

# API REST – Node.js

API TCG

## Présentation

L'objectif est de découvrir les API REST à travers Node.js.

Pour cela on va créer l'API d'une application de jeu de cartes à collectionner (TCG).

Cette API offrira les fonctionnalités de bases de création d'utilisateurs, gestion des données et diverses fonctionnalités liées aux TCG.

## Concepts théoriques

Avant de commencer la réalisation, il est nécessaire de rappeler ou présenter les différents aspects théoriques qui seront abordés.

### API REST

Une API (Application Programming Interface) est un ensemble de règles et d'outils qui permet à différents logiciels de communiquer entre eux.

Elle définit comment des fonctionnalités ou des données peuvent être utilisées par d'autres programmes, sans exposer leur fonctionnement interne.

Une API offre ses fonctionnalités à travers des urls spécifiques, des points d'entrées appelés "end-points". Chaque end-point est dédié à une fonctionnalité et expliqué dans la documentation de l'API.



REST (REpresentational State Transfer) est un style d'architecture pour concevoir des API web.

Il repose sur des principes simples :

- Utilisation du protocole HTTP et accès aux ressources via des URL
- Toutes les réponses et les données suivent la même logique (messages, erreurs, structures)
- Opérations standards (GET, POST, PUT, DELETE) utilisées selon les opérations effectuées
- Communication sans état (stateless) : chaque appel est indépendant, aucune information sur les requêtes reçues n'est gardée.

Plus d'informations : <https://restfulapi.net/rest-architectural-constraints/>

## Node.js

<https://nodejs.org/fr>

Node.js est une plateforme logicielle libre en JavaScript, permettant de créer des serveurs, des applications web, et des scripts.

Il repose sur le moteur V8 JavaScript de Google Chrome et utilise un modèle d'architecture orienté événements et non bloquant.

Pour vérifier la présence de Node.js, ouvrir un terminal et exécuter :

```
node -v
```

Cela doit afficher la version installée, sinon il est disponible ici : <https://nodejs.org/fr/download>

## Modules Node.js

Un module en Node.js est un fichier JavaScript contenant du code que vous pouvez réutiliser dans différents fichiers.

Node.js utilise un système de modules basé sur require, ce qui permet d'importer et d'utiliser des fonctionnalités provenant d'autres fichiers ou des bibliothèques tierces.

Les modules peuvent être :

- Internes : Il s'agit de ceux fournis de base avec Node.JS, exemples : fs, path, http.
- Externes : Ce sont des librairies disponibles en ligne à installer avec npm avant de les utiliser, exemple : express, socket.io, pm2
- Personnalisés : créés par vous pour organiser et réutiliser votre propre code.

Exemple d'un fichier `mathUtils.js` qui sert de petit module :

```
function add(a, b) {  
  return a + b;  
}  
  
function subtract(a, b) {  
  return a - b;  
}  
  
module.exports = { add, subtract };
```

Ce fichier peut être importé comme un module dans d'autres fichiers de l'application grâce à la dernière ligne qui permet de préciser les fonctions exportable.

Il peut être utilisé de cette manière :

```
const mathUtils = require("./mathUtils");  
  
console.log("Addition :", mathUtils.add(5, 3));  
console.log("Soustraction :", mathUtils.subtract(9, 4));
```



## Express

<https://expressjs.com/fr/>

Express est un framework minimaliste et très populaire pour Node.js qui facilite la création de serveurs web et d'API.

Il offre un ensemble de fonctionnalités pour gérer les requêtes HTTP, les routes, et les middlewares (fonctions intermédiaires pour manipuler les requêtes/réponses).

Express est un module externe disponible via npm, il peut être installé en exécutant dans le dossier du projet :

```
npm install express
```

Exemple de création d'une petite API avec Express :

<https://wordlyfusion.com/2024/08/15/construire-une-api-restful-avec-node-js-et-express/>

## TCG

TCG pour Trading Card Game, en français : jeu de cartes à collectionner.

Il s'agit d'un type de jeu où les joueurs construisent des decks personnalisés à partir de cartes qu'ils collectionnent, puis s'affrontent selon des règles spécifiques.

