node : express

javascript et son écosystème

Jean-Christophe Routier
Licence mention Informatique
Université Lille – Sciences et Technologies

express

un *framework* web pour Node.js

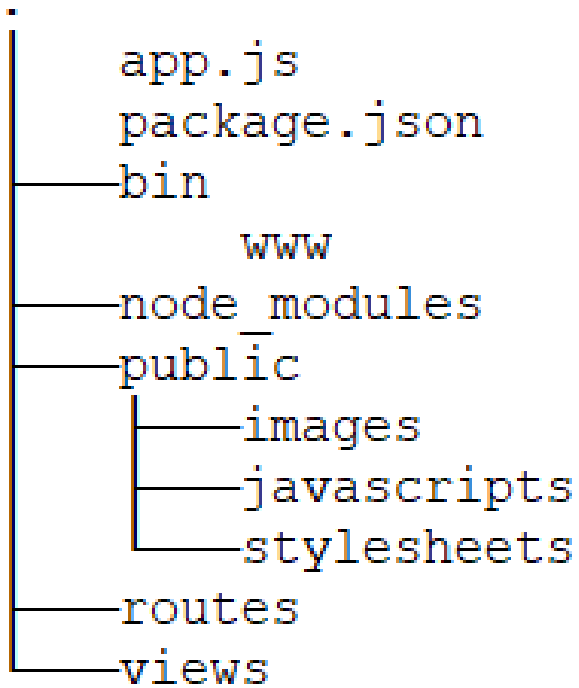
express

- permet de fixer les propriétés classiques d'un serveur
- facilite la gestion des routes
- intègre des moteurs de rendu des vues
- s'appuie sur des *middlewares* qui peuvent s'insérer dans le flux de gestion des requêtes

installation

- utilisation de l'outil `express-generator`
  `npm install express-generator -g`
- création du projet
  `express --view=pug <app_name>`
  utilisation de *pug* comme moteur de vues
  ↪ crée une structure par défaut pour les projets `express`
- installation des dépendances dans dossier *app_name*
  `npm install`
- `npm start` → `http://127.0.0.1:3000`

structure du projet



express v0

bin/www

express v0

- dans package.json

```
"scripts": {
  "start": "node ./bin/www"
}
```

- mise en place du serveur http Node.js

```
// dans /bin/www
var app = require('../app');
var http = require('http');
    (...)
app.set('port', port);
    (...)
var server = http.createServer(app);
    (...)
server.listen(port);
server.on('error', onError);
    (...)
function(onError) { ...
```

app.js

- structure générale

```
// dans /app.js
var express = require('express');
(...)
var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'pug');

// mount middlewares
app.use(...);
app.use(...);
...
```

- `__dirname` est une variable `Node.js` représentant le dossier du module courant
- `app.use(...)` insère un *middleware* dans le flux de traitement de la requête

app.use

app.use([path,] callback [, callback...])

- **path** précise le chemin auquel s'applique le middleware
  - si absent, le middleware s'applique à tous les chemins
  - peut être une expression régulière
- **callback** peut être une « **fonction middleware** »

  autres possibilités : *Express API*
- fonction middleware : `function(req, res, next)`
  - `req` l'objet requête HTTP, de type `Request`
  - `res` l'objet réponse HTTP, de type `Response`
  - `next` fonction callback pour chainer les middlewares

exemples

-

```
app.use(express.static(path.join(__dirname, 'public')));
```

`express.static` est un middleware pour gérer de manière statique les fichers (css, png, etc.) placés sous le dossier précisé (ici `public`)
http://127.0.0.1:3000/images/timoleon.jpg express v0.1

