

React

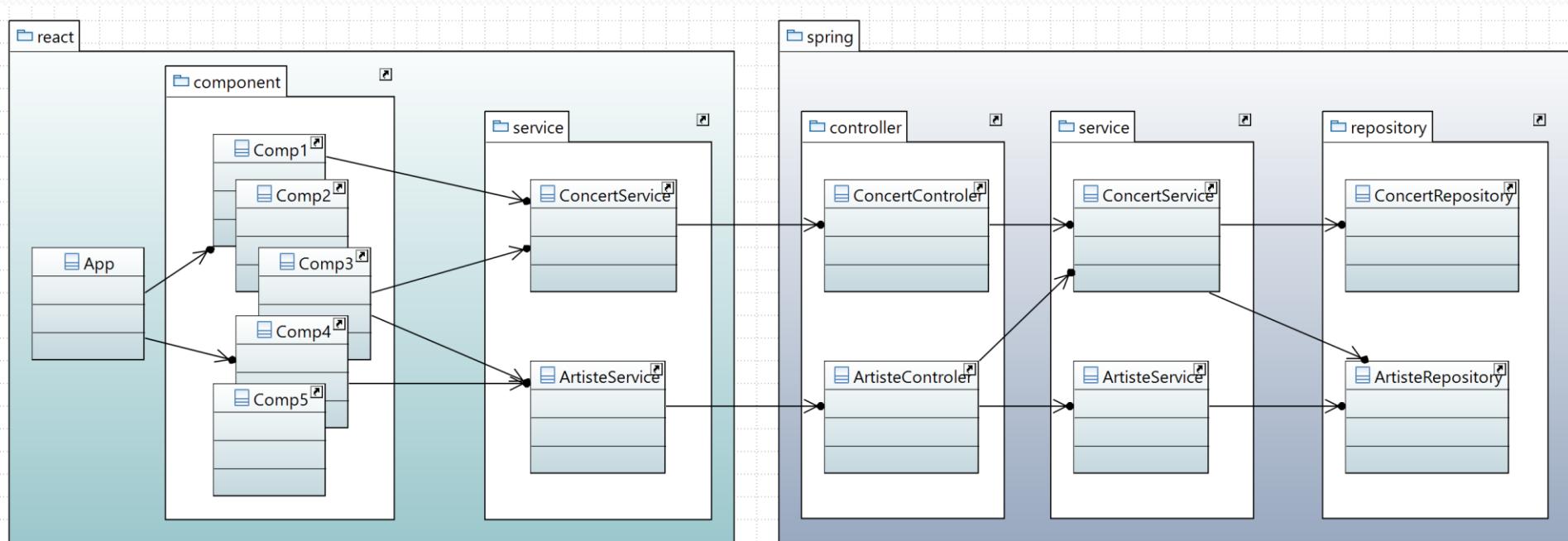
Communication client serveur
Services React
Cache des requêtes (ReactQuery)

Cedric Dumoulin

Architecture d'une application

Architecture

- La communication se fait entre
 - les services React
 - les contrôleurs Spring
- Toujours à l'initiative du client React
- Les composants utilisent les **services React**



Organiser les Composants, leurs modules ...

- Pas de consensus
- Pour commencer :

```
src/  
  components/  
  buttons/  
  textfield/  
  contexts/  
  hooks/  
  pages/  
  services/  
  utils/  
  App.js  
  index.js
```