Les cartes peuvent représenter des personnages, des sorts, des objets ou des effets, et elles ont souvent différentes raretés.

# Initialisation du projet

Créer un dossier “tcg-api” avec un fichier principal **app.js** et installer Express (voir plus haut).

Utiliser Express et ajouter une première route à la racine de l’application renvoyant un message :

```
app.get("/", (req, res) => {  
  res.json(  
    {  
      message : "Bienvenue sur l'API TCG",  
      data : {}  
    }  
  );  
});
```

On gardera ce format de réponse pour tous les appels à l’API :

- Message : Affiche un message de réponse selon l’appel, sert également pour les erreurs
- Data : Contient éventuellement les données renvoyées selon l’endpoint appelé

Ajouter cette ligne dans **app.js** :

```
app.use(express.urlencoded({ extended: true }));
```

Cela nous permet de gérer les données des formulaires correctement pour la suite.

Ajouter un fichier **.gitignore** et ignorer le dossier **node\_modules**, partager le projet sur Github à lucas-bierne.

On ignore ce dossier car il contient toutes les dépendances et alourdirait énormément les commits inutilement, on ne souhaite garder que le code réalisé.

# Gestion des utilisateurs

Créer un fichier à **data/users.json** contenant :

```
[
  {
    "id": 1,
    "username": "Exemple",
    "password": "exemple",
    "collection": []
  }
]
```

Il s'agit du fichier de stockage de nos utilisateurs, on peut y voir un utilisateur servant d'exemple pour présenter la structure.

Créer un module **users.js** (voir le précédent TP pour les modules), qui contiendra toute la gestion des utilisateurs.

## Création des utilisateurs (/register)

On va créer un premier endpoint à l'API : la création d'utilisateur via un envoi en POST.

Dans notre module users, ajouter une nouvelle fonction vide **RegisterUser(req, res)** et l'exporter.

Ensuite, l'utiliser dans le routing d'express de cette manière (sans oublier le require) :

```
app.post("/register", users.RegisterUser);
```

On notera que les paramètres de la fonction RegisterUser sont fixes et seront automatiquement gérés par Express :

- req : Objet correspondant à la requête effectuée, contient toutes les informations d'entrée
- res : Objet correspondant à la réponse, il permet de répondre avec un status, des données, etc..

Maintenant que la route est créée vers la méthode de création d'utilisateur, on peut la remplir.

Pour récupérer les informations envoyées par POST, on peut les récupérer dans `req.body`, par exemple pour récupérer le username on pourra utiliser `req.body.username`

Exemple de la fonction qui récupère le username :

```
function RegisterUser(req, res)
{
  if(!req.body)
  {
    res.status(400).json({"message": "Erreur : Aucune données"});
    return;
  }

  let username = req.body.username;

  // TODO

  res.json({"message": "OK"});
}
```

On remarquera qu'un message d'erreur est envoyé si le formulaire est vide.

#### 💡 Note :

Les mots de passe sont en clair dans le fichier JSON.

Dans le cadre de ce TP ce n'est pas un problème, mais pour les plus curieux, il est possible de les encrypter avec le module npm [bcrypt](#), par exemple.

