Star Wars API

* *Internship project –*

Documentation

Mentor: Selena Simic Nikola Tasic

Contents

[Visual representation 3](#_Toc107392707)

[Features implementation 8](#_Toc107392708)

[1. Paging 8](#_Toc107392709)

[2. Search 11](#_Toc107392710)

[3. Creating a character 12](#_Toc107392711)

[4. Deleting a character 14](#_Toc107392712)

[5. Films modal 15](#_Toc107392713)

[6. Character details page 16](#_Toc107392714)

[7. Localization 17](#_Toc107392715)

[Redux-Store 19](#_Toc107392716)

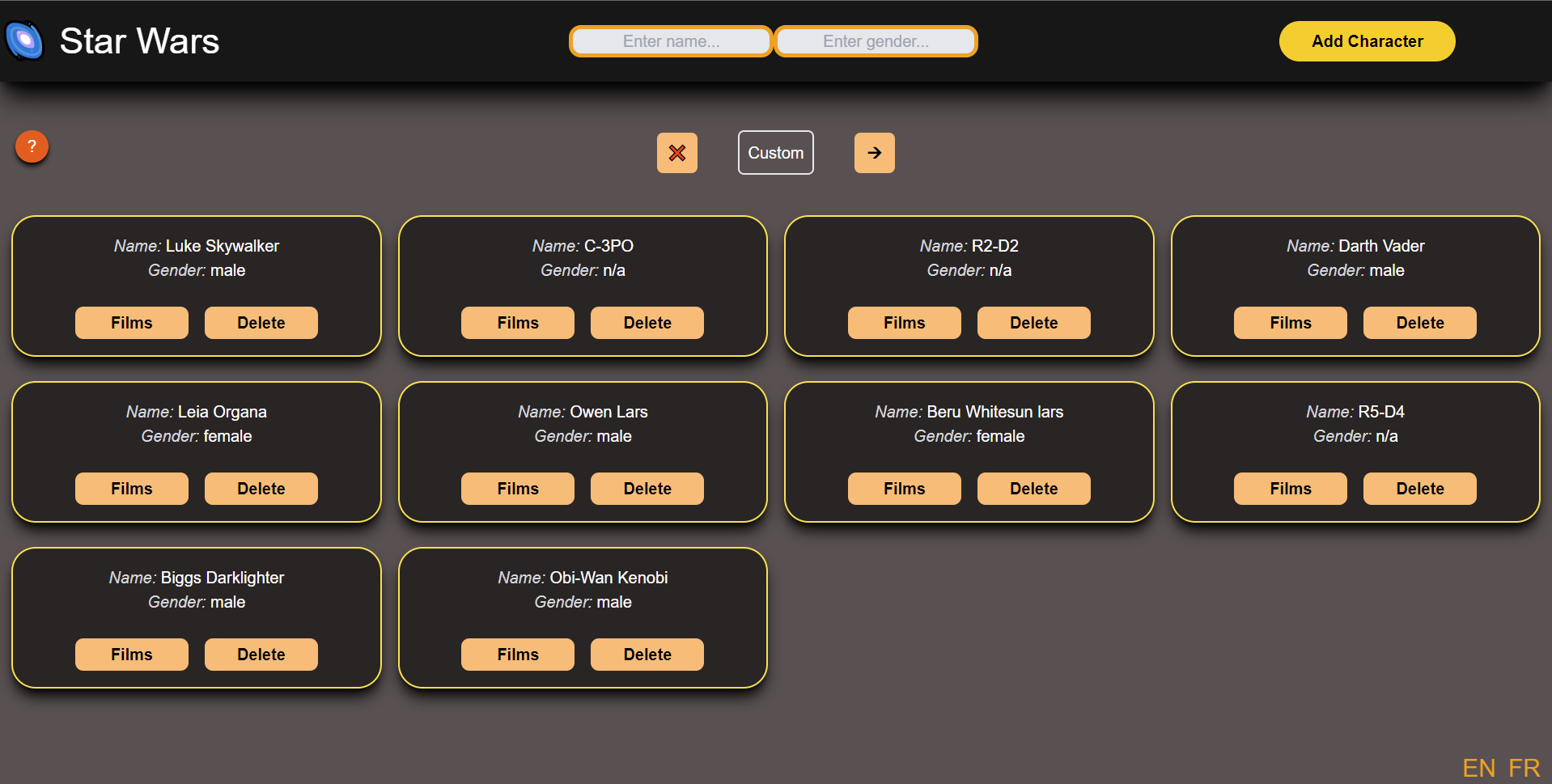
[Sagas 23](#_Toc107392717)

[React Query 25](#_Toc107392718)

[Styling 27](#_Toc107392719)

