> *Ce tutoriel nécessite des connaissances en ReactJS et React-Redux*

Qu'est-ce qu'une extension ou plugin ?

1. **Définition générale**

> *Les plugins sont de petits programmes complémentaires qui ajoutent des fonctions aux applications Web*

Source : https://www.ionos.fr/digitalguide/serveur/know-how/quest-ce-quun-plugin/

2. **Cas d'usages**

Bien que MapStore2 dispose nativement d'une large palette dd'outils et d'actions, il est parfois nécessaire d'ajouter un outil simple ou complexe aves ses propres fonctionnalités pour complèter l'application.

Ce sera le cas si vos services, souvent avec des activités uniques, souhaitent gérer des données particulières (e.g naturalisme) ou bien consulter des informations d'une façon très spécifiques avec une API distante.

Voici quelques exemples de demandes spécifiques :

- Ajouter un outil de parcours d'intinéraire
- Dispose d'un calculateur du rayonnement solaire par jour et par heure
- Gérer ou consulter des données du cadastre
- Calculer le profil en long (altitude) sur un trait terrestre
- Consulter, gérer et quantifier des données d'urbanisme
- Gestion des espaces verts
- Gestion des réseaux enterrés
- etc.

> *Un MapStore unique pour les gouverner tous...*

Les plugins permettent donc de compléter MapStore2 plutôt que de créer une application complète pour chaque besoin.

Partie 1 - Prérequis

Ces prérequis sont indiqués pour une distribution Linux Debian et permette de compiler un thème depuis les sources.

Pour pouvoir modifier ou créer un thème, vous devez avoir installé :

Nom	Version	lien
nodejs	14.x	guide ici
npm	6.x	vérifier la version
git		download

Vous devez également disposer de :

- un compte administrateur sur la plateforme MapStore2-georchestra ciblée
- un accés administrateur à votre ordinateur

Il est aussi fortement conseillé de lire ces ressources avant de commencer :

- https://github.com/geosolutions-it/MapStoreExtension
- https://mapstore.readthedocs.io/en/latest/developer-guide/extensions/

- https://react-redux.js.org/introduction/getting-started

Si vous avez besoin de découvrir ou revoir ReactJS et React Redux :

- https://www.codecademy.com/learn/react-101
- https://openclassrooms.com/fr/courses/7008001-debutez-avec-react
- https://react-redux.js.org/introduction/getting-started
- https://fr.reactjs.org/docs/hooks-intro.html
- https://fr.reactjs.org/docs/hooks-intro.html

Avec ces prérequis, nous pourrons réaliser une extension simple ou complexe avec le build obligatoire pour obtenir une archive au format ZIP que MapStore2 pourra installer.

Partie 2 - Installer le plugin de base

Récupération et installation du code source

- **Installation**

Accéder au dépôt MapStoreExtension et cloner les sources :

`git clone https://github.com/geosolutions-it/MapStoreExtension.git`

Se positionner à la racine du code source et installer les dépendances :

`npm install`

Si une erreur survient à propose d'une police (ttf), relancer la même commande.

- **Serveur de développement**

Le projet s'appui sur le bundler Webpack pour builder le code et publier une application Mapstore2 en local sur le port `8081`.

Pour le démarrer, exécutez cette commande :

`npm start`

Accédez ensuite l'URL localhost:8081 pour accéder à l'application.

Si vous cliquez sur une carte (e.g Jardin de Rennes), vous pourrez voir que le plugin par défaut `SampleExtension` est activé.

 home extension

Dans ce mode, toutes sauvegardes dans un fichier déclenche un build et un rechargement de la page web (`hot reload`).

Architecture des dossiers et fichiers utiles

- **Fichiers de configuration MapStore2**

Seuls ces fichiers vont nous intéresser dans ce tutoriel pour développer un plugin :

Nom	Description
localConfig.json	Configuration de base de MapStore2
config.json	Fichier de configuration de notre carte par défaut

- **Emplacement du plugin**

L'ensemble des sources de votre extension sera localisé dans le répertoire `/js`. Lorsque l'extension en ZIP est générée, toutes les sources sont localisées dans le répertoire `/dist` par défaut.

