React 2

08/06/2023

Les routes - reactRouter

https://reactrouter.com/docs/en/v6

Npm i react-router-dom

Aller dans le fichier index.js

Et importer BrowserRouter

import { BrowserRouter } from 'react-router-dom’;

Et nous allons englober <App />

```
<React.StrictMode>
  <BrowserRouter>
    <App />
  </BrowserRouter>
</React.StrictMode>
```

Les routes - reactRouter

On va ensuite aller dans App.js et importer Routes et Route

import {Routes, Route } from "react-router-dom“

Dans   :

```
<Routes>
    <Route />
</Routes>
```

Les routes - reactRouter

Créons maintenant un component Home que nous placerons dans un dossier Home afin de l'afficher par la suite.

Home.js

```
return (
  <div>
    <h1>Bienvenue sur notre super site React !</h1>
  </div>
)
```

Les routes - reactRouter

Pour bien comprendre la subtilité des routes créons un nouveau dossier Services et un fichier services

Ajoutons la route au App.js :

<Route path="/services" element={<Services/>} />

Les routes - reactRouter

Ajoutons un chemin dynamique :

Créons un dossier Profile et un component Profile.js

Ajoutons la route au App.js :

<Route path="/profile/:id" element={<Profile/>} />

Les routes - reactRouter

On va pouvoir utiliser ce qui est passé en paramètre :

Dans Profile.js

```
import {useParams} from 'react-router-dom'
function Profile() {
    const params = useParams()
    console.log(params);
  return (
    <div>Bonjour {params.id}, voici votre profil </div>
  )
}
```

Les routes - reactRouter

On va voir maintenant comment faire des pages d’erreur 404 notamment

Créons un component NotFound.js dans le dossier NotFound

```
<Route path="/*" element={<NotFound/>} />
```

Les autres routes sont plus restrictif, donc c’est a eux que viennent la priorité d’affichage. Des que l’url entrer par l’utilisateur ne matchera avec aucune route. Alors le component NotFound s’affichera.

Vous trouverez sur le net des templates de pages 404 sur le net, faites vous plaisir.

La navigation

Créons un dossier Navbar et un fichier Navbar.js.

On va utiliser le Link de react-router-dom. Plus simple et plus facile à utiliser.

```
import React from 'react'
import {Link} from 'react-router-dom'
function Navbar() {
  return (
    <nav>
        <Link to="/">Accueil</Link>
        <Link to="/services">Services</Link>
    </nav>
  )
}
export default Navbar
```

La navigation

Intéressons-nous maintenant aux routes imbriquées.

Créons deux dossiers dans le dossier services :

- Marketing
- Développement

J'ajoute dans l'App.js l'imbrication des routes

```
<Route path="/services" element={<Services/>} >
        <Route path="/services/developpement" element={<Developpement/>} />
        <Route path="/services/marketing" element={<Marketing/>} />
</Route>
```

La navigation

Si on affiche le component Service, les liens ne marcherons pas car il manque l'outlet. Qui est la sortie des routes imbriquées (comme une sortie électrique par exemple).

Dans service.js on import Outlet:

import {Link, Outlet} from 'react-router-dom'

Et on mets la balise Outlet après la nav :

<Outlet />

La navigation

```
<nav>
        <Link to ='/services/marketing'>Service Marketing</Link>
        <Link to ='/services/developpement'>Service Développement</Link>
    </nav>
<Outlet />
```

La navigation - useLocation

Pour récupérer données passer en url, on peut utiliser le Hook useLocation.

Dans le component de votre choix importer useLocation :

import { useLocation } from 'react-router-dom’

Et on peut la loger pour savoir ce qu’il y a à l’intérieur de ce hook.

const location = useLocation()

console.log(location);

La navigation - useLocation

```
▶ Object { pathname: "/services/marketing", search: "", hash: "", state: null, key: "4p0rxyor" Marketing.js:7
}
```

Vous avez:

- le pathname : qui est l'url
- Search: si a des paramètres pour la recherche
- hash: Navigation via des ancres (#)
- state : Possibilité de faire passer des states
- Key: permet d'identifier une location de façon unique.

