programmation asynchrone

javascript et son écosystème

Licence mention Informatique
Université Lille – Sciences et Technologies

principes de Node

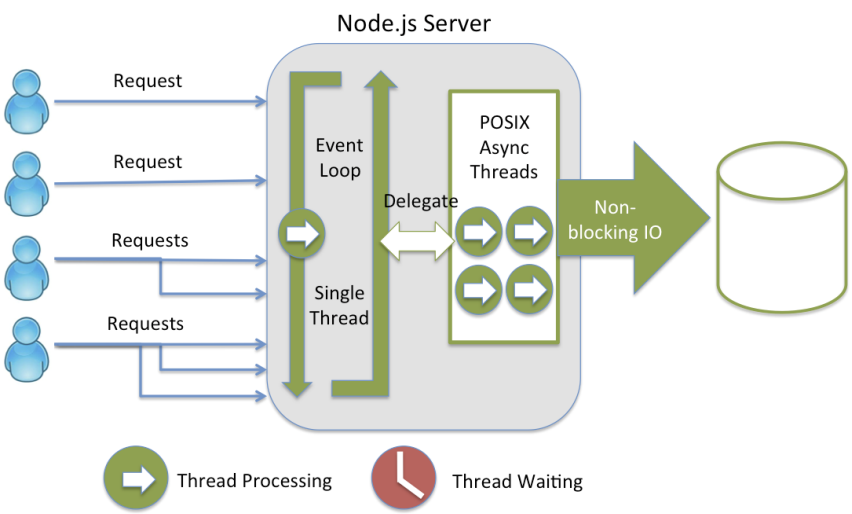
- un seul thread
- des entrées/sorties non bloquantes
- orienté événements « *event-loop* »

modèle asynchrone

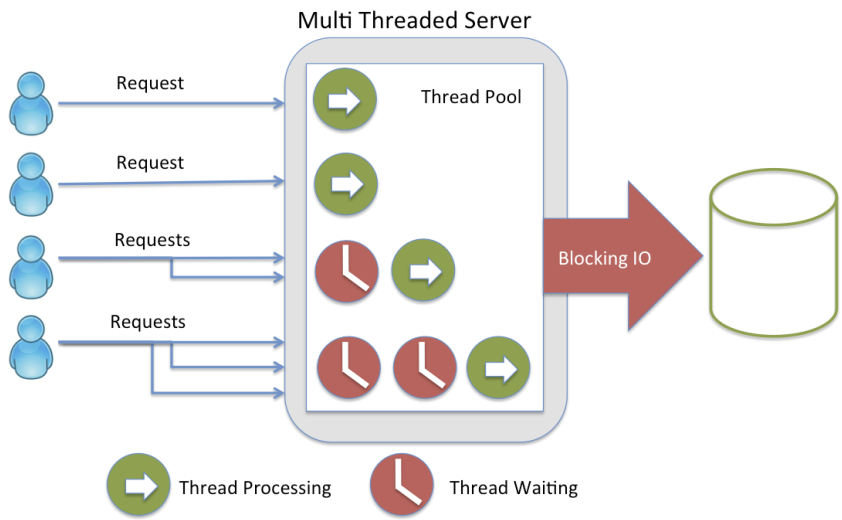
- les opérations sont déléguées tant que possible au système

