plb consultant	PLB Consultant Formation React	
Période : Du 06 au 08 Juin 2022		
Formateur : Mehdi Mtir		

Atelier : Développement d'une application isomorphe

- **Activité pratique 1 : Création d'une simple application via create-react-app pour un rendu côté serveur : React SSR (Server Side Render)**

Le rendu côté serveur (SSR) restitue une SPA (application d'une seule page) sur le serveur et l'envoie ensuite au client. Le client interprète ensuite les pages et les affiche sur le frontend, après quoi le contrôle de l'application est repris par le framework.

Avantages :

- Chargement initial rapide de l'application : Le SSR envoie une application entièrement chargée (paquets HTML) à partir du serveur lui-même afin que la puissance de calcul du serveur compense la puissance de traitement utilisée par le client.
- Visibilité de l'application de la part des robots d'exploration pour l'optimisation des moteurs de recherche (référencement).

On se propose dans cette activité d'examiner le rendu côté serveur et comment il peut être utilisé au niveau d'une simple application React.

- Créer alors un nouveau projet react via create-react-app intitulé : react-ssr-app1

```
npx create-react-app react-ssr-app
```

- Au niveau du fichier App.js, modifier le code de sorte à afficher le simple message "Hello from App !".

```
import React from 'react';

const App = () => {
  return (
    <div className="app">
      <h2>Hello from app!</h2>
    </div>
  );
}
export default App;
```

- Au niveau du fichier index.js, remplacer la fonction render() par hydrate().

```
//...
ReactDOM.hydrate(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
...//
```

Hydrate : fonction qui garantit que toutes les fonctionnalités restent intactes en attachant des écouteurs d'événements au balisage déjà existant afin que les robots des moteurs de recherche puissent indexer le balisage pré-chargé depuis le serveur.

- Installer maintenant express via l'instruction :

```
> npm install express
```

Express : framework web construit sur le dessus Nodejs pour fournir des méthodes utilitaires HTTP et des middlewares et permettre à un serveur d'être construit avec seulement quelques lignes de code.

- Ensuite, installer babel

```
npm install @babel/register @babel/preset-env @babel/preset-react ignore-
styles
```

Babel : pour transpiler notre code JSX local
ignore-styles : pour ignorer les styles importés dans notre environnement de nœuds.

- Restructurer votre code du côté serveur en créant un dossier nommé ***server*** au niveau du répertoire racine avec les deux fichiers à l'intérieur :
    - index.js : le point d'entrée
    - server.js : le code source relatif au serveur
- Au niveau du ***server.js***

```
const express=require('express');
const PORT = 8080;
const app = express();

const router = express.Router()

router.use('^/$', serverRenderer)
app.use(router)

app.listen(PORT, () => {
  console.log(`react app running on port ${PORT}`)
})
```

La configuration du serveur se fait dans l'environnement node en utilisant express :

- sélectionner un port (tant qu'aucun processus n'est déjà en cours d'exécution sur ce port)
- appeler ***express()*** pour initialiser une application express.
- utiliser la méthode du routeur pour configurer un itinéraire simple et dire à l'application d'écouter ce port.

- L'étape suivante consiste à rendre le fichier App.js se lancer à partir du serveur : Pour ce faire, on aura besoin de :
  - importer ce fichier
  - des modules fs et path pour gérer les manipulations de fichiers,
  - react et react-dom pour gérer l'application React
  - créer une fonction simple qui prend les arguments : (req, res et next). L'objectif est de créer un fichier HTML statique dans le dossier de build et de pouvoir le lancer par la suite sur le serveur à l'aide de l'application express.
  - Appeler la méthode renderToString() au niveau du ReactDOMServer pour restituer l'intégralité app.js sous forme de chaîne sur le serveur à l'itinéraire créé à l'aide de router.use().