# Postman

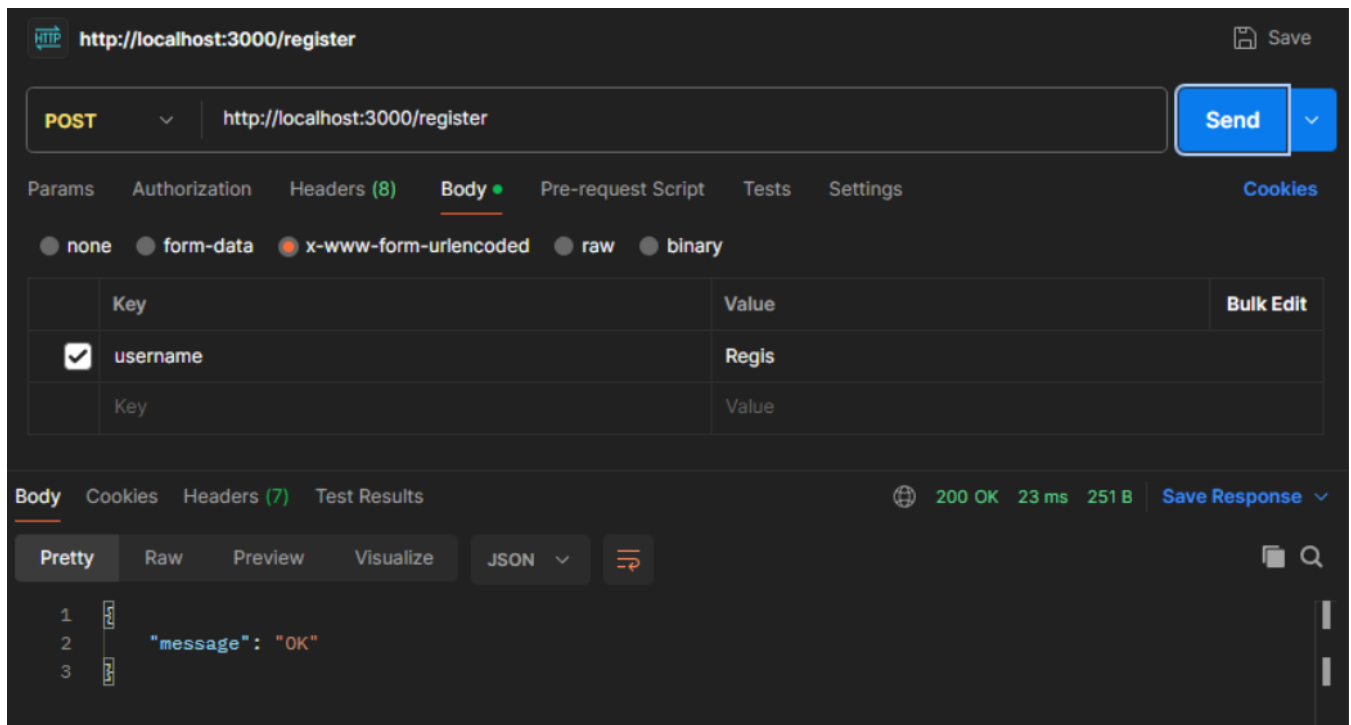
On va tester la fonction avant d'aller plus loin, pour cela on va utiliser Postman.

Postman est une application permettant de tester des API, disponible en client léger et sans création de compte.

Télécharger Postman : <https://www.postman.com/downloads/>

Une fois lancé, l'utiliser pour appeler notre API :

- sur l'url /register
- en POST
- le body en "x-www-form-urlencoded"
- notre données "username" avec une valeur



On peut maintenant utiliser Postman pour tester et debugger facilement l'API.

Une fois que l'appel fonctionne, on peut terminer la fonction, elle doit donc :

- Récupérer les données envoyées
- Lire le fichier users.json pour récupérer les utilisateurs
- Ajouter le nouvel utilisateur dans la liste
- Enregistrer la nouvelle liste

## Authentification des utilisateurs (/login)

Maintenant que les utilisateurs sont créés, on va gérer l'authentification.

L'authentification sera gérée avec un token qui sera renvoyé par l'api si un utilisateur réussit à se connecter.

Ce token sera par la suite requis pour tous les endpoints nécessitant un utilisateur connecté.

On va créer un nouvel endpoint GET **/login** qui amène à une fonction **Login(req, res)** comme la précédente.

Cette fonction récupère également un username et un password, mais cette fois elle les utilise pour vérifier l'utilisateur et s'il est valide, enregistre le token dans l'utilisateur et le renvoie.

Exemple de réponse :

```
{
  message : "Authentification réussie",
  data :
    {
      token : "H2G242"
    }
}
```

Le token sera une suite de caractères alphanumériques au choix, pour ce TP la méthode de génération est libre du moment qu'il semble assez unique.

💡 Pour les plus motivés, il est possible d'utiliser des modules spécialisés pour une gestion de token complète, exemples : [token-generator](#), [jsonwebtoken](#).

## Informations d'un utilisateur (/user)

On va maintenant créer un simple endpoint qui renvoie les informations d'un utilisateur.

