Express.js

créer et utiliser un serveur express.js

Setup

Installer express

```
npm i express
npm i -d @types/express
```

Installer express et les types associés pour typescript

Nodemon

npm i -D nodemon ts-node

“start” : “nodemon index.ts”

Installer nodemon et ts-node qui permettent à ce que le projet soit recompilé lorsque l’on sauvegarde un fichier

hello world

```
import express from “express”

const app = express();


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

  res.send("Hello world!");

});


app.listen(3000);
```

On crée une instance express avec la fonction du même nom.

Avec la méthode ‘get’ de cette instance, on ajoute du handling pour une route, ici ‘/’

On lance l’app avec la méthode `listen`

Rounting

Routing simple

```
app.get('/hello', function(req, res){
  res.send("Hello World!");
});
app.post('/hello', function(req, res){
  res.send("Hello World!");
});
app.put('/hello', function(req, res){
  res.send("Hello World!");
});
```

```
app.delete('/hello', function(req, res){
  res.send("Hello World!");
});
```

Pour le routing, on dispose de méthodes éponymes aux méthodes http sur l'objet app

Refactoring

```
// things.ts

import express from “express”;
const router = express.Router();

router.get('/', function(req, res){
   res.send('GET route on things.');
});
router.post('/', function(req, res){
   res.send('POST route on things.');
});

//export this router to use in our index.js
module.exports = router;
```

```
//index.ts

import express from “express”
import things from “things”

const app = express();


app.use('/things', things);

app.listen(3000);
```

Utiliser un Router() permet de fractionner son code en plusieur routeurs

Routing dynamique

```
import express from “express”
const app = express();

app.get('/:id', function(req, res){
   res.send('The id you specified is ' + req.params.id);
});

app.listen(3000);
```

```
import express from “express”
const app = express();

app.get('/things/:id([0-9]{5})', function(req, res){
   res.send('id: ' + req.params.id);
});


app.listen(3000);
```

Il est possible de récupérer des params dans l’URL, en les nommant avec un ‘:’

Middleware

Middleware

Un middleware est une fonction qui a accès à la requête, la réponse, et au prochain middleware.

Il peut servir à faire de l’adaptation sur la donnée entrante ou sortante

Middleware

```
//Simple request time logger

app.use(function(req, res, next){

   console.log("A new request received at " + Date.now());


   //This function call is very important. It tells that more processing is

   //required for the current request and is in the next middleware function route handler.

   next();

});
```

Middleware spécifique

```
//Middleware function to log request address

app.use('/things', function(req, res, next){

  console.log("A request for things received at " + Date.now());

  next();

});


// Route handler that sends the response

app.get('/things', function(req, res){

  res.send('Things');

});
```

On peut spécialiser un middleware pour une route en particulier

Payloads

Pour faciliter les requêtes utiliser Postman ou Insomnia

Postman

Insomnia

Parser de payload Json et multipart

```
# installer avec npm
npm i body-parser multer
npm i -D @types/multer
```

Multer permet de parser des données reçues dans le format multipart/form, notamment si elles contiennent des fichiers.

body-parser permet de parser implicitement un body de requête (depuis json par exemple)

Payloads

```
import bodyParser from “body-parser”
import multer from “multer”

const upload = multer();

// for parsing application/json
app.use(bodyParser.json());

// for parsing application/xwww-
app.use(bodyParser.urlencoded({ extended: true }));
//form-urlencoded

// for parsing multipart/form-data
app.use(upload.array());


app.post('/', function(req, res){
   console.log(req.body);
   res.send("recieved your request!");
});
```

Multer et body parser s’utilisent tel des middlewares: En effet, ils modifient `req` avant le handling de la requête.

Le résultat rendu par ces middlewares est stocké dans req.body.

Cookies

cookie-parser

npm i cookie-parser

npm i -D @types/cookie-parser

cookie-parser s’utilise comme un middleware.

```
import cookieParser from “cookie-parser”

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

set cookie

```
app.get('/', function(req, res){
   console.log(req.cookies)
   res.cookie('name', 'express')
   res.cookie(name, 'value', {expire: 360000 + Date.now()});
});
```

La fonction `cookie` sur res permet d'établir un cookie pour le client. On renseigne d'abord le nom puis la donnée.

Il est possible d'ajouter des options pour gérer, par exemple, l'expiration du cookie.

remove cookie

```
app.get('/clear_cookie_foo', function(req, res){
  res.clearCookie('foo');
  res.send('cookie foo cleared');
});
```

Avec la méthode clearCookie, on peut supprimer un cookie.

Attention à ne pas émettre et supprimer un même cookie dans la même requête.

Sessions

Sessions