Enfin, le contenu du répertoire `/js` et le fichier `/js/app.jsx` nous permettront de configurer ce que l'on va voir par défaut au démarrage de l'application (page d'accueil, carte par défaut, etc...).

Partie 3 - Configurer son environnement MapStore2

Configuration de base

- **Configurer la carte par défaut**

Nous souhaitons par défaut afficher une carte pour éviter d'avoir à chaque sauvegarde (donc build du serveur) la page d'accueil de MapStore2.

Pour cela, ouvrez le fichier `/js/app.jsx` et ajouter en haut du fichier:

`const ConfigUtils = require('@mapstore/utils/ConfigUtils').default;`

Commentez ensuite la ligne :

`appConfig = require('@mapstore/product/appConfig').default;`

Puis, ajoutez cette configuration sous la ligne que nous venons de commentér :

```
let appConfig = {
    ...require('@mapstore/product/appConfig').default,
    pages: [{
        name: "mapviewer",
        path: "/",
        component: require('@mapstore/product/pages/MapViewer').default
    }]
};
```

Rechargez la page localhost:8081 pour constater que nous arrivons maintenant sur une carte.

Ici, c'est le fichier `/config.json` qui est utilisé pour définir la carte que nous voyons par défaut. Vous pouvez le modifier pour ajouter des couches, une emprise etc...

- **Modifier les extensions et outils par défaut**

Si vous désirez rajouter un module comme par exemple l'identification ou la mesure, vous devez modifier le fichier `localConfig.json` et suivre la documentation des plugins MapStore2 pour avoir une condiguration adaptée.

Serveur de développement & proxy

Cette section permet d'avoir certaines clés d'entrées sur le proxy et le serveur de développement pour ceux qui peuvent en avoir besoin.

- **devServer Webpack**

Comme déjà vu, c'est le devServer Webpack qui est utilisé. Nous n'aborderons pas sa configuration car en l'état elle nous convient parfaitement.

Si vous souhaitez cependant rajouter des règles pour le proxy interne MapStore2, suivez la section suivante.

- **Proxy MapStore2**

Vous pouvez utiliser une URL pour votre proxy en utilisant dans le `localConfig.json` la propriété `proxyUrl`.

Vous trouverez un exemple ici.

```
  "proxyUrl": {
    // if it is an object, the url entry holds the url to the proxy
    "url": "/MapStore2/proxy/?url=",
    // useCORS array contains a list of services that support CORS and so do not need a
proxy
    "useCORS": ["http://nominatim.openstreetmap.org",
"https://nominatim.openstreetmap.org"]
  },
```

Si vous désirez configurer le proxy par défaut de MapStore2, vous devrez modifier le fichier `proxy.properties` (dans le datadir pour `mapstore2-georchestra`).

- **Liens utiles**

Pour comprendre le fonctionnement général du backend MapStore2 ou du proxy, suivez ces liens :

https://mapstore2.readthedocs.io/en/user_docs/developer-guide/infrastructure-and-general-architecture/#backend

https://mapstore2.readthedocs.io/en/user_docs/developer-guide/developing/

https://github.com/geosolutions-it/MapStore2/blob/master/project/standard/templates/web/src/main/resources/proxy.properties

Partie 4 - Préparer votre propre extension

Notre extension va afficher un bouton dans la toolbar lattéral de droite pour afficher un message dans une popup MapStore2. Une configuration simple permettra de modifier le message depuis le fichier `localConfig.json`. Nous utiliserons le système de traduction pour le text standard de notre popup.

Trouver un nom

Commencez par trouver un nom à votre extension. Il doit être court et sans caractères spéciaux. Nous prendrons le nom `AfficheHello` pour ce petit tutoriel.

Remplacer l'extension par défaut

Vous pouvez rechercher via votre éditeur de code les occurrences `SampleExtension` et les remplacer par le nom de votre extension.

