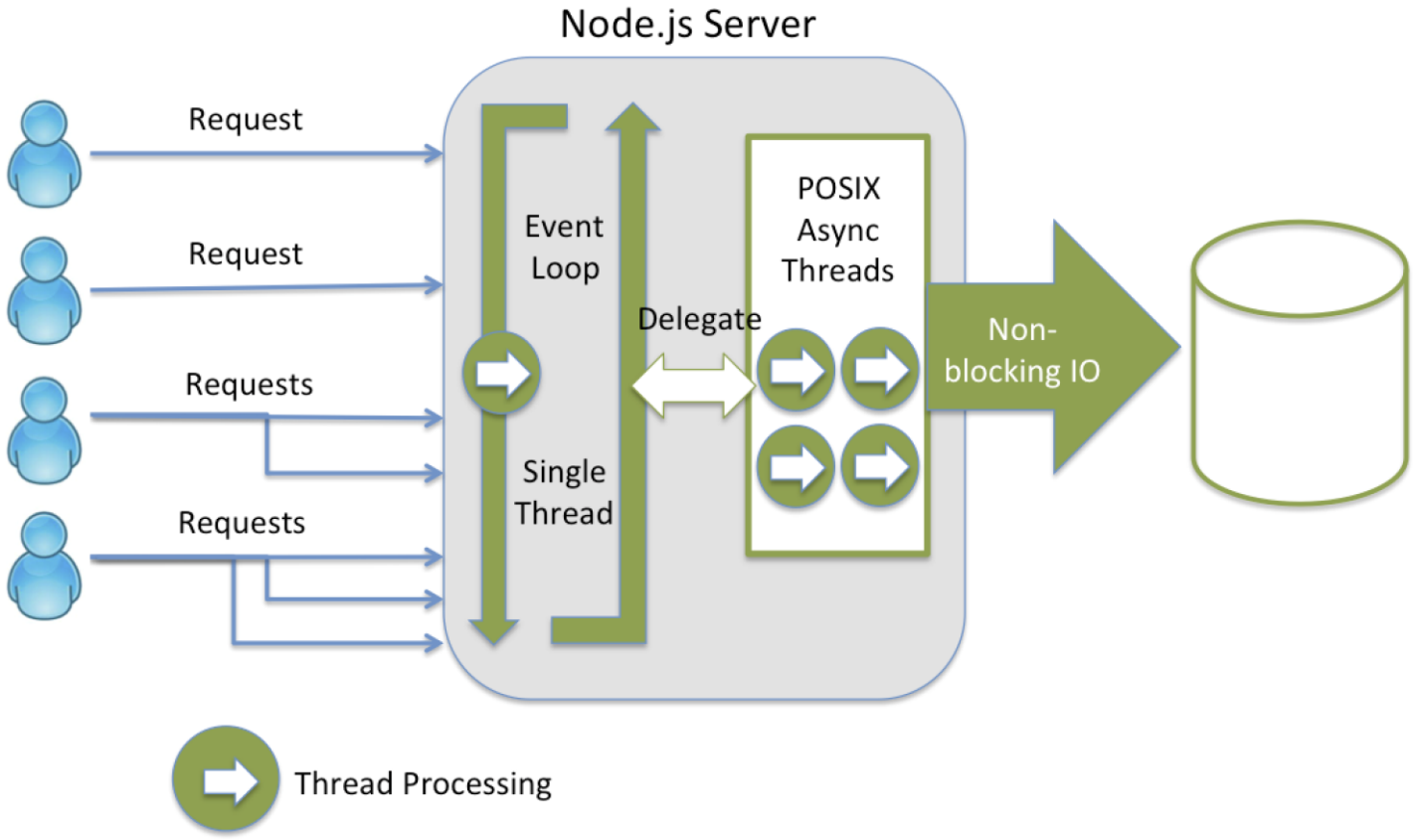
Programmation asynchrone

Traitement des requêtes

- Node.js est monothreadé : toutes les opérations sont exécutées dans un seul thread.
- La boucle d'événements (event loop) : permet à Node.js de traiter des requêtes asynchrones non bloquantes
- Il permet à Node.js de se décharger les opérations d'entrées-sorties sur le noyau système
  - Les opérations E/S : requêtes HTTP, lectures ou écritures de fichiers, requêtes en base de données, etc.