La navigation - useLocation

Exercice

Créons deux dossiers dans le dossier Admin :

- Dashboard
- GestionPost

Sur cette page L'admin pourra accéder sur la même page au dashboard et à la gestion des posts via une barre de navigation sans avoir à rafraichir la page

Bonus : Gestion des posts devra afficher le composant DataFetchAllReducer

Affichage depuis back-end

Affichage depuis un backend

Prenons notre backend précédemment crée dans le cours de nodeJS. Faisons en sorte que l'accueil nous envoie nous plus une page ejs mais un json :

```
res.json({data: data})
```

Notre backend est devenue maintenant une api et on peut enfin communiquer avec notre front :

Par le biais des routes et des useState et useReducer

Affichage depuis un backend

Affichons une donnée simple :

L'objectif afficher un film par rapport à l'id mis dans l'url dans le front

Dans le back :

```
app.get('/film/:id', function (req, res) {
    Film.findOne({
        _id: req.params.id
    }).then((data) => {res.json(data);})
    .catch((err) => {console.error(err)});
    ;
});
```

Affichage depuis un backend

Créons un composant FetchOneFilm :

On utilise useState

```
import React, {useEffect, useState } from 'react'
import axios from 'axios'
import { useParams } from 'react-router-dom'
function FetchOneFilm() {
    const params = useParams()
    const [loading, setLoading] = useState(true);
    const [error, setError] = useState("");
    const [film, setFilm] = useState({});
```

Affichage depuis un backend

```
useEffect(() => {
    axios.get('http://localhost:5000/film/'+params.id)
    .then(response =>{
        setLoading(false)
        setFilm(response.data)
        setError('')
    })
    .catch(error => {
        setLoading(false)
        setFilm({})
        setError('Something went wrong')
    })
})
```

Affichage depuis un backend

```
return (
    <div>
        {loading ? 'Loading' : film.titre}
        {loading ? 'Loading' : film.genre}
        {loading ? 'Loading' : film.nb_ventes}
        {loading ? 'Loading' : film.poster}
        {error ? 'error' : null}

    </div>
  )
```

Affichage depuis un backend

Exercice

Mettre à disposition la totalité des films (allfilm sur expressbackend) via un json

Et afficher la totalité des informations sur la vue react

Styled Component

Styled Component

npm i styled-components

On va crée un composant Button.style.js

```
const Button = styled.button`
    width: 200px;
    height: 50px;
    background-color: red;
`
```

Styled Component

Dans App.js

```
import {Button} from "./components/Button/Button.style";

 <Button> Test </Button>
```

On va pouvoir dupliquer tout cela plusieurs fois sans aucun problème.

Styled Component

Dans App.js

```
import {Button} from "./components/Button/Button.style";

 <Button> Test </Button>
```

On va pouvoir dupliquer tout cela plusieurs fois sans aucun problème.

Styled Component

Dans App.js

On va pouvoir dupliquer plusieurs boutons si nous voulons :

```
export const RedButton = styled.button`
    width: 200px;
    height: 50px;
    background-color: red;
`

export const GreenButton = styled.button`
    width: 200px;
    height: 50px;
    background-color: green;
`

export const BlueButton = styled.button`
    width: 200px;
    height: 50px;
    background-color: blue;
`
```

Styled Component

Cela peut vite être fastidieux on va préférer retourner en arrière et travailler avec les props :

```
export const Button = styled.button`
    width: 200px;
    height: 50px;
    background-color: ${(props) => props.backgroundColor};
`
```

Styled Component

On peut également gérer les containers :

On crée un component Container.style.js

```
import styled from "styled-components";

export const AppContainer = styled.div `
    width: 100vw;
    height: 100vh;
    background-color: lightblue;
`
```

Styled Component

On peut également gérer les containers :

On crée un component Container.style.js

```
import styled from "styled-components";

export const AppContainer = styled.div `
    width: 100vw;
    height: 100vh;
    background-color: lightblue;
`
```

Styled Component

Et dans App.js:

J’englobe le tout dans le component :

```
<AppContainer>
      <Button backgroundColor="red"> Test </Button>
      <Button> Test </Button>
      <Button> Test </Button>
      <Button> Test </Button>
       <Routes>
       [……]
      </Routes>
</AppContainer>
```