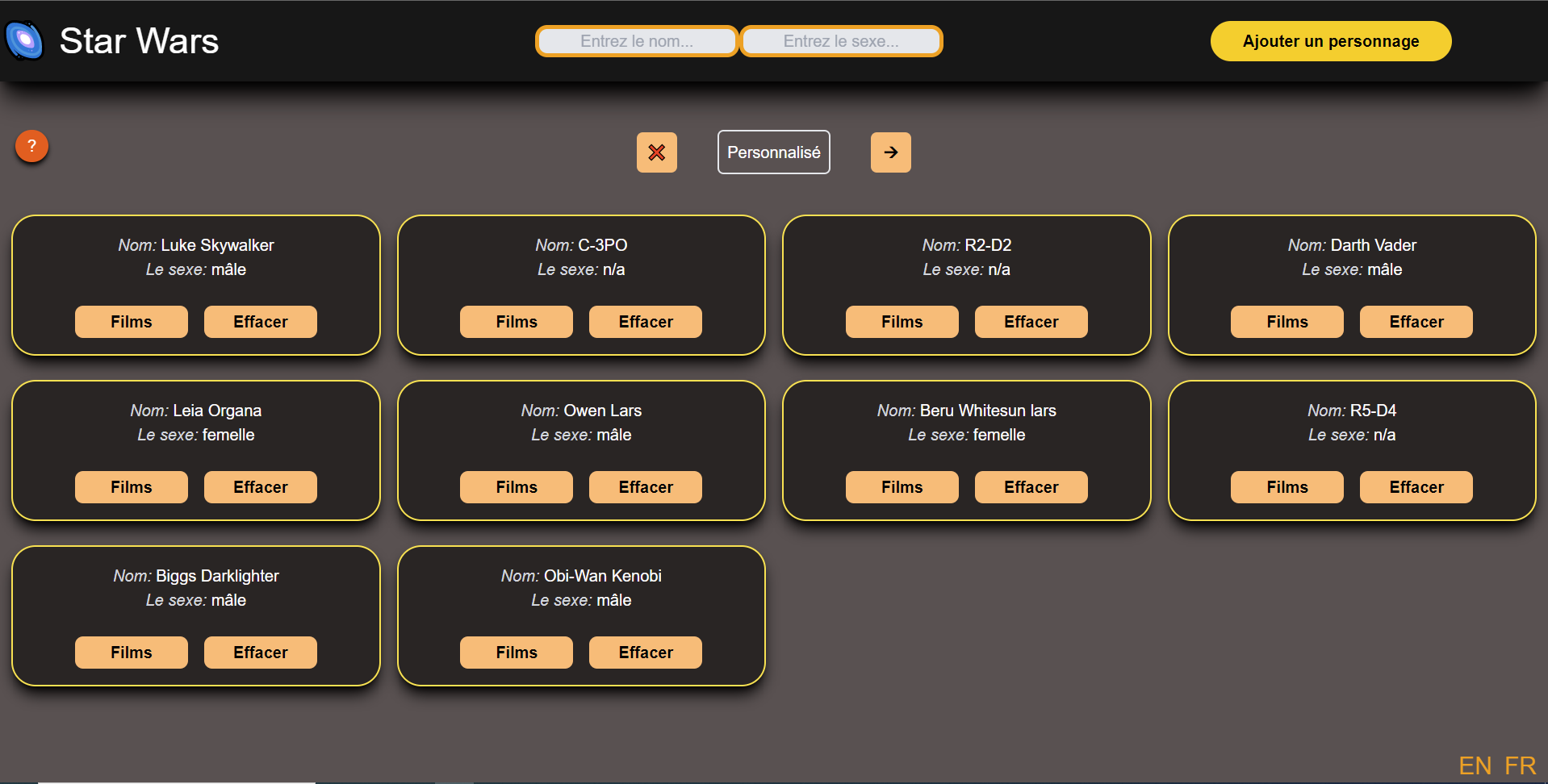
# Visual representation

Upon starting the project, we are presented with a web page that contains a fixed header and the first page of a list of Star Wars characters (Image 1.).



*Image 1.*

For now, the project supports the French and English languages to give a better user experience. It can be easily switched by clicking a button in the bottom right corner (Image 2.).

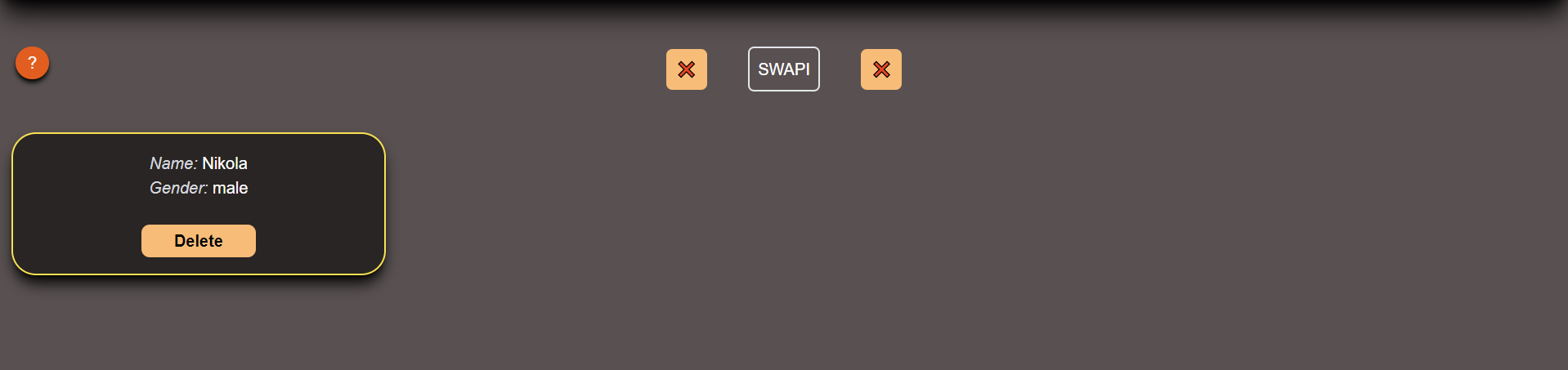


*Image 2.*

The user can page through Star Wars characters that SWAPI provides or through their custom characters that can be easily created. Clicking on a button with an arrow symbol will lead a user to the next/previous page. To switch lists, a user needs to do is click on a ‘Custom’ or ‘SWAPI’ button, depending on the list that is currently visible. (Images 3. and 4.)



