


R3.01 - dev. web - côté client (vuejs)

TP n°5 : DrMad - fait sauter la banque

Détails

Écrit par stéphane Domas

Catégorie : R3.01 - dev. Web côté client (vuejs) (/index.php/menu-cours-s3/menu-mmi3exp-sp-17468698)

 Publication : 24 janvier 2021

 Affichages : 923

Préambule

L'objectif principal de ce TP est d'implémenter une pseudo application bancaire et de compléter l'application générale DrMad, afin de mettre en oeuvre le principe des slots. Il y a donc un scénario à deux niveaux.

Le scénario principal est le suivant :

- Quand l'utilisateur se trouve sur la page principale de l'application DrMad, il apparaît :
 - en haut, une barre de navigation avec deux boutons : "Boutique" et "Banque"
 - au centre, un texte du genre "Bienvenue DrMad"
- Quand on clique sur "Boutique", le centre est remplacé par la page d'accueil de la boutique,
- Quand on clique sur "Banque", le centre est remplacé par la page d'accueil de la banque,

Le scénario d'utilisation de la boutique est quasi le même que celle du TP 4. Les différences sont :

- la barre de navigation avec les boutons login, acheter, payer, commander se trouve dans la page d'accueil de la boutique, et que cette barre de navigation utilise la nouvelle version (cf. ci-dessous)
- Quand on veut payer une commande (dans ShopPay), il faut avoir au préalable fait un virement sur le compte de la boutique (cf. section 9), du montant total de la commande. Cela permet d'obtenir un uuid de transaction bancaire.
- Ensuite, dans ShopPay, il faut ajouter un champ de saisie pour pouvoir saisir cet uuid de transaction. Quand on valide, l'API/source locale va d'abord récupérer les infos de la commande, dont son montant. Elle vérifie ensuite que l'uuid de transaction correspond bien à une transaction bancaire existante, avec le montant de la commande en négatif ET un champ destination, représentant l'_id du compte du destinataire, ET que cet _id est bien celui du compte de la boutique (cf. data.js : compte boutique _id = "65d721c44399ae9c8321832c"). Si c'est le cas, la commande est considérée comme payée, sinon elle reste en attente.

Le scénario pour utiliser la banque est le suivant :

- Quand l'utilisateur arrive sur la page d'accueil de la banque il voit apparaître :
 - en haut, une barre de navigation avec un seul bouton dont l'intitulé est "Mon compte" ou bien "Déconnexion" selon s'il y a un compte "courant" ou non
 - à gauche, un menu avec 3 boutons les uns sous les autres : "Solde", "Débit/virement", "Relevé". Ces boutons ne sont actifs que si l'utilisateur a fourni un n° de compte valide via la page "mon compte"
 - au centre, un texte du genre "bienvenue à la banque"
- Quand on clique sur "Mon compte", le centre est remplacé par une page où il est possible de fournir un n° de compte bancaire, avec un bouton pour valider.
- S'il correspond à un n° valide, alors on considère que l'on récupère les informations liées au compte (cf. le tableau bankaccounts dans datasource/data.js), que l'on stocke dans le store. Ces informations représentent "le compte courant" dans la suite de l'énoncé. De plus, l'intitulé du bouton doit devenir "Deconnexion" et les boutons du menu doivent devenir actifs (NB: ce qui implique d'avoir la possibilité d'activer ou non les boutons dans VerticalMenu. A vous de trouver comment faire) et on fait apparaître la page qui donne le solde du compte.
- Quand on clique sur "Deconnexion", cela affiche au centre un message du genre "déconnexion du compte", et après 1 seconde, cela vide le store des informations bancaires, puis revient à l'accueil de la banque.

- Quand on clique sur "Solde", le centre est remplacé par une page où le solde du compte courant apparaît, s'il existe dans le store.
- Quand on clique sur "Débit/virement", le centre est remplacé par une page avec :
 - un champ de saisie permettant de taper une somme,
 - une case à cocher avec un label "Destinataire". Quand on coche cette case, un champ de saisie doit apparaître afin de taper un n° de compte qui sera le destinataire de la somme
 - un bouton pour valider. Un clic sur ce bouton permet de vérifier si la transaction est valide, pour ensuite la créer.
 - Si la vérification échoue, un message d'erreur est affiché dans une boîte de dialogue, sinon, l'uuid de la transaction est affiché.
- Quand on clique sur "Relevé", le centre est remplacé par une page où la liste de toutes les transactions concernant le compte courant sont listées, ce qui inclut celles où le compte est débité mais aussi celles où il apparaît comme destinataire.
- La liste peut être filtrée avec un intervalle de dates.