Styled Component

On peut améliorer le code Button.style.js

```
export const Button = styled.button`
    width: 200px;
    height: 50px;
    background-color: ${(props) => props.backgroundColor};
&:hover{
    background-color :coral;
}
`
```

Styled Component

On peut améliorer le code Button.style.js

Ici lors du clique :

```
export const Button = styled.button`
    width: 200px;
    height: 50px;
    background-color: ${(props) => props.backgroundColor};
&:active{
    background-color :coral;
}
`
```

Styled Component

On peut améliorer le code Button.style.js

Ici lors du clique :

```
export const Button = styled.button`
    width: 200px;
    height: 50px;
    background-color: ${(props) => props.backgroundColor};
&:active{
    background-color :coral;
}
`
```

Styled Component

On peut rajouter un label :

```
export const ButtonLabel = styled.label`
    font-size: 25px;
    color: white;
`
```

Ajoutons dans l’app.js

```
<Button backgroundColor="red"> <ButtonLabel>Test</ButtonLabel> </Button>
```

Styled Component

On peut rajouter un label :

```
export const ButtonLabel = styled.label`
    font-size: 25px;
    color: white;
`
```

Ajoutons dans l’app.js

```
<Button backgroundColor="red"> <ButtonLabel>Test</ButtonLabel> </Button>
```

Put and Delete

Method_overide :

<form action='http://localhost:5000/post/edit/645ca0d3c3c00d479d94bb13?_method=PUT' method='post'>

    <input type="hidden" name="_method" value="PUT" />

JWT

JSONWEBTOKEN

JWT

Un site :