*Image 3.*

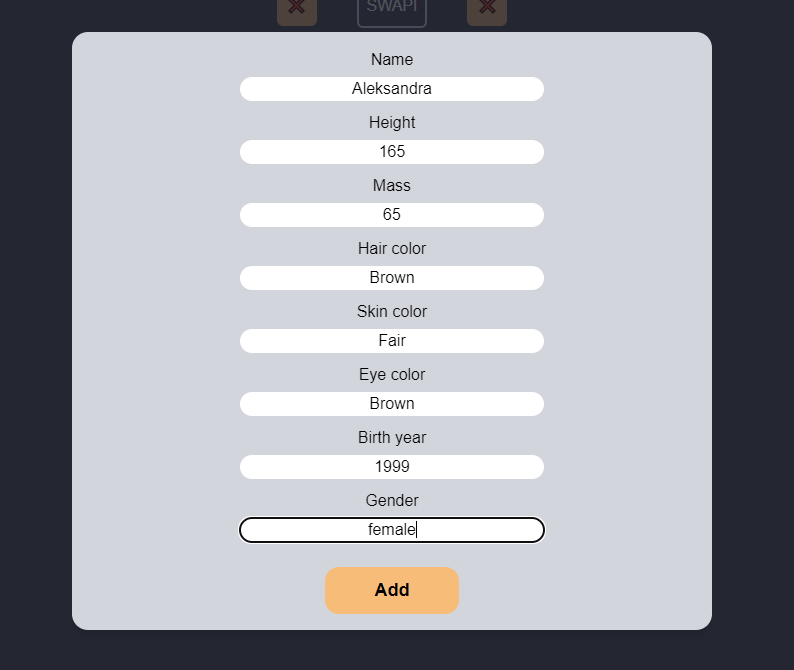


*Image 4.*

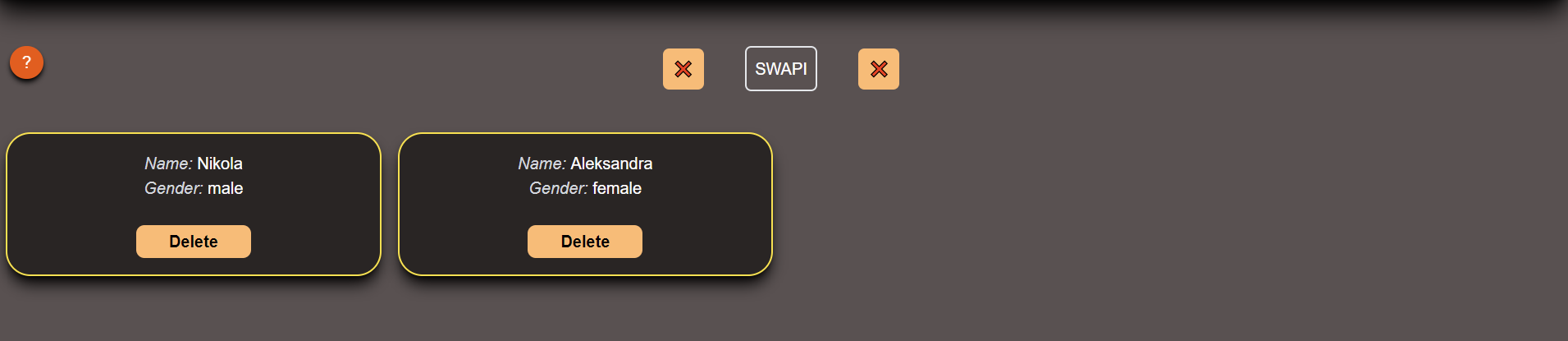
Adding a custom character is done by clicking on an ‘Add Character’ button in the top right corner. On a click, the modal is presented with input fields that need to be correctly filed to submit your character. Just by clicking on an ‘Add’ button, the new character can be found in a custom list. (Images 5. , 6. and 7.)



*Image 5.*

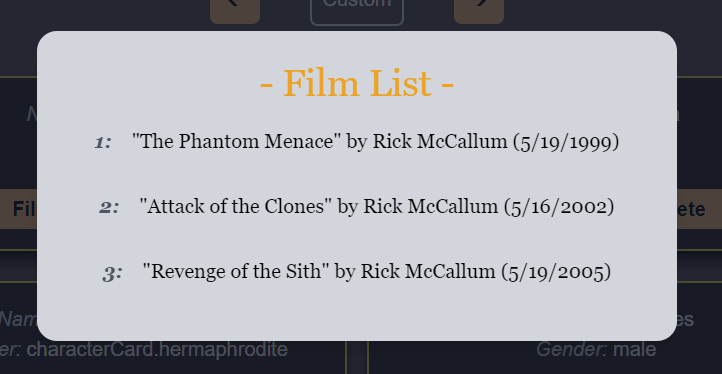


*Image 6.*



*Image 7.*

All characters have their card with the ‘Films’ and ‘Delete’ buttons. Clicking on a ‘Film’ button will open a modal with films starred by this character, and clicking a ‘Delete’ button will remove a card from the current page. (Images 8. and 9.)