Organiser les paquetages

Bibliographie

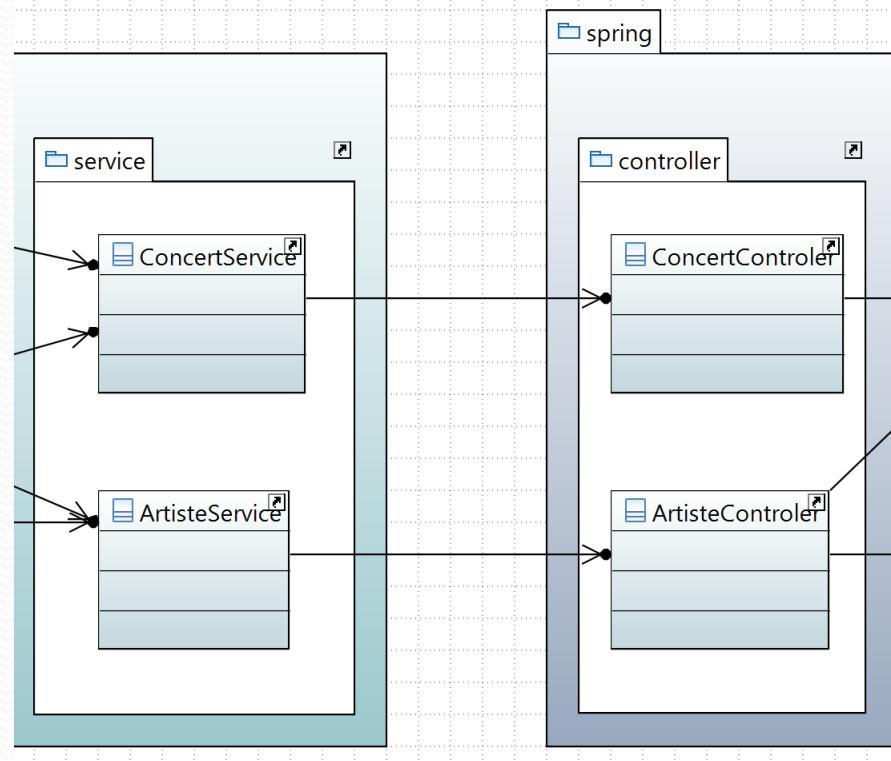
- React Arch
 - <https://www.taniarascia.com/react-architecture-directory-structure/>
Architecture: How to Structure and Organize a React Application
- Is there a recommended way to structure React projects?
 - <https://legacy.reactjs.org/docs/faq-structure.html>
- ➔ 4 folder structures to organize your React & React Native project
 - <https://reboot.studio/blog/folder-structures-to-organize-react-project>



Coté client :
Communiquer avec le serveur

Le problème

- Le client veut envoyer une requête au serveur
- Le serveur :
 - Fournit des données (ex: Json)
 - Reçoit des requêtes update/create/delete
 - Sert des données Json (Controleurs)
- Le client :
 - Affiche les données (après mise en forme)
 - Envoie des requêtes update/create/delete
- → Comment procéder côté client ?



Communiquer avec le serveur

- Un client javascript peut communiquer avec le serveur
- Utilisation de la fonction '**fetch()**' de javascript
- Il existe aussi des librairies :
 - axios
- Les appels au serveur sont asynchrones !
 - → il faut utiliser un mécanisme pour attendre la réponse
 - → on se retrouve avec plusieurs flots d'exécution

API Fetch

- Fetch() permet
 - d'envoyer une requête à un serveur
 - GET, POST, PUT, DELETE
 - Et d'attendre la réponse
 - ➔ l'appel est asynchrone
- Paramètres :
 - urlAsString : l'url de la ressource (absolue ou relative)
 - init (optionnel) : permet de passer des paramètres à l'appel
- Retour ;
 - une Promise !!
 - la Promise est remplie lorsque la réponse est reçue

```
fetch( urlAsString, init ) : Promise
```

Important : Fetch() et Erreur 40X 50x

- Fetch() ne lance pas d'exception quand il y a des erreurs de type :
 - 40x
 - Bad Request – Le serveur a reçu la requête, mais celle-ci ne peut pas être traité car elle comporte un problème (mal formed, authorization ...)
 - 50x
 - Erreur coté serveur – Une erreur est survenue du coté serveur
- ➔ **Le client doit vérifier le status de la réponse**
 - Et effectuer l'action appropriée
 - En général envoyer une exception

Fetch et React

- L'appel à fetch() doit se faire après l'initialisation du Component
 - → utiliser useEffect()
 - La réception de la réponse doit rafraîchir le Component
 - → sauve la réponse dans un état (useState())
- Les hooks facilitent l'utilisation de fetch()

```
const [concerts, setConcerts] = useState([]);
const [loading, setLoading] = useState(false);

useEffect(() => {
    setLoading(true);

    fetch('http://localhost:8080/api/concerts')
        .then(response => response.json())
        .then(data => {
            setConcerts(data);
            setLoading(false);
        })
}, []);
```

useEffect()

