Express.js

Le framework Express.js

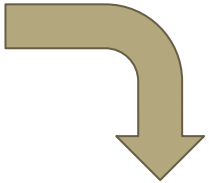
- Express.js est un micro-framework pour Node.js.
- Il met à disposition des outils de base pour aller plus vite dans la création d'applications Node.js.
- Pas assez complet comme les framework php (laravel, symphony ..) python(django).
- Pour installer express.js
    - ***npm install express***

Express.js: firstApp

```
var http = require('http');
var url = require('url');

var server = http.createServer(function(req, res) {
    var page = url.parse(req.url).pathname;
    console.log(page);
    res.writeHead(200, {"Content-Type": "text/plain"});
    if (page == '/') {
        res.write('Accueil');
    }
    res.end();
});

server.listen(3000);
```



```
var express = require('express');

var app = express();

app.get('/', function(req, res) {
    res.setHeader('Content-Type', 'text/plain');
    res.send('Accueil');
});

app.listen(3000);
```

Les routes

```
1  var express = require('express');
2  var app = express();
3
4  app.get('/', function(req, res) {
5      res.setHeader('Content-Type', 'text/plain');
6      res.write('Accueil');
7  });
8
9  app.get('/contact', function(req, res) {
10     res.setHeader('Content-Type', 'text/plain');
11     res.write('Nous contscter');
12 });
13
14 app.get('/cours/web/nodejs', function(req, res) {
15     res.setHeader('Content-Type', 'text/plain');
16     res.write('Node.js is greate');
17 });
18
19 app.use(function(req, res, next){
20     res.setHeader('Content-Type', 'text/plain');
21     res.status(404).send('Page introuvable !');
22 });
23
24 app.listen(3000);
```

Les routes dynamiques

Express permet de gérer des routes dont certaines portions peuvent varier en insérant une variable ***:mavariable*** dans l'URL de la route

Ce qui crée un paramètre accessible avec ***req.params.mavariable***.

```
26  app.get('/capteurs/:id/:temp', function(req, res) {
27      res.setHeader('Content-Type', 'text/plain');
28      res.end('La temperature mesurerpar le capteur n°' +
29                  req.params.id + ' est: ' + req.params.);
30  });
```

Routes Modulaires avec express.Router()

- Lorsque les applications Express.js deviennent complexes, organiser les routes de manière modulaire devient essentiel pour maintenir la clarté et la facilité de maintenance du code.
- ***express.Router()*** est une fonctionnalité puissante qui permet de créer des routes modulaires dans Express.js.
- Il permet de séparer les routes en fichiers distincts, offrant ainsi une meilleure organisation du code.

Routes Modulaires avec express.Router()

- Création d'un Module de Route

userRouter.js

```
const express = require('express');
const router = express.Router();
router.get('/', (req, res) => {
    res.send('Liste des utilisateurs');
});
router.get('/:id', (req, res) => {
    res.send(`Info de l'utilisateur ${req.params.id}`);
});
module.exports = router;
```

Routes Modulaires avec express.Router()

- Utilisation du Module dans l'Application Principale
  - Dans cet exemple, toutes les routes définies dans userRoutes.js seront préfixées par /users.
  - Par exemple, l'URL complète pour la liste des utilisateurs serait /users/

app.js

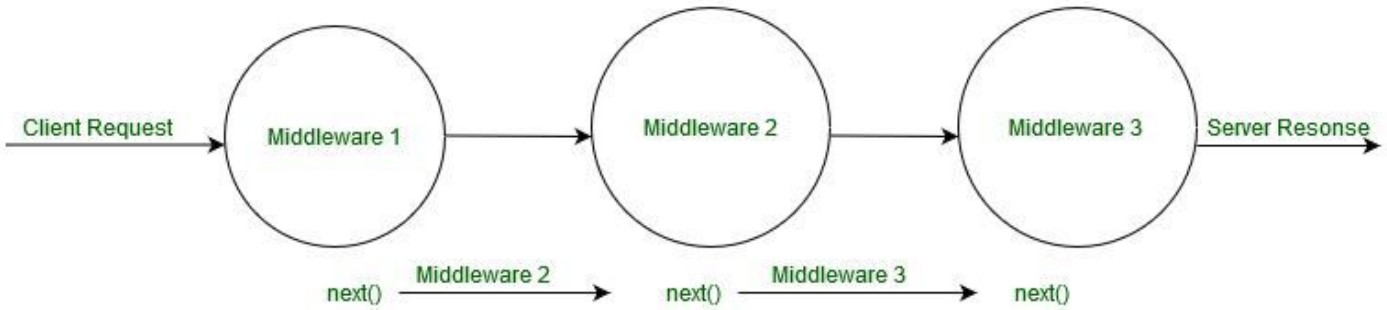
```
const express = require('express');
const userRouter = require('./userRouter'); // Chemin du module de route
const app = express();
app.get('/', (req, res) => {
   res.send('Home');
});
app.use('/users', userRouter);
app.listen(300,()=>console.log("listening on port 3000"))
```

