TIW8 - TP1 Mise en place Stack

Encadrants

- Aurélien Tabard (responsable)
- Louis Le Brun

Présentation du TP

L'objectif du TP est de mettre en place "l'enveloppe" d'une application Web avec un serveur Node/Express léger, et un framework TS côté client. Pour l'UE le client sera développé avec React, mais la "stack" que nous allons voir dans ce TP serait peu ou presque la même pour Angular ou Vue.

Nous allons voir :

- La mise en place d'un serveur Node/Express basique
- L'automatisation d'un build
- Comment configurer la transpilation du code Typescript et des composants TSX en code interprétable par n'importe quel navigateur
- Créer un projet React
- Créer deux composants React basiques
- Gérer le bundling avec Webpack
- Utiliser un linter pour vérifier votre code
- Assembler et servir le contenu
- Déployer via une image Docker sur GitLab Pages

Ce TP fera l'objet d'un premier rendu **individuel** et d'une note binaire (PASS/FAIL). Voir les critères d'évaluation en bas de la page.

Vous ferez le rendu sur la forge.

Initialisation du projet

Nous vous recommandons chaudement de vous servir d'un `version manager` pour NodeJS comme `Node Version Manager` NVM afin de pouvoir changer de version de Node facilement ou installer la dernière version `long term support (LTS)` avec `nvm install --lts`. Ce conseille s'applique à toutes les autres technologies peu importe le langage (venv pour python, sdkman! pour le sdk android, ...)

Créez un projet git sur la forge dès maintenant. Remplissez le champ Tomuss associé.

Installer Node dans sa dernière version `lts` et Express si ce n'est pas déjà fait. Si c'est le cas, pensez à les mettre à jour.

(Optionel) Selon votre OS, la version de node et d'Express que vous allez installer, il sera peut être nécessaire d'installer `express-generator` qui gère le "cli" d'Express (la possibilité de l'invoquer depuis la ligne de commande). Installez le globalement avec npm ou yarn.

À la racine de votre projet, créez deux dossiers:

- un dossier `server` avec un dossier `src` qui contiendra le code Nodejs + Express côté serveur
- un dossier `client` avec un dossier `src` qui contiendra le code React côté navigateur

Ce sont deux projets distincts.

Pensez régulièrement à ajouter les fichiers qui n'ont pas à être versionnés à votre .gitignore (*a minima* : node_modules & dist), vous. pouvez vous servir de gitignore.io pour en généré un via des tags (Visual Studio Code, nodejs, ...)

Poussez le projet sur la forge.

On va en premier configurer le serveur.

Partie serveur

Allez dans le dossier `server`.

Créez un projet node (`yarn init`), en le liant à votre dépôt Git sur la forge via le champs `repository` du `package.json`.

Pour pouvoir travailler avec Typescript, installez quelques dépendances supplémentaires:

```
# Ajoute typescript à votre projet
yarn add typescript ts-node express --dev

# Ajoute les types d'un module à typescript
yarn add @types/node @types/express

# Installe le compilateur Typescript de façon globale
```

```
npm i -g tsc

# Créé un fichier de configuration pour Typescript
tsc --init
```

Trouvez le fichier automatiquement généré `tsconfig.json`:

- Cherchez la ligne `outDir` et décommentez là pour mettre comme valeur `./dist`.
- Cherchez la ligne `rootDir` et décommentez là pour mettre comme valeur `./src`.
- Dans le fichier `package.json`, dans la partie `scripts`, mettre:

```
"scripts": {
  "start": "tsc && node dist/index.js"
}
```

Mise en place du serveur

Voici l'architecture du projet express que nous allons créer:

```
| - dist
| - src
| | - index.ts
| | - routes
|   | - hello.router.ts
| - package.json
| - yarn.lock
| - tsconfig.json
| - .gitignore
```

Dans notre `index.ts`, nous allons créer le serveur:

```
import express from 'express';
import { HelloRouteur } from './routes/hello.router';

const app = express();
const port = process.env.PORT || 3000;

app.listen(port, () => {
  process.stdout.write(`Server started on port: ${port}\n`);
});

app.use('/hello', HelloRouteur);
```

Dans `hello.router.ts`, nous allons créer un des routeurs de notre API:

```
import express from 'express';

const helloRouteur = express.Router();

// With middlewares you can ensure the user is authenticated
// before requesting secured API routes
helloRouteur.use((request, response, next) => {
  process.stdout.write('HelloRouter Middleware\n');
  if (request.ip.endsWith('127.0.0.1')) {
    process.stdout.write('Request from local IP\n');
    next();
  } else {
    next();
  }
});

helloRouteur.get('/', (request, response) => {
  response.send('Hello TIW8 !');
});
```

```
export {
  helloRouter as HelloRouteur
};
```

Testez votre serveur avec la commande `yarn start`

Vérifier que le serveur fonctionne et versionnez le sur la forge.

Projet React

Allez maintenant dans le dossier `client`.

Nous verrons plus en détail le fonctionnement de React lors de la prochaine séance. Pour le moment nous allons créer un projet simple.

