03-account

Vamos a partir del ejemplo anterior `02-account-list` e implementar la página de edición y creación de una cuenta.

Pasos a realizar

- Vamos a crear el fichero principal `account.js`:

./src/pages/account/account.js

```
console.log('account page');
```

- Y referenciarlo en el html:

./src/pages/account/account.html

```
...
    <footer id="footer">
      <div class="container">
        <img class="footer_logo"
src="../../core/content/img/logo_footer.svg" />
      </div>
    </footer>
+   <script src="account.js"></script>
  </body>
</html>
```

- Al igual que hicimos con el formulario de login, vamos a recoger los valores de los campos:

./src/pages/account/account.js

```
- console.log('account page');
+ import { onUpdateField, onSubmitForm } from '../../common/helpers';

+ let account = {
+   id: '',
+   type: '',
+   alias: '',
+ };

+ onUpdateField('type', event => {
+   const value = event.target.value;
```

```
+   account = { ...account, type: value };
+ });

+ onUpdateField('alias', event => {
+   const value = event.target.value;
+   account = { ...account, alias: value };
+ });

+ onSubmitForm('save-button', () => {
+   console.log({ account });
+ });
```

- Añadimos el esquema de validación:

./src/pages/account/account.validations.js

```
import { Validators, createFormValidation } from '@lemoncode/fonk';

const validationSchema = {
  field: {
    type: [
      {
        validator: Validators.required,
        message: 'Campo requerido',
      },
    ],
    alias: [
      {
        validator: Validators.required,
        message: 'Campo requerido',
      },
    ],
  },
};

export const formValidation = createFormValidation(validationSchema);
```

- Y validamos el formulario:

./src/pages/account/account.js

```
import {
  onUpdateField,
  onSubmitForm,
+ onSetError,
+ onSetFormErrors,
} from '../../common/helpers';
+ import { history } from '../../core/router';
```

```
+ import { formValidation } from './account.validations';

...

onUpdateField('type', event => {
  const value = event.target.value;
  account = { ...account, type: value };

+ formValidation.validateField('type', account.type).then(result => {
+   onSetError('type', result);
+ });
});

onUpdateField('alias', event => {
  const value = event.target.value;
  account = { ...account, alias: value };

+ formValidation.validateField('alias', account.alias).then(result => {
+   onSetError('alias', result);
+ });
});

onSubmitForm('save-button', () => {
- console.log({ account });
+ formValidation.validateForm(account).then(result => {
+   onSetFormErrors(result);
+   if (result.succeeded) {
+     history.back();
+   }
+ });
});
```

> Con el método `history.back` podemos volver a la página anterior de donde navegamos hacia el formulario de edición / creación de cuenta.

- El siguiente paso, podría ser crear la api para guardar los datos. Si analizamos la página, podemos ver que hay 2 flujos a implementar:
    - Por un lado, podemos navegar seleccionando la cuenta a editar. En este caso, necesitamos recuperar primero los valores actuales de la cuenta, por tanto necesitamos el método `getAccount` para recuperarlos mediante el id de la cuenta. Este `id` vendrá informado en la url mediante el párametro `?id=valor`.
    - Una vez recuperados dichos valores, podemos actualizarlos y utilizar el método `updateAccount` para actualizar la cuenta.
    - Por último, podemos navegar mediante el botón `Agregar nueva cuenta`, por tanto necesitamos recuperar los nuevos valores del formulario y utilizar el método `insertAccount` para crear la nueva cuenta en servidor.

./src/pages/account/account.api.js

```
import Axios from 'axios';

const url = `${process.env.BASE_API_URL}/account`;

export const insertAccount = account =>
  Axios.post(`${url}/${account.id}`, account).then(({ data }) => data);

export const getAccount = id =>
  Axios.get(`${url}/${id}`).then(({ data }) => data);

export const updateAccount = account =>
  Axios.put(`${url}/${account.id}`, account).then(({ data }) => data);
```

> NOTA: Cuando utilizamos los métodos `insertAccount` y `updateAccount`, se actualizará el fichero `./server/src/data.json` con los nuevos valores.

- Además, si echamos un vistazo al modelo que tenemos en servidor y en cliente, podemos ver que hay algunas diferencias:
  - Servidor:

```
{
  id: string;
  iban: string;
  type: string;
  name: string;
  balance: number;
  lastTransaction: string;
}
```

  - Cliente:

```
{
  id: string;
  type: string;
  alias: string;
}
```

- Por tanto necesitamos un `mapper` que transforme la entidad del modelo de API al de la vista (para el flujo cuando recuperamos la información de servidor). Y otro de la vista a la API, cuando queremos `insertar` una nueva cuenta o `actualizarla`:

./src/pages/account/account.mappers.js

```
export const mapAccountVmToApi = account => ({
  ...account,
  name: account.alias,
});

export const mapAccountApiToVm = account => ({
  ...account,
  alias: account.name,
});
```

> NOTA: en este caso necesitamos utilizar el `spread operator` para conservar los valores de los campos `iban`, `balance` y `lastTransaction`, ya que éstos no son utilizados en el formulario.
>
> Como propuesta, podéis utilizar el siguiente código para comprobar que ocurre si no usamos el `spread operator`:

```
export const mapAccountVmToApi = account => ({
  id: account.id,
  type: account.type,
  name: account.alias,
});

export const mapAccountApiToVm = account => ({
  id: account.id,
  type: account.type,
  alias: account.name,
});
```

- Ahora, lo siguiente sería recuperar el parámetro `id` de la url para saber si estámos en modo edición o creación:

./src/pages/account/account.js

```
import {
  onUpdateField,
  onSubmitForm,
  onSetError,
  onSetFormErrors,
+ onSetValues,
} from '../../common/helpers';
import { history } from '../../core/router';
import { formValidation } from './account.validations';
+ import { insertAccount, getAccount, updateAccount } from './account.api';
+ import { mapAccountVmToApi, mapAccountApiToVm } from './account.mappers';

let account = {
```

```
  id: '',
  type: '',
  alias: '',
};

+ const params = history.getParams();
+ const isEditMode = Boolean(params.id);

+ if (isEditMode) {
+   getAccount(params.id).then(apiAccount => {
+     account = mapAccountApiToVm(apiAccount);
+     onSetValues(account);
+   });
+ }
```

> NOTA: Con el método `onSetValues` asignamos los valores de servidor a los elementos `HTML`.

- Por último haremos un update o un insert según estemos en modo edición o no:

./src/pages/account/account.js

```
...

+ const onSave = () => {
+   const apiAccount = mapAccountVmToApi(account);
+   return isEditMode ? updateAccount(apiAccount) :
insertAccount(apiAccount);
+ };

onSubmitForm('save-button', () => {
  formValidation.validateForm(account).then(result => {
    onSetFormErrors(result);
    if (result.succeeded) {
+     onSave().then(() => {
        history.back();
      });
    }
  });
});
```