Cours Redux Toolkit — Pourquoi plusieurs slices ?

Avec un exemple clair : L'application du restaurant

1. C’est quoi un slice ?

Un **slice** =une partie de l’état +ses fonctions (reducers) pour le modifier.

Métaphore :
Imagine un grand restaurant avec plusieurs équipes.
Chaque équipe gère une tâche **indépendante** :

- Équipe **Cuisine** → prépare les plats
- Équipe **Serveurs** → gère les tables et les commandes
- Équipe **Caisses** → gère les paiements

Chaque équipe connaît **ses propres règles**, pas celles des autres.

Dans Redux Toolkit, **chaque équipe = un slice**.

2. Pourquoi on ne met pas tout dans un seul reducer ?

Si tu mets tout dans un seul reducer, c’est comme ,Une seule équipe dans le restaurant qui fait :

- Cuisine
- Service
- Nettoyage
- Caisse
- Commandes
- Stocks
- Réclamations

C’est le chaos, difficile à gérer, difficile à maintenir.
Avec les slices :

- Chaque partie est indépendante
- Le code est plus court et plus clair
- On trouve facilement les bugs

Exemple clair : L’application d’un restaurant

Ton application gère :

Gestion du Menu (plats) : Slice menuSlice

- ajouter un plat
- supprimer un plat
- changer le prix

Gestion des Commandes : Slice orderSlice

- ajouter une commande
- changer la quantité
- annuler une commande

Gestion de la Caisse : Slice cashSlice

- total des ventes
- ajouter un paiement
- rembourser

Un seul reducer = un restaurant avec une seule équipe

Si on met tout dans un seul reducer :

```
const rootReducer = (state, action) => {

   if(action.type === "ADD_PLAT") ...

   if(action.type === "DELETE_PLAT") ...

   if(action.type === "CHANGE_PRICE") ...

   if(action.type === "ADD_ORDER") ...

   if(action.type === "CANCEL_ORDER") ...

   if(action.type === "ADD_PAYMENT") ...

   if(action.type === "REFUND") ...

}
```

- Impossible de s'y retrouver
- Trop long
- Chaque changement casse autre chose
- Très mauvais pour les grands projets

Avec plusieurs slices : chaque équipe gère son travail

menuSlice.js

```
const menuSlice = createSlice({
  name: "menu",
  initialState: [],
  reducers: {
    addPlat: (state, action) => { state.push(action.payload) },
    deletePlat: (state, action) => {
      return state.filter(p => p.id !== action.payload)
    },
    changePrice: (state, action) => {
      const plat = state.find(p => p.id === action.payload.id)
      if(plat) plat.price = action.payload.price
    }
  }
})
```

orderSlice.js

```
const orderSlice = createSlice({
  name: "order",
  initialState: [],
  reducers: {
    addOrder: (state, action) => { state.push(action.payload) },
    increaseQty: (state, action) => {
      const item = state.find(o => o.id === action.payload)
      if(item) item.qty++
    },
    cancelOrder: (state, action) => {
      return state.filter(o => o.id !== action.payload)
    }
  }
})
```

cashSlice.js

```
const cashSlice = createSlice({
  name: "cash",
  initialState: { total: 0 },
  reducers: {
    addPayment: (state, action) => { state.total += action.payload },
    refund: (state, action) => { state.total -= action.payload }
  }
})
```

Assembler les tranches (slices)

Le store devient organisé comme ceci :

```
const store = configureStore({
  reducer: {
    menu: menuSlice.reducer,
    order: orderSlice.reducer,
    cash: cashSlice.reducer
  }
})
```

C'est comme un manager du restaurant qui réunit toutes les équipes.

C'est quoi Immer ?

Immer est une petite bibliothèque magique utilisée par **Redux Toolkit** pour rendre ton code plus simple.

Son rôle :

> Te permettre de *modifier le state comme si c'était mutable*,
> alors que Redux exige normalement un state *immuable*.

Donc Immer donne l'impression que tu modifies le state **directement**, mais en réalité, il crée un **nouveau state propre** derrière ton dos.

Pourquoi c'est important ?

En React + useReducer classique, tu n'as PAS le droit de faire :

```
state.push(...)
state.qty += 1
state.name = "Nouvelle valeur"
```

Car tu modifies directement l'état, ce qui est interdit :

- risque de bugs
- rendu React incorrect
- Redux ne détecte pas les changements

Donc tu devais faire :

```
return {
  ...state,
  qty: state.qty + 1
}
```

```
return state.map(...)
```

Immer arrive comme un super-héros

> “Fais comme si tu modifiais l’état directement…Je m’occupe du reste.”

Comment Immer fonctionne vraiment ?

C’est très smart.
Immer te donne un draft
Le state dans les reducers de Redux Toolkit n’est PAS le vrai state.
C’est une **copie provisoire** produite par Immer.
Tu peux faire :

```
state.push(...)

state.name = "test"

p.qty++
```

Pas de problème.

À la fin du reducer

- Immer compare le draft avec l’ancien state.

Immer génère un state IMMUTABLE neuf

- Sans que toi tu fasses de spread ...state
- Sans que tu retournes un nouvel objet

Exemple concret

Code que tu écris :

```
addProduct: (state, action) => {
  state.push(action.payload);
}
```

Ce que Immer fait secrètement :

```
const newState = [...state, action.payload];
return newState;
```

Quand Immer n'intervient PAS ?

Si tu écris un reducer comme ceci :

```
return newState;
```

Immer dit : “Ok, tu as pris le contrôle toi-même, je n’interviens plus.”

Donc si tu utilises filter(), map(), etc.

- C’est TOI qui crées le nouvel état
- Immer ne touche pas

Exemple clair avec et sans Immer

Sans Immer

```
case "increaseQty":
  return {
    ...state,
    qty: state.qty + 1
  }
```

Avec Redux Toolkit + Immer

```
increaseQty(state) {
  state.qty += 1;
}
```