comparaison

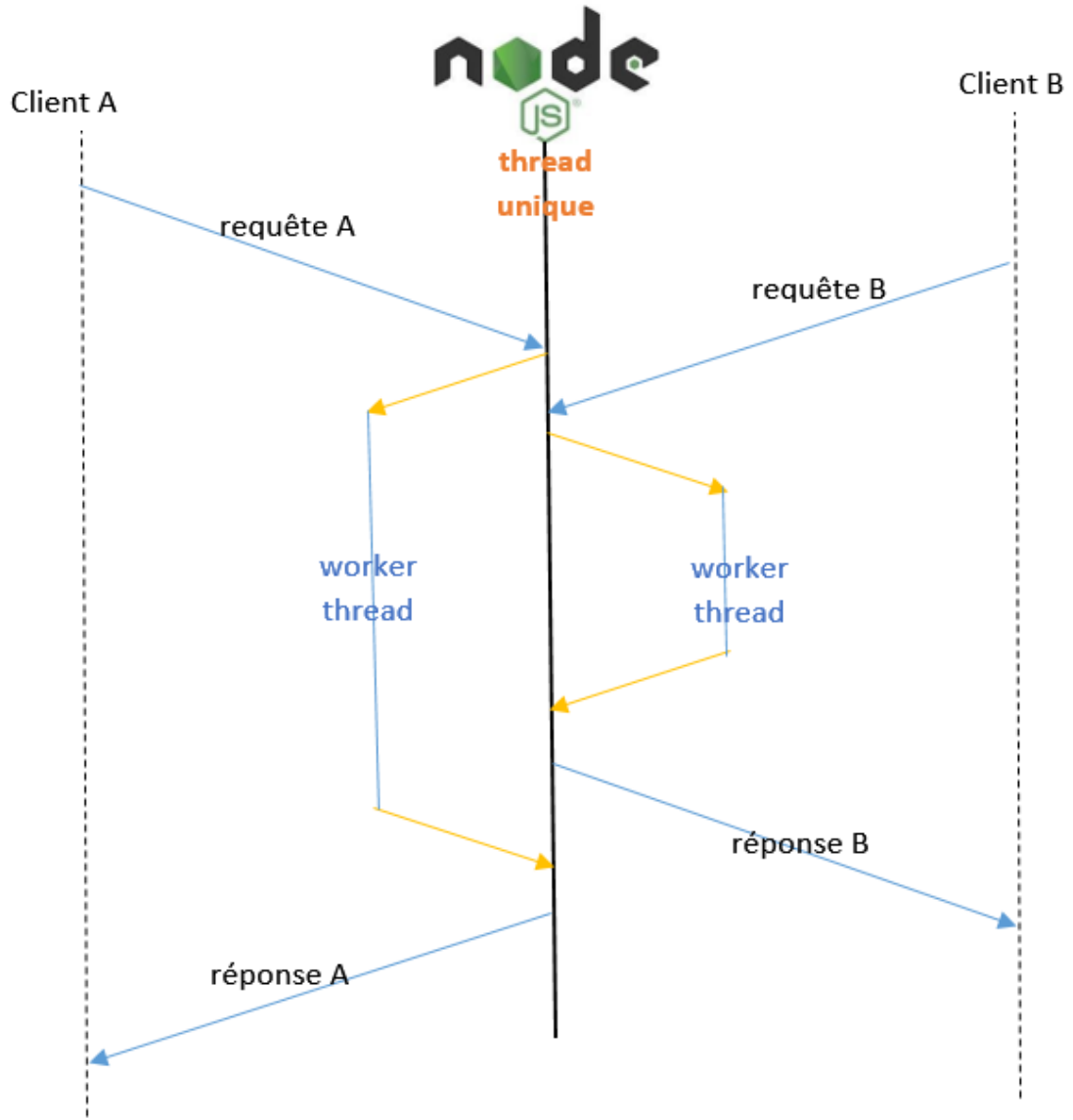
modèle *thread unique*
Node.js



modèle *multi-threads*
Apache, IIS



appels asynchrones

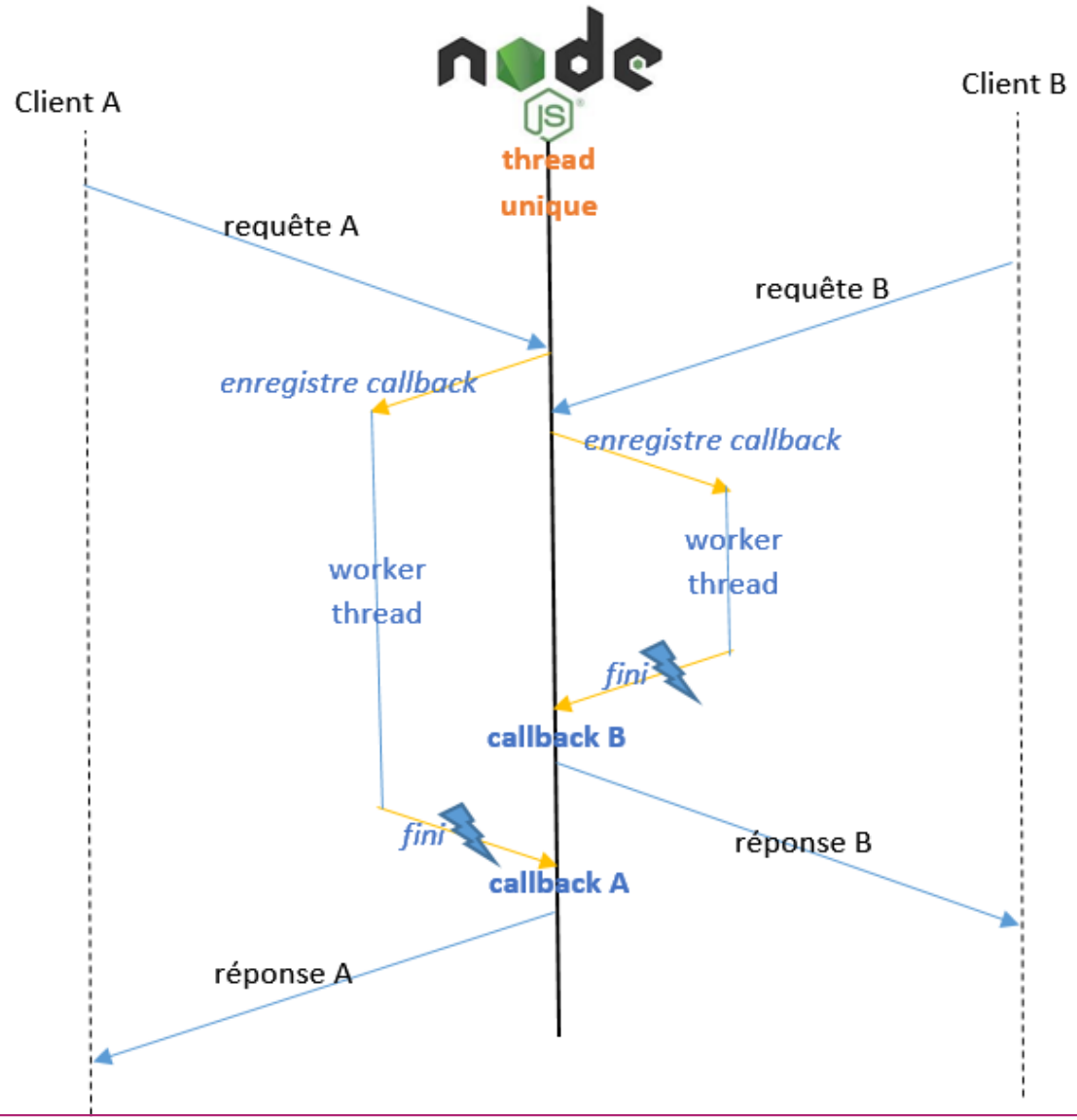


principes et conséquences

- appel asynchrone simple — async-example v0
- version avec 2 appels asynchrones "en séquence" — async-example v1
- tentative pour enchainer après un traitement asynchrone — async-example v2

ajout de **callback** aux fonctions asynchrones

callbacks


Client A
Client B
thread
unique
requête A
requête B
enregistre callback
enregistre callback
worker
thread
worker
thread
fini
callback B
fini
callback A
réponse B
réponse A

