self-service-machine-api - STEP 8

Maintenant nous allons nous intéresser à la gestion des erreurs.

Selon l'erreur que le consommateur va rencontrer, on veut que le statut HTTP retourné correspond à l'erreur en question.

Les statuts HTTP

Voici la liste complète des status HTTP : https://developer.mozilla.org/fr/docs/Web/HTTP/Status

A noter le statut 418, mon préféré : `I'm a teapot` (je suis une théière) 😉

Gestion du statut HTTP 404 - Création de notre 1er middleware

Si un consommateur de notre API REST tente d'utiliser une URL non définie, nous devons nous assurer que :

- le statut d'erreur est bien `404`
- un message d'erreur lui indique clairement que cette URL ne correspond à aucune ressource pour notre API REST.

Pour ce faire, nous allons créer notre 1er middleware !

En effet, nous avons déjà utiliser des middlewares mais nous n'en avons pas encore créé un.

Allons-y !

Dans notre fichier `app.mjs` après la définition de nos routes, vous pouvez ajouter le code suivant :

```
...
// Si aucune route ne correspondant à l'URL demandée par le consommateur
app.use(({ res }) => {
  const message =
    "Impossible de trouver la ressource demandée ! Vous pouvez essayer une autre URL.";
  res.status(404).json(message);
});
...
```

`app.use` est utilisée pour ajouter des middlewares à notre application Express.

Les middlewares sont des fonctions qui ont accès aux objets de requête (`req`), de réponse (`res`). Le `app.use` est utilisé ici sans chemin spécifique, donc ce middleware sera exécuté pour toutes les requêtes.

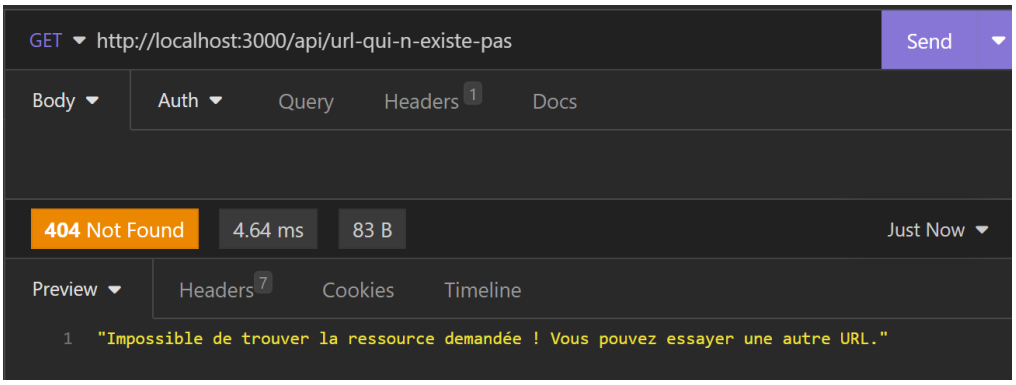
`({ res }) => { ... }` est une fonction fléchée (Arrow function) en JavaScript, et elle utilise la déstructuration pour extraire `res` de l'objet argument. Cela signifie que cette fonction prend un objet en entrée qui a au moins une propriété `res` , qui est l'objet de réponse Express

`res.status(404)` définit le code d'état de la réponse HTTP à 404. Le code 404 est standard pour "Not Found" (Non trouvé), ce qui signifie que le serveur n'a pas trouvé ce qui était demandé.

Ce middleware est exécuté lorsqu'aucune autre route n'a été trouvée correspondant à la requête entrante. C'est pourquoi ce code placé à la fin de toutes les autres définitions de route. Il envoie une réponse avec le statut 404 et un message JSON indiquant à l'utilisateur que la ressource demandée n'a pas été trouvée.

Test du middleware pour la gestion du statut HTTP 404 avec insomnia

Maintenant si avec insomnia nous tentons accéder à une URL qui ne correspond à rien pour notre API REST, nous pouvons voir que le statut et le message sont corrects.



HTTP 500 pour la route GET /api/products

Pour la route GET /api/products, nous devons gérer le cas où un problème surviendrait lors de la promesse. Dans ce cas, on doit retourner un statut HTTP 500 et indiqué un message d'erreur adéquat.

```
...
productsRouter.get("/", (req, res) => {
  Product.findAll()
    .then((products) => {
      const message = "La liste des produits a bien été récupérée.";
      res.json(success(message, products));
    })
    .catch((error) => {
      const message =
        "La liste des produits n'a pas pu être récupérée. Merci de réessayer dans quelques instants.";
      res.status(500).json({ message, data: error });
    });
});
...
```

Maintenant si nous faisons volontairement une erreur (par exemple ajoutons un s à products => productss) dans le code puis avec insomnia nous tentons accéder à GET /api/products/ nous pouvons voir que le statut 500 et le message d'erreur.

