app.listen(port, () => {
  console.log("Server app listening on port " + port);
});
```

- Pour démarrer le serveur exécutez la commande suivante dans le terminal:

```
node app.js.
```

Le message « Server app listening on port 4000 » ,s'affiche ? ➔ Votre serveur fonctionne

- Entrez l'URL suivante dans votre navigateur web : `http://localhost:4000`

Le message **Hello world !!** apparaît ? ➔ Votre serveur fonctionne.

Explications :

- La commande [res.set](#) permet de fixer la valeur d'un champ de [l'entête HTTP](#) transmis au client.
- La commande [res.send](#) permet de transmettre une réponse au client.

B. Rappel sur le protocole http

Le protocole [HTTP](#) définit le mode de communication entre le client (votre navigateur) et le serveur (votre application *Express*). Il définit le format des requêtes pouvant être émises par le client. Elles peuvent être décomposées en deux éléments :

- le [verbe](#), encore appelé *méthode*. Ici notre client n'utilisera que les verbes suivants :
 - **POST** : pour envoyer des données au serveur.
 - **PUT/PATCH** : pour modifier ou remplacer des données stockées sur le serveur.
 - **GET** : pour obtenir des données du serveur.
 - **DELETE** : pour supprimer des données stockées sur le serveur.
- l'*URI*
 - identifie le serveur en précisant le *domaine*.
 - indique le *port* utilisé : *80* par défaut.
 - précise la méthode utilisée : *GET* par défaut.
 - donne le *chemin* associé au verbe : */* par défaut.

Exemples

- L'URL `http://localhost:4000` définit une requête *GET*. `localhost` est le *domaine*, `:4000` donne le port utilisé. Aucune autre information n'étant spécifiée, le chemin par défaut est `/`.

- L'URL `http://localhost:4000/xx` transmet une requête *GET* avec le chemin `/xx`.

III. Routage

- Arrêtez votre application (ctrl-C).
- Saisissez la commande suivante : `npm install cors morgan nodemon`
- Remplacez le contenu du fichier `app.js` par le code ci après

```
const express = require("express");
//on délègue la gestion des routes à un fichier de
const router = require("./router");
const cors = require("cors");
const morgan = require("morgan");
const bodyParser = require("body-parser");
const app = express();

const port = 4000;

app.use(morgan("combined"));
app.use(cors());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));
app.use(router); // Requests processing will be defined in the file router
app.listen(port, () => console.log("Server app listening on port " + port));
```

Explications :

- Le package `morgan` permet de définir les informations que le serveur affiche dans la console à chaque fois qu'il reçoit une requête HTTP. Ce package est particulièrement utile en phase de développement de votre serveur.
- Le package `cors` permet de configurer la façon dont des applications web définies sur un autre domaine peuvent accéder aux ressources de votre serveur. Ce mécanisme est appelé CORS pour Cross-Origin Resource Sharing, d'où le nom de ce package. Faire appel à ce package sans lui passer d'arguments permet d'autoriser tous les accès à votre ressource. Pour des exemples d'utilisation plus élaborés, vous pouvez consulter cette [page](#).
- Le package `body-parser` permet de décomposer les requêtes HTTP POST, PATCH, etc. afin de pouvoir extraire les informations transmises dans des formulaires. Ces informations apparaissent dans le champ `req.body`.
- L'instruction `app.use(router)` permet d'utiliser les routes définies dans le fichier `router.js`.
- **Nodemon** est un utilitaire qui relance automatiquement le serveur à chaque modification.
- Modifiez l'entrée `scripts` de votre fichier `package.json` comme suit . Ajouter dans la section « `scripts` » la ligne

```
"start": "nodemon server.js"
```

- `npm run start` déclenchera donc l'exécution de la commande `nodemon server.js`.

- Créez un fichier « router.js » qui contient le code suivant