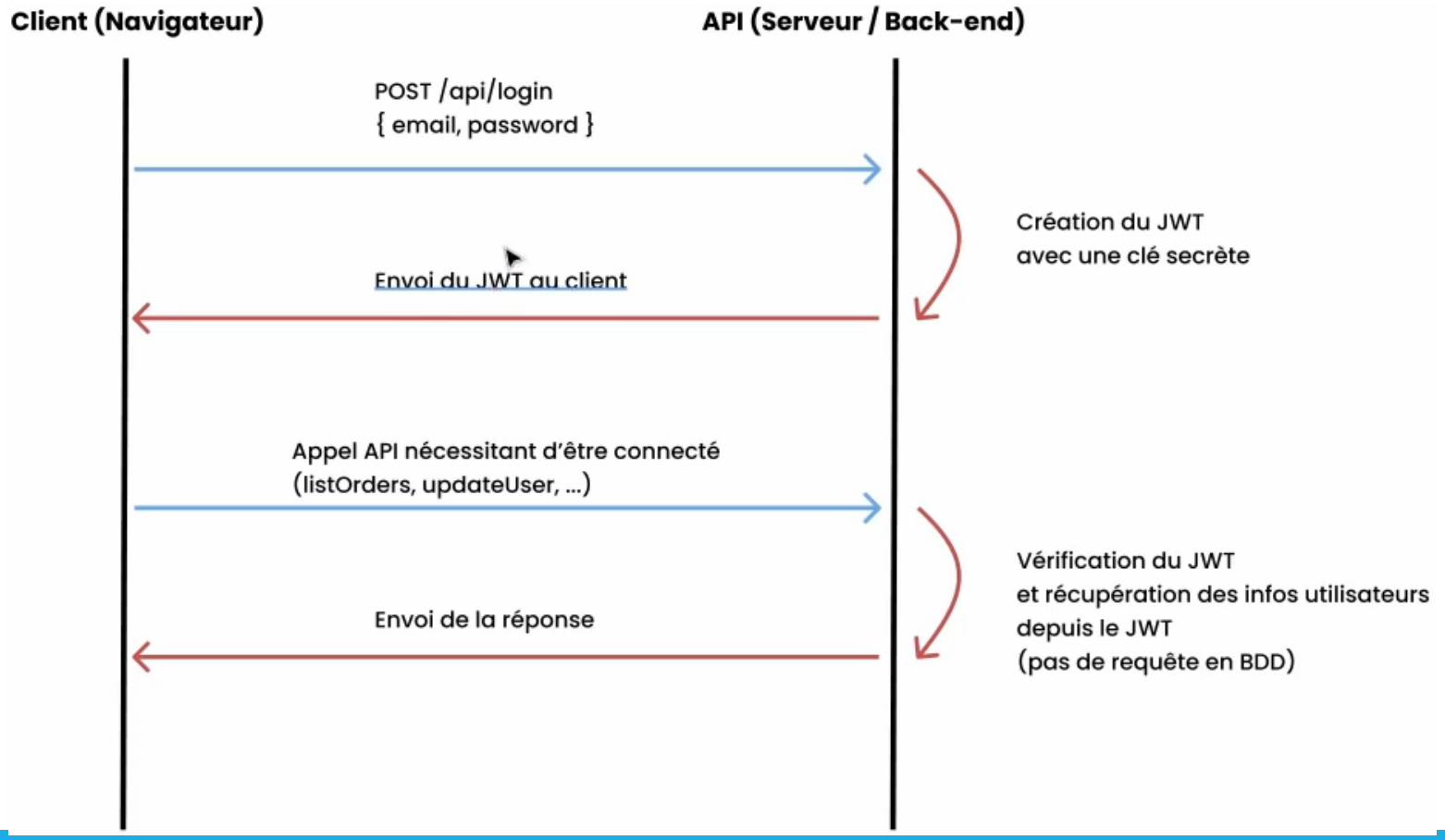
https://jwt.io/

Ce site introduit la notion JWT et comment on peut l'utiliser

Installons JWT **coté back :**

Npm i jsonwebtoken

JWT


Client (Navigateur)
API (Serveur / Back-end)
POST /api/login
{ email, password }
Création du JWT
avec une clé secrète
Envoi du JWT au client
Appel API nécessitant d'être connecté
(listOrders, updateUser, ...)
Vérification du JWT
et récupération des infos utilisateurs
depuis le JWT
(pas de requête en BDD)
Envoi de la réponse

JWT

Nous allons créer un fichier que nous allons nommer JWT.js

```
const { sign, verify } = require("jsonwebtoken");
const createTokens = (user) => {
  const accessToken = sign(
    { username: user.username, id: user.id },
    "SECRET"
  );
  return accessToken;
};
```

JWT

```
const validateToken = (req, res, next) => {
  const accessToken = req.cookies["access-token"];
  console.log(accessToken);
  if (!accessToken)
    return res.status(400).json({ error: "User not Authenticated!" });
  try {
    const validToken = verify(accessToken, "SECRET");
    if (validToken) {
      req.authenticated = true;
      return next();
    }
  }
```

JWT

```
catch (err) {
    return res.status(400).json({ error: err });
  }
};
module.exports = { createTokens, validateToken };
```

JWT

Installons maintenant cookie-parser et appelons le dans app.js :

```
const cookieParser = require("cookie-parser");

app.use(cookieParser());


const { createTokens, validateToken } = require("./JWT");
```

JWT

Une fois l’utilisateur logger et qu’il existe bien :

```
 const accessToken = createTokens(user);
      res.cookie("access-token", accessToken, {
        maxAge: 60 * 60 * 24 * 30,
        httpOnly: true,
      });
      res.json("LOGGED IN");
```

JWT

Rajoutons validateToken à votre route get / :

app.get("/", validateToken ,function (req, res){

[……]

}

Semantic UI

Mettre en place un gif de chargement

Mettre en place un gif de chargement

Dans le fichier index.html qui se trouves dans le dossier public.

```
<div id="root">
    <img src="loader-6.gif" alt="loading">
</div>
```

On peut rajouter un peu de style pour centrer le gif. Dans ce cours nous l'ajouterons en inline :

```
style="display: block; margin: auto;"
```

Upload image

UPLOAD D'IMAGE VIA UN FORMULAIRE

Upload image - Backend

```
const multer = require('multer');


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

Upload image - front

Crée un component imageUpload.js, le composant sera de cette forme la :

```
import React, {useState} from 'react'
import axios from 'axios'

const ImageUpload = () =>{
```

Upload image - front

```
function ImageUpload() {
   const [selectedFile, setSelectedFile] = useState(null);


   const handleFileChange = (event) => {
     setSelectedFile(event.target.files[0]);
   };
```

Upload image - front