Sinon, suivez ce guide :

- Dans le fichier `config.js`, remplacez `SampleExtension` par le nom de votre extension.

```
module.exports = {
    name: "SampleExtension"
};
```

... devient avec notre extension `AfficheHello` :

```
module.exports = {
    name: "AfficheHello"
};
```

- Dans `localConfig.json`, remplacez :

```
"desktop": ["Details", "SampleExtension",
```

... par ...

```
"desktop": ["Details",
    {
        "name": "AfficheHello",
        "cfg": {}
    },
```

Ici, nous venons d'indiquer que MapStore2 doit charger notre extension au chargement d'une carte et utiliser le plugin `AfficheHello` la configuration décrite dans la propriété `cfg` actuellement vide.

- Dans `/assets/index.json`, remplacez aussi "SampleExtension" par "AfficheHello"

Dans ce fichier `/assets/index.json`, il est indiqué que notre plugin `AfficheHello` nécessite le plugin générique MapSotre2 "Toolbar". Ce qui est obligatoire pour ajouter notre bouton dans la toolbar standard de MapStore2.

La configuration du projet est terminée, nous devons maintenant développer le plugin pour que le serveur (actuellement en erreur) charge correctement les ressources.

Par où commencer ?

Tout se réalisera dans le répertoire `/js/extension`.

- **Bonnes pratiques**

Il est au préalable nécessaire d'observer certaines **bonnes pratiques** :

1. Le fichier d'entrée du plugin pour MapStore2 est localisé dans `/plugins/Extension.jsx`
2. Les components React seront dans `/components`
3. Les images et autres ressources seront dans `/assets`
4. Organisez à l'avance l'architecture de vos composants pour essayer d'avoir une structure de dossier rapidement compréhensible et qui mettra à l'aise les développeurs (*Regroupez par répertoire tous les fichiers `css`, `jsx` et `js` d'un composant.*).
5. Si le fichier JSX d'un composant est trop long, n'hésitez-pas à le découper en sous-composants de façon **structurée**
6. Conserver un répertoire unique et séparé dans `/extension` pour les `/actions`, `/epics`, `/reducers`, `/selectors`
7. Utiliser des composant fonctionnel React
8. Respecter le code style et utiliser les outils(e.g ESLint) de MapStore2 pour la qualité de code (voir doc Debugging the frontend)
9. Respects général des bonnes pratiques et conventions MapStore2 et ReactJs (e.g cet article)

- **Fichier principal : déclarer notre plugin**

Le fichier principal est dans le répertoire `/js/extension/plugin/Extension.js`. C'est dans ce fichier que l'on commencera pas déclarer notre plugin. Il sera ensuite connectable au

store MapStore2 (via redux). C'est notamment ici qu'on indiquera où afficher notre bouton.

> *Les notions Redux (store, actions, reducers, selector, epics) seront abordées plus tard*

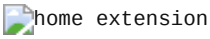
On commencera donc par ce fichier...

Partie 5 - Développer notre extension

L'objectif est de réaliser un plugin simple qui permet de :

- Ajouter un plugin dans MapStore2
- Afficher / masquer une modal
- Ajouter du contenu dans le plugin
- Rendre son plugin dynamique avec Redux dans MapStore2

Voici le résultat que l'on souhaite obtenir :


home extension

Pour notre extension nous commencerons par modifier le fichier principal `extension.jsx` en suivant globalement cette ordre :

1. Imports (librairies, component, actions, etc...)
2. Déclarer notre component principal React
3. Connecter l'extension au store avec Redux
4. Rendre disponible notre extension
5. Interaction principale
6. Développer le contenu

Imports

En haut du fichier, importer React et Redux :

```
import React from "react";
import { connect } from "react-redux";
import {Glyphicon} from 'react-bootstrap';
```

Déclarer notre component principal React

- **Composant principal de l'extension**

Dans `Extension.jsx`, ajouter la fonction utilitaire sous les imports :

`const compose = (...functions) => args => functions.reduceRight((arg, fn) => fn(arg), args);`

Ajouter notre composant principal AfficheHello via une fonction (gràace à l'import React précédent):

```
function AfficheHello({}) {
    return (
        <div></div>
    );
}
```

C'est dans le return de cette fonction que sera ajouter le contenu de notre extension. Nous y reviendront dans quelques étapes.