```
const express = require("express");
const router = express.Router();

router.get("/", (req, res) => {
  res.send("Bienvenue sur la page d'accueil");
});

module.exports = router;
```

Explications :

Ce code se compose de plusieurs parties qu'il convient de bien comprendre :

- **.get** est une méthode de l'objet router qui permet de répondre aux requêtes GET. Comme les autres méthodes permettant de répondre aux différentes requêtes (POST, PUT, ...), cette méthode comprend deux arguments.
 - Le premier argument de la méthode **router.get**, il définit le chemin auquel le routeur réagit. Ici « / »
 - Le deuxième argument (req,res) => {...} est une fonction, dite « [middleware](#) », qui sera déclenchée par le routeur lorsque qu'une requête http, constituée de la bonne méthode et du bon chemin, sera envoyée au serveur. Cette fonction comprend deux arguments :
 - L'objet **req** (**request**) est utilisé pour représenter la requête **HTTP entrante**,
 - L'objet **res** (**response**) est utilisé pour envoyer la **réponse http** au client

A. L'objet request

- **req.params**: Contient des propriétés définies dans la partie de l'URL de la route.
- **req.query**: Contient les paramètres de requête de l'URL.
- **req.body**: Contient les données envoyées dans le corps de la requête. Pour accéder à **req.body**, vous devez utiliser un body-parser middleware.
- **req.headers**: Contient les en-têtes HTTP de la requête.
- **req.cookies**: Contient les cookies envoyés par le client.
- **req.method**: Contient la méthode HTTP de la requête (GET, POST, PUT, DELETE, etc.).
- **req.path**: Contient le chemin de l'URL demandée.
- **req.url**: Contient l'URL demandée.
- **req.hostname**: Contient le nom d'hôte de l'URL demandée.
- **req.protocol**: Contient le protocole de la requête (HTTP ou HTTPS).
- **req.ip**: Contient l'adresse IP du client.

B. L'objet response

- **res.send()**: Envoie une réponse HTTP au client. Cette méthode peut envoyer divers types de données (texte, JSON, HTML, etc.).
- **res.sendFile()**: Envoie un fichier
- **res.json()**: Envoie une réponse JSON au client.
- **res.render()**: Rend un modèle (template) avec les données spécifiées et l'envoie au client.
- **res.status()**: Définit le code d'état HTTP de la réponse.
- **res.redirect()**: Redirige l'utilisateur vers une autre URL.
- **res.cookie()**: Définit un cookie dans la réponse.
- **res.clearCookie()**: Efface un cookie.
- **res.setHeader() / res.header()**: Définit un en-tête de réponse.
- **res.get()**: Récupère la valeur d'un en-tête de requête.
- **res.locals()**: Un objet contenant des variables locales accessibles dans les modèles lors du rendu.
- **res.download()**: Déclenche le téléchargement d'un fichier.
- **res.end()**: Termine le processus de réponse sans envoyer de données.

NB : Une fonction middleware devra toujours faire une réponse au client sous peine de le voir attendre indéfiniment

IV. Route inexistante

- Ouvrez la page <http://localhost:4000/accueil>. Que se passe-t-il ?
- Ajoutez la méthode suivante au routeur afin que celui-ci retourne le [statut HTTP approprié](#) au client :