Avantages des Routes Modulaires

- **Meilleure Organisation** : Les routes modulaires favorisent une meilleure organisation du code en regroupant des fonctionnalités liées dans des fichiers séparés.
- **Facilité de Maintenance** : Les modifications et les ajouts de fonctionnalités peuvent être effectués plus facilement car chaque module de route est autonome.
- **Réutilisabilité** : Vous pouvez réutiliser des modules de route dans différentes parties de votre application ou même dans d'autres projets.
- **Lisibilité** : La lisibilité du code est améliorée car chaque fichier se concentre sur une tâche ou une fonctionnalité spécifique.
- **Évolutivité** : Les applications complexes peuvent évoluer de manière plus saine en utilisant des modules de route, car ils facilitent l'ajout de nouvelles fonctionnalités sans affecter le reste de l'application.

les middlewares

- Ce sont des petits morceaux d'application qui rendent chacun un service spécifique.



les middlewares

- Les middlewares de base dans Express ne sont pas nombreux:
  - ***compression*** : permet la compression gzip de la page
  - ***cookie-parser*** : permet de manipuler les cookies
  - ***cookie-session*** : permet de gérer des informations de session (durant la visite d'un visiteur)
  - ***serve-static*** : permet de renvoyer des fichiers statiques contenus dans un dossier (images, fichiers à télécharger...)
  - ***serve-favicon*** : permet de renvoyer la favicon du site
  - etc.

les middlewares

- Ces middlewares sont interconnectés et peuvent communiquer entre eux en se renvoyant jusqu'à 4 paramètres
  - ***err***: les erreurs
  - ***req***: la requête du visiteur
  - ***res***: la réponse à renvoyer (la page HTML et les informations d'en-tête)
  - ***next***: un callback vers la prochaine fonction à appeler

les middlewares

```
1  var express = require('express');
2  var favicon = require('serve-favicon'); // Charge le middleware de favicon
3
4
5  var app = express();
6  // Indiquer que le dossier /public contient des fichiers
7  //statiques (middleware chargé de base)
8  app.use(express.static(__dirname + '/public'));
9
10 // Activer la favicon indiquée
11 app.use(favicon(__dirname + '/public/favicon.ico'));
12
13 // Répondre a la requete
14 app.use(function(req, res){
15     res.send('Hello');
16 });
17
18 app.listen(3000);
```

Créer un Middleware

- Dans Express.js, un middleware est simplement une fonction avec trois arguments : ***req*** (la requête), ***res*** (la réponse), et ***next*** (la fonction à appeler pour passer à la fonction middleware suivante).
- Un middleware peut effectuer des opérations sur la requête et la réponse, et peut appeler ***next()*** pour passer la main au middleware suivant.

```
const monMiddleware = (req, res, next) => {
   // Effectuez des opérations sur la requête ici
   console.log('Middleware fonctionne !');
   next(); // Appel à next pour passer au middleware suivant
};
```

Utiliser un Middleware

- **Application-Level Middleware** : Ces middlewares s'appliquent à chaque requête dans l'application

```
app.use(monMiddleware);
```

Utiliser un Middleware

- **Route-Level Middleware** : Ces middlewares s'appliquent uniquement aux routes spécifiques où ils sont définis.

```
app.get('/route', monMiddleware, (req, res) => {
    // Gestionnaire de route
});
```