Connecter l'extension au store avec Redux

Connecter notre extension avec Redux :

```
const AfficheHelloPlugin = compose(
    connect((state) => ({}), {}),
    // setup and teardown due to open/close
    compose()
)(AfficheHello);
```

Rendre disponible notre extension

Nous devons rendre notre extension disponible pour que MapStore2 puisse l'utiliser comme un composant React utilisable.

Ajouter à la suite ces lignes :

```
export default {
    name: "AfficheHello",
    component: AfficheHelloPlugin,
    reducers: {},
    epics: {},
    containers: {
        Toolbar: {
            name: "AfficheHello",
            position: 1,
            icon: <Glyphicon glyph="th" />,
            doNotHide: true,
            alwaysVisible: true,
            action: null,
            priority: 1,
            tooltip: "Extension pour afficher une popup"
        }
    }
};
```

- **Explications**

La propriété `containers` permet d'indiquer que notre plugin sera intégré dans le plugin Toolbar de MapStore2. On pourra alors définir l'icône de notre plugin, la position dans la toolbar et la tooltip au survol de la souris.

Dans cette propriété, on retrouvera la propriété `action` qui nous aidera à faire quelque chose au clic sur le bouton.

> *Les plugins peuvent être ajoutés dans d'autres composants, mais pour notre exemple c'est la Toolbar que nous ciblons.*

La propriété `component` indique quel component utiliser.

Les propriétés `reducers` et `epics` sont indispensables pour connecter ces éléments au store et détecter les changements d'état utile au refresh des composant. C'est entre autre ce qui permettra de rendre dynamique notre composant.

Interaction principale

Par interaction principale, on entendra ici la première interaction qu'un utilisateur aura à réaliser pour gérer l'activation de votre extension.

Nous nous servirons donc ici de la propriété `containers.action` de notre composant principal (null par défaut).

Nous utiliserons ici une action qui sera déclenché au clic sur le bouton dans la toolbar pour activer ou non le plugin.

En haut du fichier, ajouter cette constante :

`export const CONTROL_NAME = 'afficheHello';`

Repérez la ligne :

`const AfficheHelloPlugin = compose(`

Et modifier le code pour obtenir :

```
const AfficheHelloPlugin = compose(
    connect((state) => ({
        // selector
    }), {
        // actions
    }),
    // setup and teardown due to open/close
    compose()
)(AfficheHello);
```

Ajoutez en haut toujours l'import suivant :

```
import { toggleControl } from "@mapstore/actions/controls";
```

> *toogleControl permettra d'appelez une action par son identifiant.*

Remplacez alors dans la propriété `container.action` :

`action: null,`

par ...

`action: toggleControl.bind(null, CONTROL_NAME, null),`

Développer le contenu

Notre composant n'a pas de contenu, on va lui ajouter des choses à afficher.

Dans le répertoire `/component`, créer un fichier `PopupAfficheHello.jsx`. Ce sera notre popup à afficher au clic au sein d'un composant dédié.

Ajouter en haut les imports nécessaires :

```
import React from "react";
```

Créer un composant dans une fonction :

```
export default function PopupAfficheHello({}) {
    return (
        <div></div>
    )
}
```

Pour le moment nous avons juste créé la structure de base du composant, il n'affichera qu'une `<div>` vide, donc rien.

- **Utiliser le composant**

Importer le composant dans le fichier `Extension.jsx`.

`import PopupAfficheHello from "../components/PopupAfficheHello";`

Utiliser ce composant dans le composant `AfficheHello` tel que :

```
// Composant fonctionnel
function AfficheHello({ ...props }) {
    // retourne un contenu et lui passe ses props
    return (
        <PopupAfficheHello {...props} />
    );
}
```

Les propriétés `props` sont passées en tant qu'argument de la fonction `AfficheHello`.

> *On trouvera dans les props les dviers propriétés (`props`), `selectors` et `action` passés via `AfficheHelloPlugin`.*

Nous avons à présent l'essentiel de la structure du plugin. Le bouton dans la toolbar en haut devrait être accessible :

 btn extension

- **Ajoutons une modal**

Notre popup sera en réalité une modal.

Dans le composant `PopupAfficheHello`, nous allons créer une modal à afficher selon si l'état actif ou non de l'extension.