-

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

analyse le corps de la requête (au format json ou url-encoded) et le restitue dans `req.body`

-

```
app.use(cookieParser());
```

analyse l'entête `Cookie` de la requête et alimente `req.cookies`

- les middlewares sont exécutés séquentiellement
- pour que l'on passe au suivant il faut qu'un middleware appelle `next()`
- exemples (triviaux) de middlewares
  - ajout d'un middleware trivial pour la route `/first`

```
// dans /app.js
app.use('/first', function(req, res, next) {
  res.writeHead(200, {"Content-Type": "text/html"});
  res.write('<h1>first middleware</h1>');
  res.write('<p>I am alive and using /first</p>');
  res.end('bye');
});
```

express v1

  - on peut « chainer » des middlewares avec `next()`

```
// dans /app.js
app.use(
  function(req, res, next) {
    req.body.witness = 'Timoleon was here';
    next();
  }
);
```

express v1.1

routeurs

un routeur

- est créé par la méthode `Router()` de l'objet `express`
- est un middleware dédié à la gestion des routes
- se comporte comme une "*mini application*"
  dispose d'une méthode `use()`
- gère les routes pour les différentes requêtes HTTP (GET, POST, ...)
  ↪ méthodes `.get()`, `.post()`, etc.

bonne pratique

- définir chaque routeurs dans un module à part et exporter ce routeur
- l'importer et le déclarer, avec sa « route racine », comme middleware dans l'application

```
// dans app.js
var booksRouter = require('./routes/books');
...
app.use('/books', booksRouter);
```

- NB : dans le routeur, les chemins sont relatifs à la route racine déclarée pour le middleware

- cf. dossier `routes` généré par Express

express v2

ajout de middleware au routeur

express v2.1

```
// dans /routers/books.js
var books = require('../data/books'); // to simulate data acquisition
var router = express.Router();

var allBooks;
var bestBook;

router.use(
(req, res, next) => {
    bestBook = books[0];
    allBooks = Array.of(...books);
    allBooks.sort( (book1, book2) => book1.author.localeCompare(book2.author) );
    next();
  }
)

router.get('/',... );

router.get('/best',...);
```

chemins des routes

- routes “joker” ’?’, ’+’, ’*’

```
router.get('/root/ti?mo*on',   ... );
```

/root/timoleon, /root/tiXmoLLEEon, /root/tiXmoon, etc.

- routes patterns ou expressions régulières

```
router.get(/[bB]e(st|ST)/, ...);
```

/best, /Best, /beST, /BeST

```
router.get(/.*[bB]est.*/, ...);
```

/best, /Best, /theBestOne, /it/is/reallythebest/one , etc.

express v2.2

chemins des routes

- routes paramétrées

```
// dans /routers/books.js
router.get('/details/:bookId',
  (req, res, next) => res.render('bookdetail',
                                 { title: 'Book detail',
                                   id: req.params.bookId,
                                   book:  books[req.params.bookId - 1]
                                  })
);
```

  - pas forcément en fin de chemin : `/details/:bookId/other`
  - plusieurs paramètres possibles : `/details/:bookId/:nextId`

express v2.2

bonne pratique

séparation des préoccupations

- séparer la gestion des routes de leur logique de traitement
  - définir un **contrôleur** pour chaque route
    dans un dossier `controllers`
  - le contrôleur définit et exporte les fonctions de traitement
  - le routeur importe le contrôleur pour en utiliser les fonctions

express v2.3

définition de la logique des routes dans le contrôleur

```
    // dans /controllers/books.js
let prepare = (req, res, next) => { ... };
let list = (req, res) => res.render( ...  ); // controller function for '/'
let best = (req, res) => res.render( ... );
let details = (req, res) => res.render( ... );
    // export controller functions
module.exports.prepare = prepare;
module.exports.list = list;
module.exports.best = best;
module.exports.details = details;
```

utilisation par le routeur

```
  // dans /routers/books.js
const booksController = require('../controllers/books');

  // link controllers to route paths
router.use( booksController.prepare );
router.get('/', booksController.list );
router.get(/[bB]est/, booksController.best );
router.get('/details/:bookId', booksController.details );
```

express v2.3

principales méthodes de `Response`

- `res.download()` Prompt a file to be downloaded.
- `res.end()` End the response process.
- `res.json()` Send a JSON response.
- `res.redirect()` Redirect a request.
- `res.render()` Render a view template.
- `res.send()` Send a response of various types.
- `res.sendFile()` Send a file as an octet stream.
- `res.status()` Set the response status code

réponse json et autres

- `res.status()` fixe la code du statut de la réponse, renvoie une réponse (« chainable »)
- `res.json()` envoie une réponse au format JSON