Créer un endpoint GET **/user** avec la fonction correspondante, toujours dans la même logique.

La fonction reçoit un token et récupère l'utilisateur correspondant, puis le renvoi dans la réponse.

## Approche REST

Sur cet endpoint `/user`, créer une autre route qui sera en PATCH et qui permet de modifier l'username de l'utilisateur.

Le fait d'utiliser la même syntaxe d'endpoint mais avec un type de requête différent et courant dans les API REST.

## Renforcer les endpoints

Maintenant que la base de la gestion des utilisateurs est en place, on va renforcer le fonctionnement, une API doit être robuste !

Gérer au minimum les cas suivants en renvoyant un message adapté (comme dans l'exemple de la fonction `RegisterUser`) et un code d'erreur HTTP adapté :

- `RegisterUser`
  - o Pas de données envoyées
  - o Username ou mot de passe absent
  - o Utilisateur déjà existant
- `Login`
  - o Pas de données envoyées
  - o Username ou mot de passe absent
  - o Utilisateur introuvable
- `User`
  - o Token invalide

## Déconnexion (`/disconnect`)

Créer un endpoint **`/disconnect`** qui reçoit un token, et le supprime de l'utilisateur si valide.

Si le token est supprimé côté utilisateur, cela effectue une déconnexion.

# Documentation


A ce stade de l'avancement de l'API il est essentiel de commencer une partie important : la documentation de l'API.

Toute API possède une documentation, aussi appelée “contrat d'interface” qui décrit son fonctionnement et détaille l'utilisation de tous ses endpoints.

Quelques exemples de documentations :

- Google Maps: <https://developers.google.com/maps/documentation/javascript>
- Twilio: <https://www.twilio.com/docs/usage/api>
- Discord: <https://discord.com/developers/docs/reference>

Utiliser la page d'accueil de l'API (le message de bienvenu sur / ) pour afficher une documentation de l'API sur une page web concernant les premiers endpoints réalisés.

 Il est possible d'utiliser des outils de génération automatique tels que swagger, du moment qu'ils sont maîtrisés et cohérents avec l'API réalisée.

# Gestion des cartes

On va maintenant s'occuper de la partie cartes à collectionner.

## Cartes disponibles (/card)

On va commencer par créer la liste des cartes disponibles, pour cela créer un fichier **data/cards.json** et le remplir avec quelques données.

Exemple :

```
[
  { "id": 1, "name": "Bulbizarre", "rarity": "common" },
  { "id": 4, "name": "Salamèche", "rarity": "common" },
  { "id": 7, "name": "Carapuce", "rarity": "common" },
  { "id": 25, "name": "Pikachu", "rarity": "rare" },
  { "id": 150, "name": "Mewtwo", "rarity": "legendary" }
]
```

Créer un nouveau module **cards.js**

Ajouter un nouveau endpoint GET **/card** qui renvoie toutes les cartes disponibles.

## Ouverture d'un booster (/booster)

On va maintenant créer l'ouverture d'un booster.

Dans le domaine des jeux de cartes à collectionner un booster est une pochette de cartes aléatoires destinée à rejoindre la collection d'un joueur.

Ces cartes obtenues peuvent être des doublons, d'où le principe de rareté de chaque carte, ce qui rend certaines cartes plus difficiles à obtenir que d'autres.

Créer un nouvel endpoint PUT **/booster** qui appelle une fonction **OpenBooster** dans le nouveau module **cards**.

Fonctionnement :

- Recoit un token utilisateur et le vérifie
- Tire aléatoirement 5 cartes dans la liste des cartes disponibles
- Attribue les cartes à la collection de l'utilisateur
  - o `user.collection` : liste d'id de cartes possédées par l'utilisateur
- Réponse : message de confirmation et les cartes du booster

Une fois la fonction opérationnelle, il doit être possible de consulter la collection d'un utilisateur depuis le endpoint **/user**

## Délai

On va ajouter un délai de 5 minutes entre chaque ouverture de booster.