- utilisation des callbacks
  - ajout d'un callback pour gérer un traitement exécuté à la suite de l'appel asynchrone
    async-example v3
  - premier aperçu du *callback hell*
    enchaînement séquentiel de deux appels asynchrones
    async-example v3.1
  - utilisation des paramètres du callback pour « renvoyer » une valeur
    async-example v3.2
- il faut que les callbacks « s'exécutent rapidement »
  ne pas bloquer la boucle événementielle ni le pool de workers

exemples

Node.js : callbacks généralement de la forme `function(err, data)`

fs.readFile(path[, options], callback)

- *path* `<string>|<Buffer>|<URL>|<integer>` filename or file descriptor
- *options* `<Object> | <string>`
  - *encoding* `<string> | <null>` Default: `null`
  - *flag* `<string>` Default: `'r'`
- *callback* `<Function>`
  - *err* `<Error>`
  - *data* `<string> | <Buffer>`

Asynchronously reads the entire contents of a file.(...)
The callback is passed two arguments `(err, data)`, where data is the contents of the file.

cf. Node.js API

exemples

- version initiale — async-readfile v0
  réponse avec un fichier statique lu de manière asynchrone
- version avec fichier à charger passé dans l'url — async-readfile v1
  + utilisation du paramètre `error` en cas de fichier non trouvé
  tester les accès concurrents
- comparaison avec la version synchrone — async-readfile v1.1
  readFileSync
- chargement des fichiers par morceaux « template » — async-readfile v2
  mise en « séquence » des lectures de fichiers
  et donc imbrication des appels asynchones
  ⇒ **callback hell**

réduire le *callback hell*

- nommer les callbacks — async-readfile v2.1
- utilisation des **promesses** — async-readfile v3

promesses

- les **promesses** (***promises***) permettent de simplifier la gestion de code asynchrone
- une promessse est une valeur qui sera disponible *dans le futur* ou jamais
- une promesse est un objet qui gère la réussite ou l'échec d'un appel asynchrone
  - une promesse peut être
    - **résolue** appel asynchrone réussi
    - **rejetée** appel asynchrone en échec
  - réussite et échec sont gérées par deux callbacks
    généralement appelés `resolve` et `reject`
- la valeur d'une promesse est immuable et peut-être obtenue plusieurs fois
- les promesses peuvent être **chaînées**

principe

```
var firstPromise = new Promise(
  (resolve, reject) => {
    if (isItOk()) {
      let theResult = doSomethingAndProduceTheResult();
      resolve(theResult);
    }
    else {
      reject(new Error('problem'));
    }
  }
);
```

- le calcul de la promesse débute à sa création

then / catch

- le *callback* `resolve` est défini et appelé via `then`
- le *callback* `reject` est défini et appelé via `catch`

```
firstPromise
   .then( res => useResultFromPromise(res) )   // resolve callback
   .catch( err => handleError(err));           // reject callback
```

- si, et lorsque, la valeur de `firstPromise` est disponible, la fonction paramètre de `then` est appelée avec le résultat : `resolve(...)`
- si une erreur est survenue elle est gérée par la fonction paramètre de `catch` : `reject(...)`

exemple (resolve/then)

async-example v4

```
var somePromisedFunction = value => {
  return new Promise(
    (resolve, reject) => {
       window.setTimeout(
            () => {
                    let result = 2 * value;
                    resolve(val *2);
                  },
            randint(250)
       );
    }
  )
}

let firstPromise = somePromisedFunction(10);

firstPromise.then( result => output(‘return value is $result‘ ) );
```

exemple (reject/catch)

async-example v4.1

```
var somePromisedFunction = value => {
  return new Promise(
    (resolve, reject) => {
        if (value >= 0 ) {
          window.setTimeout(
              ...
          );
        }
        else {
          reject (new Error(‘ value is negative ($value)‘));
        }
    }
  )
}

for (val of [10, 20, -5]) {
  let firstPromise = somePromisedFunction(val);
  firstPromise
    .then( result => output(‘return value is \${result}‘ ) )
    .catch( err => output(‘error :  ${err.message}‘ ) );
}
```