- <https://react.dev/reference/react/useEffect>
- useEffect is a React Hook that lets you synchronize a component with an external system.

useEffect(setup, dependencies?)

```
import { useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]);
  // ...
}
```

useEffect()

```
import { useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();

    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]);
  // ...
}
```

Setup code

cleanup code

List of dependencies

You need to pass two arguments to `useEffect`:

1. A *setup function* with `setup code` that connects to that system.
 - It should return a *cleanup function* with `cleanup code` that disconnects from that system.
2. A `list of dependencies` including every value from your component used inside of those functions.

React calls your setup and cleanup functions whenever it's necessary, which may happen multiple times:

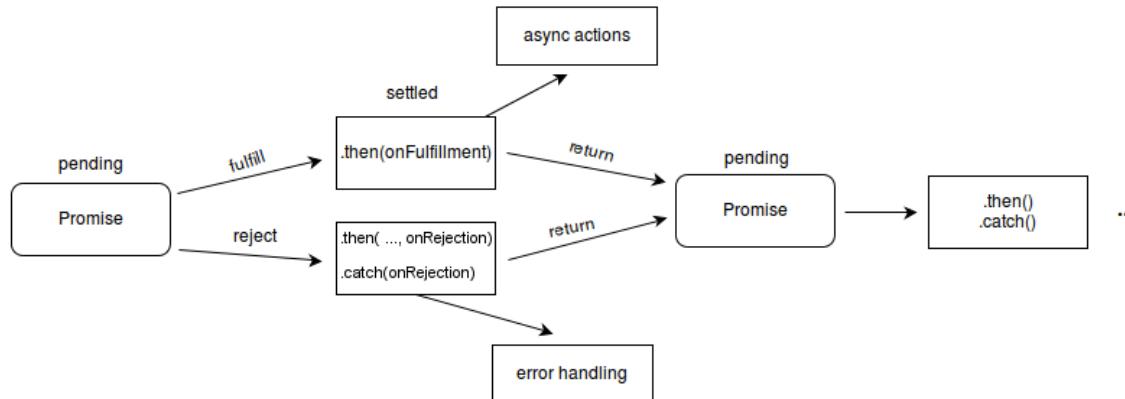
1. Your `setup code` runs when your component is added to the page (*mounts*).
2. After every re-render of your component where the `dependencies` have changed:
 - First, your `cleanup code` runs with the old props and state.
 - Then, your `setup code` runs with the new props and state.
3. Your `cleanup code` runs one final time after your component is removed from the page (*unmounts*).

Plusieurs façon d'utiliser fetch()

- Avec les promises
- Avec await / async
- Avec useFetch()

Requête avec fetch() et Promises

- Fetch renvoie un objet de type Promise
 - https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Promise
- Plusieurs méthodes sur cette classe Promise :
 - `then(onFulfillment, onRejection)` -
 - On peut enchaîner les appels à `then()`
 - `catch(onRejection)`
 - `onFulfillment` – fonction appelée en cas de succès.
 - `onRejection` – fonction appelée en cas d'échec



Exemple fetch() + Promise

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

```
fetch('https://api.example.com/data')
```

- Effectue l'appel asynchrone. Renvoie une Promise
 - Si succès, la Promise contient la réponse

```
.then(response => response.json())
```
 - Traite la réponse, si succès (ne traite pas si échec)
 - response : la réponse de l'appel précédent
 - response.json() : déserialise la réponse (json->objet JS). Renvoie une Promise avec le résultat

```
.then(data => console.log(data))
```
 - Fait un traitement avec l'objet récupéré (ici log dans la console)

```
.catch(error => console.error(error));
```
 - En cas d'erreur, cette partie est appelée