```
const handleSubmit = (event) => {
    event.preventDefault();
    if (selectedFile) {
      const formData = new FormData();
      formData.append('image', selectedFile);

      axios.post('http://localhost:5000/upload', formData)
        .then((response) => {
          console.log(response.data);
})
                                                .catch((error) => {
                                                  console.error(error);
                                              });
                                                  }
                                                };
```

Upload image - front

```
return (
    <form onSubmit={handleSubmit}>
      <input type="file" onChange={handleFileChange} />
      <button type="submit">Envoyer</button>
    </form>
  );
```

Upload image - Backend

Installons maintenant mutler

npm i multer

// 1

const multer = require('multer')

// 2

Crée le dossier uploads

//3

app.use(express.static('uploads'))

Upload image - Backend

```
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'uploads/'); // Destination folder for uploaded files
  },
  filename: (req, file, cb) => {
    cb(null, file.originalname); // Use original file name
  },
});
```

Upload image - Backend

```
const upload = multer({ storage });

app.post('/upload', upload.single('image'), (req, res) => {
  if (!req.file) {
    res.status(400).send('No file uploaded.');
  } else {
    res.send('File uploaded successfully.');
  }
});
```

Plusieurs images

```
function FileUploadForm() {
  const [selectedFiles, setSelectedFiles] = useState([]);

  const handleFileChange = (event) => {
    setSelectedFiles(Array.from(event.target.files));
  };
```

Plusieurs images

```
const handleSubmit = (event) => {
  event.preventDefault();
  if (selectedFiles.length > 0) {
    const formData = new FormData();
    selectedFiles.forEach((file) => {
      formData.append('images', file);
    });
    axios
      .post('http://localhost:5000/uploadFiles', formData)
      .then((response) => {
        console.log(response.data);
      })
      .catch((error) => {
        console.error(error);
      });
  }
};
```

Plusieurs images

```
return (
  <form onSubmit={handleSubmit}>
    <input type="file" multiple onChange={handleFileChange} />
    <button type="submit">Upload</button>
  </form>
);
```

Plusieurs images - Back

```
// Handle file upload route
app.post('/uploadFiles', upload.array('images', 5), (req, res) => {
    if (!req.files || req.files.length === 0) {
      res.status(400).send('No files uploaded.');
    } else {
      res.send('Files uploaded successfully.');
    }
  });
```

Ajout d’une image en plus des données

Créons d'abord un nouveau Model Blog.js

```
const mongoose = require('mongoose');


const blogSchema = mongoose.Schema({
    titre : {type: 'String'},
    username : {type: 'String'},
    imageName : {type: 'String'},
})


module.exports = mongoose.model('Blog', blogSchema);
```

Ajout d’une image en plus des données

Dans le backend créons la route pour la création d'une donnée :

```
const Blog = require('./modeles/Blog');

app.post("/submit-blog", function (req, res) {
    const Data = new Blog({
        titre: req.body.titre,
        username: req.body.username,
        imageName: req.body.imagename,
})
    Data.save().then(() => {
res.redirect('http://localhost:3000/myblog')
    }).catch(err => {console.log(err)});


});
```

Ajout d’une image en plus des données

```
app.post("/submit-blog", upload.single('image'), function (req, res) {
    const Data = new Blog({
        titre: req.body.titre,
        username: req.body.username,
        imageName: req.body.imagename,
    })
    if (!req.file) {
        res.status(400).send('No file uploaded.');
    } else {
        res.send('File uploaded successfully.');
        Data.save().then(() => {

res.redirect('http://localhost:3000/myblog')
        }).catch(err => {console.log(err)});
    }
});
```

Ajout d’une image en plus des données

Créons d'abord un nouveau Model Blog.js

```
const mongoose = require('mongoose');


const blogSchema = mongoose.Schema({
    titre : {type: 'String'},
    username : {type: 'String'},
    imageName : {type: 'String'},
})


module.exports = mongoose.model('Blog', blogSchema);
```

Ajout d’une image en plus des données

```
if (err) {
        return res.status(500).json(err)
    }
    console.log(req.file.filename);
    res.status(200).json({Data});
  })
});
```

Affichage de l’image