comparons

callback « classique »

async-example v3.2

```
var doAsyncJobAndCallCallback =
  (value, callback) => {
    window.setTimeout(
        () => {
          let something = value;
          callback(something);
        },
        randint(250)
    );
  }
```

avec une promesse

async-example v4.2

```
var somePromisedFunction = val => {
    return new Promise(
      (resolve, reject) => {
          window.setTimeout(
            () => {
              let something = val;
              resolve(something);
            },
          randint(250)
        );
      }
  )
}
```

comparons

- version callback « classique »

```
doAsyncJobAndCallCallback( 10,
   (result) => {
     x = x + result;
     output(‘I am the callback: result is ${result} and x is ${x}‘);
   }
);
```

- version promesse

```
  somePromisedFunction(10).then(
     result => {
         x = x + result;
         output(‘I am the resolve promise callback: result is ${result} and
              x is ${x}‘);
  );
```

- bof : gain pas évident...

chainage

- `then` et `catch` produisent une nouvelle promesse
- elles sont résolues avec le résultat de la fonction appelée
- il est donc possible de chainer les appels

exemple

- on retrouve une écriture *séquentielle* des enchainements de traitements asynchrones

async-example v4.3

```
let promise = somePromisedFunction(10);

promise
  .then( value => result = x + value )
  .then( () => output(‘after first resolve: x is ${x}‘ )
  .then( ()  => somePromisedFunction(20) )
  .then( result => x = x + result )
  .then( () => output(‘after second resolve: x is ${x}‘) );
```

- cf *callback hell*

async-example v3.2

- gain beaucoup plus évident !

gestion erreurs

- `catch` aussi produit une promesse et donc *thenable*
- comportement type *finally*
- les erreurs de la chaine sont propagées et interceptées

async-example v4.4

```
firstPromise
  .then( result => { ... })
  .then( result => {
            if (result > 30) {
                throw new Error(‘trouble with ${result}‘)
            } else return result
          })
  .then( result =>  output(‘second return value is ${result}‘ )  )
  .catch( err => output(‘error : ${err.message}‘ ) )
  .then( () =>  output('--- this is always written ---')  ) ;
```

exemple

chargement des images pour `ImageSlider`

- sans promesse, utilisation des callbacks

```
class ImageSlider {
  initImages (src1, src2) {
    this.imageAvant = new Image();
    this.imageApres = new Image();
    this.imageAvant.addEventListener('load',
      () => {                                    // callback hell
        this.imageApres.addEventListener('load',
                                          () => this.init()
        );
        this.imageApres.src = src2;
      });
    this.imageAvant.src = src1;
  }
}
```

exemple

chargement des images pour `ImageSlider`

- « *promisification*» du chargement des images pour `ImageSlider`

```
let setImageSrc = (img, src) =>       // promisification
  new Promise( resolve => {
    img.addEventListener( 'load', resolve );
    img.src = src;
  });
```

travail fait une fois pour toute...

```
class ImageSlider {
  initImages(src1, src2) {
    this.imageAvant = new Image();
    this.imageApres = new Image();
    setImageSrc(this.imageAvant, src1)
      .then( () => setImageSrc( this.imageApres, src2 ) )
      .then( () => this.init() );
  }
}
```

Promise.all

La méthode `Promise.all()` renvoie une promesse qui est résolue lorsque l'ensemble des promesses contenues dans l'itérable passé en argument ont été résolues ou qui échoue avec la raison de la première promesse qui échoue au sein de l'itérable. *(MDN)*

```
class ImageSlider {
  initImages(src1, src2) {
    this.imageAvant = new Image();
    this.imageApres = new Image();
    Promise.all([setImageSrc(this.imageAvant, src1),
                 setImageSrc(this.imageApres, src2)])
      .then( this.init.bind(this) );
  }
}
```

intérêt : les promesses sont exécutées en parallèle

autre exemple

- utilisation des promesses pour la lecture asynchrone des fichiers

  async-readfile v3

  cf. `ContentReader.promisedReadFile(path)`
  « séquentialisation » des traitements asynchrones
- *promisification* du code synchrone avec `Promise.resolve()`