```
router.use((req, res) => {  
  res.status(404);  
  res.json({ error: "Page not found" });  
});
```

Le terme use signifie que la *fonction middleware* est exécutée quelle que soit la méthode *HTTP* utilisée. Ici, le premier argument (le *chemin*) n'est pas précisé. En conséquence, cette méthode sera exécutée systématiquement.

Elle doit donc être la dernière de la liste . Si aucune méthode du routeur n'a déclenché sa *fonction middleware*, alors le routeur retourne le message **Page not found** avec un statut d'erreur **404**.

NB : Remarquez l'utilisation de `res.json` qui permet de fournir la réponse au format json

Le serveur doit respecter les [codes de réponse HTTP](#), ce qui permet au client de réagir correctement en fonction du code qu'il reçoit.

V. Récupération des données des routes

Le client transmet des données au serveur pour que ce dernier exécute un traitement
Ces données peuvent être transmises via la chaîne d'url

Une URL se compose de différents fragments dont certains sont obligatoires et d'autres optionnels. Pour commencer, voyons les parties les plus importantes d'une URL :

<http://www.exemple.com:80/chemin/vers/monfichier.html?clé1=valeur1&clé2=valeur2#QueIquePartDansLeDocument>

http:// correspond au protocole. Ce fragment indique au navigateur le protocole qui doit être utilisé pour récupérer le contenu. Généralement, ce protocole sera HTTP ou sa version sécurisée : HTTPS

www.exemple.com correspond au nom de domaine. Il indique le serveur web auquel le navigateur s'adresse pour échanger le contenu. À la place du nom de domaine, on peut utiliser une adresse IP, ce qui sera moins pratique (et qui est donc moins utilisé sur le Web).

:80 correspond au port utilisé sur le serveur web. Il indique la « porte » technique à utiliser pour accéder aux ressources du serveur. Généralement, ce fragment est absent car le navigateur utilise les ports standards associés aux protocoles (80 pour HTTP, 443 pour HTTPS). Si le port utilisé par le serveur n'est pas celui par défaut, il faudra l'indiquer.

/chemin/vers/monfichier.html est le chemin, sur le serveur web, vers la ressource. Au début du Web, ce chemin correspondait souvent à un chemin « physique » existant sur le serveur. De nos jours, ce chemin n'est qu'une abstraction qui est gérée par le serveur web, il ne correspond plus à une réalité « physique ».

?clé1=valeur1&clé2=valeur2 sont des paramètres supplémentaires fournis au serveur web. Ces paramètres sont construits sous la forme d'une liste de paires de clé/valeur dont chaque élément est séparé par une esperluette (&). Le serveur web pourra utiliser ces paramètres pour effectuer des actions supplémentaires avant d'envoyer la ressource. Chaque serveur web possède ses propres règles quant aux paramètres. Afin de les connaître, le mieux est de demander au propriétaire du serveur.

Dans notre serveur nous allons pouvoir récupérer la ressource et les paramètres

url	route	récupération
http://localhost:4000/product/115	/product/: attribut	req.params.attribut → 115
http://localhost:4000/product/?attribut=115	/product/	req.query.attribut → 115

NB : 'attribut' est un exemple de nom de variables, il faudra le remplacer par le nom de variable que vous souhaitez utiliser par rapport à votre situation

NB : si les données proviennent d'un formulaire elles sont à récupérer dans l'objet req.**body**

- Remplacez le contenu du fichier « router.js » pour intercepter les routes

```
const express = require("express");
const router = express.Router();
module.exports = router;

router.get("/", async(req, res) => {console.log("tous les produits");});

router.get("/:val", async(req, res) => { console.log("un produit par id :val");});

router.use((req, res) => {
  res.status(404);
  res.json({
    error: `${req.method} + ":" + req.originalUrl} Page not found`,
  });
});