Télécharger la vidéo de démonstration des scénarii : [[vuejs-scenario-TP5.mkv \(/upload/supports/S3/Vuejs/TP/tp5/vuejs-scenario-TP5.mkv\)](#)]

0°/ Mise en place

La structuration générale reste la même que pour les TP's précédents. De plus, certains composants vont être réutilisés. La mise en place initiale la plus simple consiste donc à :

- créer un nouveau projet avec vue-cli,
- remplacer le répertoire src de ce nouveau projet, par celui du TP 4.

1°/ Modifications sur App et la boutique

Dans le TP4, les boutons de la boutique se trouvent dans une NavBar, elle-même dans le template de App. Il suffit de tout déplacer dans ShopView. D'après les scénarios ci-dessus, App ne contient au final qu'une NavBar avec deux boutons permettant soit d'afficher l'accueil de la boutique, soit de la banque.

Comme le comportement de NavBar va être modifié (cf. section 4), le bouton Logout de la boutique pose problème. En effet, dans le TP4, c'est App qui décide quoi faire quand on clique sur un bouton. Dans ce TP, c'est NavBar qui suit une route quand on clique sur un bouton. Or, suivre une route consiste à afficher un composant et pas exécuter du code, en l'occurrence appeler l'action de shop.js qui fait le logout. A vous de trouver un moyen pour appeler cette action.

Pour payer une commande, il faut fournir un uuid de transaction bancaire. Cela impose des changements dans plusieurs fichiers :

- `controller.js` : la fonction `payOrder()` doit être modifiée pour suivre le scénario donné en préambule, en utilisant la valeur de `data.transactionId` pour vérifier qu'il y a bien une transaction bancaire correspondant au paiement de la commande.
- `ShopPay` :
 - il faut ajouter un champ de saisie dans le template, devant le bouton "Payer".
 - il faut modifier fonction appelée lorsque l'on clique sur ce bouton : lors de l'appel à la fonction de service pour payer, il faut ajouter dans l'objet donné en paramètre un champ `transactionId` dont la valeur est celle du champ de saisie.

2°/ Ajout des services liés à la banque

Le principe reste le même que dans les TP précédents. On crée des fonctions (cf. liste ci-dessous) dans `datasource/controller.js` pour manipuler les données de la source locale (dans `data.js`) liées à la banque. Les fonctions de service sont placées dans `services/bankaccount.service.js`, dans lequel il suffit d'ajouter une paire de fonction pour chaque service :

- La fonction principale, qui sera appelée par les actions du store (ou directement par un composant dans certains cas), se contente d'appeler la fonction secondaire et retourne son résultat,
- La fonction secondaire, qui appelle une des fonctions de `controller.js`, et retourne son résultat.

Remarque : il est conseillé d'utiliser les règles de nommage des fonctions principales et secondaires déjà utilisées lors des TP précédents.

Exemple pour récupérer les informations d'un compte : la fonction principale `getAccount(number)` appelle la fonction secondaire `getAccountFromLocalSource(number)`, qui elle-même appelle la fonction `getAccount(number)` de `controller.js`.

Rappel : les fonctions de `controller.js` doivent retourner un résultat avec une structure qui doit être identique à celle retournée par l'API. Pour rappel, cette structure est du type `{error: num_erreur, status: ..., data: ...}`. S'il y a une erreur, `data` contient le message d'erreur. Sinon, `data` contient un objet dont la structure dépend du type de requête.

Voici la liste des fonctions à définir dans `controller.js` pour gérer la banque à partir de la source de données locale, avec le descriptif de ce qu'elles retournent :