Traitement des requêtes

- Les opérations synchrones sont exécutées séquentiellement en exploitant la pile d'appels (call stack). Tant que des opérations sont présentes sur la pile d'appels celles-ci sont exécutées.
- Il est donc essentiel de ne pas avoir des opérations qui monopolisent cette pile puisqu'il n'y a qu'un seul thread.
- Les opérations les plus pénalisantes doivent donc être exécutées de manière asynchrone. Elles sont alors gérées par l'event loop qui communique avec le noyau du système d'exploitation hôte pour lui déléguer l'exécution de ces opérations asynchrones dans des threads système



Tester un traitement asynchrone


```
function asyncFunction(msg, delay) {
  setTimeout(
    // use setTimeout to simulate the asynchronous code
    () => console.log(`I am ${msg}, delaid for ${delay} ms`), // callback
    delay
  );
}
asyncFunction("hello1",1000);
asyncFunction("hello2",600);
```

```
PS F:\Mes documents prof\Enseignements\AI16\Contenu 2023\TDs\projet-web> node .\asyn.js
I am hello2, delaid for 600 ms
I am hello1, delaid for 1000 ms
```

Tester un traitement asynchrone récupérer un résultat


```
let value;
function asyncValue(delay) {
  setTimeout(() => (value = 100), delay);
}
asyncValue(200);
console.log(`value =${value}`);
```

```
value =undefined
```

- Le programme essaie d'afficher le résultat d'un traitement asynchrone avant son terminaison

Utilisation d'une fonction callback


```
let value;
callback = (value) => console.log(`value
=${value}`);


function asyncValue(delay, callback) {
  setTimeout(() => {value = 100;
callback(value);}, delay);

}

asyncValue(200,callback);
```

- La fonction callback récupère le résultat du traitement asynchrone à la fin de ce dernier.

```
PS F:\Mes documents prof\Enseignements\AI16\Contenu 2023\TDs\projet-web> node .\asyn.js
value =100
```

CALLBACK HELL


```
const makeBurger = nextStep => {
  getBeef(function (beef) {
    cookBeef(beef, function (cookedBeef) {
      getBuns(function (buns) {
        putBeefBetweenBuns(buns, beef, function(burger) {
          nextStep(burger)
        })
      })
    })
  })
}

// Make and serve the burger
makeBurger(function (burger) => {
  serve(burger)
})
```

Solution 1 : ajouter des commentaires

```
// Fait un hamburger
// makeBurger contient quatre étapes :
// 1. Obtenez du boeuf
// 2. Cuire le boeuf
// 3. Obtenez des petits pains pour le burger
// 4. Mettez le boeuf cuit entre les pains
// 5. Servir le burger (depuis le rappel)
// Nous utilisons des rappels ici car chaque étape est asynchrone.
// Nous devons attendre que l'assistant termine la première étape
// avant de pouvoir passer à l'étape suivante
const makeBurger = nextStep => {
  getBeef(function (beef) {
    cookBeef(beef, function (cookedBeef) {
      getBuns(function (buns) {
        putBeefBetweenBuns(buns, beef, function(burger) {
          nextStep(burger)
        })
      })
    })
  })
}
// Make and serve the burger
makeBurger(function (burger) => {
  serve(burger)
})
```

Solution 2 : moins de fonctions imbriquées & de fonctions anonymes


```
app.get("/", function (req, res) {
  Users.findOne({ _id: req.body.id }, function (err, user) {
    if (user) {
      user.update(
        {
          /* params to update */
        },
        function (err, document) {
          res.json({ user: document });
        }
      );
    } else {
      user.create(req.body, function (err, document) {
        res.json({ user: document });
      });
    }
  });
});
```