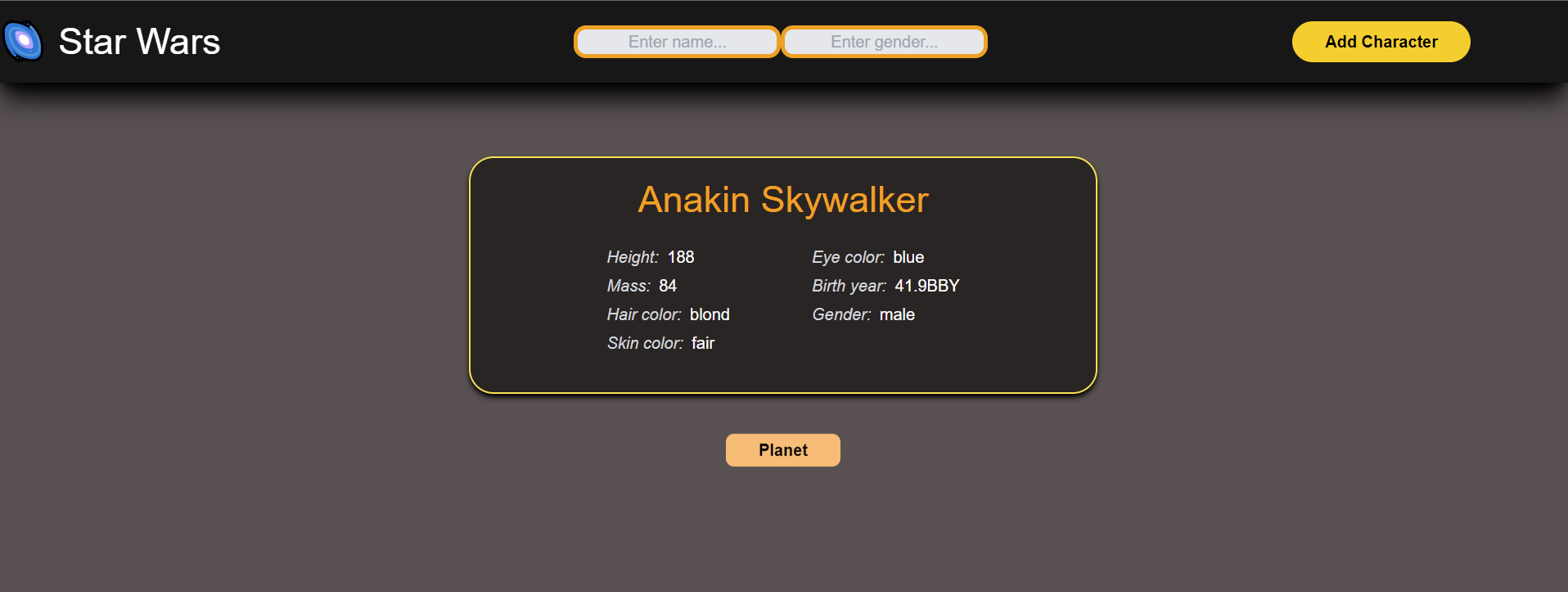


*Image 8.*

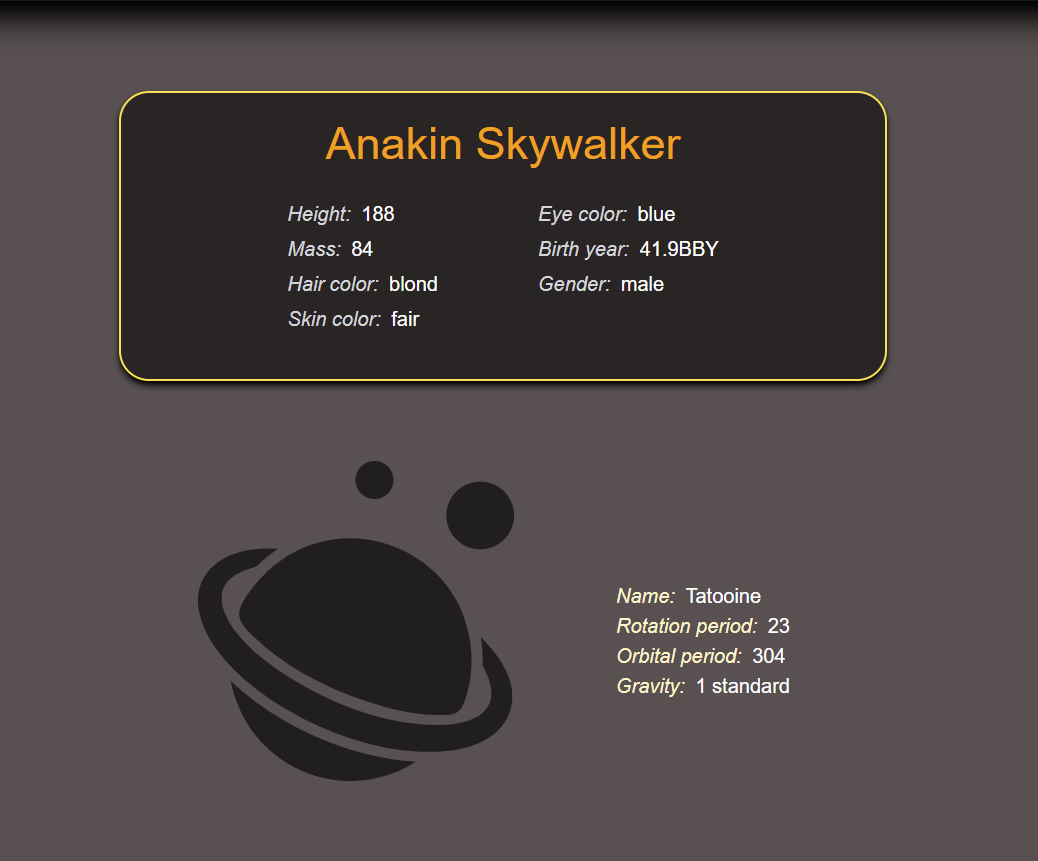


*Image 9.*

Every card is clickable and will lead a user to a page with more information about a character, and if a user is interested in the character’s home planet, it can find information by clicking a ‘Planet’ button. (Images 10. and 11.)



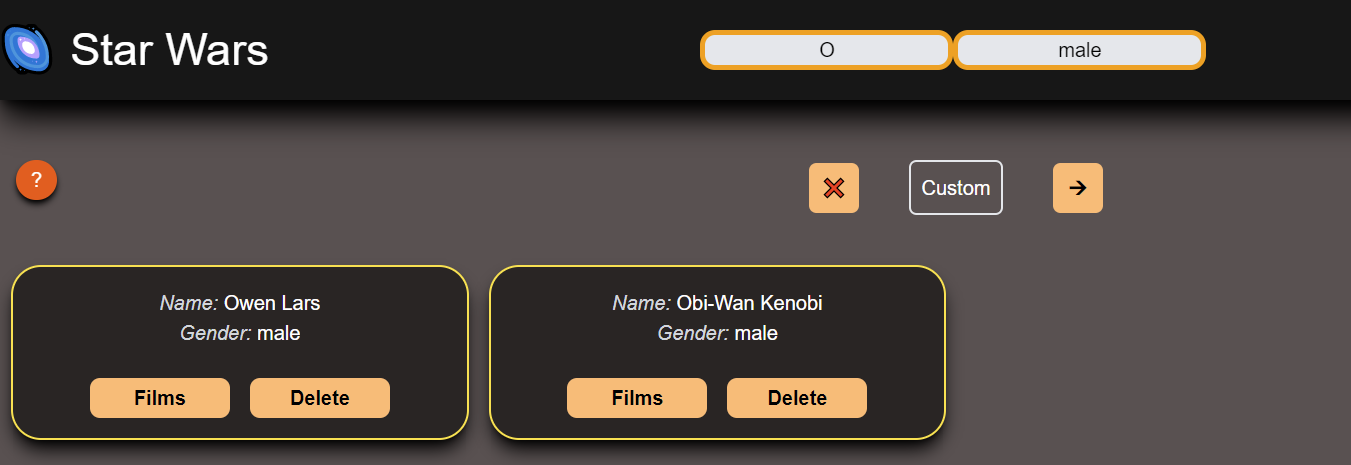
*Image 10.*



*Image 11.*

Not finding a character is not a problem because in the header section there are two input fields for filtering cards by name or gender. Filters are only applied on the page a user is on. (Image 12.)





*Image 12.*

# Features implementation

## Paging

Pagination.jsx

  const [isFromApi, setIsFromApi] = useState(true);

  const [currentPageNumber, setCurrentPageNumber] = useState(1);

  const [pageNumberApi, setPageNumberApi] = useState(0);

In Pagination.jsx we manage three states, isFromApi to tell us if data came from a server or it is a custom data that is preserved just in redux-store. Next, currentPageNumber as the name says keeps track of a current page user is on in a custom list, it’s the same as pageNumberApi but for the SWAPI list.

const charactersDataArray = useSelector((state) => state.swData.results);

  const charactersData =

    charactersDataArray.length > 0 ? charactersDataArray[pageNumberApi] : [];

  const next = charactersData?.next;

  const previous = charactersData?.previous;

App.js

useEffect(() => {

    if (initialRender) {

      initialRender = false;

      dispatch({

        type: "FETCH\_SWDATA",

        payload: "https://swapi.dev/api/people",

      });

    }

  }, [dispatch]);

For the SWAPI list, we are getting data from redux-store and storing it in charactersData but only based on pageNumberApi. On initial render, in App.js, we are dispatching an action to fetch data from a swapi.dev and store it in redux-store.

The next and previous constants are for enabling/disabling paging buttons.

 const customCharactersData = useSelector(

    (state) => state.swData.customCharacters

  );

 const lastPage = Math.ceil(customCharactersData.length / dataLimit);

  const startIndex = currentPageNumber \* dataLimit - dataLimit;

  const endIndex = startIndex + dataLimit;

There is a different approach for custom data. Since custom data is not divided as arrays of 10 elements like data from swapi.dev, startIndex and endIndex are keeping a track of the slice that needs to be displayed.