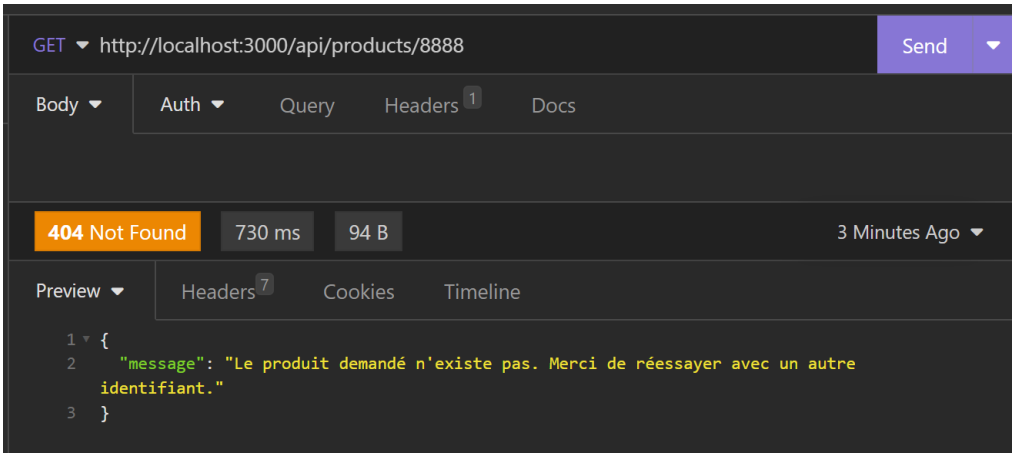
HTTP 500 et HTTP 404 pour la route GET /api/products/:id

Pour la route GET /api/products/:id, nous devons gérer les cas :

- où un problème surviendrait lors de la promesse comme précédemment.
- mais également le cas où le consommateur a demandé un produit avec un id qui n'existe pas.

```
...

productsRouter.get("/:id", (req, res) => {
  Product.findByPk(req.params.id)
    .then((product) => {
      if (product === null) {
        const message =
          "Le produit demandé n'existe pas. Merci de réessayer avec un autre identifiant.";
        // A noter ici le return pour interrompre l'exécution du code
        return res.status(404).json({ message });
      }
      const message = `Le produit dont l'id vaut ${product.id} a bien été récupéré.`;
      res.json(success(message, product));
    })
    .catch((error) => {
      const message =
        "Le produit n'a pas pu être récupéré. Merci de réessayer dans quelques instants.";
      res.status(500).json({ message, data: error });
    });
});
...
```



HTTP 500 pour la route POST /api/products/

Pour la route POST /api/products/, nous devons gérer le cas où un problème surviendrait lors de la promesse comme précédemment.

```
...
productsRouter.post("/", (req, res) => {
  Product.create(req.body)
    .then((createdProduct) => {
      // Définir un message pour le consommateur de l'API REST
      const message = `Le produit ${createdProduct.name} a bien été créé !`;

      // Retourner la réponse HTTP en json avec le msg et le produit créé
      res.json(success(message, createdProduct));
    })
```

```
    .catch((error) => {
      const message =
        "Le produit n'a pas pu être ajouté. Merci de réessayer dans quelques instants.";
      res.status(500).json({ message, data: error });
    });
});
...
```

HTTP 500 et HTTP 404 pour la route PUT /api/products/:id

Comme dans le cas de l'update nous utilisons :

- `Product.update()`
- `Product.findByPk()`

nous avons 2 cas de HTTP 500 à gérer.

De même, comme nous utilisons `findByPk()` le consommateur a pu renseigné un id qui n'existe pas. Nous avons donc un HTTP 404 à gérer.

```
...
productsRouter.put("/:id", (req, res) => {
  const productId = req.params.id;
  Product.update(req.body, { where: { id: productId } })
    .then((_) => {
      Product.findByPk(productId)
        .then((updatedProduct) => {
          if (updatedProduct === null) {
            const message =
              "Le produit demandé n'existe pas. Merci de réessayer avec un autre identifiant.";
            // A noter ici le return pour interrompre l'exécution du code
            return res.status(404).json({ message });
          }
          // Définir un message pour l'utilisateur de l'API REST
          const message = `Le produit ${updatedProduct.name} dont l'id vaut ${updatedProduct.id} a été mis à jour avec s

          // Retourner la réponse HTTP en json avec le msg et le produit créé
          res.json(success(message, updatedProduct));
        })
        .catch((error) => {
          const message =
            "Le produit n'a pas pu être mis à jour. Merci de réessayer dans quelques instants.";
          res.status(500).json({ message, data: error });
        });
    })
    .catch((error) => {
      const message =
        "Le produit n'a pas pu être mis à jour. Merci de réessayer dans quelques instants.";
      res.status(500).json({ message, data: error });
    });
});
...
```

HTTP 500 pour la route DELETE /api/products/:id

Nous laissons volontairement de côté cette route pour l'instant.

Car nous n'en avons pas encore terminé avec la gestion des statuts HTTP.

Nous allons continuer et améliorer cela dans l'étape 9 (step9).

Passons à l'étape n°9