Utiliser un Middleware

- **Error-Handling Middleware** : Ces middlewares sont utilisés pour gérer les erreurs.

```
app.use((err, req, res, next) => {
    // Gestion des erreurs
});
```

Utilisation Pratique des Middlewares

Authentification : Vérifiez l'authenticité des utilisateurs avant de permettre l'accès aux ressources protégées.

```
const vérifierAuthentification = (req, res, next) => {
    // Vérification de l'authentification
    if (utilisateurAuthentifié) {
        next();
    } else {
        res.status(401).send('Non autorisé');
    }
}
```

Utilisation Pratique des Middlewares

Journalisation : Enregistrez les détails des requêtes pour des raisons de suivi et de débogage.

```
const journalisation = (req, res, next) => {
    console.log(`Requête ${req.method} sur ${req.url}`);
    next();
};
```

Utilisation Pratique des Middlewares

- Body-Parser est un middleware pour Express.js qui:
  - analyse le corps des requêtes entrantes dans différents formats
  - expose ces données à travers l'objet req.body de l'application Express.
- Il peut gérer divers types de données, y compris:
  - les données JSON,
  - les données de formulaire,
  - les données de requête de l'URL (query parameters)
  - d'autres types de données.

Le middleware Body-parser

- Installer body-parser
  - $ npm i body-parser
- Importer et utiliser le module en tant que middleware.

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();

// Utilisation de Body-Parser Middleware
app.use(bodyParser.json()); // Pour les données JSON
app.use(bodyParser.urlencoded({ extended: true })); // Pour les données de formulaire
```

Le middleware Body-parser

- Accéder aux données de la requête dans vos routes Express. Les données analysées sont disponibles dans req.body

```
app.post('/exemple', (req, res) => {
   const jsonData = req.body;
   // Faites quelque chose avec jsonData
   res.send('Données reçues : ' + JSON.stringify(jsonData));
 });
```

Avantages des Middlewares

- **Modularité** : Les middlewares favorisent la modularité en permettant de découper l'application en petites fonctions réutilisables.
- **Gestion des Requêtes** : Ils offrent un contrôle total sur les requêtes entrantes, permettant des validations, des transformations et des filtrages.
- **Réutilisabilité** : Les middlewares peuvent être réutilisés dans différentes parties de l'application, garantissant la cohérence dans le traitement des requêtes.

Les Sessions

- Les sessions sont un mécanisme essentiel dans le développement web permettant de maintenir l'état de l'utilisateur entre les requêtes.
- Express.js offre une gestion de session simple et efficace grâce à différents middlewares et à au module express-session.
- Pour installer ce module :
    - $ npm i express-session

Les Sessions: Configuration

- Dans l'application (par exemple, app.js), il faut configurer express-session en tant que middleware.
- Il faut également spécifier un secret pour signer les cookies de session, ce qui est essentiel pour la sécurité.

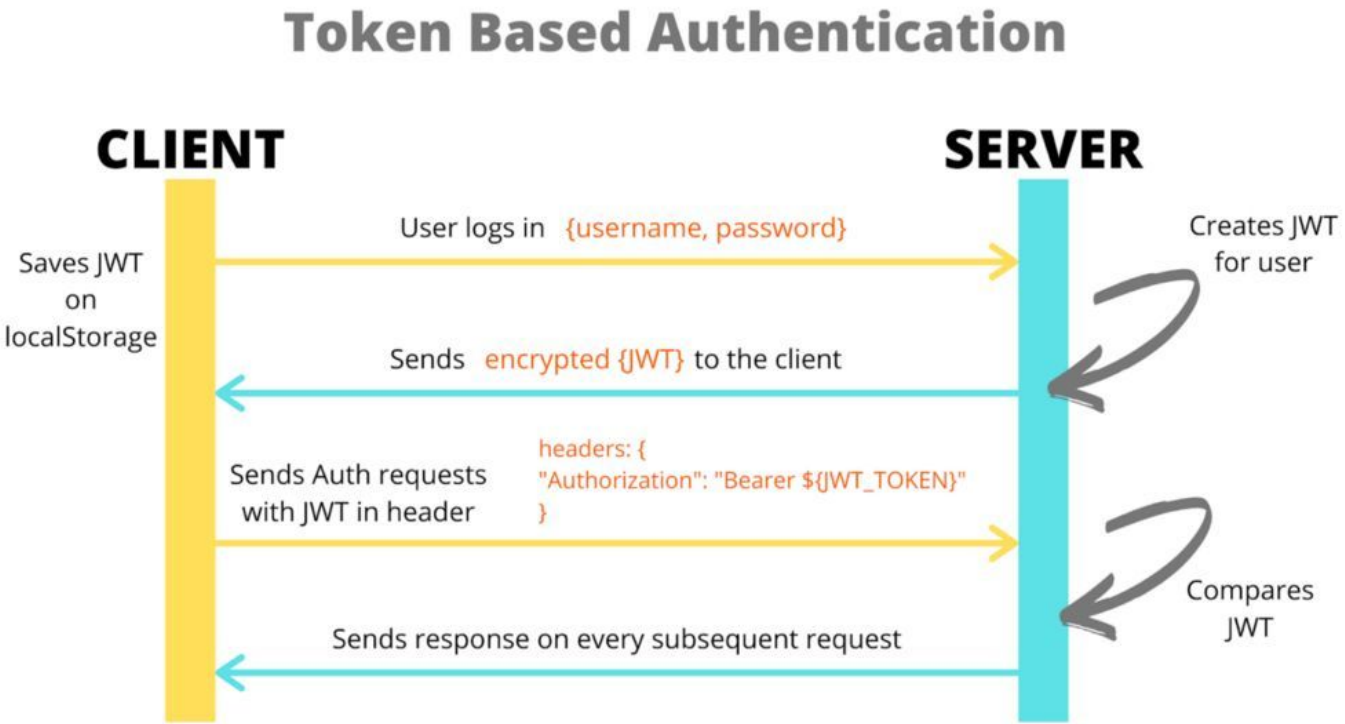
```
const express = require('express');
const session = require('express-session');
const app = express();
app.use(session({
   secret: 'votre_secret',
   resave: false,
   saveUninitialized: true,
   cookie: { maxAge: 60*60000 } // Durée de vie de la session en millisecondes (ici, 1 huere)
}));
```

Les Sessions : Utilisation

- Une fois configurées, les sessions sont accessibles via l'objet req.session. Vous pouvez y stocker des données liées à l'utilisateur.