- `getAccount(data)` : permet de récupérer toutes les informations d'un compte si `data.number` correspond à un numéro valide dans le tableau `bankaccounts`, importé du fichier `datasource/data.js`. Si c'est le cas, le champ `data` de la réponse contient l'objet correspondant dans le tableau. Sinon, `data` contient un message du type "numéro de compte invalide"
- `getTransactions(data)` : permet de récupérer toutes les transactions liées à un id de compte (**et pas son numéro**). Pour cela, il faut vérifier dans les objets du tableau `transactions` si `data.idAccount` apparaît dans le champ `account`. S'il existe de tels objets, le champ `data` de la réponse contient un tableau de ces objets. Sinon, `data` contient un message du type "aucune transaction pour ce compte".
- `createWithdraw(data)` : permet de créer un objet transaction dans le tableau `transactions`, avec un montant négatif dont la valeur positive est donnée par `data.amount`, puis de débiter le compte de ce montant. Si `data.idAccount` est valide, le champ `data` de la réponse contient un objet avec comme structure `{ uuid: ..., amount: ...}`. Le champ `uuid` a la même valeur que celle du champ `uuid` de la nouvelle transaction. Le champ `amount` contient le nouveau solde disponible du compte. Si `data.idAccount` n'est pas valide, `data` contient un message du type "id de compte invalide". ATTENTION ! si cette fonction est appelée avec succès par le store, il ne faut pas oublier ensuite de modifier le solde disponible du compte courant.
- `createPayment(data)` : idem que la fonction précédente mais en créant deux transactions. Si `data.idAccount` n'est pas valide, le champ `data` de la réponse contient un message du type "id de compte invalide". Si `data.destNumber` n'est pas valide, `data` contient un message du type "compte destinataire inexistant". Sinon, la fonction ajoute deux objet dans le tableau `transaction`. Le premier est identique à celui créé par la fonction `createWithdraw()`, **MAIS** avec en plus, un champ `destination` qui est l'_id du compte bancaire de destination des fond. Cet _id doit être récupéré grâce à `data.destNumber` qui est le n° de compte du destinataire. Le deuxième objet représente le crédit qui est fait sur le compte du destinataire. Il n'a donc pas de destinataire. Son montant est égal au montant du premier objet mais en positif. Il a la même date, mais un `uuid` différent.

Attention !

Dans le tableau `transactions` de `data.js`, les objets `transaction` ont un champ `date` qui est un objet, contenant lui-même un champ `$date` avec la vraie date. Cela implique que :

- pour comparer la date de deux objets `transaction t1` et `t2`, il faut utiliser quelque chose du genre `if (t1.date.date === t2.date.date) ...`
- quand on crée une nouvelle transaction, il **FAUT** utiliser le même format.

Les objets `transaction` ont également un champ `account`, qui référence le **champ `_id`** d'un compte bancaire et pas le numéro de compte (= champ `number`)

Après avoir écrit des fonctions, il faut créer les fonctions de service associées dans `services/bankaccount.service.js`, en utilisant le principe des TP précédents, à savoir ajouter une paire de fonction pour chaque service :

- La fonction principale, qui sera appelée par les actions du store (ou directement par un composant dans certains cas), se contente d'appeler la fonction secondaire et retourne son résultat,
- La fonction secondaire, qui appelle une des fonctions de `controller.js`, et retourne son résultat.

Remarque : il est conseillé d'utiliser les règles de nommage des fonctions principales et secondaires déjà utilisées lors des TP précédents.

Exemple pour récupérer les informations d'un compte : la fonction principale `getAccount(number)` appelle la fonction secondaire `getAccountFromLocalSource(number)`, qui elle-même appelle la fonction `getAccount(number)` de `controller.js`.

3°/ Modification du store

Dans le TP 3, le store a déjà été modularisé avec un fichier `bank.js` qui représente le module concernant la banque. Cependant, ce qui a été mis en place dans les TP précédents n'est pas totalement adapté au sujet présent. Il faut donc modifier ce module afin de prendre en compte les fonctionnalités demandées :

- Le state doit contenir au minimum deux objets : le compte courant et le tableau des transactions liées à ce compte
- les mutations permettent au minimum de mettre à jour les deux objets du state, et de modifier le solde du compte courant.
- il y a au minimum 4 actions, chacune appelant une des fonctions service de `bankaccount.service.js`

4°/ La barre de navigation

L'objectif est de modifier `NavBar.vue` pour en faire un composant utilisant les props et les scoped-slot pour le rendre facilement paramétrable par le composant parent. De plus, `NavBar` n'envoie plus un signal au parent pour signaler qu'un des boutons a été cliqué : il suit directement une route de vue-router donnée via une props.