Solution 2 : moins de fonctions imbriquées & de fonctions anonymes

```
const updateUser = (req, res) => {
  user.update(
    {
      /* params to update */
    },
    function () {
      if (err) throw err;
      return res.json(user);
    }
  );
};

const createUser = (req, res, err, user) => {
  user.create(req.body, function (err, user) {
    res.json(user);
  });
};
```

```
  app.get("/", function (req, res) {
    Users.findOne({ _id: req.body.id },
(err, user) => {
      if (err) throw err;
      if (user) {
        updateUser(req, res);
      } else {
        createUser(req, res);
      }
    });
```

Promesse (Promise)

- Une promesse est un objet qui renverra une valeur dans le futur.
- Les promesses sont bien adaptées aux opérations JavaScript asynchrones.

- Exemple

```
getSomethingWithPromise()
  .then(data => {/* do something with data */})
  .catch(err => {/* handle the error */})
```

  - Lorsque la promesse se résout, nous faisons quelque chose avec les données qui reviennent. Sinon si la promesse est rejetée, nous traitons l'erreur

Construction d'une promesse


```
const acheterGateau = gateauType => {
    return new Promise((resolve, reject) => {
      setTimeout(()=> {
        if (gateauType
   === 'tartelette') {
          resolve('Une délicieuse tartelette!')
        } else {
          reject('pas de gâteau 😢')
        }
      }, 1000)
    })
  }
```

```
const promise = acheterGateau("tartelette")
  .then((gateau) => console.log(gateau))
  .catch((aucungateau) => console.log(aucungatea
```

- Vous pouvez faire une promesse en utilisant la nouvelle promesse.
- Ce constructeur Promise prend en charge une fonction qui contient deux arguments : résoudre et rejeter.

- Si la résolution est appelée, la promesse réussit et continue dans la chaîne then. Le paramètre que vous transmettez à resolve serait l'argument dans le prochain appel then.

- Si le rejet est appelé, la promesse échoue et continue dans la chaîne de capture. De même, le paramètre que vous passez à rejeter serait l'argument dans l'appel catch.

Promise vs Callback

- Les promesses réduisent la quantité de code imbriqué
- Les promesses vous permettent de visualiser facilement le flux d'exécution
- Les promesses vous permettent de gérer toutes les erreurs à la fois à la fin de la chaîne.

Exemple


```
asyncValuePromise = (delay) => {
  return new Promise((resolve, reject) => {
    if (delay >= 0) {
      setTimeout(() => {
        resolve(100);
      }, delay);
    } else reject("delay <0");
  });
};

asyncValuePromise(200).then((data) => console.log(`value =${data}`));
```

Utilisation async & await


```
asyncValuePromise = (delay) => {
  return new Promise((resolve, reject) => {
    if (delay >= 0) {
      setTimeout(() => {
        resolve(100);
      }, delay);
    } else reject("delay <0");
  });
};

const asyncAwaitfunction = async function (delay) {
  const value = await asyncValuePromise(delay);
  console.log(`value =${value}`);
};
asyncAwaitfunction(200);
```

Representational State Transfer (REST)

Representational State Transfer (REST)

- **Un style d'architecture logicielle pour les systèmes distribués hypermedia comme WWW**
- **Introduit par Roy Fielding dans sa thèse de doctorat**
  - Un des auteurs des spécifications HTTP.
- **Un ensemble de principes décrivant comment les ressources sont définies et adressées**

REST et HTTP

- **La motivation de REST est de s'appuyer sur les caractéristiques qui ont fait le succès du Web.**
  - Ressources adressables par URI
  - HTTP
  - Faire une Requête – Recevoir une Réponse – Afficher Réponse
- **Exploite l'utilisation de HTTP**
  - HTTP POST, HTTP GET, HTTP PUT, HTTP DELETE

REST – n'est pas un Standard

- **REST n'est pas un standard**
  - Node.js & Express
  - Autres modules
- **Mais utilise plusieurs standards:**
  - HTTP
  - URL
  - XML/HTML/GIF/JPEG/etc (Représentation de Ressources)
  - text/xml, text/html, image/gif, image/jpeg, etc (Types de Ressources, MIME Types)

Stateless ou stateful ?