```
app.get('/login', (req, res) => {
   // Vérifiez les informations d'authentification de l'utilisateur
   // Si les informations sont correctes, enregistrez l'utilisateur dans la session
   req.session.utilisateur = utilisateur; // utilisateur est l'objet de l'utilisateur
   res.send('Connecté avec succès');
});
app.get('/profil', (req, res) => {
   // Accédez aux données de l'utilisateur à partir de la session
   const utilisateur = req.session.utilisateur;
   if (utilisateur) {
       res.send('Bienvenue sur votre profil, ' + utilisateur.nom);
   } else {
       res.send('Non connecté');
   }
});
```

Json Web Token



https://www.freecodecamp.org/news/how-to-sign-and-validate-json-web-tokens/

JWT: signature

```
…..
const jwt=require("jsonwebtoken")
//const TOKEN_SECRET= require('crypto').randomBytes(64).toString('hex')
// console.log(TOKEN_SECRET)
const TOKEN_SECRET="un secret"
app.post('/api/login', (req, res) => {
  Const token=jwt.sign(
                              { username: req.body.username },
                              TOKEN_SECRET,
                              { expiresIn: '24h' }
                          );
  res.json(token);
}
```

JWT: verification

```
const jwt = require('jsonwebtoken');
function verifyToken(req, res, next) {
 const authHeader = req.headers['authorization'] //req.header("Authorization")
 const token = authHeader && authHeader.split(' ')[1]
 if (token == null) return res.sendStatus(401)
 jwt.verify(token,TOKEN_SECRET, (err, user) => {
   if (err) return res.sendStatus(403)
   req.user = user
   next()
 })
}
```