<CardList

          fromApi={isFromApi}

          pageNumberApi={pageNumberApi}

          openModal={props.openModal}

          filter={props.filter}

          data={

            isFromApi

              ? charactersData?.results

              : customCharactersData.slice(startIndex, endIndex)

          }

        ></CardList>

Now we have prepared data and based on data origin we just pass it down to a CardList component.

  const loadPageHandler = (e) => {

    if (isFromApi) {

      const testPage =

        e.target.name === "next" ? pageNumberApi + 1 : pageNumberApi - 1;

      if (!charactersDataArray[testPage])

        dispatch({

          type: "FETCH\_SWDATA",

          payload: e.target.name === "next" ? next : previous,

        });

      setPageNumberApi(testPage);

    } else

      setCurrentPageNumber((prevPageNum) =>

        e.target.name === "next" ? prevPageNum + 1 : prevPageNum - 1

      );

  };

To give a smoother experience, and without the use of any libraries for paging, a loadPageHandler function is responsible for paging. If data come from a server, we check if it’s loaded to redux-store and just increase the page number, and if not we dispatch an action to fetch it from the server. For custom data, we just need to increase the page number and the startIndex/endIndex constant will do the work for us.

CardList.jsx

const cards = characterData.map((character, i) => {

      return (

        <CharacterCard

          key={i}

          fromApi={props.fromApi}

          characterData={character}

          loadCharacter={loadCharacterHandler}

          selectFilms={loadFilms}

          deleteCharacter={deleteCharacterHandler}

        ></CharacterCard>

      );

    });

Once CardList components receive data, it maps it and creates an array of cards that we see on the home page. CharacterCard receives a bunch of props, among them the data for a specific character, and renders that data.

## Search

App.jsx

const [filter, setFilter] = useState({

    name: "",

    gender: "",

  });

CardList.jsx

const cards = characterData

    .filter(

      (character) =>

        character.name

          .toLowerCase()

          .startsWith(props.filter.name.toLowerCase()) &&

        character.gender

          .toLowerCase()

          .startsWith(props.filter.gender.toLowerCase())

    )

    .map((character, i) => {

      return (

        <CharacterCard

          key={i}

          fromApi={props.fromApi}

          characterData={character}

          loadCharacter={loadCharacterHandler}

          selectFilms={loadFilms}

          deleteCharacter={deleteCharacterHandler}

        ></CharacterCard>

      );

    });

The App.jsx file keeps a filter state that is passed down to a Header component so it can fill it through input fields, and the Pagination component to render a list. Through props, the filter is passed down to the CardList component that filters the list before it maps it and creates an output array.

## Creating a character

App.js

  const [showAddCharacterModal, setShowAddCharacterModal] = useState(false);

const openAddCharacterModalHandler = () => {

    setShowAddCharacterModal(true);

  };

  const closeAddCharacterModalHandler = () => {

    setShowAddCharacterModal(false);

  };

As stated in visual representation, by clicking on an ‘Add Character’ button, a user open a modal with input fields that need to be filled. The opening and closing modal are managed by App.js and its showAddCharacterModal state. When it’s opened, a NewCharacter component is rendered.

NewCharacter.jsx

<Formik

        initialValues={{

          name: "",

          height: "",

          mass: "",

          hair\_color: "",

          skin\_color: "",

          eye\_color: "",

          birth\_year: "",

          gender: "",

        }}

        validationSchema={ValidationSchema}

        onSubmit={(values) => addCharacterHandler(values)}

      >

Using a Formik library, a NewCharacter component creates a form with initial values, validation schema, and action that is triggered on a submit.

const DisplayingErrorMessagesSchema = Yup.object().shape({

  name: Yup.string()

    .min(2, "Too Short!")

    .max(50, "Too Long!")

    .required("Required"),

  height: Yup.number("Need a number")

    .typeError("Must be a number")

    .required("Required")

    .positive("Must be positive")

    .integer("Must be integer"),

  mass: Yup.number()

    .typeError("Must be a number")

    .required("Required")

    .positive("Must be positive")

    .integer("Must be integer"),

  hair\_color: Yup.string().required("Required"),

  skin\_color: Yup.string().required("Required"),

  eye\_color: Yup.string().required("Required"),

  birth\_year: Yup.string().min(4, "Too Short!").required("Required"),

  gender: Yup.string().required("Required"),

});

A validation schema is created using the Yup library that creates an object with basic instructions on what data in fields (name, height, etc.) should look like.

  const addCharacterHandler = (value) => {

    dispatch(swDataActions.addCustomCharacter({ fromDB: false, ...value }));

    props.onClose();

  };

On submit, the action is dispatched to store this new character in redux-store. Alongside value, there is an attached fromDB attribute that is used to not render the ‘Planet’ button in the character page as it is part of the feature. After dispatching, the onClose function is called to close the modal.

## Deleting a character

CharacterCard.jsx

 <Button

            onClick={() =>

              props.deleteCharacter(

                props.fromApi

                  ? props.characterData.name

                  : props.characterData.id,

                props.fromApi

              )

            }

          >

            {t("characterCard.delete")}

 </Button>

The button in CharacterCard when clicked calls the deleteCharacter function and sends a payload containing a character name or id, depending on if data came from a server, to find a single character, and fromApi so the redux-store knows which list should iterate.

CardList.jsx

const deleteCharacterHandler = (characterID, fromDB) => {

    dispatch(

      swDataActions.deleteCharacter({

        characterID,

        pageNumberApi: props.pageNumberApi,

        fromDB,

      })

    );

  };

The deleteCharacter function is passed from CardList down to CharacterCard and its purpose is to dispatch a delete action with the proper payload.