```
const handleSubmit = (event) => {
  event.preventDefault();

  if (file && titre && username) {
    const formData = new FormData();
    formData.append('file', file);
    formData.append('titre', titre);
    formData.append('username', username);

    axios.post('/upload', formData, {
      headers: {
        'Content-Type': 'multipart/form-data',
      },
    })
      .then((response) => {
        console.log(response.data);
        // Handle successful upload
      })
      .catch((error) => {
        console.error(error);
        // Handle upload error
      });
  }
};
```

Affichage de l’image

```
import React, { useState } from 'react';
import axios from 'axios';


function FileUploadForm() {
  const [file, setFile] = useState(null);
  const [titre, setTitre] = useState('');
  const [username, setUsername] = useState('');
```

Affichage de l’image

```
const handleFileChange = (event) => {
  setFile(event.target.files[0]);
};

const handleTitreChange = (event) => {
  setTitre(event.target.value);
};

const handleUsernameChange = (event) => {
  setUsername(event.target.value);
};
```

Affichage de l’image

```
return (
  <form onSubmit={handleSubmit}>
    <div>
      <label htmlFor="file">File:</label>
      <input type="file" id="file" onChange={handleFileChange} />
    </div>
    <div>
      <label htmlFor="titre">Titre:</label>
      <input type="text" id="titre" value={titre} onChange={handleTitreChange} />
    </div>
    <div>
      <label htmlFor="username">Username:</label>
      <input type="text" id="username" value={username} onChange={handleUsernameChange} />
    </div>
    <button type="submit">Upload</button>
  </form>
);
```

Affichage de l’image

```
return (
    <form method="post" onSubmit={onSubmit}>
[..]
</form>
```

Affichage de l’image

```
<Card
                image={['//localhost:5000/'+voiture.img]}
                href={['/cars/'+voiture._id]}
                header={voiture.modele}
                meta={voiture.marque}
                description={voiture.description}
            />
        </div>
```

Mini projets

Les projets

Maintenant vous avez les connaissances nécessaires pour créer une application basique Fullstack JS avec Express – React.

- Créer votre GitHub pour le back et un GitHub pour le front.
- Créer votre projet backend avec expressJS et créer un CRUD.
- Créer une page qui test chacune des routes.
- Créer votre projet React et afficher les données de votre backend sur votre projet.
- Mettre en place une connexion par JWT

Les projets fullstack JS

Continuer sur votre lancer. Implémenter semantic ui react, et compléter votre fullstack js.

Voici vos objectifs :

Possibilité d’enregistrer une donnée dans la base via un formulaire réalisé via semantic

Visibilité de la totalité des données sur une page

Typescript

COMPRÉHENSION DE TYPESCRIPT EN RÉALISANT UNE TODOLIST

Typescript

TypeScript se veut un sur-ensemble de JavaScript qui se transforme en JavaScript. En générant un code conforme à ECMAScript, TypeScript peut introduire de nouvelles fonctionnalités de langage tout en conservant la compatibilité avec les moteurs JavaScript existants. ES3, ES5 et ES6 sont actuellement des cibles prises en charge.

Les types facultatifs sont une caractéristique principale. Les types permettent une vérification statique dans le but de détecter rapidement les erreurs et peuvent améliorer les outils avec des fonctionnalités telles que la refactorisation du code.

TypeScript est un langage de programmation open source et multi-plateforme développé par Microsoft. Le code source est disponible sur GitHub .

Typescript

https://create-react-app.dev/docs/adding-typescript/

npx create-react-app todo-list --template typescript

Typescript

Ouvrons App.tsx, c’est le nouveau App.js sauf qu’ici les instructions sont écrit en Typescript :

```
import React, {FC} from 'react';
import "./App.css"
const App: FC = () => {
  return (
    <div className="App">
     <div className='header'></div>
     <div className='todoList'></div>
    </div>
  );
}
export default App;
```

Typescript

```
<div className='header'>
    <input type="text" placeholder='Task....'/>
    <input type="number" placeholder='Deadline (in Days)'/>
    <button>Add Task</button>
</div>
```

Typescript

```
<div className='inputContainer'>
    <input type="text" placeholder='Task....'/>
    <input type="number" placeholder='Deadline (in Days)'/>
</div>
```

Typescript