```
app.get("/api/products", verifyToken,(req,res)=>{
   res.status.send(req.use)
})
```

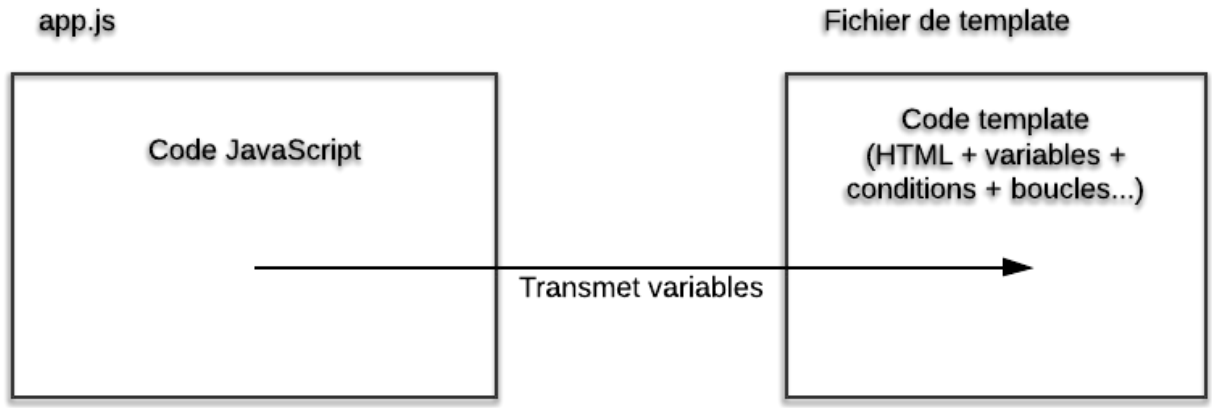
Les templates

- Envoyer un contenu Html en utilisant cette méthode n’est pas une tâche agréable
- Les templates permettent de produire du HTML et d'insérer au milieu du contenu, des variables (javascript).
- Il existe beaucoup de langages de templates: Mustache, Underscore, EJS, JSP, HandlebarsJS, pug (Jade)...

```
res.write('<!DOCTYPE html>'+
'<html>'+
'    <head>'+
'        <meta charset="utf-8" />'+
'        <title>Ma page Node.js !</title>'+
'    </head>'+
'    <body>'+
'       <p>Voici un paragraphe <strong>HTML</strong> !</p>'+
'    </body>'+
'</html>');
```

Les templates

- Principe:
  - depuis un fichier JavaScript, on appelle une template en lui transmettant les variables dont il a besoin pour construire la page.



PUG

1. Pug est un moteur de templates implémenté en JavaScript
2. Il permet de générer dynamiquement du HTML
3. Il est minimaliste et basé sur les indentations.
4. Pug était auparavant connu sous le nom de Jade

Pug: Exemple

- Installer ***pug*** dans le dossier de votre projet
  - $npm install pug
  - $npm install pug-cli -g
- Premier execution de pug
  - $pug index.pug -w -P
- Dans l'application express il faut définir le moteur de templates utilisé et le chemin d'accès aux templates
  - `app.set("view engine", "pug");`
  - `app.set("views", "path/to/views");`
- Finalement on appel la méthode *render(template,params)* de l'objet *response*

Pug: Exemple

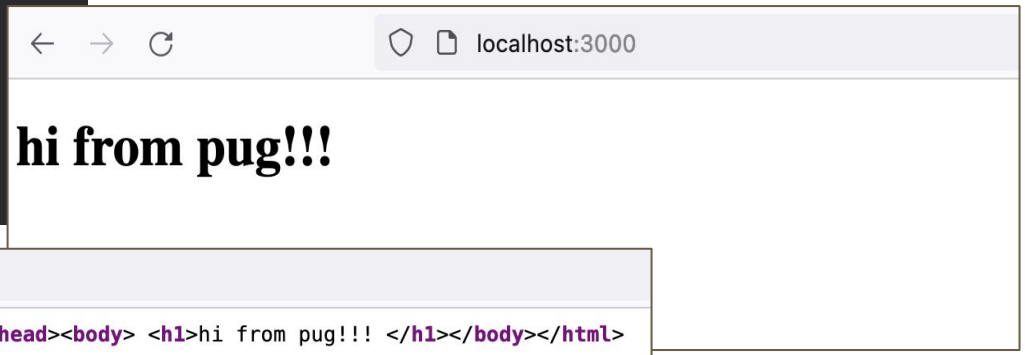
app.js