D'où le code du server.js prendra cette allure :

```
const path=require('path');
const fs=require('fs');
const express=require('express');
const React=require('react');
const ReactDOMServer=require('react-dom/server');
import App from '../src/App'

const PORT = 8080
const app = express()
const router = express.Router()

const serverRenderer = (req, res, next) => {
  fs.readFile(path.resolve('./build/index.html'), 'utf8', (err, data) => {
    if (err) {
      console.error(err)
      return res.status(500).send('An error occurred')
    }
    return res.send(
      data.replace(
        '<div id="root"></div>',
        `<div  id="root">${ReactDOMServer.renderToString(<App  />)}</div>`
      )
    )
  })
}
router.use('^/$', serverRenderer)
```

```
router.use(
  express.static(path.resolve(__dirname, '..', 'build'))
)

app.use(router)

app.listen(PORT, () => {
  console.log(`SSR running on port ${PORT}`)
})
```

- Au niveau de server/index.js

```
require('ignore-styles')

require('@babel/register')({
  ignore: [/(node_modules)/],
  presets: ['@babel/preset-env', '@babel/preset-react']
})

require('./server')
```

- Il ne reste maintenant qu'à tester l'application sur localhost :8080 :

```
npm run build
node server/index.js
```

Remarque :

- En cas de problème, vérifier le fichier package.json qui devrait contenir les dépendances suivantes (sans oublier de taper la commande : npm i pour le mettre à jour) :

```
 {
 ..//
  "dependencies": {
    "@babel/preset-env": "^7.6.3",
    "@babel/preset-react": "^7.6.3",
    "@babel/register": "^7.6.2",
    "express": "^4.17.1",
    "ignore-styles": "^5.0.1",
    "isomorphic-fetch": "^2.2.1",
    "react": "^16.11.0",
    "react-dom": "^16.11.0",
    "react-router-dom": "^5.1.2",
    "react-scripts": "3.2.0",
    "serialize-javascript": "^2.1.0"
  }
  ..//
}
```

- **Activité pratique 2 : Développement d'une application isomorphe avec recherche de données à partir d'une API.**

---

1. Configuration du projet :

- Créer un nouveau dossier intitulé : react-router-ssr
- Ouvrir le dossier à partir de votre éditeur
- Afin de créer et initier un projet npm, au niveau du terminal, taper :

```
npm init –y
```

- Créer au niveau de la racine du projet :
  - Un dossier src
  - Les dossiers : src/browser, src/shared, src/server
  - le fichier : .babelrc
  - le fichier : webpack.config.js
- Installer les dépendances suivantes :

```
npm i -D babel-core babel-loader@7 babel-preset-env babel-preset-react
babel-plugin-transform-object-rest-spread webpack webpack-cli webpack-
node-externals nodemon

npm i react react-dom react-router-dom isomorphic-fetch serialize-
javascript express cors
```

- Au niveau du fichier package.json, vous ajoutez la ligne :

```
"start": "webpack && nodemon server.js"
```

- Au niveau du fichier : .babelrc

```
{
  "presets": [
    "env",
    "react"
  ],
  "plugins": [
    "transform-object-rest-spread"
  ]
}
```

- Au niveau du fichier webpack.config.js :

```
const  path  =  require('path')
const webpack = require('webpack')
const nodeExternals = require('webpack-node-externals')

const browserConfig = {
  mode : 'development',
  entry: './src/browser/index.js',
  output: {
    path: path.resolve(__dirname, 'public'),
```

```
    filename: 'bundle.js',
    publicPath: '/'
  },
  module: {
    rules: [
      { test: /\.(js)$/, use: 'babel-loader' },
    ]
  },
  plugins: [
    new webpack.DefinePlugin({
      __isBrowser__: "true"
    })
  ]
}

const serverConfig = {
  mode : 'development',
  entry: './src/server/index.js',
  target: 'node',
  externals: [nodeExternals()], // ignores node_modules when bundling in
Webpack
  output: {
    path: __dirname,
    // this file is the server entry,
    filename: 'server.js',
    publicPath: '/'
  },
  module: {
    rules: [
      { test: /\.(js)$/, use: 'babel-loader' }
    ]
  },
  plugins: [
    new webpack.DefinePlugin({
      __isBrowser__: "false"
    })
  ]
}
module.exports = [browserConfig, serverConfig]
```