module.exports = router;
```

Il est aussi possible de regrouper les routeurs par ressources avec la fonction « route »

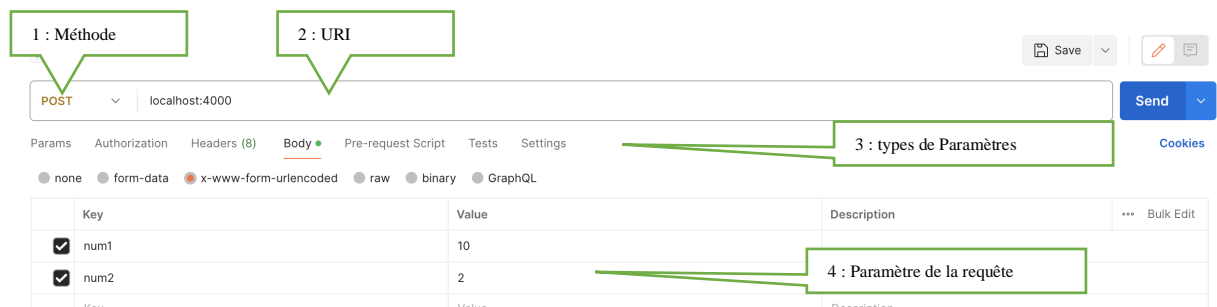
```
/** Produits */
router.route("/product")
  .get((req, res) => {
    console.log("GET");
  })
  .post((req, res) => {
    console.log("POST");
  });
```

VI. Utilisation de notre api avec un client type postman

Pour rappel, Postman, comme thunderclient, ou cUrl, est un client http qui permet de tester les requêtes http en dehors du navigateur.

L'intérêt est pouvoir utiliser les méthodes autres que GET

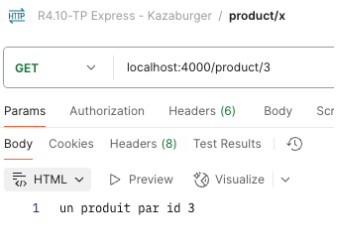
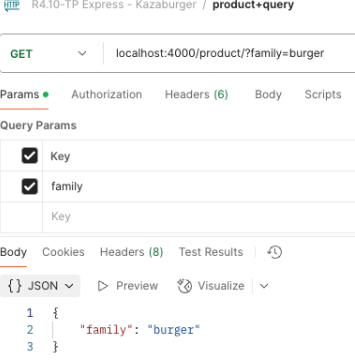
- Dans postman créez une nouvelle requête



Le serveur construit précédemment respecte, à minima les recommandations d'une API REST. Nous n'étudions pas cette API en détail, néanmoins, le fait de respecter les bases nous

permet de créer des interfaces que tout client peut utiliser. Pour plus de détails sur les API REST, vous pouvez vous référer à cette [page](#). Les réponses obtenues respectant le format REST, n'importe quelle application peut interroger le serveur et exploiter les résultats obtenus.

- Testez les routes avec postman. Attention les résultats s'affichent dans le terminal du serveur
- Modifiez le code pour renvoyer le message au client plutôt que dans la console
- Modifiez le code pour récupérer les paramètres envoyés par le client et les afficher

Verbe	Chemin	Action
GET	/product/3	 <p>R4.10-TP Express - Kazaburger / product/x</p> <p>GET localhost:4000/product/3</p> <p>Params Authorization Headers (6) Body Scr</p> <p>Body Cookies Headers (8) Test Results</p> <p>HTML Preview Visualize</p> <p>1 un produit par id 3</p>
GET	/product/?family=burger	 <p>R4.10-TP Express - Kazaburger / product+query</p> <p>GET localhost:4000/product/?family=burger</p> <p>Params Authorization Headers (6) Body Scripts</p> <p>Query Params</p> <p><input checked="" type="checkbox"/> Key</p> <p><input checked="" type="checkbox"/> family</p> <p>Key</p> <p>Body Cookies Headers (8) Test Results</p> <p>JSON Preview Visualize</p> <pre> 1 { 2 "family": "burger" 3 }</pre>

- En vous inspirant de la ressource /product, ajoutez les routes pour traiter la ressource /testimony

Verbe	Chemin
GET	/testimony
GET	/testimony /?user=Ellon M
GET	/testimony/3
POST	/testimony
PATCH	/testimony/3
DELETE	/testimony/6

VII. Accès aux données

Cette fois les données seront accessibles via l'api <https://kazaburge.e-mingo.net/api/sec/>

NB : Vous allez tous utiliser la même base de données, donc les données sont communes à tout le monde.

Par conséquent, évitez de faire n'importe quoi, comme tenter de supprimer les témoignages ou suggestions qui ne vous appartiennent pas.

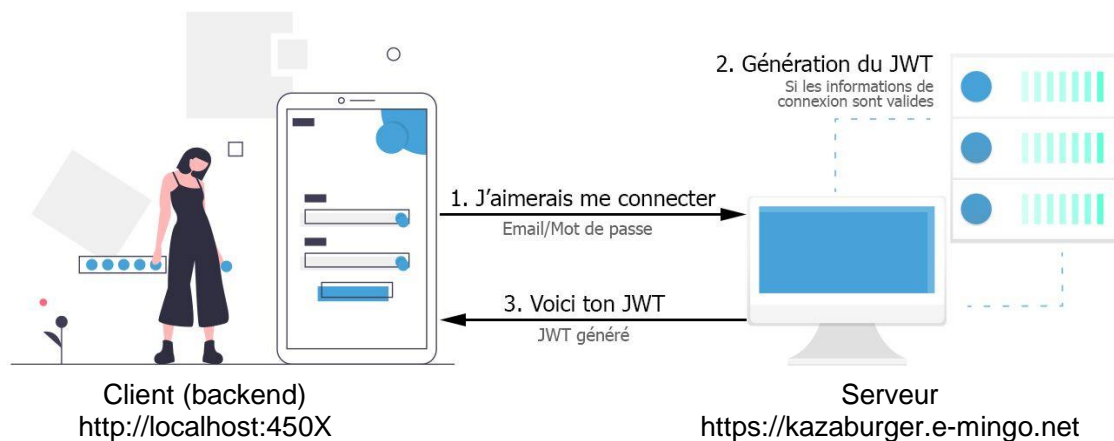
L'accès à l'api de modification des données est sécurisé.

Pour accéder aux données il faudra :

- Enregistrer un utilisateur pour votre backend sur l'api kazaburger. Bien noter ses paramètres car vous en aurez besoin pour vous authentifier
- Vous authentifier pour récupérer un jeton de connexion (valable 4h)
- Envoyer le jeton de connexion à chaque requête de modification

A. Mise en place de la sécurité entre votre api et kazaburger

Votre API Express sera considérée comme un client de l'api Kazaburger



1. Création du compte

Pour créer un compte sur kazaburger il faut faire appel à la route suivante

Actions	verbe	URL	Retour
Création d'un compte	PUT	/auth/	Objet utilisateur

Pour passer les valeurs en PUT, POST, PATCH, vous utiliserez l'onglet body de postman et l'entête de requête x-www-form-urlencoded

PUT ▼ | https://kazaburger.e-mingo.net/auth/

Params Authorization Headers (8) **Body** • Scripts Tests Settings

☐ none ☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

Key	Value
<input checked="" type="checkbox"/> login	[REDACTED]
<input checked="" type="checkbox"/> password	[REDACTED]
<input checked="" type="checkbox"/> email	david1@nowhere.is
<input checked="" type="checkbox"/> name	test 1
Key	Value

Le serveur vous renvoie les informations de l'utilisateur

Body Cookies Headers (8) Test Results ↺

{ } JSON ▼ ▶ Preview 🔗 Visualize ▼