- la props `titles` est renommée en `links` et devient un tableau contenant des objets au format : `{ label: ..., to: ... }`. La valeur de `label` est un texte qui s'affiche par défaut dans un bouton et `to` est une route pour vue-router. : `[{ label: "boutique", to: "/shop" }]`
- dans le `<template>`, au lieu d'utiliser directement le `label` pour chacun des boutons, on définit un `scoped-slot` à l'intérieur de la balise `<button>`.
- Par défaut le slot affiche le `label`, mais en tant que `scoped-slot`, il doit donner l'accès au `label` au composant parent, afin que ce dernier puisse customiser son aspect graphique.

On obtient quelque chose du genre :

```
1 <button v-for="(link, index) in links" :key="index" ... >
2   <slot name="nav-button" :label="link.label" >{{link.label}}</slot>
3 </button>
```

- Il faut ensuite ajouter la gestion des clics sur les boutons :
 - en cas de clic, on appelle une fonction `goTo(dest)`, définie dans `methods`, avec comme paramètre `dest` la valeur de `link.to`.
 - la fonction `goTo()` suit la route donnée en paramètre.

Pour mettre en place la barre de navigation principale, il faut modifier `App.vue` pour indiquer à `NavBar.vue` le contenu du slot pour chaque bouton :

- passer comme valeur de la props `links` le tableau `[{ label: "boutique", to: "/shop"}, { label: "banque", to: "/bank"}]`,
- pour le premier bouton, donner comme contenu au slot le mot "boutique" écrit en gras
- pour le deuxième bouton, donner comme contenu au slot une icône ressemblant à une banque.

Remarques :

- dans une application vue, les ressources de type icônes se trouvent généralement dans le sous-répertoire `assets`.
- une fois placées dans ce répertoire, on peut facilement y faire référence via un chemin d'accès

Exemple :

```

1  <template>
2    <div>
3      
4      ...
5    </div>
6  </template>
```

5°/ Un menu vertical

- Dans le répertoire `components`, créer un composant `VerticalMenu.vue`.
- Ce composant fait la même chose que `NavBar.vue`, excepté que la présentation des boutons est verticale afin de ressembler à un menu et que l'on peut intercaler des "titres" entre les boutons.
- Pour ce faire, le composant reçoit une props nommée `items`, qui est un tableau contenant des objets avec la structure `{ type: "...", label: "...", to: "..." }`
- Le champ `type` contient comme valeur "title" ou bien "link".
 - dans le premier cas, le composant doit juste afficher un `scoped-slot` nommé `menu-title`, dont le contenu par défaut est la valeur de `label`. Ce slot donne au parent l'accès à `label`.
 - dans le second cas, le composant affiche une balise `` contenant elle-même un `scoped-slot` nommé `menu-link`. dont le contenu par défaut est un bouton dont le texte est la valeur de `label`. Ce slot donne au parent l'accès à `label`. Un clic sur la balise `` (ou son contenu) permet de suivre la route indiquée par le champ `to`.

6°/ Le composant racine de la banque

- Dans le répertoire `views`, créer un composant `BankView.vue`.
- Ce composant affiche :
 - en haut une barre de navigation avec un seul bouton intitulé "Mon Compte",
 - à gauche, un menu vertical,
 - au centre, un des composants décrit dans la suite, grâce à une balise `<router-view>` dont le nom est `bankmain`.
- Pour la barre de navigation, on utilise le composant `NavBar`, auquel on donne en props le tableau `[{label: "Mon compte", to: "/bank/account"}]`
- Pour le menu, on utilise le composant `VerticalMenu`, auquel on donne en props le tableau

```
[ {type: "title", label: "Opérations"}, {type: "link", label: "Solde", to: "/bank/amount"}, {type: "link", label: "Débit/Virement", to: "/bank/operation"}, {type: "title", label: "États"}, {type: "link", label: "Historique", to: "/bank/history"} ]
```

- On ne définit que le contenu du `scoped-slot menu-title` : on utilise `label` pour créer du HTML qui affiche la valeur de `label` en gras, souligné.
- Pour le `scoped-slot menu-link`, on garde le comportement par défaut, à savoir afficher un bouton.
- Grâce à `<router-view>`, le composant affiche différents composants grâce à une route à 2 niveaux.
- Il faut donc modifier `router/index.js` en conséquence.
- la route `/bank` est la route racine. Elle doit afficher le composant `BankView` (NB : grâce à `<router-view>` se trouvant dans `App.vue`) et elle a 6 enfants, qui vont tous s'afficher dans l'emplacement `bankmain`.
- la route `/bank/home` doit permettre d'afficher `BankHome.vue`. Elle a comme alias `/bank`
- la route `/bank/account` doit permettre d'afficher `BankAccount.vue`.
- la route `/bank/amount` doit permettre d'afficher `BankAmount.vue`.
- la route `/bank/operation` doit permettre d'afficher `BankOperation.vue`.
- la route `/bank/history` doit permettre d'afficher `BankHistory.vue`.
- la route `/bank/logout` doit permettre d'afficher `BankLogout.vue`.