2. Rendu côté serveur : React SSR

A ce niveau, on se propose d'afficher un rendu contenant un simple message "Hello World" du côté serveur.
Pour ce faire,

- Créer un fichier : src/shared/App.js

```
import React, { Component } from 'react';

class App extends Component {
  render() {
    return (
      <div>
        Hello World
      </div>
    );
```

```
  }
}
export default App;
```

- Créer également un nouveau fichier : src/server/index.js

```
import express from "express"
import cors from "cors"

const app = express()

app.use(cors())
```

- Afin de pouvoir servir les fichiers statiques dans express, nous passerons le répertoire public comme argument à cette instruction :

```
app.use(express.static("public"))

app.listen(3000, () => {
  console.log(`Server is listening on port: 3000`)
})
```

- Avec l'intégration du code HTML, le code source aurait l'allure suivante :

```
import express from "express"
import cors from "cors"
import React from 'react'
import { renderToString } from "react-dom/server"
import App from "../shared/App"

const app = express()

app.use(cors())


app.use(express.static("public"))

app.get("*", (req, res, next) => {
  const markup = renderToString(<App />)
  res.send(`
    <!DOCTYPE html>
    <html>
      <head>
        <title>SSR with RR</title>
        <script src="/bundle.js" defer></script>
      </head>

      <body>
        <div id="app">${markup}</div>
      </body>
    </html>
  `)
})

app.listen(3000, () => {
  console.log(`Server is listening on port: 3000`)
```

```
})
```

- Il reste à créer maintenant le fichier : src/browser/index.js

```
import React from 'react'
import { hydrate } from 'react-dom'
import App from '../shared/App'

hydrate(
  <App />,
  document.getElementById('app')
);
```

- En lançant maintenant : npm start, vous pouvez consulter sur localhost:3000 le rendu serveur : "Hello World".

3. Passage des propriétés : props

- Ajouter la propriété data contenant la valeur 'server' au niveau du fichier : src/server/index.js.

```
  const markup = renderToString(
    <App data='server' />
  )
```

- Côté client, au niveau du fichier : src/browser/index.js, mettre la valeur 'client' au niveau de cette propriété.

```
hydrate(
  <App data='client' />,
  document.getElementById('app')
);
```

En actualisant l'application, on devrait normalement afficher d'abord le message "Hello world, server" (rendu de la part du serveur), puis le message "Hello world, client".

4. Partage des données initiales du serveur au client

React attend que le contenu rendu soit identique entre le serveur et le client. Il peut corriger les différences de contenu textuel, mais vous devez traiter les décalages comme des bogues et les corriger. En mode développement, React met en garde les décalages lors de l'hydratation. Il n'y a aucune garantie que les différences d'attributs seront corrigées en cas de disparités. Ceci est important pour des raisons de performances car dans la plupart des applications, les décalages sont rares, et donc valider tout le balisage serait prohibitif.

Pour partager les données initiales du serveur avec le client, nous faisons recours à l'objet window comme suit :

- Au niveau du fichier : src/server/index.js

```
// ...
import serialize from "serialize-javascript"

app.get("*", (req, res, next) => {
  const name = 'Anis'
  const markup = renderToString(
    <App data={name}/>
  )

  res.send(`
    <!DOCTYPE html>
    <html>
      <head>
        <title>SSR with RR</title>
        <script src="/bundle.js" defer></script>
        <script>window.__INITIAL_DATA___= ${serialize(name)}</script>
      </head>

      <body>
        <div id="app">${markup}</div>
      </body>
    </html>
  `)
})
```

- Côté client, nous pouvons récupérer maintenant les données via : `window.__INITIAL_DATA__`.

```
src/browser/index.js

hydrate(
  <App data={window.__INITIAL_DATA__} />,
  document.getElementById('app')
);
```