```
<div className='inputContainer'>
    <input type="text" placeholder='Task....'/>
    <input type="number" placeholder='Deadline (in Days)'/>
</div>
```

Typescript

```
.App {
  display: flex;
  align-items: center;
  flex-direction: column;
  width: 100vw;
  height: 100vh;
  font-family: Arial, Helvetica, sans-serif;
}
```

Typescript

```
.header{
  flex: 30%;
  background-color: tomato;
  width: 100%;
  display: flex;
  justify-content: center;
  align-items: center;
}
```

Typescript

```
.header .inputContainer {
  display: flex;
  flex-direction: column;
}
```

Typescript

```
.header input {
  width: 200px;
  height: 40px;
  border: none;
  border-bottom-left-radius: 8px;
  border-top-left-radius: 8px;
  padding-left: 10px;
  font-size: 17px;
  border: 1px solid grey;
}
```

Typescript

Pour éviter la bordure blanche tout autour nous allons faire :

```
body{
  margin: 0;
  padding: 0;
}
```

Typescript

Nous voulons faire en sorte que le header prends 30% de l'écran et la todoList le reste:

```
.todoList {
  flex: 70%;
  width: 100%;
}
```

Typescript

Nous allons utiliser les usestate pour permettre de stocker les valeurs plus simplement :

```
import React, {FC, useState} from 'react’;

 const [task, setTask] = useState<string>("")

  const [deadline, setDeadline] = useState<number>(0)

  const [todoList, setTodoList] = useState([])
```

En Typescript il est pas obligatoire de mettre le type de Valeur mais il est fortement recommandé.

Typescript

Pour remplir les states il faut réaliser des fonctions handles :

```
import React, {FC, ChangeEvent ,useState} from 'react';

 const handleChange = (event: ChangeEvent<HTMLInputElement>) =>{

    setTask(event.target.value)

  }


<input type="text" placeholder='Task....' name="task" onChange={handleChange}/>

<input type="number" placeholder='Deadline (in Days)' name="deadline"
onChange={handleChange}/>
```

Typescript

```
const handleChange = (event: ChangeEvent<HTMLInputElement>) =>{
   if(event.target.name == "task"){
     setTask(event.target.value)
   }
   else{
     setDeadline(event.target.value)
   }
 }
```

Ce code donne une erreur event.target.value n'est pas sur qu'il soit de type « Number » il faut donc le « caster » en nombre. En d'autres termes, transformer en nombre le résultat obtenue.

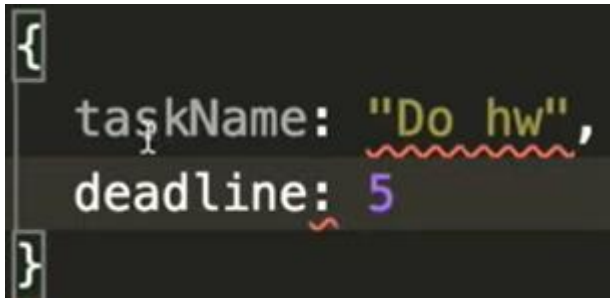
Typescript

```
const handleChange = (event: ChangeEvent<HTMLInputElement>) =>{
  if(event.target.name == "task"){
    setTask(event.target.value)
  }
  else{
    setDeadline(Number(event.target.value))
  }
}
```

Typescript

Passons maintenant à l'ajout d'une tache :

Il faut tout d'abord créer une interface. L'interface va nous permettre de crée un objet et de sauvegarder les informations de cette façon la :



Créons un fichier interfaces.ts dans lequel nous mettrons toutes nos interfaces à la suite.

Typescript

```
export interface ITask{
    taskName: string;
    deadline: number;
}
```

Importons le dans l’App.tsx

Typescript

```
import {ITask} from './Interfaces’

const [todoList, setTodoList] = useState<ITask[]>([])

  const addTask = ():void => {

    const newTask = {taskName: task, deadline: deadline}

    setTodoList([...todoList, newTask])

    console.log(todoList);

  }
```

Ici on a mis un type de retour : « void » car la fonction ne renvoit rien en particulier

```
<button onClick={addTask}>Add Task</button>
```

Typescript

Nous allons maintenant faire en sorte que lors d’un ajout d’une tâche les inputs soit vide.

Pour cela il faut changer les values des inputs :

