

<a href="#">Syllabus</a>	<a href="#">Doc E2E tests</a>	<a href="#">Planning et Fonctionnement</a>	<a href="#">Démarche centrée utilisateurs</a>
<a href="#">Modalités de rendus et évaluations</a>	<a href="#">Sujet</a>	<a href="#">Des exemples de Vidéos</a>	<a href="#">Cours</a>
<a href="#">Back</a>	<a href="#">Technologie Angular NodeJs</a>		
<a href="#">Index</a>	<a href="#">Informations Techniques</a>		


## Architecture du projet

### Fonctionnement

Pour gérer un type de données, il faut créer un modèle correspondant à ce type, un manager contenant des fonctions pour créer/modifier/supprimer les éléments de ce type, et des routes pour appeler les fonctions du manager. Pour cela :

- Dans le dossier models :
  - Créer un modèle contenant la description des éléments du type choisi
  - L'ajouter dans l'index.js du même dossier
- Dans le dossier api :
  - Créer un dossier correspondant au modèle, à l'intérieur duquel il faudra 2 fichiers : manager.js contenant les fonctions d'ajout/modification/suppression, et un fichier index.js choisissant selon la route voulue quelle fonction du manager appeler.
  - Ajouter au router le dossier nouvellement créé. Pour cela, il faut modifier le fichier app/api/index.js. Dans l'exemple ci-dessous, on peut voir qu'on importe les routeurs QuizzesRouter et UserRouter.

[ps6-correction-td1-td2-v2 / back-end / app / api / index.js](#) 

 NablaT Correction back-end & front-end

Code

Blame

10 lines (8 loc) · 308 Bytes

```
1  const { Router } = require('express')
2  const QuizzesRouter = require('./quizzes')
3  const UserRouter = require('./users')
4
5  const router = new Router()
6  router.get('/status', (req, res) => res.status(200).json('ok'))
7  router.use('/quizzes', QuizzesRouter)
8  router.use('/users', UserRouter)
9
10 module.exports = router
```

### Lancement

#### index.js

Le fichier index.js est le point d'entrée de l'application. Il fait appel au fichier **build-server.js** qui va construire le serveur de l'application. Ensuite, il utilise le module **logger.js** pour afficher un message de confirmation indiquant que le serveur est prêt à écouter les requêtes sur le port spécifié.

#### build-server.js

Le fichier **build-server.js** construit le serveur de l'application en utilisant le framework express. Il utilise les modules **cors**, **morgan**, **body-parser**, et le fichier **api.js** pour créer les routes de l'API.

La fonction exportée **buildServer()** crée l'application express et définit les paramètres de base tels que l'activation de CORS et la configuration des logs. Ensuite, elle définit la route de l'API, spécifiant que toutes les requêtes commençant par **/api** doivent être gérées par le fichier **api.js**. Si la requête ne correspond à aucune route connue, elle répond avec une erreur **404**.

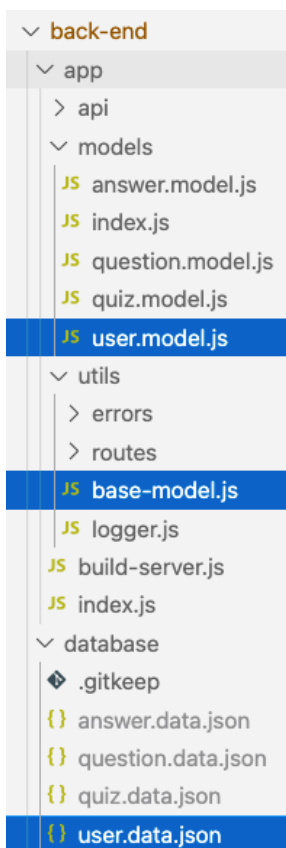
Enfin, le serveur est démarré sur le port spécifié ou sur le port **9428** par défaut. Lorsque le serveur est prêt à écouter les requêtes, la fonction de rappel cb est appelée, qui peut être utilisée pour effectuer des actions supplémentaires, telles que l'affichage d'un message de confirmation.

## How-tos

---

### Comment créer un modèle ?

Plusieurs fichiers interviennent lors de la création d'un modèle :



La classe BaseModel (app/utils/base-model.js) fournit une base pour stocker des données sous forme de fichiers JSON dans un dossier appelé "database". Cette classe permet de créer, lire, mettre à jour et supprimer des éléments dans la base de données.

Pour utiliser BaseModel, il faut créer un modèle personnalisé dans `app/models` en utilisant cette classe. La création d'un modèle nécessite deux éléments : un nom et un schéma. Le nom sera utilisé pour définir le nom du fichier JSON contenant les données stockées, et le schéma décrira les propriétés et les types de données autorisés pour chaque élément du modèle.

```
const Joi = require('joi')  
const BaseModel = require('../utils/base-model.js')
```

```
module.exports = new BaseModel('User', {
  firstName: Joi.string().required(),
  lastName: Joi.string().required(),
})
```

Le nom du modèle doit être passé comme premier argument pour le constructeur de BaseModel. Le schéma doit être fourni en tant que deuxième argument, sous la forme d'un objet Joi.

Joi permet de définir et de valider des schémas de données avec des règles de validation pour s'assurer que les données ont le bon format avant d'être utilisées.

Une fois que le modèle est créé, il est possible d'utiliser les méthodes de BaseModel pour manipuler les données stockées (voir documentation "Comment créer une nouvelle route"). Les méthodes disponibles sont get(), getByld(id), create(obj), update(id, obj) et delete(id).