```

1  {
2    "login": [REDACTED]
3    "email": "david1@nowhere.is",
4    "name": "test 1",
5    "role": "user",
6    "password": [REDACTED] "tXKhUPV0biL3BVH.U0kuS3FQlnvxNCABLxLnypxpRQ/13H2",
7    "_id": "67c54e83bf285a1[REDACTED]",
8    "created": "2025-03-03T06:38:59.577Z",
9    "__v": 0
10 }
```

2. Récupération d'un jeton

Actions	verbe	URL	Retour
Authentification d'un utilisateur	POST	/auth/	Token JWT

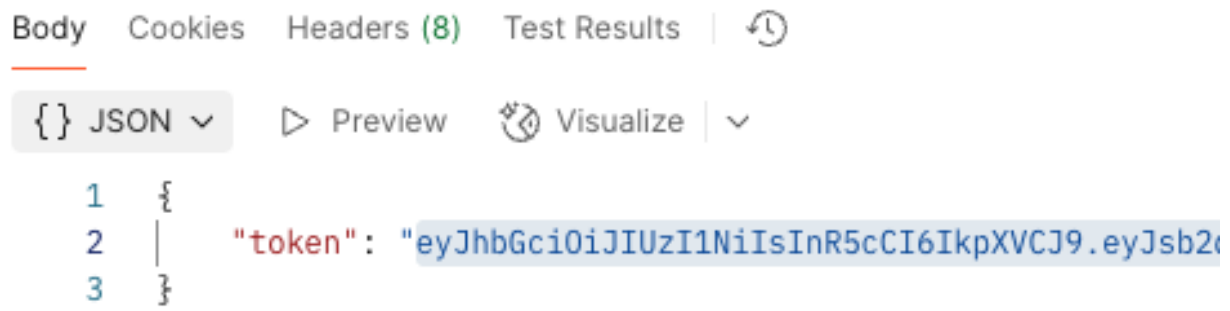
POST ▼ | https://kazaburger.e-mingo.net/auth/

Params Authorization Headers (8) **Body** • Scripts Tests Settings

☐ none ☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

Key	Value
<input checked="" type="checkbox"/> login	
<input checked="" type="checkbox"/> password	
Key	Value

Le serveur vous renvoie le token à utiliser pour accéder aux ressources protégées



3. Paramétrage du jeton dans postman

Actions	verbe	URL	Retour
Accès à une ressource protégé	*	/ressource/	Objet ressource

Pour envoyer le jeton au serveur pour chaque requête sécurisée, vous pouvez utiliser l'onglet « Authorization » de postman ou l'onglet Headers

a) Onglet Authorization

PATCH

https://kazaburger.e-mingo.net/auth/

Params

Authorization

Headers (9)

Body

Scripts

Tests

Settings

Auth Type

Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about [Bearer Token](#) authorization.

Token

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ...

b) Onglet Headers

PATCH ▼ <https://kazaburger.e-mingo.net/auth/>

Params Authorization **Headers (9)** Body ● Scripts Tests Settings

Headers 👁 8 hidden

Key	Value
<input checked="" type="checkbox"/> Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJsb2dpbi

NB : Vous pourrez récupérer le token dans votre route via l'objet req