```
 <div className='inputContainer'>

      <input type="text" placeholder='Task....' name="task" value={task}
onChange={handleChange}/>

      <input type="number" placeholder='Deadline (in Days)' name="deadline" value={deadline}
onChange={handleChange}/>

    </div>
```

Typescript

```
const addTask = ():void => {
  const newTask = {taskName: task, deadline: deadline}
  setTodoList([...todoList, newTask])
  setTask("");
  setDeadline(0);
}
```

Typescript

Créons maintenant l'affichage de la TodoList. On va crée un nouveau composant : TodoTask.tsx

Créer un composant avec le snippet (rfce)

Et modifier la fonction par une fonction fléché :

```
const TodoTask = () => {
  return (
    <div>TodoTask</div>
  )
}
```

Typescript

Dans l'App.tsx, on va ajouter dans la div :

```
<div className='todoList'>
    {todoList.map(()=>{
      return <TodoTask />
    })}
   </div>
```

Pour que chaque élément de la Todo devra être afficher

Typescript

Dans l'App.tsx, on va ajouter dans la div :

```
<div className='todoList'>
    {todoList.map(()=>{
      return <TodoTask />
    })}
   </div>
```

Pour que chaque élément de la Todo devra être afficher

Typescript

Allons un peu plus loin et essayons de faire passer la task en props directement dans le composant :

```
 <div className='todoList'>
      {todoList.map((task: ITask, key: number)=>{
        return <TodoTask key={key} />
      })}
     </div>
```

(le key, est juste la pour enlever le warning en react, aucune utilité sinon)

Typescript

Revenons a notre composant il va falloir ajouter une interface :

```
interface Props {
  task: ITask;
}
const TodoTask = ({ task, completeTask }: Props) => {
  return (
    <div className="task">
      <div className="content">
        <span>{task.taskName}</span>
        <span>{task.deadline}</span>
      </div>
    </div>
  );
};
```

Typescript

Il faut bien sur passer la task en props :

return <TodoTask key={key} task={task} />

Typescript

Un peu de style :

```
.header button {
  width: 70px;
  height: 87px;
  border: none;
  border-bottom-right-radius: 8px;
  border-top-right-radius: 8px;
  padding-left: 10px;
  cursor: pointer;
}
```

Typescript

```
.task {
  width: 500px;
  height: 50px;
  display: flex;
  color: white;
  margin: 15px;
}
```

Typescript

```
.task .content {
  flex: 80%;
  height: 100%;
  display: flex;
  justify-content: center;
  align-items: center;
}
```

Typescript

```
.task .content span {
  display: grid;
  place-items: center;
  border: 1px solid white;
  width: 100%;
  height: 100%;
  font-size: 18px;
  border-right: none;
  background-color: tomato;
}
```

Typescript

```
.task button {
  flex: 20%;
  height: 100%;
  border: none;
  background-color: lightseagreen;
  border-top-right-radius: 8px;
  border-bottom-right-radius: 8px;
  color: white;
  cursor: pointer;
}
```

Typescript

Créons un boutons pour supprimer une tache dans le component TodoTask :

```
<div className="task">
  <div className="content">
    <span>{task.taskName}</span>
    <span>{task.deadline}</span>
  </div>
  <button>
    X
  </button>
</div>
```

Typescript

Dans APP.tsx :

```
const completeTask = (taskNameToDelete: string):void => {
    setTodoList(todoList.filter((task)=>{
      return task.taskName != taskNameToDelete
    }))
  }
```

Filter va permettre de garder les autres et enlever celui qui match

Typescript

Il faut ensuite retourner dans le component, et ajouter la fonction dans l'interface :

Ainsi on fait passer l'information que la tache est complété.

```
interface Props {
  task: ITask;
  completeTask(taskNameToDelete: string): void;
}
```

Typescript

```
<button
      onClick={() => {
        completeTask(task.taskName);
      }}
    >
      X
    </button>
```

On appelle la fonction completeTask avec le nom de la tache à supprimer

Typescript

Il reste plus qu’à ajouter la fonction en props dans App.tsx

return <TodoTask key={key} task={task} completeTask={completeTask}/>