Ajoutez des imports dans le compossant `PopupAfficheHello.jsx` :

`import Modal from '@mapstore/components/misc/Modal';`

> *Nous venons d'importer un composant issu du coeur de MapStore2. De nombreux composants MapStore2 sont ainsi réutilisables.*

Ajoutez maintenant la Modal ouverte par défaut et l'argument `props` passé par le composant parent :

```
export default function PopupAfficheHello(props) {
    return (
        <div className="container-fluid query-toolbar">
            <Modal show bsSize="large">
                <Modal.Header closeButton>
                    <Modal.Title>Mon premier Plugin !</Modal.Title>
                </Modal.Header>
                <Modal.Body>
                    <div>Hello World !</div>
                </Modal.Body>
            </Modal>
        </div>
    );
}
```

Raffraîchissez la page localhost:8081 et voir que la modal est maintenant toujours affichée.

Maintenant, nous souhaitons l'afficher uniquement au clic sur le bouton de notre plugin.

Nous allons donc utiliser un selector, notion du chapitre suivant.

Partie 6 - Créer un selector

- **Principes de Store et de Selector**

Un Selector est une méthode qui permet d'accéder aux valeur du store Redux.

Pour définir le store, la documentation Redux relative indique :

> *A store holds the whole state tree of your application. The only way to change the state inside it is to dispatch an action on it. A store is not a class. It's just an object with a few methods on it. To create it, pass your root reducing function to createStore.*

Un store est donc un objet principal qui permet de stocker des informations.

Pour comprendre les Selectors, parcourons la documentation Redux relative :

> *A "selector function" is any function that accepts the Redux store state (or part of the state) as an argument, and returns data that is based on that state.*

En traduction, c'est une fonction qui prend en argument l'état (state) actuel du store (un objet) et qui retourne une propriété du store :

```
// Arrow function, direct lookup, return store property
const getCurrentCountry = state => state.myCountryPlugin.country
```

Pour nos besoins donc, ajoutez un répertoire `/selectors` et un fichier `/js/extension/selectors/selectors.js` .

Ajoutez alors dans ce fichier `selectors.js` une fonction pour savoir si le plugin est actif ou non:

```
export const CONTROL_NAME = 'afficheHello';
export function isAfficheHelloActif(state) {
    return (state.controls && state.controls[CONTROL_NAME] &&
state.controls[CONTROL_NAME].enabled) || (state[CONTROL_NAME] &&
state[CONTROL_NAME].closing) || false;
}
```

Ensuite, pour utiliser ce selector, rajoutez un import dans le fichier `Extension.jsx` :

`import { isAfficheHelloActif } from "../selectors/afficheHello";`

Puis, modifiez `AfficheHelloPlugin` pour obtenir :

```
const AfficheHelloPlugin = compose(
    connect((state) => ({
        // selector
        isAfficheHelloActif: isAfficheHelloActif // notre selector
    }), {
        // actions
    }),
    // setup and teardown due to open/close
    compose()
)(AfficheHello);
```

Ajoutez un `console.log` avant le `return` dans le composant PopupAfficheHello et observez le contenu des props qui n'est plus vide. On y voit que le selecteur `isAfficheHelloActif` est disponible.

Nous allons nous en servir pour savoir si le plugin est actif.

Utilisez un selecteur

En l'état, notre selecteur n'est qu'une méthode. La documentation indique qu'il faut appeler cette méthode en lui passant l'état du store actuel (le state) pour accéder à la valeur souhaitée (ici, si le plugin est actif ou non).

Dans Extension.jsx, modifiez :

```
    connect((state) => ({
        // selector
        isAfficheHelloActif: IsAfficheHelloActif
```

par ...

```
    connect((state) => ({
        // selector
        isAfficheHelloActif: IsAfficheHelloActif(state)
    }), {
```

Raffraîchir la page et voir que le `console.log` retourne un booléen, qui est bien le résultat du selecteur `IsAfficheHelloActif`.

Modifiez maintenant la propriété `show` de la modal pour qu'elle ne soit visible que si le composant est actif :

`<Modal show={props.IsAfficheHelloActif} bsSize="large">`

On affiche maintenant la Modal au clic sur le bouton du composant.