  async-readfile v3.1

```
end() {
  return  Promise.resolve().then(this.response.end())
}
```

  et « enrichissement » de l'API de `ContentReader`

fs.promises

depuis node v11, `fs` fournit une *promisification* des fonctions :

```
    const fs = require('fs').promises;

    fs.readFile(path)
      .then( content => this.response.write(content) );
```

async-readfile v3.2

```
// fichier /scripts/contentReader.js
const fs = require('fs').promises;

class ContentReader {

  prepareFile(path) {
        return fs.readFile(path)
                 .then( content => this.response.write(content) );
  }
  ...
}
```

async/await

Ecmascript2017 ES8 introduit les mots-cles `async`/`await` pour simplifier l'utilisation des promesses.

- `async` permet une fonction asynchrone
- le résultat d'une telle fonction est une promesse

- `await` permet d'attendre qu'une promesse soit résolue ou rejetées
- `await` ne peut être utilisée qu'au sein d'une fonction `async`

exemple

```
// définition d'une fonction asynchrone
const simple = async name => { return name; }
// simple a pour résultat une Promise
simple('timoleon')
     .then( value => console.log(`hello ${name}`) );  // hello timoleon

// await ne peut être utilisé que dans une fonction async
const useSimple = async msg => {
  const name = await simple(msg);    // attend réponse appel asynchrone
  console.log(`hello ${name}`);
}
useSimple('timoleon');                                 // hello timoleon
```

exemple

code d'exploitation des promesses plus «naturel» async-example v4.3

```
const usePromiseToMakeAsynchronousJob = () => {
  const x = 5;
  const promise = somePromisedFunction(10);
  promise
    .then( value => result = x + value )
    .then( () => output(`after first resolve: x is ${x}` )
    .then( ()  => somePromisedFunction(20) )
    .then( result => x = x + result )
    .then( () => output(`after second resolve: x is ${x}`) );
}
```

devient async-example v5

```
const usePromiseToMakeAsynchronousJob = async () => {
     let x = 5;
     let result = await somePromisedFunction(10);
     x = x + result;
     output(`I am after first resolve promise callback : x is ${x}`);
     result =  await somePromisedFunction(20);
     x = x + result;     // ou      x = x +  await somePromisedFunction(20);
     output(`I am after second resolve promise callback : x is ${x}`) ;
 }
```

async/await

async-readfile v3.3 en partant de la version 3

```
  // fichier /index.js
const fs = require('fs').promises;
const server = http.createServer(
  async (request, response) => {
    ...
    response.writeHead(200, {"Content-Type": "text/html"});
    try {
      const header = await fs.readFile('data/header_html');
      response.write(header);
      const content = await fs.readFile(path);
      response.write(content) ;
      const footer = await fs.readFile('data/footer_html');
      response.write(footer);
    }
    catch(e) { // in case of error while reading files
      const error = await fs.readFile('data/error_html');
      response.write(error);
    }
    finally {
      response.end();
    }
```

API Fetch

- permet de récupérer des ressources de manière **asynchrone**
- alternative à/successeur de `XMLHttpRequest`
- l'API Fetch fournit des objets `Request`, `Response`, `Header`, `Body` pour manipuler les requêtes et leurs résultats
- la méthode `fetch()` a pour résultat une **promesse** résolue avec la réponse (objet `Response`)
  - la promesse échoue (`reject`) en cas d'erreur réseau
  - la propriété `Response.ok` permet de savoir si la requête a réussi
    une requête qui échoue ne se traduit pas par un rejet de la promesse

exemple

express v3

côté **client** chargé par `http://127.0.0.1:3000/books`

```
// dans /public/javascripts/bookdetails.js
const id = book.dataset.id;

fetch(`http://127.0.0.1:3000/books/details/${id}`)
    .then( response => response.json() )
    .then( book => displayDetails(book) )
    .catch( error => console.log(error.msg) );
```