## Films modal

CardList.jsx

  const loadFilms = (characterID) => {

    const character = characterData.filter((c) => c.id === characterID);

    dispatch({

      type: "FETCH\_FILMS",

      payload: character[0].films,

    });

    props.openModal();

  };

By clicking on a “Films” button, a user calls function an action that filters through data and finds the right character, then action is dispatched with URLs as payload for films that need to be loaded. As final instruction, it calls a function that opens a modal.

Films.jsx

  const filmsData = useSelector((state) => state.filmsData.filmsArray);

{filmsData.map((film, i) => {

          return (

            <p key={i} className="mb-7 font-serif">

              <span className="text-gray-600 font-semibold italic mr-3">

                {i + 1}:{" "}

              </span>

              "{film.title}" by {film.producer} (

              {new Date(film.release\_date).toLocaleDateString()})

            </p>

          );

        })}

The Films component is rendered inside the opened modal, it takes data from redux-store and iterates through it. As a result, it renders information for each movie.

## Character details page

CardList.jsx

const loadCharacterHandler = (characterData) => {

    dispatch(

      characterActions.addCharacter({ characterData, fromApi: props.fromApi })

    );

    navigate("/character");

  };

When a user clicks on a character card on the home page, the loadCharacterHandler function is called and action is dispatched that loads character data the in redux store. The user is then navigated to a character page.

CharacterInfo.jsx

  const characterData = useSelector((state) => state.character.characterData);

      <CharacterInfoDisplay

        characterData={characterData}

      ></CharacterInfoDisplay>

When a user arrives on the character page, the CharacterInfo component takes character data from redux-store and renders relevant information.

const fromApi = useSelector((state) => state.character.fromApi);

const characterHomeworld = useSelector((state) => state.character.homeworld);

 {fromApi ? (

        characterHomeworld ? (

          <PlanetInfo planet={characterHomeworld} />

        ) : (

          <div className="self-center  mt-10">

            {spinner ? (

              spinner

            ) : (

              <button

                className="animate-bounce text-black font-semibold px-1 py-1 rounded-lg w-28 bg-orange-300 m-auto"

                onClick={loadPlanetHandler}

              >

                {t("characterInfo.planet")}

              </button>

            )}

          </div>

        )

      ) : null}

const loadPlanetHandler = () => {

    dispatch({ type: "FETCH\_PLANET", payload: characterData.homeworld });

  };

The CharacterInfo components also render a button, if a character is fetched from a server, that dispatch an action to fetch a character’s home planet from a server. If a planet is successfully fetched, the components render some basic information about it.

## Localization

config.js

import i18n from "i18next";

import { initReactI18next } from "react-i18next";

i18n.use(initReactI18next).init({

  fallbackLng: "en",

  lng: "en",

  resources: {

    en: {

      translations: require("./locales/en/translations.json"),

    },

    fr: {

      translations: require("./locales/fr/translations.json"),

    },

  },

  ns: ["translations"],

  defaultNS: "translations",

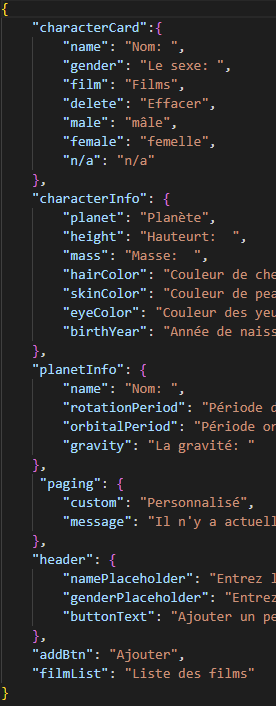
});

i18n.languages = ["en", "fr"];

export default i18n;

Localization is done using an i18next library. The use(initReactI18next) will bind react-i18next to the i18n instance. The first two properties of the init object parameter are a fallback language (fallbackLng) and default language (lng).

The resources are the JSON files with the translations that are created in a separate folder. The namespaces (ns) and default namespace (defaultNS) are the keys from the resources object.

translation.json

These JSON files hold hard-coded translation text for specific parts of a website

const { t, i18n } = useTranslation();

text1={t("characterInfo.height")}

To implement localization all that needs to be done is to use the useTranslation hook and pull the text based on a key that’s provided.

 const changeLanguage = (e) => {

    i18n.changeLanguage(e.target.value);

  };

Switching between languages is done through a function that i18next provides, we just need to pass the value corresponding to a JSON file in config.js.

## Redux-Store

index.js

const store = configureStore({

  reducer: {

    swData: swDataSlice.reducer,

    filmsData: filmsSlice.reducer,

    character: characterSlice.reducer,

  },

  middleware: [...getDefaultMiddleware(), sagaMiddleware],

});

The project store consists of 3 slices, swDataSlice for lists of characters, characterSlice for a selected character upon clicking on a card, and filmSlice for characters’ films.

swData-slice.js

const swDataSlice = createSlice({

  name: "swData",

  initialState: {

    count: 1,

    results: [],

    customCharacters: [],

    isLoading: false,

    errorOccurred: false,

  },

  reducers: {

    addNewData(state, action) {

      state.results = [...state.results, action.payload];

      state.isLoading = false;

      state.errorOccurred = false;

    },

    deleteCharacter(state, action) {

      if (action.payload.fromDB)

        state.results[action.payload.pageNumberApi].results = state.results[

          action.payload.pageNumberApi

        ].results.filter((entry) => entry.name !== action.payload.characterID);

      else

        state.customCharacters = state.customCharacters.filter(

          (entry) => entry.id !== action.payload.characterID

        );

    },

    addCustomCharacter(state, action) {

      state.customCharacters = [

        ...state.customCharacters,

        { id: state.count, ...action.payload },

      ];

      state.count += 1;

    },

    isLoading(state) {

      state.isLoading = true;

    },

    setError(state) {

      state.errorOccurred = true;

      state.isLoading = false;

    },

  },

});

The initial state of swDataSlice consists of a counter as a generator for data ids, a results array for characters data coming from the server, customCharacter array for custom characters data, and isLoading and errorOccurred flags.