- REST est-elle une architecture *stateless* (sans état) ou *stateful* (avec état) ?
- La réponse: est « stateless ».
  - Il n'y a pas de notion de session: le serveur ne se souvient pas des enchaînements des requêtes, ne fait pas varier les réponses selon un enchaînement particulier et ne stocke pas d'informations autres que les actions demandées sur les ressources (création, modification, suppression...).

Concepts

Nom(Ressources)
sans contraintes ex.,
http://example.com/employees/12345

Verbes
contraintes ex.,
GET

Représentations
contraintes ex., XML

Ressources

- **L'abstraction clé de l'information dans REST est la Ressource**
- **Une ressource est un mapping conceptuel vers un ensemble d'entités**
  - Toute info qui peut être nommée est une ressource: un document ou image, un service temporaire (météo à Paris), une collection d'autres ressources, etc.
- **Repésentée par un identifiant global (URI)**
  - http://www.boeing.com/aircraft/747

Nommage des Ressources

- **REST utilise les URI pour nommer les ressources**
  - http://localhost/books/
  - http://localhost/books/ISBN-0011
  - http://localhost/books/ISBN-0011/authors
- **On navigue dans les données en traversant le chemin du plus générique au plus spécifique.**

Verbes

- **Représentent les actions possibles sur les ressources**
  - HTTP GET
  - HTTP POST
  - HTTP PUT
  - HTTP DELETE
  - HTTP HEAD
  - HTTP OPTIONS
  - HTTP TRACE
  - HTTP CONNECT

HTTP GET

- **Permet aux Clients de demander une information**
- **Transfère les données du Serveur vers le Client sous une certaine représentation**
- **GET http://localhost/books**
  - Récupérer tous les livres
- **GET http://localhost/books/ISBN-0011021**
  - Récupérer le livre identifié par ISBN-0011021
- **GET http://localhost/books/ISBN-0011021/authors**
  - Récupérer les auteurs du livre identifié par ISBN-0011021

HTTP PUT, HTTP POST

- **HTTP POST crée une ressource**
- **HTTP PUT modifie une ressource**
- **POST http://localhost/books/**
  - Content: {title, authors[], …}
  - Crée un nouveau livre avec les propriétées données
- **PUT http://localhost/books/isbn-111**
  - Content: {isbn, title, authors[], …}
  - Modifie le livre identifié par isbn-111 avec les propriétés soumises

HTTP DELETE

- **Supprime la ressource identifiée par l'URI**
- **DELETE http://localhost/books/ISBN-0011**
  - Supprimer le livre identifié par ISBN-0011