Fetch + Promise

Exemple plus complet

```
const [concerts, setConcerts] = useState([]);  
const [loading, setLoading] = useState(true);  
const [error, setError] = useState(false);  
  
useEffect(() => {  
    setLoading(true);  
    fetch(SERVER_URL + '/api/concerts')  
        .then(response => {  
            if(response.ok) {  
                return response.json();  
            }  
            throw response  
        })  
        .then(data => {  
            setConcerts(data);  
            setLoading(false);  
        })  
        .catch(error => {  
            console.error(  
                'erreur fetching data : ', error);  
            setError(true)  
        })  
        .finally(() => {  
            setLoading(false);  
        })  
}, []);  
  
if (loading) { return <p>Loading...</p>; }  
if (error) {return <p>Error !!</p>;}
```

Les états

Dans un useEffect()

Pour que l'appel se fasse après
l'initialisation du composant

Appel asynchrone

Attente de la réponse

Vérification du status OK

Deserialisation Json -> objet JS
Renvoie d'une promesse avec le résultat

Traitement de l'objet retourné par json()

Gestion des erreurs

Dans tous les cas, quand c'est fini :
loading ← false

On vérifie les status.
On affiche le status si on n'a pas encore la
réponse

Fetch() et await

Exemple complet

```
const [joke, setJoke] = useState("");

const fetchJoke = async () => {
  try {
    const response = await fetch('https://api.example.com/data',
      {
        method: 'GET',
        headers: { /* optionel */ }
      });
    if ( ! response.ok) {
      throw new Error('Network response was not ok');
    }
    const data = await response.json();

    setJoke(data[0].joke);
  } catch (error) {
    console.error('Error:', error);
  }
}
```

Un état qui recevra la réponse.

Crée une méthode **async**.
Obligatoire

Try / catch

await : indique d'attendre le résultat de l'appel asynchrone

Optionnel : on peut indiquer le type de requête, les headers ...

Verifie le status de la réponse

Deserialise la réponse. Appel asynchrone.

await : attend le résultat de l'appel asynchrone

Range le résultat dans un état
➔ rafraîchissement du comp.

Traitement en cas d'erreurs

Utilisation

```
useEffect(() => {  
  fetchJoke();  
}, []);
```

Dans un useEffect()

Pour que l'appel se fasse après
l'initialisation du composant

Appel de notre méthode asynchrone.
Le résultat sera disponible dans l'état.
En cas d'erreur, pas de résultat

Avec useFetch()

- Nécessite la librairie **react-fetch-hook**
 - <https://www.npmjs.com/package/react-fetch-hook>
 -

```
import useFetch from "react-fetch-hook";

const { isLoading, data : concerts, error } = useFetch(SERVER_URL + '/api/concerts');

if (isLoading) {
  return <p>Loading...</p>;
}
if (error) {
  return <p>Error !!</p>;
}
```

Retourne les status et le résultat

Les useState(), le useEffect() et la déserialisation sont encapsulé dans useFetch()

CrossOrigin

- Si le serveur React et le serveur de Spring sont différent :
 - → problème de CrossOrigin
 - certain navigateur bloquent ce type de requêtes
- Solutions :
 - Autoriser le CrossOrigin
 - Au niveau du Contrôleur Spring :
 - `@CrossOrigin(origins = "http://localhost:3000")`

```
@RestController()
@RequestMapping("api/concerts")
@CrossOrigin(origins = "http://localhost:3000")
public class ConcertController {
    ...
}
```

Lecture et tutoriaux

- Fetching data with Effects
 - <https://react.dev/reference/react/useEffect#fetching-data-with-effects>
- How to Fetch Data in React: Cheat Sheet + Examples
 - <https://www.freecodecamp.org/news/fetch-data-react/>

Requête de type Read

Requête pour demander des données au serveur

- L'appel à fetch() doit se faire après l'initialisation du Component
 - → utiliser useEffect()
 - La réception de la réponse doit rafraîchir le Component
 - → sauve la réponse dans un état (useState())
- Les hooks facilitent l'utilisation de fetch()

```
const [concerts, setConcerts] = useState([]);
const [loading, setLoading] = useState(false);

useEffect(() => {
    setLoading(true);

    fetch('http://localhost:8080/api/concerts')
        .then(response => response.json())
        .then(data => {
            setConcerts(data);
            setLoading(false);
        })
}, []);
```