7°/ L'accueil de la banque

- Dans le répertoire `views`, créer un composant `BankHome.vue`.
- Ce composant affiche juste un message de bienvenue à la banque.

8°/ Afficher le solde

- Dans le répertoire `views`, créer un composant `BankAmount.vue`.
- Ce composant affiche juste le solde du compte courant.
- L'aspect du texte affiché doit être paramétrable par le composant parent via un `scoped-slot` nommé `account-amount`. Ce `scoped-slot` affiche par défaut le solde, sans formatage. Il donne également accès au solde au parent.
- Dans le cas présent, c'est `BankView` qui est le parent. Il doit donner à son fils du HTML pour créer un champ de saisie non éditable, contenant le solde disponible, suivi du symbole €, en rouge si le solde est négatif, en vert si le solde est positif (NB: pour gérer la couleur, il suffit d'utiliser un nom de classe conditionnel dans le champ `<input>` qui va utiliser un style `route` ou `vert` en fonction du solde donné par le composant fils)

9°/ Débit & virements

- Dans le répertoire `views`, créer un composant `BankOperation.vue`.
- Ce composant affiche :
 - un titre, qui est par défaut "Débit / Virement", mais qui peut être changé via un slot.
 - un champ de saisie pour un montant,
 - une case à cocher avec comme label "Destinataire", suivie d'un champ de saisie qui n'apparaît que si la case est cochée.
 - un bouton "Valider".
- Le fonctionnement du composant doit simplement suivre le comportement donné dans le scénario, sachant que :
 - si l'opération est valide, un texte est affiché sous le bouton "valider", du type : "L'opération est validée avec le n° : xxx-xxx-xxx. Vous pouvez la retrouver dans l'historique". Le n° est l'uuid renvoyé dans la réponse du service. Ce texte doit être affiché pendant 5 secondes puis s'effacer.
 - sinon, une boîte de dialogue apparaît avec un message d'erreur.

10°/ Un tableau dynamique avec case à cocher

L'objectif est de faire un composant `DataTable.vue`, utilisant les props `items`, `headers`, `itemCheck`, `itemButton`, `tableButton` ainsi que slots (cf. description ci-dessous) pour le rendre facilement paramétrable par le composant parent.

Ce composant permet d'afficher :

- une table avec X , $X+1$ ou $X+2$ colonnes et $Y+1$ lignes, sachant que :
 - la première ligne sert à afficher les entêtes de colonne, avec au minimum X entêtes provenant des X objets dans `headers`,
 - les données à affichées sont dans `items`, qui contient Y objets, l'objectif étant d'afficher la valeur de ses champs correspondant aux noms de ceux dans `headers`. Chaque champ doit être sur une même ligne de la table. La structure de `items` est décrite plus loin.
 - si la props `itemCheck` vaut `true`, la première colonne, avec comme entête "select.", doit afficher des cases à cocher, et les X suivantes, la valeur des champs des objets reçus via la props `items`. Sinon, on commence chaque ligne avec directement les champs des objets.
 - si la props `itemButton` vaut `true`, il faut ajouter une colonne à droite, avec comme entête "actions", contenant des boutons. Le label (= texte) de ce bouton doit être paramétrable via un slot.
 - selon la valeurs des 2 props précédentes, on a donc bien entre X et $X+2$ colonnes et entêtes à afficher.
 - la props `headers` contient des objets décrivant les entêtes de chaque colonne, excepté celles des cases à cocher et boutons, qui n'ont pas d'entête. La structure de `headers` est décrite plus loin.
- un éventuel bouton après la table, qui apparaît ou non selon la valeur de la props `tableButton`. Le label de ce bouton doit être paramétrable via un slot.