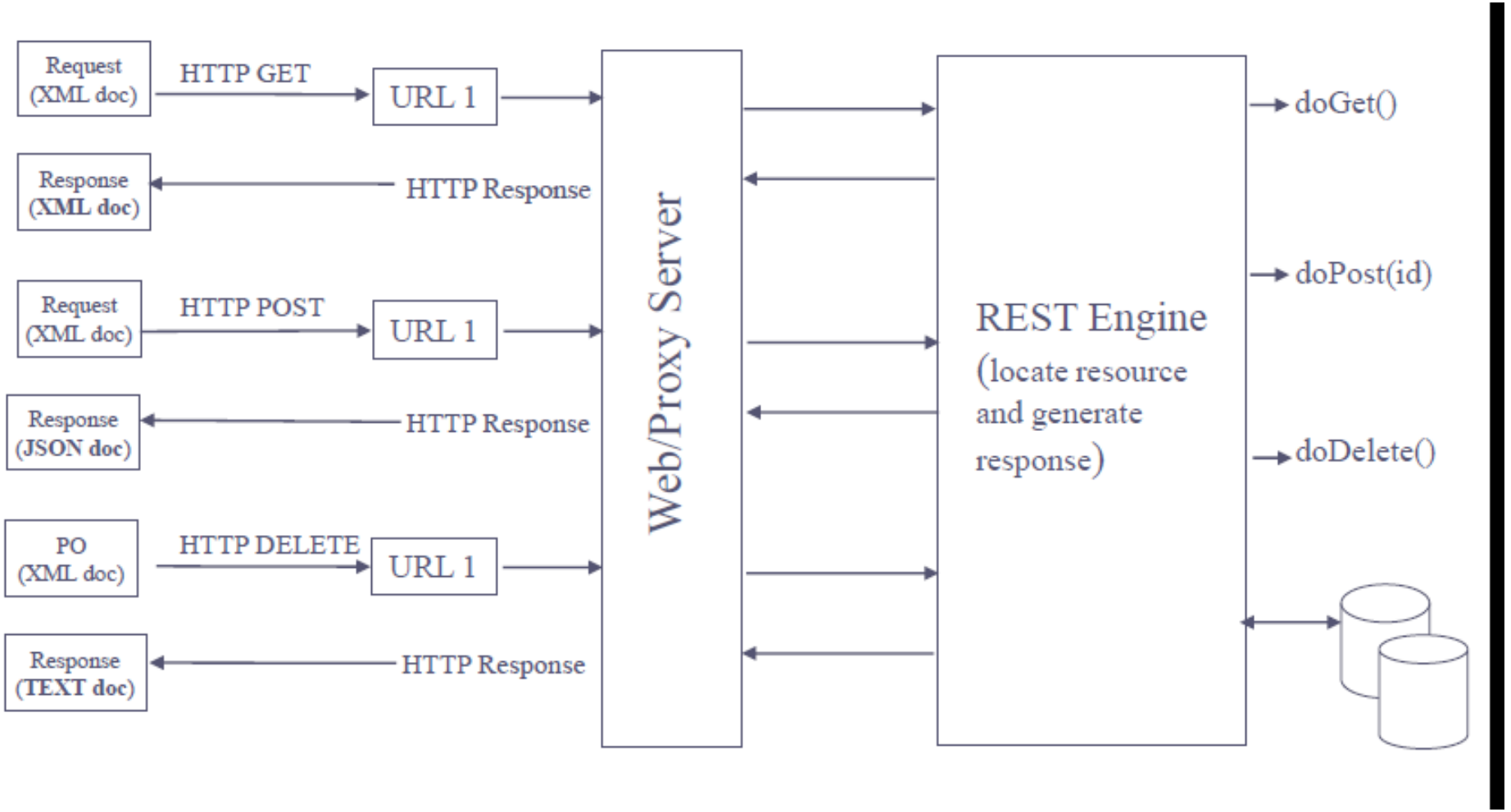
Représentations

- **Comment les données sont représentées ou retournées au client pour présentation.**
- **Deux formats principaux:**
  - JavaScript Object Notation (JSON)
  - XML

Représentations

- **XML**
  - <COURSE>
    - <ID>CS2650</ID>
    - <NAME>Distributed Multimedia Software</NAME>
  - </COURSE>
- **JSON**
  - {course
    - {id: CS2650}
    - {name: Distributed Multimedia Sofware}
  - }

Architecture



Implémentation
Node.js & Express

- app.get(), app.post(), app.delete(), app.put(),…
- Ou
  - const api =express.Router()
  - api.get() ….

```
app.get('/users', function (req, res, next) {

    result=userModel.readall(function(result){

      res.status(200).json(result);

    });
});
```

- On peut utiliser la méthode json(obj) ou send (obj) de l'objet réponse (res) pour retourner la réponse sous format json