Requête de type Update et Create

Requête de type Update et Create

- Principalement utilisées pour la soumission de formulaires
- Il faut :
 - gérer le formulaire
 - soumettre la requête quand on soumet le formulaire
- Pour **edit** :
 - On veut commencer par afficher les anciennes valeur
 - → faire un read pour charger la valeur initiale
- Pour **create** :
 - il faut initialiser le formulaire avec un objet contenant les valeurs par défaut (généralement des chaînes vides)

Update et Create (1)

Initialisation

```
import React, { useEffect, useState } from 'react';
import {useNavigate, useParams} from 'react-router-dom';
//import { Button, Container, Form, Form.Group, Form.Control, Form.Label } from 'react-bootstrap';
import Form from 'react-bootstrap/Form';
import Button from 'react-bootstrap/Button';

const SERVER_URL = "http://localhost:8080/";

const ConcertEdit = () => {
    // Création d'un concert vide
    const initialFormState = {
        name: '',
        price: '',
        date: '',
    };
    // etat
    const [concert, setConcert] = useState(initialFormState);
    const navigate = useNavigate();
    // Recupere l'id du concert a éditer
    const { id } = useParams();
    // Charge le concert si id existe
    useEffect(() => {
        if (id !== 'new') {
            fetch(SERVER_URL + `/api/concerts/${id}`)
                .then(response => response.json())
                .then(data => setConcert(data));
        }
    }, [id, setConcert]);
```

Update et Create (2)

Handlers changement et submit

```
// Gere les changement de valeurs dans le formulaire
const handleChange = (event) => {
  const { name, value } = event.target

  setConcert({ ...concert, [name]: value })
}

// Methode appelee quand le formulaire est soumis
const handleSubmit = async (event) => {
  // Disable submit default (html) behavior
  event.preventDefault();
  // Do fetch , wait the response (await)
  // We set method type according to the presence of an id
  // Form values are sent as a json object
  await fetch( SERVER_URL + '/api/concerts' + (concert.id ? '/' + concert.id : ''),
    {
      method: (concert.id) ? 'PUT' : 'POST',
      headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
      },
      body: JSON.stringify(concert)
    });
  // Reset form
  setConcert(initialFormState);
  // Move away
  navigate('/concerts');
}
```

Requêtes POST et PUT avec le SecurityManager Spring

- Si la librairie 'org.springframework.boot:spring-boot-starter-security' est dans le classpath
 - il faut désactiver (cross references forgery)

```
package fil.ipint.resaconcert;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    /**
     * Configure protected area, login and logout pages ...
     */
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            // We disable csrf (cross references forgery) when we test without password.
            // Otherwise, PUT, POST and DELETE requests don't work (error 403)
            // We should try to reenable csrf when passwords are enabled for supervisors
            .csrf().disable()
            .authorizeRequests()
                .anyRequest().permitAll()
                .and()
                .exceptionHandling().accessDeniedPage("/403");
    }
}
```

Requête Delete

Delete

- Ajouter un bouton ‘delete’ pour chaque élément de la liste

```
<button className='btn btn-primary btn-sm'  
        type="button" onClick={() => deleteById(concert.id)}>Delete</button>
```

- Envoyer la requête au serveur

- si la requête est ok, on met à jour la liste locale (en enlevant aussi le concert)

```
const deleteById = async (id) => {  
    console.log('delete called');  
    await fetch(SERVER_URL + '/api/concerts/' + id, {  
        method: 'DELETE',  
        headers: {  
            'Accept': 'application/json',  
            'Content-Type': 'application/json'  
        }  
    }).then(() => {  
        let updatedConcert = [...concerts].filter(i => i.id !== id);  
        setConcerts(updatedConcert);  
    });  
}
```

Les requêtes dans un module Service

Module Service

- Il est préférable de regrouper les requêtes dans un module
- Un service par type d'objet manipulé
 - → ex: services/ConcertService, services/ArtisteService
 - → ex: services/StageService, services/StudentService

Exemple de service avec une classe

- Pas forcément la meilleure façon de faire
- On peut privilégié un service proposant des hooks ...