```
console.log(req.headers["authorization"]?.split(" ")[1]);
```

4. Gestion du compte

Les opérations de modification / suppression d'un compte sont protégées.

Les informations sont à fournir dans l'onglet body

Actions	verbe	URL	Retour
Modification (mot de passe, nom)	PATCH	/auth/	Objet ressource
Suppression	DELETE		

NB : La ressource /auth permet de gérer le compte de votre backend. Un seul compte est nécessaire par étudiant.

Pour les visiteurs créés via votre frontend, nous utiliserons la ressource /user

- A la racine de votre projet, créez le fichier « .env » et mettez y les clefs suivantes ;
`KAZABURGER_API_URL='https://kazaburger.e-mingo.net/api'`
`KAZABURGER_API_TOKEN=VOTRE_TOKEN_ATTENTION_IL_CHANGE_TOUTES_LES_4H`
`API_SECRET=VOTRE_SECRET`
`PORT=4501`
- A ce stade, modifiez la valeur de votre TOKEN

NB : Pensez à mettre à jour le port dans votre client d'api (postman ou autre)

5. Récupération des variables d'environnement

- Installez le module dotenv

```
npm i dotenv
```

- Importez le module dans votre script

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

Les variables sont maintenant accessibles dans l'objet process.env. **MA_VARIABLE**

```
const port = process.env.PORT
```

B. Développement des routes

Votre API devra fournir les ressources qui seront utilisées plus tard par votre frontend. Vous allez devoir coder et tester chaque route qui sera susceptible d'être appelée par le frontend.

Les ressources pourront être séparées dans des fichiers différents si besoin.

En général, la ressource que je vous demande de coder a les mêmes nom et format que la ressource de mon api .

Maintenant que vous avez mis en place les prérequis, vous pourrez récupérer et modifier les données de l'api kazaburger **de manière sécurisée** .

1. Fetch vers l'api

Vous utiliserez la fonction [fetch](#) et ses options, pour accéder aux données de l'api.

Nous avons déjà étudié l'appel de fetch pour des requêtes type GET. Pour les requêtes de modification (PATCH, PUT, POST , ...) nous allons devoir utiliser le deuxième paramètre de la fonction

```
fetch(resource, options)
```

```
const donnees = {
  nom: 'John Doe',
  age: 30
};

fetch(url, {
  method: 'POST', // Méthode de la requête
  headers: {
    'Content-Type': 'application/json', // Type de contenu
    'Authorization': 'Bearer votre_token' // Token d'autorisation
  },
  body: JSON.stringify(donnees) // Transforme les données en JSON
})
.then(response => {
  if (!response.ok) {
    throw new Error('Là = y a un problème :');
  }
  return response.json(); // Transforme la réponse en JSON
})....
```

2. Les produits

Les produits sont fournis depuis la ressource « /product ».

Cette ressource n'est pas sécurisée, ne nécessite pas de token

Actions	verbe	URL
Tous les produits	GET	/product/
Produits par famille		/product/?family=XXX
Produits par nom		/product/?title=XXX
produit par Id		/product/999

Créez les routes pour fournir les produits en fonction des différents critères, depuis votre api. Vous aurez peut-être besoin de fonctions intermédiaires pour contrôler les entrées et adapter la route . A vous de jouer

Structure de product

```

_id: ObjectId('6788a121667f58558151f6e9')
id: "29544"
family: "pizza"
title: "Pâte à pizza fine "
rating: "4.8"
▼ pictures: Array (3)
  0: "https://assets.afcdn.com/recipe/20171206/75873_origin.jpg"
  1: "https://assets.afcdn.com/recipe/20180712/80809_origin.jpg"
  2: "https://assets.afcdn.com/recipe/20130823/42626_origin.jpg"
tags: Array (4)
  0: "De saison"
  1: "Plat principal"
  2: "Vegetarian"
  3: "Pizza"
price: 14.074345630020002

```

NB : Dans les bases de données mongodb, l'identifiant unique de chaque entité est accessible via l'attribut « id » **c'est cet attribut qu'il faudra utiliser pour enregistrer les produits**

NB : xxx correspond à la valeur récupérée depuis les paramètres de l'url

NB : 999 correspond à l'id récupéré depuis les paramètres de l'url

3. Les témoignages

Ils sont disponibles depuis la ressource « /testimony »

NB : Sans le token vous n'aurez accès qu'en lecture seule aux témoignages de démo.

Actions	verbe	URL
Tous les témoignages	GET	/testimony/
Témoignages par produit		/testimony/?product=999
Témoignages par Auteur		/testimony/?user=XXX
Témoignage par Id		/testimony/999
Ajout d'un témoignage	POST	/testimony/
Modification d'un témoignage à partir de son id	PATCH	/testimony/999
Suppression d'un témoignage à partir de son id	DELETE	/testimony/999

Structure de testimony