Implémentation
Node.js & Express

- router.get('users/:id', (req,res) => {
  - const id = req.params.id;

});

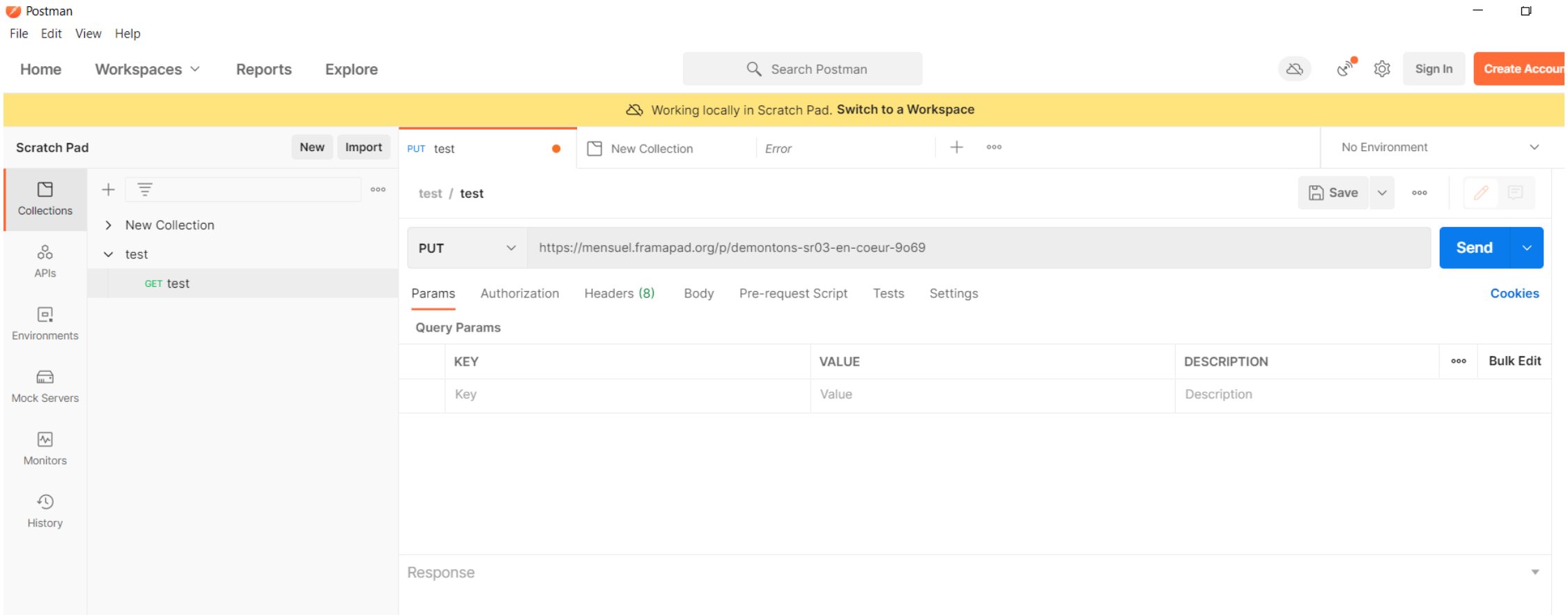
- router.post('users/',(req, res) => {
  - let data = req.body…
  - …// utiliser creat de votre modèle

});

Tester une API REST

Postman

- Fabriquer des requêtes http
  - get, delete, post, put etc
  - Ajouter des paramètres, body (post, put, path, etc.), etc.
  - Visualiser des réusltats



Cross-Origin Resource Sharing (CORS)

- Partage de ressources entre origines
  - Localhost :3000 vs localhost:3001
  - Site1.com vs site2.com
- Il permet de relâcher la sécurité appliquée à une API.
- Mise en place
  - > npm install cors
  - Ajouter dans app.js
    - const cors = require('cors’);
    - ..
    - app.use(cors()); // premier use de vottre app