JavaScript côté serveur

Support APP3

1. Introduction

HTTP définit un ensemble de méthodes de requête qui indiquent l'action que l'on souhaite réaliser sur la ressource indiquée. Ces méthodes sont souvent appelées verbes HTTP (HTTP verbs).

Dans le protocole HTTP, une méthode est une commande spécifiant un type de requête, c'est-à-dire qu'elle demande au serveur d'effectuer une action. En général l'action concerne une ressource identifiée par l'URL qui suit le nom de la méthode.

- **GET :** La méthode GET demande une représentation de la ressource spécifiée. Les requêtes GET doivent uniquement être utilisées afin de récupérer des données.
- **POST :** La méthode POST est utilisée pour envoyer une entité vers la ressource indiquée, cela entraîne généralement un changement d'état.
- **PUT :** La méthode PUT remplace toutes les représentations actuelles de la ressource visée par le contenu de la requête.
- **PATCH :** La méthode PATCH est utilisée pour appliquer des modifications partielles à une ressource.
- **DELETE :** La méthode DELETE supprime la ressource indiquée.

2. CRUD

Essayons de créer de nouvelles routes (en suppriment les anciennes routes), en mettant à jour le code précédant, permettant de manipuler l'entité Game comme suit :

HTTP verb	Route	Description	CRUD
GET	/game	Récupération de la liste des jeux	READ
GET	/game/:name	Récupérer les détails du jeu passé en paramètre	READ
POST	/game	Ajout d'un nouveau jeu	CREATE
PUT	/game/:name	Remplacer le jeu passé en paramètre	UPDATE
PATCH	/game/:name	Modifier partiellement le jeu passé en paramètre	UPDATE
DELETE	/game/:name	Supprimer le jeu passé en paramètre	DELETE

```
class Game {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
}

const games = [new Game("dmc5", 2019), new Game("re8", 2021), new Game("nfs", 2019)];

app.get('/game', (req, res) => {
  res.status(200).json(games);
})

app.get('/game/:name', (req, res) => {
  res.status(200).json(games.find(val => val.name === req.params.name));
})

app.post('/game', (req, res) => {
  const game = new Game(req.body.name, req.body.year);
  games.push(game);
  res.status(201).json({ message: "Created !", entity: game});
})

app.put('/game/:name', (req, res) => {
  res.status(200).json({ message: "Updated !", name: req.params.name});
})

app.patch('/game/:name', (req, res) => {
  res.status(200).json({ message: "Updated !", name: req.params.name});
})

app.delete('/game/:name', (req, res) => {
  res.status(200).json({ message: "Deleted !", name: req.params.name});
})
```

Maintenant, essayez de les tester avec Postman. Pour la requête de POST, essayez avec l'exemple suivant :

```
{
    "name": "gta5",
    "year": 2013
}
```

Qu'est-ce que vous remarquez ?

Pourquoi avons-nous eu cette erreur ?

```
TypeError: Cannot read properties of undefined (reading 'name')
```

3. Envoyez du JSON

Comme nous l'avons vu précédemment (APP2), il est assez facile de renvoyer du JSON à l'aide de .json(), mais il est un peu plus délicat d'accepter les entrées (données) de l'utilisateur lorsqu'elles sont envoyées sous forme de JSON ou d'autres valeurs.

Pour ce faire, nous devons inclure un autre module qui analyse le "corps" de la réponse en tant que "application/json". Le module en question est utilisé via « express.json() ».

Ce type de module est appelé middleware car il se situe au milieu du cycle requête/réponse et peut modifier la requête ou la réponse. Nous en verrons plus à ce sujet prochainement dans le cours.