côté **serveur**

```
// dans /controllers/books.js
details(req,res){
  res.status(200).json( books[req.params.bookId - 1] );
}

// dans /routes/books.js
const booksController = require('../controllers/books');
router.get('/details/:bookId', booksController.details );
```

gestion erreurs de requêtes

côté **serveur**

express v3.1

```
// dans /controllers/books.js
  details (req,res) {
    let book = books[req.params.bookId - 1];
    if (book)
      res.status(200).json(book);
    else
      res.status(404).end();
  }
```

côté **client**

```
// dans /public/javascripts/bookdetails.js
fetch(`http://127.0.0.1:3000/books/details/${id}`)
 .then( response => {
   if (response.ok) {
     return response.json();          // successfull fetch
   }
   throw new Error('book unknown');  // else: response not ok, fetch failed 404
 })
 .then( book => displayDetails(book) )
 .catch( error => displayError(`problem with book id ${id}`) );
```

options de requêtes

express v3.2

côté client

```
// dans /public/javascripts/bookdetails.js
const id = book.dataset.id;

const options = { method : 'GET' }

fetch(`http://127.0.0.1:3000/books/details/${id}`, options)
  .then( response => {
    if (response.ok) {
      ...
    }
  })
  .then( book => displayDetails(book) )        // decoded book is displaid
  .catch( error => console.log(error.msg) );
```

autre exemple

côté **client**, utilisation d'une route avec la méthode PUT

```
// dans /public/javascripts/bookdetails.js
const updateTitle =
  (input) => {
    const id = input.dataset.id;
    const body = JSON.stringify({ newTitle : input.value });
    const requestOptions = { method :  'PUT',
                             headers: { "Content-Type": "application/json" },
                             body : body };
    fetch(`http://127.0.0.1:3000/books/details/${id}`, requestOptions)
      .then( ... )
```

côté **serveur**, définition de la route pour la méthode PUT

```
// dans /routes/books.js
router.put('/details/:bookId', booksController.updateTitle );
```

```
// dans /controllers/books.js
class BookController {
  constructor() { ...
    this.updateTitle = this.updateTitle.bind(this);
  }
  updateTitle(req, res) { ... }
}
```

async/await

express v3.3

côté **client**

```
// dans /public/javascripts/bookdetails.js
const getDetails = book => {
  const id = book.dataset.id;
  fetch(‘http://127.0.0.1:3000/books/details/\$\{id\}‘)
    .then( response => response.json() )
    .then( book => displayDetails(book) )
    .catch( error => console.log(error.msg) );
}
```

devient

```
const getDetails = async book => {
  const id = book.dataset.id;
  try {
    const response = await fetch(‘http://127.0.0.1:3000/books/details/\$\{id\}‘);
    const book = await response.json();
    displayDetails(book);
  }
  catch(error) { console.log(error.msg) }
}
```

requêtes *cross-origin*

- quand une ressource chargée depuis un domaine fait une requête vers un autre domaine

exemple :

chargement `http://localhost:3000/books`

avec dans `public/javascripts/bookdetails.js` :

```
fetch(`http://127.0.0.1:3000/books/details/${id}`, ...)
```

la requête échoue car les url de base ne sont pas les mêmes :

cross-origin

CORS

express v3.4

- **Cross-Origin Request Sharing**
- c'est un standard W3C
- le standard CORS définit des règles pour déterminer si une requête cross-origin peut avoir lieu
  basé sur la définition de nouveaux entêtes de réponses

avec Express, activation CORS sur toutes les routes :

- `npm install cors --save`
- ```
  // dans /app.js
  const cors = require('cors');
  const app = express();
  app.use(cors());
  ```
  ```

alternative

définir un middleware :

```
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  //res.header("Access-Control-Allow-Origin", "http://localhost:3000");
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-
      Type, Accept");
  next();
});
```

- possibilité de choisir les sites autorisés