Comme pour le projet serveur, il faut installer quelques dépendances avant tout:

```
yarn add typescript --dev
yarn add react-dom react @types/react-dom @types/react --dev
```

Dans le dossier `src`, créez un `index.html`. Ce sera le seul fichier HTML du projet, il sera "peuplé" dynamiquement par React.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1, maximum-scale=1"
    />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>React TP1</title>
```

```
  </head>
  <body>
    <!-- L'id de ce div est important -->
    <div id="root"></div>
  </body>
</html>
```

3 Dans le même dossier nous allons créer un premier composant React, on l'appellera `index.tsx` (l'extension de fichier est très importante):

```
import * as React from "react";
import * as ReactDOM from "react-dom";

const Index = () => {
  return <div>TIW 8 TP1!</div>;
};
ReactDOM.render(<Index />, document.getElementById("root"));
```

Générer un bundle avec Webpack

Mise en place de Webpack

Installer Webpack en dev (pas la peine d'avoir les dépendances pour le déploiement) :

- `webpack` (Le bundler)
- `webpack-cli` (Command Line Interface pour lancer les commandes webpack)

Installez également le module html-webpack-plugin pour faciliter la création de fichier HTML avec Webpack.

```
yarn add --dev ts-loader css-loader html-webpack-plugin mini-css-extract-plugin source-map-loader webpack webpack-
cli
```

Configuration de webpack

Même si les dernières versions de webpack peuvent fonctionner sans fichier de configuration (avec des défauts), vous aurez de toutes façons à spécifier une config dans ce TP. Mettez donc en place un fichier `webpack.config.js` avec une configuration minimale (entry, output), que vous allez modifier par la suite.

Dans ce fichier de configuration, pointez vers le point d'entée React (le fichier index.jsx) et indiquez ou l'appliquer (`template: "./client/index.html"`). Ci-dessous une partie de ce fichier, qui sera complétée par la suite :

```
const HtmlWebPackPlugin = require("html-webpack-plugin");
const path = require('path');
const htmlPlugin = new HtmlWebPackPlugin({
  template: "./client/index.html",
  filename: "./index.html"
});
module.exports = (env, argv) => {
  console.log(argv.mode);
  return {
    entry: "./client/index.tsx",
    output: {
      path: path.join(__dirname, 'dist'),
      filename: "bundle.js"
    },
    plugins: [htmlPlugin],
    module: {
      rules: [
        {
          ...
        }
      ]
    }
  };
};
```

Transpilation et configuration de Babel

React s'appuie "normalement" sur JSX pour lier la logique de rendu, la gestion d'évènement et les changements d'états pour un élément donné. Ces éléments seraient normalement séparés entre langages et technos différentes. Babel permet de traduire ce code (et au passage de transformer du ES6 en ES5).

Nous allons reprendre la même logique mais avec du Typescript (un fichier typescript simple se termine en .ts, nous allons créer des .tsx). JSX, Typescript ou TSX ne sont pas interprétés par les navigateurs, nous devons donc le "traduire" ou transpiler en HTML+JS pour que le code devienne compréhensible.

Nous avons normalement déjà installé les dépendances typescript, reste à installer un dépendance (de développement) pour l'intégration à Webpack: `ts-loader`

Configurez le transpileur Typescript à l'aide d'un fichier `tsconfig.json` à la racine de votre projet, en indiquant les pré-configurations utilisées pour le reste du projet.

```
{
  "compilerOptions": {
    "jsx": "react",
    "module": "commonjs",
    "noImplicitAny": true,
    "outDir": "./dist",
    "preserveConstEnums": true,
    "removeComments": true,
    "sourceMap": true,
    "target": "es5"
  },
  "include": ["client"],
  "exclude": ["node_modules", "**/*.spec.ts", "**/*.test.ts"]
}
```

Il faut spécifier à Webpack la transpilation des fichiers .ts et .tsx du projet lors du build. Cela se fait dans le fichier `webpack.config.js` :

```
module.exports = {
  module: {
```

```
    rules: [
      {
        test: /\.(ts|tsx)$/,
        exclude: /node_modules/,
        use: {
          loader: "ts-loader",
        },
      },
    ],
  },
};
```

Il faut aussi spécifier à votre bundler (Webpack) comment résoudre les liens vers les modules, il faut lui dire de quelles extensions de fichier rajouter et dans quel ordre les traiter. Pour cela on utilise la directive resolve de webpack.

```
  resolve: {
    extensions: [".js", ".jsx", ".json", ".ts", ".tsx"],
  },
```

Bundling

Il faut maintenant assembler le code React.

Dans `package.json` ajouter une commande `build` (dans `scripts`)

```
"scripts": {
  "build": "webpack --mode production"
}
```

Cette commande doit vous générer un fichier HTML et un fichier JS dans `dist`.

Servir le contenu

Il faut maintenant dire à Express où aller chercher le contenu. Pour cela il faut lui dire que sa route `/` est maintenant `dist/index.html`

Rajouter les constantes suivantes (selon vos noms de fichiers et de dossiers) :

```
const path = require("path");

const DIST_DIR = path.join(__dirname, "../dist");
const HTML_FILE = path.join(DIST_DIR, "index.html");

// TODO Modifier la route '/' pour qu'elle pointe sur HTML_FILE
```

Nous allons aussi rajouter la commande suivante dans `package.json` pour distinguer un build de dev et un de production.