```
_id: ObjectId('67a7e626261c28e045830a43')
product: ObjectId('6788a121667f58558151f844')
rating: 5
review: "Meilleure pizza que j'ai mangée depuis longtemps ! La croûte était par..."
user: "Mathieu V."
```

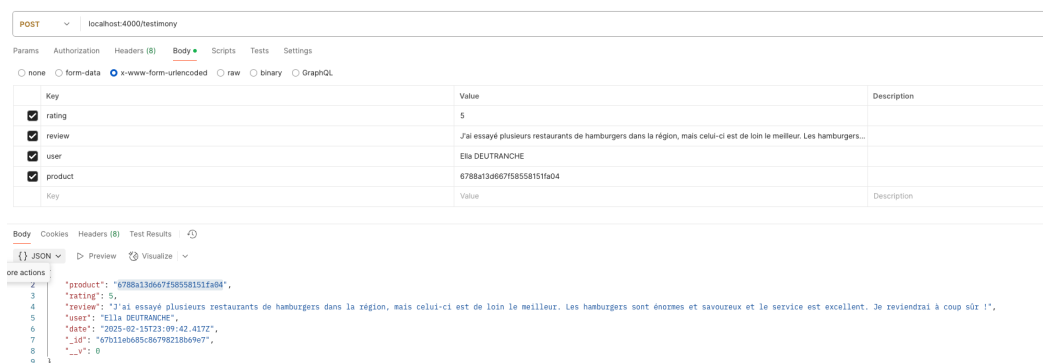
NB : Dans les bases de données mongodb, l'identifiant unique de chaque entité est accessible via l'attribut « _id »

NB : xxx correspond à la valeur récupérée depuis les paramètres de l'url

NB : 999 correspond à l'id récupéré depuis les paramètres de l'url

NB : * correspondant à un dictionnaire comprenant les attributs nécessaires, certains sont obligatoires (user,product, review, rating)

NB : Pensez à contrôler les actions et leurs résultats via postman,



4. Les visiteurs

Pour pouvoir liker les produits et laisser des témoignages les utilisateurs auront besoin. De s'identifier. Pour cela nous allons mettre en place une gestion des comptes.

Nb : Dans la première partie nous avons authentifié votre backend sur kazaburger avec la ressource /auth. Maintenant nous utiliserons la ressource /user pour éviter la confusion

NB : En situation professionnelle il faudrait mettre en place des systèmes plus robuste de sécurisation, à minima jwt + ssl, mais ce n'est pas l'objet de ce module.

Ici vos 2 applications sont en local sur le même serveur, l'utilisateur se connectera avec un identifiant et un mot de passe.

Nous allons juste faire un hash du mot de passe pour le transmettre à la base en utilisant le module bcrypt

- Installez le module

```
npm i bcrypt
```

- Importez le module dans votre router

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

- Consultez la doc pour [hasher](#) et vérifier le mot de passe
- Inutile de vous dire quoi faire pour que votre api réponde aux requêtes suivantes.

Actions	verbe	URL
Liste des utilisateurs	GET	/user/
Utilisateur par login		/user/?login=xxx
Utilisateur par mail		/user/?email=xxx
Utilisateur par Id		/user/999
Ajout d'un utilisateur	PUT	/user
Modification d'un utilisateur à partir de son id	PATCH	/user/999
Suppression d'un utilisateur à partir de son id	DELETE	/user/999

Structure de user

```
{
  "_id": "67cb3290857202d94bd8235f",
  "login": "david-visit",
  "email": "david1@nowhere.isa",
  "name": "david1",
  "password": "$2b$10$MgUCohLPYHXUVxcvWaSIVe4XbxKlX3fYRFgCoGl1B7MIiVFgfo0hG",
  "..."
}
```

NB : Les champs email, login et password sont obligatoires pour la création

Les champs email et le login doivent être uniques

L'utilisateur ne peut modifier que name ou password

- Créez la route et la fonction nécessaires à l'authentification de l'utilisateur avec login/mot de passe