5. Recherche de données à partir d'une API

- Créer le fichier : src/shared/api.js

```
import fetch from 'isomorphic-fetch'

export default function fetchPopularRepos(language = 'all') {
  const encodedURI =
encodeURI(`https://api.github.com/search/repositories?q=stars:>1+languag
e:${language}&sort=stars&order=desc&type=Repositories`)

  return fetch(encodedURI)
    .then((data) => data.json())
    .then((repos) => repos.items)
    .catch((error) => {
      console.warn(error)
      return null
    });
}
```

- Faire appel à la fonction fetchPopularRepos au niveau du fichier : src/server/index.js

```
// ...
import fetchPopularRepos from "../shared/api"
// ...
app.get("*", (req, res, next) => {
  fetchPopularRepos()
    .then((data) => {
      const markup = renderToString(
        <App data={data} />
      )

      res.send(`
        <!DOCTYPE html>
        <html>
          <head>
            <title>SSR with RR</title>
            <script src="/bundle.js" defer></script>
            <script>window.__INITIAL_DATA___=
${serialize(data)}</script>
          </head>

          <body>
            <div id="app">${markup}</div>
          </body>
        </html>
      `)
    })
})
```

- Créer maintenant le composant : src/shared/Grid.js qui servira pour l'affichage des données récupérées.

```
import React, { Component } from 'react'

class Grid extends Component {
  render() {
    const repos = this.props.data

    return (
      <ul style={{ display: 'flex', flexWrap: 'wrap' }}>
        {repos.map(({ name, owner, stargazers_count, html_url }) => (
          <li key={name} style={{ margin: 30 }}>
            <ul>
              <li><a href={html_url}>{name}</a></li>
              <li>@{owner.login}</li>
              <li>{stargazers_count} stars</li>
            </ul>
          </li>
        ))}
      </ul>
    )
  }
}
export default Grid
```

- Mettre à jour le fichier : src/shared/App.js afin d'importer le nouveau composant au niveau de l'application.

```
import React, { Component } from 'react'
import Grid from './Grid'

class App extends Component {
  render() {
    return (
      <div>
        <Grid data={this.props.data} />
      </div>
    )
  }
}

export default App
```

En relançant l'application, on devrait afficher le rendu des données récupérées auprès de l'API github.

6. Routage

React Router est une approche déclarative et basée sur les composants du routage. Cependant, lorsque nous avons affaire au rendu côté serveur avec React Router, nous devons abandonner ce paradigme et déplacer toutes nos routes vers une configuration de route centrale. La raison en est que le client et le serveur doivent être conscients des itinéraires. On devrait savoir du côté client quels composants rendre lorsque l'utilisateur navigue dans notre application et le serveur, de son côté devrait savoir quelles données récupérer lorsque l'utilisateur demande un chemin spécifique.

Pour ce faire, on passera par les étapes suivantes :
- récupérer les données sur le serveur en fonction de l'itinéraire demandé par l'utilisateur
- rendre les routes côté client
- passer les props par contexte, accéder aux données via props.staticContext
- récupérer des données côté client
- tenir compte des mises à jour des données lors de la navigation côté client

6.1. Recupération des données côté serveur

- Créer le fichier : src/shared/Home.js qui contiendra le code suivant :

```
import React from 'react'

export default function Home () {
  return (
    <div>
      Select a Language
    </div>
  )
}
```

- Créer également le fichier : src/shared/routes.js

```
import Home from './Home'
import Grid from './Grid'
import { fetchPopularRepos } from './api'

const routes = [
  {
    path: '/',
    exact: true,
    component: Home,
  },
  {
    path: '/popular/:id',
    component: Grid,
    // pass a fetchInitialData attribute -> to fetch initial data by
using the 'id' parameter from the URL
    fetchInitialData: (path = '') =>
fetchPopularRepos(path.split('/').pop())
  }
]
export default routes
```

Faites attention à la propriété fetchInitialData lorsqu'un utilisateur fait une demande GET avec ce chemin depuis le serveur, nous irons de l'avant et invoquerons fetchInitialData en lui passant le chemin. Le résultat est une promesse qui sera finalement résolue avec les données à rendre.