Pour qu'un composant parent fournisse des données à `DataTable`, il doit respecter une structure stricte concernant les props `headers` et `items` :

- `headers` est de type tableau d'objets. Chaque objet est associé à une des X colonnes, avec comme format `{ label: "...", name: "..."}`. `label` est le texte qui doit être affiché dans l'entête d'une colonne, et `name` est son nom, à savoir celui qui est utilisé comme nom de champ dans la props `items`.
- `items` est de type tableau d'objets, chaque objet contenant les données à afficher sur une même ligne de la table, avec comme format `{ nom_colonne1: val1, nom_colonne2: val2, ...}`. Le nom des champs doit être le même que celui indiqué dans les champs `name` de `headers`. NB: il est possible que `items` contienne d'autres champs, auquel cas ils ne seront pas utilisés dans le tableau.

Par ailleurs, les clics sur :

- les boutons doivent envoyer un événement au parent, nommé `itemClicked`, avec comme valeur l'objet `item` de la ligne où le bouton a été cliqué.
- le bouton d'après `table` doit envoyer un événement au parent, nommé `tableClicked`, avec comme valeur un tableau contenant tous les objets `item` sélectionnés.

Exemple d'utilisation dans un composant parent qui veut afficher une table avec 4 colonnes (1 pour les cases à cocher, 2 pour les champs de chaque item, 1 pour les boutons), ainsi qu'un bouton après la table. Il y a 4 lignes (1 pour l'entête, 3 pour les données). A noter que dans cet exemple, on ne capture aucun des événements :

```

1  <template>
2    ...
3    <DataTable :headers="headers" :items="items" itemCheck itemButton tableButton></DataTable>
4    ...
5  </template>
6  <script>
7    ...
8    data: () => ({
9      headers : [ {label: "nom", name: "virus"}, { label: "prix", name: "price"} ],
10     items: [
11       { virus: "grippe", price: 1000, strength: 5 }, // NB: strength ne sera pas visualisé da
12       { virus: "covid", price: 150, strength: 5 },
13       { virus: "cholera", price: 6666, strength: 20 },
14     ]
15   })
16   ...
17 </script>

```

11°/ L'historique des transactions

- Dans le répertoire `views`, créer un composant `BankHistory.vue`.
- Ce composant affiche :
 - un titre, qui est par défaut "Opérations passées", mais qui peut être changé via un slot
 - une case à cocher avec comme label "Filtrer par période", suivie de deux champs de saisie qui n'apparaissent que si la case est cochée, avec comme labels respectifs "Du", "Au".
 - un composant `DataTable`, contenant des transactions associées au compte courant. Par défaut, elles sont toutes affichées sauf si le filtrage est actif.
 - Que le filtrage soit actif ou non, les transactions passées sont classées par order chronologique décroissant, donc avec les plus récentes en premier.
- Le comportement du filtrage est le suivant :
 - si seul le premier champ est rempli, on sélectionne toutes les transactions à partir de cette date,
 - si seul le deuxième champ est rempli, on sélectionne toutes les transactions jusqu'à cette date,
 - si les deux champs sont remplis, on sélectionne les transactions entre les deux dates.
 - si le premier champ est déjà rempli, et que l'on tape une date antérieure dans le second champ, ce dernier est vidé puisque invalide.
 - si le deuxième champ est déjà rempli, et que l'on tape une date postérieure dans le premier champ, ce dernier est vidé puisque invalide.
 - dès que l'on valide la saisie d'un des champs, le filtrage doit se faire et donc impacter l'affichage de la table.
- La table doit afficher :
 - 1 colonne avec des cases à cocher,
 - 1 colonne avec le montant des transactions,
 - 1 colonne avec la date des transactions,
 - 1 colonne avec une lettre, qui doit être S si le compte courant est la source de la transaction (= montant négatif) et D si elle est la destination (= montant positif).
 - 1 colonne avec des boutons, dont le label est "Détails"
 - 1 bouton après la table avec comme label "Voir".
- Quand une des boutons d'item est cliqué, une boîte de dialogue s'ouvre avec dedans l'uuid de la transactions.
- Quand le bouton d'après table est cliqué, une boîte de dialogue s'ouvre avec dedans l'uuid de toutes les transactions dont la case est cochée.

12°/ Déconnexion

- Dans le répertoire `views`, créer un composant `BankLogout.vue`.
- Ce composant affiche simplement un texte signifiant la déconnexion, puis au bout de 1 seconde, utilise la mutation de `bank.js` pour "effacer" le compte courant et ensuite rediriger vers l'accueil de la banque.
- NB : il faut que l'attente et la redirection se fasse sans intervention de l'utilisateur.