Ex: ConcertService (1)

```
const SERVER_URL = "http://localhost:8080";

class ConcertService {

    async fetchAllConcerts() {
        return fetch(SERVER_URL + '/api/concerts')
            .then(response => response.json());
    }

    async fetchConcert( id ) {
        return fetch(SERVER_URL + `/api/concerts/${id}`)
            .then(response => response.json());
    }

    async updateConcert( concert ) {
        return fetch( SERVER_URL + '/api/concerts' + (concert.id ? '/' + concert.id : ''), {
            method: (concert.id) ? 'PUT' : 'POST',
            headers: {
                'Accept': 'application/json',
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(concert)
        });
    }
}
```

Ex: ConcertService (2)

```
/**
 *
 * @param {*} id
 */
async deleteConcert (id) {
    return await fetch(SERVER_URL + '/api/concerts/' + id, {
        method: 'DELETE',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        }
    });
}

export default new ConcertService();
```

Ex: ConcertService Utilisation

```
import ConcertService from '../services/ConcertService';
const ConcertWithFetch = () => {
// . .
  useEffect(() => {
    setLoading(true);
    ConcertService.fetchAllConcerts()
      .then(data => {
        setConcerts(data);
        setLoading(false);
      })
  }, []);
  if (loading) {
    return <p>Loading...</p>;
  }

  const deleteById = async (id) => {
    console.log('delete called');
    ConcertService.deleteConcert(id)
      .then(() => {
        let updatedConcert = [...concerts].filter(i => i.id !== id);
        setConcerts(updatedConcert);
      });
  };
}
```

Atelier

- Vous avez un serveur (back office) acceptant des requêtes pour lister, créer et détruire une entité (par exemple pour Student et Stage)
- Vous avez des composants React (Front) pour afficher des collections
- Vous allez connecter la partie Front et le serveur afin d'obtenir une application qui doit :
 - afficher la liste des entités (d'abord Student, puis Stage)
 - avec les boutons edit et delete pour chaque entité
 - permettre la destruction d'un concert
 - ~~permettre la création d'un concert (vue plus loin)~~
 - ~~Il manque le formulaire !!~~
- Votre application doit utiliser, côté React, des services

Cache des données avec tanstack-query

Le problème

- Pour des raisons de performances
 - mettre en cache les données téléchargées
- Permet aussi à plusieurs composants de partager des données
 - via le cache
 - au lieu de partager une variable ou un état
- Solution
 - utiliser un mécanisme de cache
 - On peut l'implémenter
 - ou utiliser des bibliothèques existantes
 - TanStack-Query (React-query)

TanStack-Query (React-query)

- **Powerful asynchronous state management**
- Installation
 - npm i react-query
 - v3.39
 - import {*} from 'react-query'
 - ou npm i @tanstack/react-query
 - V5.xx
 - import {*} from '@tanstack/react-query'
- Site
 - <https://github.com/TanStack/query>
 - <https://tanstack.com/query/latest>
- Doc :
 - <https://tanstack.com/query/latest/docs/react/overview>

Attention : exemple en V4 !

- Dernière version : V5

Mise en place

- Un composant englobant tout les appels à tanstack-query
 - Dans App.js

```
function App() {  
  
    // Create a client  
    const queryClient = new QueryClient();  
  
    return (  
        // Provide the client to your ConcertsListWithCache  
        <QueryClientProvider client={queryClient}>  
            <BrowserRouter>  
                <Routes>  
                    <Route path="/" element={<Layout />}>  
                        <Route index element={<Home />} />  
                        <Route path="component1" element={<Component1 />} />  
                        <Route path="*" element={<NoPage />} />  
                    </Route>  
                </Routes>  
            </BrowserRouter>  
        </QueryClientProvider>  
    );  
}
```

Utilisation

- Read

- `useQuery({ ... })`
- return un objet avec :
 - `data` : le résultat
 - `isLoading` : boolean
 - `isError` : boolean