```
  "dev": "webpack --mode development && node server/index.js",
```

Gérer les fichiers statiques

Pour que Express trouve plus tard son chemin "de base" et les fichiers statiques générés par Webpack (images, css...) rajouter la ligne suivante:

```
app.use(express.static(DIST_DIR));
```

Installez le module `file-loader` (toujours en dev).

Et rajoutez la règle suivante dans `webpack.config.js:` pour que webpack place les images dans un dossier `/static/`.

```

```

```
{
  test: /\.(png|svg|jpg|gif)$/,
  loader: "file-loader",
  options: { name: '/static/[name].[ext]' }
}
```

> Note: depuis Webpack 5, vous pouvez préférer Asset Modules qui évite d'utiliser un loader externe.

Il faudra les importer dans vos composants. Voici comment cela se fait au sein d'un composant React:

```
// Import de l'image
import LOGO from "./logo.png";

// Utilisation
<img src={LOGO} alt="Logo" />;
```

Pour que l'import marche, il faut spécifier à Typescript et Webpack de traiter les images comme des modules:

- Créer un fichier `index.d.ts` qui contient la définition des modules/types associé aux extensions de fichiers :

```
declare module '*.png';
declare module '*.jpg';
```

- Dans `tsconfig.json`, modifier la ligne d'include en rajoutant le fichier créé : `["client", "index.d.ts"],`

CSS

Nous allons utiliser une bibliothèque dérivée de Tailwind CSS qui fonctionne bien avec React: Windmill React UI. Cette bibliothèque fournit des composants, mais permet aussi de s'appuyer sur Tailwind pour la création de nouveaux composants/widgets.

Linting

Pour vérifier que votre code se conforme aux bonnes pratiques, nous allons utiliser eslint, et son plugin react.

Pour créer votre fichier de configuration `eslint` taper `yarn run eslint --init` votre configuration devrait ressembler à cela

```
✔ How would you like to use ESLint? · style
✔ What type of modules does your project use? · require
✔ Which framework does your project use? · react
✔ Does your project use TypeScript? · Yes
✔ Where does your code run? · browser
✔ How would you like to define a style for your project? · guide
✔ Which style guide do you want to follow? · standard
✔ What format do you want your config file to be in? · JSON
```

Vous pouvez tester eslint à la "main" avec

```
yarn run eslint client/*.tsx
```

Ajouter ensuite eslint à Webpack. Installez le module `eslint-webpack-plugin` en dev. Importez le dans votre webpack config et rajouter les lignes suivantes au blog plugin. eslint se lancera maintenant lors du build (vous pouvez rajouter une erreur dans votre index.tsx et tester le build).

```
  plugins: [
    ...,
    new ESLintPlugin({
      extensions: ["js", "jsx", "ts", "tsx"],
    }),
  ],
```

Déployer sur GitLab Pages

Afin de rendre notre application disponible sur le Web, nous allons la déployer sur le GitLab Pages de votre projet.

Depuis la page de votre projet GitLab, allez dans `settings > pages` et suivez les indications pour héberger le site.

N'oubliez pas de désactiver l'option `watch` de webpack si vous lancez Webpack en `--mode production` voir ici.

Des composants Reacts

Créer deux composants basiques sans aucune logique. Le premier affichera un titre. Le deuxième affichera des images prises dans un dossier statique. On placera ces composants dans un dossier `components`.

```
import React from "react";
import ReactDOM from "react-dom";
import Header from "./components/Header/index.jsx";
import Content from "./components/Content/index.jsx";
const Index = () => {
  return (
    <div className="container">
      <Header />
      <Content />
    </div>
  );
};
ReactDOM.render(<Index />, document.getElementById("root"));
```

React Developer Tools

Installez l'extension React Developer Tools dans votre navigateur préféré. Inspectez l'application.

Rendu et évaluation

Le TP est individuel. **Il est évalué sur une base binaire PASS/FAIL** et compte pour 10% de la note de Controle Continu (CC) totale. Il est à rendre pour dimanche 19 23h59.

Les critères d'évaluation sont les suivants pour avoir un PASS (=20), si un des critères n'est pas rempli c'est un FAIL (=0):

- Le rendu est effectué avant ce soir minuit. Pensez à remplir les deux champs Tomuss associés au TP1 (lien forge pour clone, et lien heroku).
- Les responsables de l'UE sont ajoutés au projet forge (le projet est clonable)
- Le lien vers la forge fournit sur Tomuss permet un `git clone` sans aucune modification
- Le projet ne contient que des éléments nécessaire (.gitignore est bien défini)
- `yarn run build` construit le projet
- `yarn run start` lance le serveur.
- `eslint` ne retourne pas d'erreur
- l'application Web est bien déployée sur Heroku au lien fournit dans le rendu