Pour inclure le middleware, nous utilisons la fonction « app.use() ».

```
class Game {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
}

const games = [new Game("dmc5", 2019), new Game("re8", 2021), new Game("nfs", 2019)];

app.use(express.json()); // Pour analyser (parsing) les requetes application/json

app.get('/game', (req, res) => {
  res.status(200).json(games);
})

app.get('/game/:name', (req, res) => {
  res.status(200).json(games.find(val => val.name === req.params.name));
})

app.post('/game', (req, res) => {
  const game = new Game(req.body.name, req.body.year);
  games.push(game);
  res.status(201).json({ message: "Created !", entity: game});
})
```

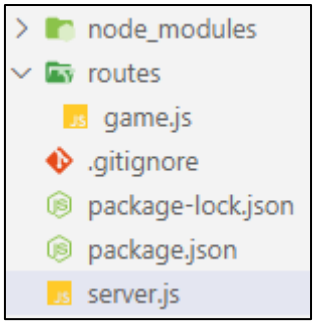
Lorsque les données sont envoyées au serveur, nous pouvons désormais collecter ces données à l'intérieur de req.body.

4. Express.js Router

Jusqu'à présent, nous avons vu comment créer des routes en Express.js. Au fur et à mesure que nous commençons à créer de plus en plus de routes, notre server.js peut rapidement devenir très désordonné.

Afin d'organiser cela, nous allons déplacer nos routes vers un fichier dans un dossier appelé routes. Les fichiers de ce dossier contiendront tous nos routes et nous les exporterons à l'aide de Express.js Router.

La structure de notre projet va devenir ainsi :



Après avoir déplacé les routes sur un fichier à part en utilisant Express.js Router, le fichier routes/game.js contient le code suivant :

```
import express from 'express';

/**
 * Router est un objet de base sur le module express.
 * Cet objet a des méthodes similaires (.get, .post, .patch, .delete)
 * à l'objet app de type "express()" que nous avons utilisé précédemment.
 */
const router = express.Router();

class Game {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
}

const games = [new Game("dmc5", 2019), new Game("re8", 2021), new Game("nfs", 2019)];

// au lieu de app.get...
router.get('/game', (req, res) => {
  res.status(200).json(games);
})

// au lieu de app.get...
router.get('/game/:name', (req, res) => {
  res.status(200).json(games.find(val => val.name === req.params.name));
})

// au lieu de app.post...
router.post('/game', (req, res) => {
  const game = new Game(req.body.name, req.body.year);
  games.push(game);
  res.status(201).json({ message: "Created !", entity: game});
})
```

```
// au lieu de app.put...
router.put('/game/:name', (req, res) => {
  res.status(200).json({ message: "Updated !", name: req.params.name});
})

// au lieu de app.patch...
router.patch('/game/:name', (req, res) => {
  res.status(200).json({ message: "Updated !", name: req.params.name});
})

// au lieu de app.delete...
router.delete('/game/:name', (req, res) => {
  res.status(200).json({ message: "Deleted !", name: req.params.name});
})

/**
 * Maintenant que nous avons créé toutes ces routes,
 * exportons ce module pour l'utiliser dans server.js
 * puisque c'est lui notre entrée principale "main".
 */
export default router;
```

Et en mettent à jour le fichier server.js, nous obtenons :

```
import express from 'express';

const app = express();

const port = process.env.PORT || 9090;

import gameRoutes from './routes/game.js'; // importer le router du fichier routes/game.js

app.use(express.json()); // Pour analyser (parsing) les requetes application/json

app.use(gameRoutes); // Utiliser les routes créés

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

a. Une syntaxe de Router plus déclarative

Nous pourrions également écrire ces routes en utilisant une syntaxe alternative, plus déclarative, pour le routage express.

Lorsque nous disons "plus déclaratif" dans ce cas, nous disons essentiellement que le code utilise des méthodes (c'est-à-dire « .route('/game') ») pour être plus lisible et implicite plutôt que redondant et explicite.

Ainsi, nous mettrons à jour le fichier routes/game.js comme suit :

```
const games = [new Game("dmc5", 2019), new Game("re8", 2021), new Game("nfs", 2019)];

// Déclarer d'abord la route, puis toutes les méthodes dessus
router
  .route('/game')
  .get((req, res) => {
    res.status(200).json(games);
  })
  .post((req, res) => {
    const game = new Game(req.body.name, req.body.year);
    games.push(game);
    res.status(201).json({ message: "Created !", entity: game});
  });

router
  .route('/game/:name')
  .get((req, res) => {
    res.status(200).json(games.find(val => val.name === req.params.name));
  })
  .put((req, res) => {
    res.status(200).json({ message: "Updated !", name: req.params.name});
  })
  .patch((req, res) => {
    res.status(200).json({ message: "Updated !", name: req.params.name});
  })
  .delete((req, res) => {
    res.status(200).json({ message: "Deleted !", name: req.params.name});
  })