- Update/create

- utiliser les mutations
- On déclare la mutation et son handler en cas de succès
- On appelle la mutation au moment où l'on veut faire l'update

```
function Todos() {
  // Access the client
  const queryClient = useQueryClient()

  // Queries
  const query = useQuery({ queryKey: ['todos'], queryFn: getTodos })

  // Mutations
  const mutation = useMutation({
    mutationFn: postTodo,
    onSuccess: () => {
      // Invalidate and refetch
      queryClient.invalidateQueries({ queryKey: ['todos'] })
    },
  })

  return (
    <div>
      <ul>
        {query.data?.map((todo) => (
          <li key={todo.id}>{todo.title}</li>
        ))}
      </ul>
      <button
        onClick={() => {
          mutation.mutate({
            id: Date.now(),
            title: 'Do Laundry',
          })
        }}
      >
        Add Todo
      </button>
    </div>
  )
}
```

useQuery({ ... })

```
function Todos() {
  // Access the client
  const queryClient = useQueryClient()

  // Queries
  const query = useQuery({ queryKey: ['todos'], queryFn: getTodos })

  return (
    <div>
      <ul>
        {query.data?.map((todo) => (
          <li key={todo.id}>{todo.title}</li>
        ))}
      </ul>
    </div>
  )
}
```

- `useQuery({ queryKey: [], queryFn: fetchFct })`
 - `queryKey`:
 - Un tableau de clés
 - Permet de donner un identifiant au résultat
 - `fetchFct`
 - La fonction retournant un résultat
 - Peut retourner une Promise
- return un objet avec :
 - `data` : le résultat
 - `isLoading` : boolean
 - `isError` : boolean

Update : useMutation(...)

```
function Todos() {
  // Access the client
  const queryClient = useQueryClient()
  // Mutations
  const mutation = useMutation({
    mutationFn: postTodo,
    onSuccess: () => {
      // Invalidate and refetch
      queryClient.invalidateQueries({ queryKey: ['todos'] })
    },
  })

  return (
    <div>
      <button
        onClick={() => {
          mutation.mutate({
            id: Date.now(),
            title: 'Do Laundry',
          })
        }}
      >
        Add Todo </button></div>  )}
}
```

- **useMutation({ mutationFn: fetchFct, onSuccess: fct })**
 - mutationFn: la fonction appelé après l'appel à mutation.mutate(data)
 - onSuccess: La fonction appelé quand le fetch c'est bien passé
- **mutation.mutate(data)**
 - A appeler lorsque l'on veut effectuer la mutation
 - Data : les données à envoyer. Sont passé à la fonction mutationFn

useQuery()

Prise en compte de isPending, isError ...

```
function Todos() {
  //
  const { isPending, isError, data, error } = useQuery({
    queryKey: ['todos'],
    queryFn: fetchTodoList,
  })

  if (isPending) {
    return <span>Loading...</span>
  }

  if (isError) {
    return <span>Error: {error.message}</span>
  }

  // We can assume by this point that `isSuccess === true`
  return (
    <ul>
      {data.map((todo) => (
        <li key={todo.id}>{todo.title}</li>
      )))
    </ul>
  )
}
```

Comportement du cache

- Par défaut :
 - quand on recharge le composant :
 - useQuery retourne le contenu actuelle
 - useQuery recharge le cache
 - quand le nouveau contenu est mis à jour, le composant est rafraîchi
 - ➔ il y a une requête à chaque fois !
- Pour éviter les rechargements (dans App.js) :

```
// Query client used to interact with the cache
// We specify the staleTime in order to avoid
// reloading each time the page is re-open.
// Note : we can change to use a delay.
const queryClient = new QueryClient({
    defaultOptions: {
        queries: {
            staleTime: Infinity,
        },
    },
})
```