- Au niveau du fichier : src/server/index.js

```
// ...
app.get("*", (req, res, next) => {

  const activeRoute = routes.find((route) => matchPath(req.url, route))
|| {}
  // check if the activeRoute has a fetchInitialData property
  const promise = activeRoute.fetchInitialData
    ? activeRoute.fetchInitialData(req.path)
    : Promise.resolve()

  promise.then((data) => {
    const markup = renderToString(
      <App data={data} />
    )

    res.send('...')
  }).catch(next)
})
```

- Maintenant, au niveau de notre navigateur (localhost:3000/popular/javascript) le serveur affiche la page correcte avec les données demandées.

En actualisant localhost:3000, vous allez voir une erreur qui s'affiche : (error Cannot read property 'map' of undefined).

Ces données ne se dirigent pas vers la page et donc, les données props ne sont pas définies au niveau de Grid.
Afin d'en remédier :

6.2. Rendre les routes côté client et transmettre les données globales au client

- Tout d'abord, utilisez BrowserRouter de React Router pour envelopper l'application de navigateur.

```
// src/browser/index.js
import React from 'react'
import { hydrate } from 'react-dom'
import App from '../shared/App'
import { BrowserRouter } from 'react-router-dom'

hydrate(
  <BrowserRouter>
    <App data={window.__INITIAL_DATA__} />
  </BrowserRouter>,
  document.getElementById('app')
);
```

- Tout comme BrowserRouter du côté client, on utilisera StaticRouter au niveau du fichier : ***src/server/index.js***

```
//...
import { StaticRouter, matchPath } from "react-router-dom"
//...

const markup = renderToString(
  <StaticRouter location={req.url} context={{}}>
    <App data={data}/>
  </StaticRouter>
)
//...
```

Ensuite, transmettez aux composants rendus par React Router la propriété fetchInitialData si elle existe afin que le client puisse également l'invoquer s'il ne dispose pas déjà des données du serveur.

- Le fichier : ***src/shared/App.js*** s'écrira alors comme suit :

```
import React, { Component } from 'react'
import Grid from './Grid'
import routes from './routes'
import { Route } from 'react-router-dom'

class App extends Component {
  render() {
    return (
      <div>
        {/* <Grid data={this.props.data} /> */}
        {routes.map(({ path, exact, component: C, ...rest }) => (
```

```
          <Route key={path} path={path} exact={exact} render={(props) =>
(<C {...props} {...rest} />)} />
        ))}
      </div>
    )
  }
}

export default App
```

- Créer un nouveau composant : ***src/shared/Navbar.js***

```
import React from 'react'
import { NavLink } from 'react-router-dom'

export default function Navbar () {
  const languages = [{
    name: 'All',
    param: 'all'
  }, {
    name: 'JavaScript',
    param: 'javascript',
  }, {
    name: 'Ruby',
    param: 'ruby',
  }, {
    name: 'Python',
    param: 'python',
  }, {
    name: 'Java',
    param: 'java',
  }]

  return (
    <ul>
      {languages.map(({ name, param }) => (
        <li key={param}>
          <NavLink activeStyle={{fontWeight: 'bold'}}
to={`/popular/${param}`}>
            {name}
          </NavLink>
        </li>
      ))}
    </ul>
  )
}
```

- Créer également le composant : ***src/shared/NoMatch.js***

```
import React from 'react'
export default function NoMatch() {
  return (
    <div>Not found</div>
  )
}
```

- Mettre à jour le fichier : ***src/shared/App.js***

```
import React, { Component } from 'react'
import routes from './routes'
import { Route, Switch } from 'react-router-dom'
import Navbar from './Navbar'
import NoMatch from './NoMatch'

class App extends Component {
  render() {
    return (
      <div>
        <Navbar />

        <Switch>
          {routes.map(({ path, exact, component: C, ...rest }) => (
            <Route
              key={path}
              path={path}
              exact={exact}
              render={(props) => (
                <C {...props} {...rest} />
              )}
            />
          ))}
          <Route render={(props) => <NoMatch {...props} />} />
        </Switch>
      </div>
    )
  }
}
export default App
```

- Ajouter : this.props.data à data.