/**
 * Maintenant que nous avons créé toutes ces routes,
 * exportons ce module pour l'utiliser dans server.js
 * puisque c'est lui notre entrée principale "main".
 */
export default router;
```

b. Syntaxe de Router déclarative avec un préfixe

Encore mieux, nous pouvons préfixer toutes nos routes pour commencer par /game.

Tout ce que nous avons à faire est d'apporter une légère modification à server.js :

```
// préfixe chaque route ici avec /game
app.use('/game', gameRoutes); // Utiliser les routes créés
```

Voici à quoi cela ressemble dans notre routes/game.js :

```
// Déclarer d'abord la route, puis toutes les méthodes dessus (préfixe spécifié dans server.js)
router
  .route('/')
  .get((req, res) => {
    res.status(200).json(games);
  })
  .post((req, res) => {
    const game = new Game(req.body.name, req.body.year);
    games.push(game);
    res.status(201).json({ message: "Created !", entity: game});
  });

router
  .route('/:name')
  .get((req, res) => {
    res.status(200).json(games.find(val => val.name === req.params.name));
  })
  .put((req, res) => {
    res.status(200).json({ message: "Updated !", name: req.params.name});
  })
  .patch((req, res) => {
    res.status(200).json({ message: "Updated !", name: req.params.name});
  })
  .delete((req, res) => {
    res.status(200).json({ message: "Deleted !", name: req.params.name});
  })
```

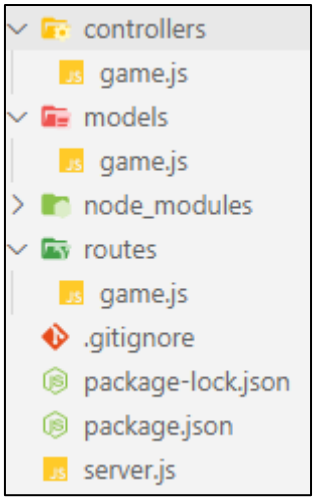
5. Organisation du service web

Jusqu'à présent, encore une fois, nous avons vu comment créer des routes en Express.js avec leurs implémentations. Au fur et à mesure que le nombre des routes augmente, notre routes/game.js peut rapidement devenir illisibles.

Afin d'organiser cela, nous allons déplacer les implémentations des routes vers un fichier dans un dossier appelé controllers. Les fichiers de ce dossier contiendront tous nos contrôleurs (implémentations des routes) et nous les exporterons pour pouvoir les utiliser dans nos fichiers des routes. Le code devient plus "auto-documenté" pour ainsi dire, parce que nous sommes plus préoccupés par ce qu'il nous dit (les définitions des routes) plutôt que par la façon dont il le fait.

Aussi, nous déplacerons les entités (ici Game) vers un fichier dans un dossier appelé models. Les fichiers de ce dossier contiendront tous nos entités et nous les exporterons pour pouvoir les utiliser dans nos fichiers de contrôleurs.

Ainsi, la structure de notre projet va devenir :



Après les modifications, nous aurons les codes suivants :

```
import express from 'express';

import { getAll, addOnce, getOnce,
  putOnce, patchOnce, deleteOnce } from '../controllers/game.js';

/**
 * Router est un objet de base sur le module express.
 * Cet objet a des méthodes similaires (.get, .post, .patch, .delete)
 * à l'objet app de type "express()" que nous avons utilisé précédemment.
 */
const router = express.Router();

// Déclarer d'abord la route, puis toutes les méthodes dessus (préfixe spécifié dans server.js)
router
  .route('/')
  .get(getAll)
  .post(addOnce);

router
  .route('/:name')
  .get(getOnce)
  .put(putOnce)
  .patch(patchOnce)
  .delete(deleteOnce);