Affichage des concerts revisité

- isLoading et isError
 - fourni par tanstack-query

```
function ConcertsListWithCache() {
    const navigate = useNavigate();
    // Access the client
    const queryClient = useQueryClient()
    // Queries
    const { isLoading, isError, data: concerts, error } =
        useQuery(['concerts'], ConcertService.fetchAllConcerts)
    // update Mutations
    const updateMutation = useMutation(ConcertService.updateConcert, {
        onSuccess: () => {
            queryClient.invalidateQueries(['concerts'])
        },
    })
    // delete Mutations
    const deleteMutation = useMutation(ConcertService.deleteConcert, {
        onSuccess: () => {
            queryClient.invalidateQueries(['concerts'])
        },
    })

    if (isLoading) {
        return (<div>Loading ...</div>)
    }

    if (isError) {
        return (<div>Error while loading ...</div>)
    }
}
```

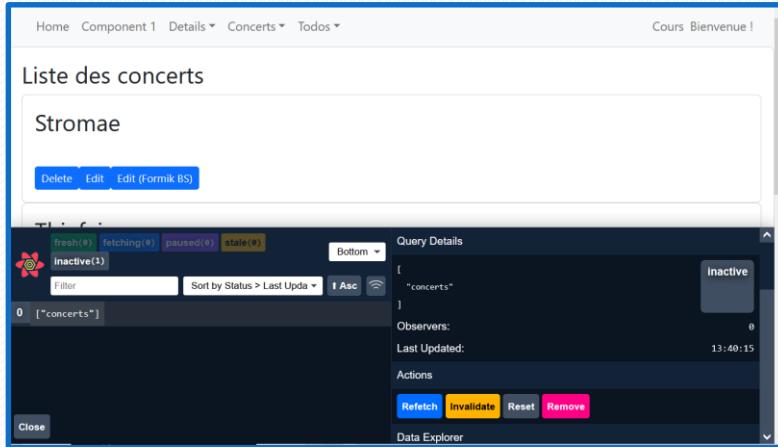
Affichage des concerts revisité (2)

- isLoading et isError
 - fourni par tanstack-query
- L'appel à mutate() entraîne l'appel de la fonction associée (delete ou update)

```
<button className='btn btn-primary btn-sm'  
        type="button"  
        onClick={() => deleteMutation.mutate(concert.id)}  
        >  
        Delete  
</button>
```

Debuguer le cache

- Utiliser ReactQueryDevTool
 - <https://tanstack.com/query/v5/docs/react/devtools>
 - npm i @tanstack/react-query-devtools
- Visible en mode dev
 - invisible en mode prod



```
import {
  QueryClient, QueryClientProvider,
} from '@tanstack/react-query'
import { ReactQueryDevtools }
      from "@tanstack/react-query-devtools";

function App() {
  // Create a client
  const queryClient = new QueryClient();
  return (
    // Provide the client to your ConcertsListWithCache
    <QueryClientProvider client={queryClient}>
      <BrowserRouter>
        <Routes>
          <Route path="/" element={<Layout />}>
            <Route index element={<Home />} />
          </Route>
        </Routes>
      </BrowserRouter>
      <ReactQueryDevtools position="bottom-right" />
    </QueryClientProvider>
  );
}
```

tanstack-query bonne pratiques

- Voir l'article
 - **Effective React Query Keys**
 - <https://tkdodo.eu/blog/effective-react-query-keys>
- **Use Query Key factories**

```
const todoKeys = {
  all: ['todos'] as const,
  lists: () => [...todoKeys.all, 'list'] as const,
  list: (filters: string) => [...todoKeys.lists(), { filters }] as const,
  details: () => [...todoKeys.all, 'detail'] as const,
  detail: (id: number) => [...todoKeys.details(), id] as const,
}
```

```
// Get all
useQuery( todoKeys.list('all'), fetchAllTodo )
// Get filtered
useQuery( todoKeys.list(filters), () => fetchTodos(filters) )
// Get by id
useQuery( todoKeys.detail(id), () => fetchTodoById(id) )

// ✎ remove everything related to the todos feature
queryClient.removeQueries(todoKeys.all)

// ⚡ invalidate all the lists
queryClient.invalidateQueries(todoKeys.lists())
```

Atelier

- Modifier votre application afin de :
 - relocaliser les appels vers le serveur dans des **services** reacts.
 - ~~Utiliser les formulaires Formik et la validation client Yup~~
 - Utiliser les caches tanstack-query