Mais cette modal ne se ferme pas. Pour la faire disparaître nous devons désactiver le plugin et donc modifiez la valeur dans le store.

Pour cela, nous avons besoin de déclencher une action qui permettra de modifier l'état du store.

Rappelez-vous de la documentation sur le lien entre Action -> Store :

> *The only way to change the state inside it is to dispatch an action on it.*

On utilisera donc l'action de désactivation du plugin par défaut car le selecor retourne un booléen qui permettra de définir si la modal est visible ou non.

Modifiez `AfficheHelloPlugin` pour utiliser le `toggleControl` et obtenir :

```
const AfficheHelloPlugin = compose(
    connect((state) => ({
        // selector
        IsAfficheHelloActif: isAfficheHelloActif(state)
    }), {
        // actions
        closeModal: toggleControl.bind(null, CONTROL_NAME, null)
    }),
    // setup and teardown due to open/close
```

```
    compose()
)(AfficheHello);
```

Modifier la Modal dans `PopupAfficheHello.jsx` pour ajouter une croix et une méthode `onHide` qui permettra de détecter quand la modale est fermée.

Cet événment déclenchera alors notre action `closeModal` passée dans les `props` qui est en réalité `toogleControl` :

`<Modal onHide={props?.closeModal} show={props.IsAfficheHelloActif} bsSize="large">`

> *Notez ici que nous utilisons une action standard de MapStore2 pour interagir l'application et en modifier l'état*

La modal peut maintenant être masquée ou affichée via le bouton dans la toolbar.

En dernière étape, on souhaite modifier la le contenu text de notre modal.

Créer une action

Les actions sont à considérer comme des événements comme expliqué dans [la documentation Redux relative] :

> *Actions are plain JavaScript objects that have a type field. [...] you can think of an action as an event that describes something that happened in the application.*

Simplement, c'est un objets comme celui-ci qui contient une clé type obligatoire et autant de clés que nécessaires pour apporter une valeur dans le store (le nom de la clé est égal au nom de l'argument):

```
export const changeValeur = (uneValeurA, uneValeurB) => {
    return {
        type:"NOM_ACTION",
        uneValeurA,
        uneValeurB
    };
};
```

Nous allons donc créer une action `CHANGE_PRENOM` qui permettra de changer le text à afficher dans la modal par une saisie libre.

Créez donc un répertoire `/js/extension/actions` et ajoutez le fichier `action.js`.

Ajoutez enfin dans ce fichier :

```
export const CHANGE_PRENOM = "CHANGE_PRENOM";

export const changePrenom = (prenom) => {
    return {
        type: CHANGE_PRENOM,
        prenom
    };
};
```

Au déclenchement, cette action retournera un objet comprenant le nom de l'action et la valeur de la propriété du store `prenom`.

Cependant, nous avons vu que les actions sont comme des événements. Lorsqu'elle se déclenchent, elles peuvent créer un objet mais rien ne permet de modifier le store directement.

C'est le rôle des `Reducers` qui devront "Ecouter" les actions et modifier l'état du store.

Créer des Reducers

Pour comprendre, la documentation Redux indique à propos des Reducers :

> *Reducers are functions that take the current state and an action as arguments, and return a new state result. In other words, (state, action) => newState.*

Un Reducer permet donc de modifier l'état du store. C'est ce qui nous intéresse pour modifier l'état du plugin et le changer le prenom dans le store et dans l'interface de la modal.

Créez pour commencer un répertoire `/js/extension/reducers` et ajoutez le fichier `reducers.js`.