```
npm i express-session
npm i -D @types/express-session

app.use(cookieParser());
app.use(session({secret: "Shh, its a secret!"}));

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

  if(req.session.page_views){
    req.session.page_views++;
    res.send("You visited this page " + req.session.page_views + " times");

  } else {
    req.session.page_views = 1;
    res.send("Welcome to this page for the first time!");

  }
});

app.listen(3000);
```

express-session ajoute un attribut `session` a req, sur lequel on peut stocker des valeurs liées à la session.

Ici on enregistre le nombre de fois que l'utilisateur a utilisé cette route API.

Authentification

Authentification simple

```
router.post("/signin", (req, res) => {
 const user: IUser = {
   age: req.body.age,
   password: req.body.password,
   username: req.body.username,
 };
 users.push(user);

 res.status(201);
 res.send({ message: "User created!", status: "Created" });
});
```

La route /signin permet à un utilisateur de créer un compte.

Login

/login permet à un utilisateur existant de s'identifier sur l'API. On peut alors mettre sur sa session le fait qu'il est identifié.

```
app.post('/login', function(req, res){
  console.log(Users);
  if(!req.body.username || !req.body.password){
    res.render('login', {message: "Please enter both username and password"});
  } else {
    Users.filter(function(user){
      if(user.username === req.body.username && user.password === req.body.password){
        req.session.user = user;
        res.send("connected !")
      }
    });
    res.send('login', {message: "Invalid credentials!"});
  }
});
```

logout

```
router.delete("/signout", (req, res) => {
 if (!(req.session as any).user) {
  res.status(401);
  res.send({
    message: "You're not logged in !",
    status: "BadRequest",
  });
 }
 const username = (req.session as any).user.username;
 (req.session as any).user = undefined;
 req.session.destroy(() => {
  console.log(`disconnecting ${username}`);
 });
 res.status(200);
 res.send({
  message: "You are being disconnected",
  status: "OK",
 });
});
```

Pour déconnecter l'utilisateur, on détruit la session à laquelle il est connecté.

Contenu protégé

```
app.use("/things", (req, res, next) => {
  (req.session as any).user ?
  next() :
  res.status(401)
  res.send(
    {
      message: "Please login to access this resource",
      status: "unauthorized"
    }
  )
})
```

Pour protéger du contenu d' un utilisateur qui n’y est pas autorisé, on fait un middleware. Dans le middleware on utilise la session pour vérifier l’habilitation de l’utilisateur. S’il ne doit pas accéder à la ressource alors on lui envoie directement une réponse.

TP

Créer un API qui simule un un système de gestion d'hôtel:

CRUDS: chambres, utilisateurs, réservations

Il y a trois type d’utilisateurs : employé, client, et admin

Un admin seulement peut ajouter ou supprimer des chambres

un client peut uniquement créer ou annuler une réservation

TP

Un utilisateur répond à l'interface suivante:

nom : string

prénom: string

id : nombre unique

role: admin | employé | client

Seul un admin peut créer un employé

Personne ne peut créer un admin

Seul un employé peut créer un user

Le rôle est utilisé dans la session pour vérifier qu'un utilisateur a le droit de faire une requête

TP

Une chambre répond à l’interface suivante :

numéro: number unique

étage: number

prix: number

Seul un admin peut créer ou modifier une chambre, mais tout le monde peut checker la liste des chambres

Une route /chambres/:nb/available doit permettre de savoir si la chambre est disponible

TP

Une réservation doit répondre à l’interface suivante:

id: nombre unique

dateDebut: date

dateFin: date

prix: number

cancelled : boolean (default = false)

user: number correspondant à l'ID de l’utilisateur ayant réservé

chambre: number - numéro de la chambre louée

Seul un client peut créer une réservation, ou l’annuler (pas en la supprimant, mais en mettant son champ cancelled a `true`)

Lors de la création de la réservation, on doit vérifier que la chambre est disponible sur le créneau demandé.

TP

Un utilisateur doit pouvoir estimer le prix que cela lui coûterait de réserver sur un créneau. Pour ça on fera une route /reservations/estimate qui renverra un prix sans créer de réservation.

Comme nous n'avons pas encore vu les bases de données, les entités seront stockées pour le moment dans des listes, dans la mémoire.

bonus: Votre hôtel est dans une station balnéaire. Gérer des prix fluctuants tels qu'ils sont diminués de 20% en période hivernale et augmentés de 20% en période estivale.