/**
 * Maintenant que nous avons créé toutes ces routes,
 * exportons ce module pour l'utiliser dans server.js
 * puisque c'est lui notre entrée principale "main".
 */
export default router;
```

Figure 1. routes/game.js

```
import Game from '../models/game.js';

const games = [new Game("dmc5", 2019), new Game("re8", 2021), new Game("nfs", 2019)];

export function getAll(req, res) {
    res.status(200).json(games);
}

export function addOnce(req, res) {
    const game = new Game(req.body.name, req.body.year);
    games.push(game);
    res.status(201).json({ message: "Created !", entity: game});
}

export function getOnce(req, res) {
    res.status(200).json(games.find(val => val.name === req.params.name));
}

export function putOnce(req, res) {
    res.status(200).json({ message: "Updated !", name: req.params.name});
}

export function patchOnce(req, res) {
    res.status(200).json({ message: "Updated !", name: req.params.name});
}

export function deleteOnce(req, res) {
    res.status(200).json({ message: "Deleted !", name: req.params.name});
}
```

Figure 2. controllers/game.js

```
export default class Game {
    constructor(name, year) {
        this.name = name;
        this.year = year;
    }
}
```

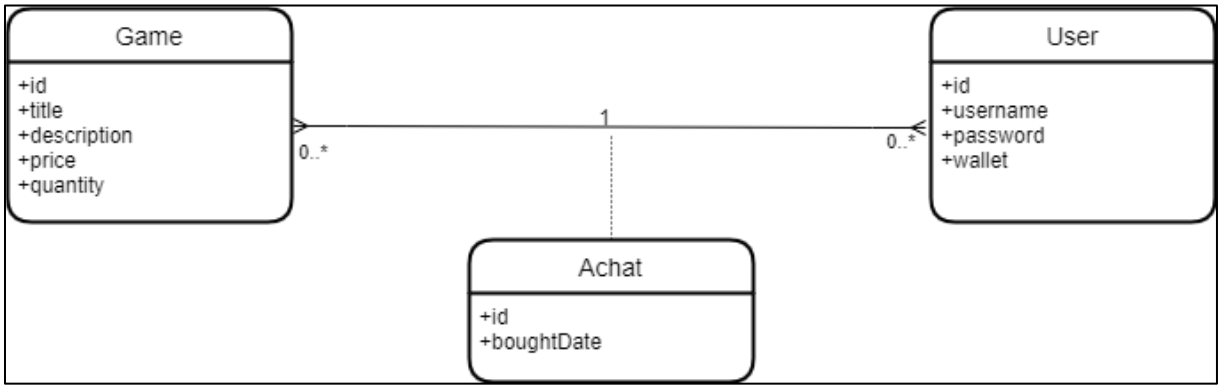
Figure 3. models/game.js

6. Exercice

On se propose de réaliser un service web qui permet à un utilisateur de créer un compte, de s'authentifier et de modifier son profil sur une plateforme de vente de jeux vidéo. Il peut aussi consulter la liste des jeux disponibles, leurs détails et les acheter.

Ce service web permet à un modérateur de gérer les jeux vidéo.

En vous référent au diagramme de classes suivant :



Créez un service web avec les Endpoint décrites ci-dessous :

Endpoint	Résultat
Créer un compte pour l'utilisateur	Retourner un JSON avec les données de l'utilisateur sans l'id
S'authentifier	Retourner un JSON avec les données de l'utilisateur
Modification de profil	Retourner un JSON avec les nouvelles données de l'utilisateur
Afficher la liste des jeux	Retourner un JSON avec l'id, le nom et le prix pour chaque jeu
Afficher les détails du jeu	Retourner un JSON avec les données du jeu
Ajouter un nouveau jeu	Retourner un JSON avec les données du jeu sans l'id
Modifier un jeu	Retourner un JSON avec les nouvelles données du jeu
Acheter un jeu s'il est disponible et si l'utilisateur a suffisamment d'argent	Retourner un JSON avec les données de l'achat

N.B : l'id de toutes les entités est incrémenté automatiquement, pas d'insertion manuelle.