Ajoutez ces lignes dans ce fichier et analysez les commentaires) :

```
import { set } from '@mapstore/utils/ImmutableUtils';
// On importe les actions à écouter
import { CHANGE_PRENOM } from '../actions/actions';

// Etat par défaut si non défini à l'initialisation du plugin
const initialState = {
    // notre variable pour savoir si la popup est visible ou non
    prenom: 'World'
};
// reducer
export default function affichePopup(state = initialState, action) {
    // on parcours la valeur de l'action déclenchée
    switch (action.type) {
    // Si c'est pour afficher la popup, alors on change la valeur de visibilité selon
l'action passée
    case CHANGE_PRENOM:
        // on va ici modifié l'état dans le store de MapStore2 et refresh l'UI, notre popup
        return set('prenom', action.prenom, state);
    default:
        return state;
    }
}
```

> *Notez que nous n'utilisons qu'un seul reducer, mais plusieurs peuvent être utilisés*

Dans le fichier Extension.jsx, nous allons ajouter les reducers :

`import affichePopup from "../reducers/reducers";`

Recherchez ces lignes dans Extension.jsx :

`reducers: {},`

Et modifiez les par :

`reducers: { affichePopup: affichePopup },`

Cette fois, notre Reducer est prêt et détectera le déclenchement de notre action.

Utilisons-le dans la section suivante...

Utiliser les Reducers

Maintenant que nous pouvons modifiez le store via un reducer, il nous reste à l'utiliser.

Le workflow est le suivant :

> *Click bouton --> Action --> Reducers --> Modification du store --> Update component --> Update UI*

Nous utiliserons en premier un Hook d'état pour stocker le prénom saisie.

Modifiez l'import React dans le fichier `PopupAfficheHello.jsx` pour importer aussi le Hook `useState` :

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

Ajoutez ensuite la constante telle que :

```
export default function PopupAfficheHello(props) {
    const [inputPrenom, setInputPrenom] = useState("World"); // <-- Constante à rajouter, affiche "World" par défaut
    return (
```

Rajoutez ces imports pour utiliser le composant `<Button>` de MapStore2 :

```
import ButtonRB from '@mapstore/components/misc/Button';
import tooltip from '@mapstore/components/misc/enhancers/tooltip';
const Button = tooltip(ButtonRB);
```

Enfin, rajoutez un `<Input>` et le `<Button>` dans l'élément `Body` pour obtenir :

```
<Modal.Body>
    <h3>Hello { props.prenom }</h3>
    <input
        onChange={e => setInputPrenom(e.target.value)}
        placeholder="Saisir un prénom..."
    /><br/>
    <Button style={{marginTop: "5px"}}
        onClick={() => props.changePrenom(inputPrenom)}
    >Changer de prénom !</Button>
</Modal.Body>
```

> *Notez que dans ce code, la saisie du prenom va mettre à jour la valeur de `inputPrenom`. En effet, il faut considérer cette propriété comme une propriété du state (état) local de notre composant. Cela permet évite manipuler inutilement le store quand ce n'est pas utile et surtout, de garder la main sur la mise à jour de notre composant.*

Pour le moment nous utilisons une action `changePrenom` et un selecteur `prenom` qui ne sont pas connus des propriétés (`props`) de notre extension.

Pour passer ces éléments dans les props, retournez donc dans `Extension.jsx` et suiviez les commentaires de ces lignes :

```
// ajoutez cet import
import { changePrenom } from '../actions/actions';

// rajouter ici getPrenom dans l'import existant
import { isAfficheHelloActif, getPrenom } from "../selectors/afficheHello";
```

Enfin, modifiez `AfficheHelloPlugin` pour y ajouter notre action `changePrenom` et notre selecteur `prenom` comme ceci :

```
const AfficheHelloPlugin = compose(
    connect((state) => ({
        // selector
        IsAfficheHelloActif: isAfficheHelloActif(state),
        prenom: getPrenom(state) // <-- SELECTEUR A AJOUTER
    }), {
        // actions
        closeModal: toggleControl.bind(null, CONTROL_NAME, null),
        changePrenom: changePrenom // <-- ACTION A AJOUTER
    }),
    // setup and teardown due to open/close
    compose()
)(AfficheHello);
```

On pourra alors accéder au `prenom` et à l'action `changePrenom` via les props du component `PopupAfficheHello`.

Vous savez maintenant utiliser une action, un selector, un reducer et un plugin MapStore2 !

Partie 7 - Build du plugin

> *Vous devez avoir des droits administrateurs pour installer un plugin dans MapStore2*

Pour builder le plugin :

`npm run ext:build`

Vous obtiendrez alors un `ZIP` dans `/dist` à utiliser pour l'installer dans votre MapStore2.

 load extension

Partie 8 - Initialisation et configuration d'une extension

Etape 1 - Charger la configuration à l'initialisation

Nous devons récupérer la propriété `pluginCfg` qui contiendra tout ce qui est dans la propriété `cfg` du `localConfig.json`. Pour cela, nous allons déclencher une action à l'initialisation du plugin.