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

- `res.sendFile()` voir dans `/controllers/example.js`
- `res.download()` voir dans `/controllers/example.js`
- `render` utilisation d'un moteur de vue

express v2.4

res.render

- express v2.5

```
// dans /controllers/index.js
module.exports.home =
     (req, res) => res.render('index', { title:  'Express' });
```

`res.render(view, locals)`

  - *view* : la vue à construire
  - *locals* : un objet qui définit des propriétés locales à la vue

-

```
//dans /app.js
      // view engine setup
app.set('views', path.join(__dirname,'views'));
app.set('view engine', 'pug');
```

- bonne pratique : un dossier pour les vues `/views`
- **pug** est un **moteur de templates** pour code HTML
- le processus de pug génère du code HTML pour des templates écrits dans la syntaxe de pug

pug : syntaxe html allégée

```
// dans /views/about.pug
doctype html

html
  head
    title about page
    link(rel="stylesheet", href="/stylesheets/style.css")
                                                  // attributs entre ()
  body
                                                  // indentation pour "emboitement"
    h1 about page                                 // balise html avec contenu
    p.text Introduction to Express                // avec une classe CSS
    .example Licence 3 Informatique - option      // div implicite
      em Javascript et son écosystème
    #otherexample                                 // id avec div implicite
      | voir                                    // texte simple sur plusieurs lignes
      |
      a(href="http://portail.fil.univ-lille1.fr/ls6/js") sur le portail
```

template

- notion de **block** que les *templates enfants* peuvent (ou non) remplacer

```
///views/layout.pug
doctype html

html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content

  footer
    block footblock
      p Licence 3 - #[em Javascript et son écosystème]
```

- NB : *tag interpolation* : `#[em ...]`

héritage

- héritage partiel

```
// dans /views/index.pug
extends layout

   block content
      h1= title
      p Welcome to #{title}
```

- héritage avec extension (prepend existe aussi)

```
//  dans /views/booklayout.pug
extends layout
 (...)
 block content
    h1 about books block bookcontent

 append footblock
    p you are in #[em book zone]
```

interpolation

utilisation des valeurs passées dans le paramètre `locals` de `render`

-

```
//  dans /controllers/index.js
module.exports.home =
   (req, res) => res.render('index', { title:  'Express' });
```

-

```
// dans /views/index.pug
extends layout

  block content
    h1= title
    p Welcome to #{title}
```

itération

structure itérative pour aider à la définition des templates

```
// dans /controllers/books.js
var list =
  (req, res) => res.render('books', { title: '...', books :  allBooks });
```

```
// dans /views/books.pug
extends booklayout

block bookcontent
  h1= title
  p Welcome to #{title.toLowerCase()}
  table.booklist
    each book in books
      tr.book
        td.author= book.author
        td.title= book.title
```

code javascript

préfixer les lignes de code par « - »

```
// dans /views/users.pug
extends layout

block content
  - var users = ['Tim Oleon', 'Timo Léon', 'Ti Moléon'];

  h1= title
  p Welcome to #{title}
  p here comes the user list...
  ul
    - for (let i = 0 ; i < users.length ; i++)
      li user-#{i} = #{users[i]}
```

mixin

les mixins sont des blocs réutilisables et paramétrables « mini composants »

-

```
// dans /views/booklayout.pug
...
mixin createCover(imgSrc)
  .cover
    img(src=imgSrc)
...
```

- utilisation de + pour « appeler » le mixin

```
// dans /views/bookdetail.pug
block bookcontent
  h1= title
  ...
  +createCover(‘$book.cover‘)

// dans /views/betsbook.pug
block bookcontent
  ...
  +createCover(‘$book.cover‘)
```

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)

```
// dans /public/javascripts/bookdetails.js
let 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
var details =
    (req,res) => res.status(200).json( books[req.params.bookId - 1] );
module.exports.details = details;

// 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
var 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
let id = book.dataset.id;

let 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
var updateTitle =
  (input) => {
    let id = input.dataset.id;
    let body = JSON.stringify({ newTitle : input.value });
    let 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
var updateTitle = (req, res) => { ... }
module.exports.updateTitle = updateTitle;
```

requêtes *cross-origin*

express v3.3

- 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 v3.3.5

- **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');
var 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