<C {...props} {...rest} data={this.props.data} />

6.3. Passage des props par context et accès aux données via props.staticContext

- Au niveau du fichier : ***src/server/index.js***

```
//...

promise.then((data) => {
  const context = { data }

  const markup = renderToString(
    <StaticRouter location={req.url} context={context}>
      <App />
    </StaticRouter>
  )

//...
```

- Changer le fichier : src/shared/Grid.js en remplaçant :
  const repos = this.props.data par const repos = this.props.staticContext.data

```
import React, { Component } from 'react'

class Grid extends Component {
  render() {
    const repos = this.props.staticContext.data

    return (
      <ul style={{ display: 'flex', flexWrap: 'wrap' }}>
        {repos.map(({ name, owner, stargazers_count, html_url }) => (
          <li key={name} style={{ margin: 30 }}>
            <ul>
              <li><a href={html_url}>{name}</a></li>
              <li>@{owner.login}</li>
              <li>{stargazers_count} stars</li>
            </ul>
          </li>
        ))}
      </ul>
    )
  }
}
export default Grid
```

6.4. Récupération des données côté client

L'application ne transmet plus ces données au composant Grid. Nous pouvons simplement les extraire de l'objet window à l'intérieur du composant Grid lui-même.
D'où :

- Au niveau de src/browser.index.js, la ligne :
  <App data={window.__INITIAL_DATA__}/> va être remplacée par : <App/>.

```
import React from 'react'
import { hydrate } from 'react-dom'
import App from '../shared/App'
import { BrowserRouter } from "react-router-dom"

hydrate(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
  document.getElementById('app')
);
```

- Au niveau du fichier : src/shared/Grid.js

```
import React, { Component } from 'react'
class Grid extends Component {
  constructor(props) {
    super(props)

    let repos
    // the global variable has been inject by webpack plugin
```

```
    // plugins: [new webpack.DefinePlugin({ _isBrowser_: "true" })]
    if (_isBrowser_) {
      repos = window._INITIAL_DATA__
      delete window.__INITIAL_DATA__
    } else {
      // get data from content of router
      repos = this.props.staticContext.data
    }
    this.state = {
      repos,
      loading: repos ? false : true,
    }

    this.fetchRepos = this.fetchRepos.bind(this)
  }
  componentDidMount() {
    if (!this.state.repos) {
      // when react takes over the page, fetch data in client
      this.fetchRepos(this.props.match.params.id)
    }
  }
  fetchRepos(lang) {
    this.setState(() => ({
      loading: true
    }))

    this.props.fetchInitialData(lang)
      .then((repos) => this.setState(() => ({
        repos,
        loading: false,
      })))
  }
  render() {
    const { repos, loading } = this.state

    if (loading === true) {
      return <p>LOADING</p>
    }

    return (
      <ul style={{ display: 'flex', flexWrap: 'wrap' }}>
        {repos.map(({ name, owner, stargazers_count, html_url }) => (
          <li key={name} style={{ margin: 30 }}>
            <ul>
              <li><a href={html_url}>{name}</a></li>
              <li>@{owner.login}</li>
              <li>{stargazers_count} stars</li>
            </ul>
          </li>
        ))}
      </ul>
    )
  }
}
export default Grid
```

On remarque que lors de la navigation d'un langage à un autre, la mise à jour ne se fait pas.
En effet, props a changé de valeur mais le composant n'est pas remonté. Pour faire face à ce problème, nous pouvons utiliser la méthode appropriée au cycle de vie de React : componentDidUpdate

6.5. Mise à jour des données côté client

- Au niveau du fichier : src/shared/Grid.js

```
componentDidUpdate (prevProps, prevState) {
  if (prevProps.match.params.id !== this.props.match.params.id) {
    this.fetchRepos(this.props.match.params.id)
  }
}
```

A ce niveau, veuillez vérifier que tout devrait fonctionner convenablement.