Pour cela, lors de l'ouverture, enregistrer dans l'utilisateur le moment d'ouverture :

```
user.lastBooster = Date.now();
```

Ce nouvel attribut dans les utilisateurs stocke la date actuelle d'ouverture (en format timestamp). Il faut ajouter une vérification au début de l'ouverture pour vérifier que le dernier "lastBooster" de l'utilisateur a plus de 5 minutes.

Ainsi, l'appel à /booster doit gérer le délai selon le token fourni et envoyer une réponse adaptée.


## Rareté

Gérer la rareté des cartes.

Il s'agit de définir des chances d'apparition différentes selon la rareté, ce qui rend le tirage plus réaliste.

Exemple :

- Common : 80%
- Rare : 15%
- Legendary : 5%

 Il est possible de trouver des sources externes pour cet algorithme, cependant elles doivent être complètement comprises si elles sont intégrées



# Monnaie & doublons

On ajoute une nouvelle fonctionnalité qui permet d'exploiter les doublons de cartes possédés : permettre de les transformer en une monnaie interne.

Cette monnaie sera utilisée plus tard.

## Gestion des doublons

Il s'agit de gérer les doublons au sein d'une collection.

Il est donc nécessaire de faire évoluer la collection des utilisateurs, il s'agit maintenant d'une liste d'objets, ou chacun contient l'id de la carte et le nombre d'exemplaires, par exemple :

```
collection : [  
  { id : 1, nb : 3 },  
  { id : 2, nb : 5 },  
  { id : 42, nb : 1 }  
]
```

Mettre à jour l'ouverture des boosters pour gérer les doublons et augmenter le compteur de chaque cartes.

## Transformation en monnaie (/convert)

Ajouter un champ "currency" aux utilisateurs, mettre à jour les données dans users.json, ainsi qu'à la création et l'affichage.

Nouveau endpoint en POST à l'API : **/convert**

Fonctionnement :

- Reçoit le token user et l'id de la carte devant être convertie
- Vérifie l'utilisateur et la validité de la conversion :
  - o L'utilisateur doit obligatoirement posséder la carte en plusieurs exemplaires
- Décrémente le compteur de la carte dans la collection de l'utilisateur
- Incrémente la monnaie de l'utilisateur en fonction de la rareté de la carte (valeurs au choix)

# Enchères (/bid)

## Fonctionnement

Les enchères vont utiliser la monnaie juste avant.

Un utilisateur peut mettre en vente une de ses cartes, elle est alors retirée de sa collection et placée en vente dans une enchère pour une durée limitée.

Lorsqu'un utilisateur participe à une enchère en augmentant le cout, le montant choisi est débité de son solde et placé dans l'enchère.

S'il y a un précédent enchérisseur il est alors remboursé.

Le vendeur ne peut pas enchérir sur sa propre enchère.

Si une enchère est terminée, le vendeur ou l'acheteur peut la clôturer.

Lors de la clôture le vendeur reçoit la somme en monnaie et la carte est ajoutée au gagnant.

## Enchère

Nouvelle structure d'objet pour les enchères :

Attribut	Type	Description
<b>id</b>	int	Identifiant unique d'une enchère
<b>card_id</b>	int	Identifiant de la carte vendue
<b>seller_id</b>	int	Identifiant du vendeur
<b>end_date</b>	date	Date de fin de l'enchère
<b>bidder_id</b>	int	Identifiant de l'acheteur
<b>bid</b>	int	Montant de l'enchère

Créer un nouveau fichier JSON pour stocker les enchères.

## Nouveaux endpoints

Route	Méthode	Description	Paramètres
/bid	POST	Créer une enchère	token, id de la carte
/bid	PUT	Placer une enchère	token, id de l'enchère, montant
/bid	GET	Récupère toutes les enchères	token
/bid	GET	Récupère une enchère	token, id de l'enchère
/bid	DELETE	Clôture une enchère	token, id de l'enchère

Penser à gérer les cas spéciaux avec des codes et message d'erreurs adaptés, voici quelques cas à gérer :

- Créer une enchère sans avoir la carte
- Enchérir avec une monnaie insuffisante
- Enchérir sur sa propre enchère
- Tenter de clôturer une enchère non terminée
- Etc...