- get() renvoie un tableau contenant tous les éléments du modèle stockés dans le fichier JSON.
- getByld(id) renvoie l'élément correspondant à l'ID passé en argument.
- create(obj) crée un nouvel élément en utilisant les propriétés de l'objet fourni et en les validant par rapport au schéma.
- update(id, obj) met à jour l'élément correspondant à l'ID fourni en utilisant les propriétés de l'objet fourni et en les validant par rapport au schéma.
- delete(id) supprime l'élément correspondant à l'ID fourni.

Ces fonctions modifient directement le dossier database qui contient l'ensemble des fichiers qui stockent vos données

```
[
  {
    "firstName": "Marie",
    "lastName": "Doe",
    "id": 1677770643390
  },
  {
    "firstName": "John",
    "lastName": "Doe",
    "id": 1677770643332
  }
]
```

## Comment créer une nouvelle route ?

Pour créer de nouvelles routes pour un nouveau modèle dans notre API Node.js :

```
const { Router } = require('express')

const { User } = require('../models')
const manageAllErrors = require('../utils/routes/error-management')

const router = new Router()

router.get('/', (req, res) => {
```

```

    try {
      res.status(200).json(User.get())
    } catch (err) {
      manageAllErrors(res, err)
    }
  })

router.get('/:userId', (req, res) => {
  try {
    res.status(200).json(User.getById(req.params.userId))
  } catch (err) {
    manageAllErrors(res, err)
  }
})

module.exports = router

```

- Créer un nouveau fichier dans le dossier approprié (par exemple, api/nomDuModele/index/js) pour définir les nouvelles routes pour ce modèle.
- Déclarer la route dans api/index.js :

```

const { Router } = require('express')
const QuizzesRouter = require('./quizzes')
const UserRouter = require('./users')

const router = new Router()
router.get('/status', (req, res) => res.status(200).json('ok'))
router.use('/quizzes', QuizzesRouter)
router.use('/users', UserRouter) <----- déclaration de la route users

module.exports = router

```

- Retourner dans votre nouveau fichier et importer le Router d'Express dans le fichier nouvellement créé (*const {Router} = require('express')*)
- Définir les routes nécessaires en utilisant les méthodes de Router (get, post, put, delete)

- Pour chacune des routes :

```

router.get('/:myModelParam', (req, res) => {
  try {
    res.status(200).json(MyModel.getById(req.params.myModelParam))
  } catch (err) {
    manageAllErrors(res, err)
  }
})

```

- Définir la méthode HTTP GET/POST/PUT/DELETE et la route, ici `('/:myModel')` utilisé pour récupérer un modèle spécifique en utilisant son ID. Le paramètre de la route `:myModelParam` doit être remplacé par un nom compréhensible.
- Dans le bloc `try`, on appelle la méthode `getById` du modèle correspondant en passant l'ID récupéré depuis la route (`req.params.myModelParam`).

- Si aucune erreur n'est levée (voir documentation "Gestion des erreurs du BaseModel"), la réponse est renvoyée au client avec le code de statut 200 et le modèle en question sous forme de JSON.
- Si une erreur est levée, elle est récupérée dans le bloc catch et transmise à la fonction `manageAllErrors` qui se charge de renvoyer une réponse d'erreur au client. Il est important d'utiliser un bloc try/catch pour gérer les erreurs potentielles lors de l'accès à la base de données ou lors de l'exécution de la logique métier.
- Exporter le Router nouvellement créé (`module.exports = router`)
- Il est également important de créer un fichier `manager.js` dans le dossier correspondant au model pour y stocker la logique métier de notre API. Cela permet de séparer la logique métier de la définition des routes, ce qui facilite la maintenance et l'évolutivité de l'API

Dans l'exemple de notre modèle quizzes, le fichier `manager.js` contient les fonctions `buildQuizz` et `buildQuizzes` qui agrègent les données des questions et des réponses pour chaque quiz. Ces fonctions sont utilisées dans les routes définies dans le fichier `index.js` pour renvoyer les données du quiz et des questions associées aux clients de l'API.

En résumé, pour créer de nouvelles routes pour un nouveau model, il suffit de créer un nouveau fichier `index.js` pour les routes et un fichier `manager.js` pour la logique métier dans le dossier correspondant. Ensuite, définir les routes nécessaires en utilisant le Router d'Express et exporter le Router. Enfin, utiliser les fonctions du fichier `manager.js` pour agréger les données et les renvoyer dans les réponses des routes définies dans le fichier `index.js`.

## Gestion des erreurs de BaseModel

Le BaseModel est une classe de base qui sert à implémenter les fonctionnalités communes à tous les modèles. Elle peut lever deux types d'erreurs :

- `NotFoundError` : cette erreur est levée lorsque la ressource recherchée n'a pas été trouvée dans la base de données. Elle est utilisée dans la méthode `getById` pour indiquer que l'objet demandé n'a pas été trouvé.
- `InvalidBodyError` : cette erreur est levée lorsque le corps de la requête ne contient pas les propriétés requises pour créer un nouvel objet dans la base de données. Elle est utilisée dans la méthode `create` pour indiquer que la requête est invalide et que la création de l'objet a échoué.

Ces erreurs sont gérées par le middleware `error-management.js` qui renvoie une réponse JSON contenant un message d'erreur approprié et un code d'erreur HTTP correspondant.

## Remplacement de console.log

Le fichier `logger.js` est un module utilitaire qui remplace la fonction de console `console.log` et ajoute un horodatage à chaque message affiché dans la console. Cela permet de mieux suivre les messages de log et de savoir quand ils ont été générés. Le module exporte simplement la fonction `console`, qui peut être utilisée comme d'habitude pour écrire des messages de log dans la console.

✉ [Contacter l'assistance du site](#) 

---

Connecté sous le nom « [duong Thi Thanh Tu](#) » ([Déconnexion](#))

[Résumé de conservation de données](#)

[Obtenir l'app mobile](#)

---

Fourni par [Moodle](#)