Cette phase d'initialisation pourra aussi servir à réaliser d'autres tâche pour les besoins propres du plugin (e.g ajouter une projection, appeler une configuration distante, etc...).

- **Fichier init.jsx**

Créez un fichier /extension/init.jsx et y ajouter :

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

export default () => (Component) => ({ setUp = () => { }, close = () => { }, ...props }) =>
{
    // configuration load and initial setup
    useEffect(() => {
        if (props.IsAfficheHelloActif) {
            // pass action on setUp
            setUp(props?.pluginCfg);
        }
```

```
        return () => {
            // pass action on close
            close();
        };
    }, [props.IsAfficheHelloActif]);
    return <Component {...props} />;
};
```

- **Modifier le store à l'initialisation**

Dans `Extension.jsx` Ajoutez cet import :

`import init from "../init";`

Créez ensuite une action dans le fichier `action.js` :

```
export const setUp = (pluginCfg) => {
    return {
        type: SETUP,
        pluginCfg
    };
};
```

... et l'ajouter cette action dans le fichier `Extension.jsx` :

`import { changePrenom, setUp } from '../actions/actions';`

Toujours dans `Extension.jsx`, remplacez :

`compose()`

par ...

```
// setup and teardown due to open/close
compose(
    connect( () => ({}), {
        setUp // action SETUP
    }),
    init() // <-- Va déclencher l'action
)
```

Enfin, dans les reducers (fichié dédié), rajoutez :

```
case SETUP:
    return set(`pluginCfg`, action.pluginCfg, state);
```

(Ne pas faire) - Pour ajouter une action à la fermeture du plugin de type `close` on ajoutera une action en plus:

```
compose(
    connect( () => ({}), {
        setUp,
        close // Action à la fermeture
    }),
    init()
)
```

- **Explications**

Nous avons fait en sorte qu'à l'initialisation du plugin, une action `setUp` soit executée. Cette action une fois déclenchée va contenir la propriété `pluginCfg` et va la positionner dans le `store` avec les autres propriétés de notre plugin.

On pourra alors accéder au `state` de notre plugin (initialState dans le fichier `selector`) et à la configuration initial qui sera disponible dans le `state` via :

`state.pluginCfg`

Chaque nouvelle propriété du `localConfig` pour notre extension sera alors directement accessible via un `selector`.

C'est ce que nous allons réaliser dans la prochaine étape...

Etape 2 - Manipuler la configuration

Dans le `localConfig.json`,

Ajouter cette configuration dans notre plugin pour avoir :

```
{
  "name": "AfficheHello",
  "cfg":{
    "defaultText": "hello"
  }
},
```

Créez un selector pour récupérez la configuration :

```
export function getPluginCfg(state) {
    return state?.affichePopup.pluginCfg;
}
```

Ajoutez l'import dans le fichier `Extension.jsx` :

`import { isAfficheHelloActif, getPrenom, getPluginCfg } from "../selectors/afficheHello";`

Passez ensuite ce `selector` dans les props du composant `AfficheHelloPlugin` tel que :

```
const AfficheHelloPlugin = compose(
    connect((state) => ({
        // selector
        IsAfficheHelloActif: isAfficheHelloActif(state),
        prenom: getPrenom(state),
        config: getPluginCfg(state)
    }), {
        // actions
        closeModal: toggleControl.bind(null, CONTROL_NAME, null),
        changePrenom: changePrenom
    }),
    // setup and teardown due to open/close
    compose(
        connect( () => ({}), {
            // action à l'init
            setUp
        }),
        init() // <-- Iinit du plugin, va déclencer l'action setUp
    )
)(AfficheHello);
```

Modifiez ensuite le text `<h3>` de la modal pour accéder à la valeur `defaultText` ajoutée dans le `localConfig` :

```
<h3>{ props.config.defaultText + " " + props.prenom }</h3>
```