```
const { application } = require("express")
const express=require("express")

const app=express()
app.set("view engine","pug")
app.set("views","./templates")
app.use((req,res)=>{
    let data={
        name: "pug"
    }
    res.render("index",data)
})
app.listen(3000);
```

templates/index.pug

```
doctype html
html
    head
        title my #{name} test
    body
        h1 hi from #{name}!!!
```



Classes et ID

- Pour attribuer une classe à un élément html, on utilise le “.”
  - *h1.titre … => <h1 class=”titre”> … </h1>*
- Pour attribuer un ID à un élément html, on utilise le “#”
  - *h1#titre … => <h1 id=”titre”> … </h1>*
- *Si on ne précise pas l’élément Html pug utilise la balise *
  - *.titre … => … *
  - *#titre … => … *
- *Pour ajouter un attribut à un élément on utilise les parenthèses “(attribut=valeur)”*
  - *img.garde(src=”logo.png”, alt=”mon logo”) => *

Texte

- On peut ajouter un texte à une template par plusieurs façons:

```
p Pug rocks!
P
    | You are logged in as
    | user@example.com
P.
    Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
    incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
    exercitation ullamco laboris nisi ut aliquip ex ea commodo consequa
```

Commentaires

- On peut ajouter des commentaires à une template par plusieurs façons:

```
// My wonderful navbar
nav#navbar-default
//- My wonderful navbar (silent comment)
nav#navbar-default
//
    My wonderful navbar
    It is just so, awesome!
nav#navbar-default
nav#navbar-default // My wonderful navbar
```

Javascript dans pug

- Une caractéristique importante de pug est la possibilité d'exécuter du javascript dans ses templates.
- On peut insérer des variables,des objets, des boucles, des conditions ...ect
- Pour utiliser js dans pug il faut distinguer entre
    - **Buffered code** (le code tamponné) :
    - **Unbuffered code** (le code non tamponné)

unbuffered et buffered code

- unbuffered code (le code tamponné) :
  - commence par (-)
  - N'ajoute rien à la sortie, mais ses valeurs peuvent être utilisé dans le reste de la template
- buffered code (le code non tamponné)
  - Commence par (=)
  - Le javascript est évalué et rendu en sortie

```
//-unbuffered code
- const name = "said"
//- On peut maintenant utiliser name dans le reste de la template pug
//-buffered code
p= 'Hi my name is: ' + name
p= 'my age is: ' + 2*10 +'years old'
//-Pour des raison de securité le buffered code est échappé (escaped)
p= '<script>alert("Hi")</script>'
//- Donne en sortie:  <p>&lt;script&gt;alert(&quot;Hi&quot;)&lt;/script&gt;</p>p>
```

Interpolation

- L’interpolation est le processus de remplacer des gabarits (placeholders) par des valeurs des expressions js.
- L’interpolation se fait par plusieurs méthodes:
  - Le buffered code (déjà vu)
  - L’utilisation de #{expression js}

```
- const name = "said"
p Hi #{name}
p Hi #{name.toUpperCase()}
p Hi #{name.charAt(0).toUpperCase() + name.slice(1)}
//on peut omettre #{} pour affecter la valeur d'une var à un attribut d'un élément
img(src="portrait.png", alt=name)
```

Les boucles

- Pour parcourir les élément d'un tableau dans pug on utilise le mot clé ***each***

```
-const days=["lundi", "mardi", "mercredi", "jeudi", "vendredi",
"vendredi", "samedi", "dimanche"]
ol
    each day in days
        li= day
    else
        li le tableau est vide
```

Les boucles

- On peut aussi parcourir les éléments d’un objet

```
-
    const employee = {
        'name': 'said',
        'email': 'said@upm.ma',
        'age':21
    }
ul
each value, key in employee
    li= `${key}: ${value}`
each l’objet est {}
```

Conditionnels

- Les conditionnels offrent un moyen très pratique de rendre différents HTML en fonction du résultat d'une expression JavaScript :

```
-
   const student = {
        'name': 'said',
        'email': 'said@upm.ma',
        'age':12
   }
if student.age<11
   p kid
else if student.age<15
   p junior
else
   p senior
```