The reducer of this slice holds the addNewData function that as the name implies, adds data coming from a server to the results array. The addCustomCharacter function for custom data, and the deleteCharacter function for deleting a character for the designated array.

The isLoading and setError function, like in every slice, sets flags based on a flow of actions.

character-slice.js

const characterSlice = createSlice({

  name: "character",

  initialState: {

    characterData: null,

    fromApi: false,

    homeworld: null,

    isLoading: false,

    errorOccurred: false,

  },

  reducers: {

    addCharacter(state, action) {

      state.characterData = action.payload.characterData;

      state.fromApi = action.payload.fromApi;

      state.homeworld = null;

    },

    addHomeworld(state, action) {

      state.homeworld = action.payload;

      state.isLoading = false;

      state.errorOccurred = false;

    },

    isLoading(state) {

      state.isLoading = true;

    },

    setError(state) {

      state.errorOccurred = true;

      state.isLoading = false;

    },

  },

});

The initial state of characterSlice consists of a characterData object for a single character data that is passed on a card click, fromApi flag if data came from a server or not, homeworld for data of characters home planet, and isLoading and errorOccurred flags.

The reducer of this slice holds the addCharacter function that as the name implies, adds data for a selected character, and the addHomeworld self-explanatory function.

The isLoading and setError function, like in every slice, sets flags based on a flow of actions.

filmsSlice.js

const filmsSlice = createSlice({

  name: "films",

  initialState: {

    filmsArray: [],

    isLoading: false,

    errorOccurred: false,

  },

  reducers: {

    addFilms(state, action) {

      state.filmsArray = action.payload;

      state.isLoading = false;

      state.errorOccurred = false;

    },

    isLoading(state) {

      state.isLoading = true;

      state.filmsArray = [];

      state.openFilmModule = true;

    },

    setError(state) {

      state.errorOccurred = true;

      state.isLoading = false;

    },

  },

});

export const filmsActions = filmsSlice.actions;

export default filmsSlice;

The approach is the same as other slices.

## Sagas

root-saga.js

export default function\* rootSaga() {

  yield spawn(watchFetchSWDataSaga);

  yield spawn(watchFetchFilmsDataSaga);

  yield spawn(watchFetchPlanetSaga);

}

Behind the scene, three worker sagas are simultaneously running and watching for asynchronous data fetching calls.

sw-saga.js

export function\* fetchSWDataSaga(url) {

  try {

    yield put(swDataActions.isLoading());

    const response = yield call(axios.get, url);

    yield put(swDataActions.addNewData(response.data));

  } catch (error) {

    yield put(swDataActions.setError());

  }

}

export function\* watchFetchSWDataSaga() {

  while (true) {

    const req = yield take("FETCH\_SWDATA");

    yield fork(fetchSWDataSaga, req.payload);

  }

}

The watchFetchSwDataSaga generator function is paused until "FETCH\_SWDATA" is dispatched and then it calls the fetchSWDataSaga generator function. The calls fetchSWDataSaga set isLoading flag, calls axios.get to fetch data, and then dispatches an action to add newly fetched data to redux-store.

planet-saga.js

export function\* fetchPlanetSaga(planetURL) {

  try {

    yield put(characterActions.isLoading());

    const response = yield call(axios.get, planetURL);

    yield put(characterActions.addHomeworld(response.data));

  } catch (error) {

    yield put(characterActions.setError());

  }

}

export function\* watchFetchPlanetSaga() {

  while (true) {

    const req = yield take("FETCH\_PLANET");

    yield fork(fetchPlanetSaga, req.payload);

  }

}

The approach is the same as sw-saga.js.

films-saga.js

export function\* fetchFromSWAPI(url) {

  try {

    const response = yield call(axios.get, url);

    return response;

  } catch (error) {

    yield put(filmsActions.setError());

  }

}

export function\* fetchFilmsDataSaga(filmsURL) {

  yield put(filmsActions.isLoading());

  const response = yield all(filmsURL.map((url) => call(fetchFromSWAPI, url)));

  const films = response.map((film) => {

    return film.data;

  });

  yield put(filmsActions.addFilms(films));

}

export function\* watchFetchFilmsDataSaga() {

  while (true) {

    const req = yield take("FETCH\_FILMS");

    const task = yield fork(fetchFilmsDataSaga, req.payload);

    yield take("STOP\_FETCHING\_FILMS");

    cancel(task);

  }

}

The only difference is the additional generator function (fetchFilmsDataSaga) that accepts an array of URLs and calls the “all” effect that waits for all Promises coming from the fetchFromSWAPI generator function to be resolved.

## React Query

The project is also refactored to use React Query for caching server-side data as a replacement for the saga-store duo.

use-intinite-people.js

const fetchUrl = async (url) => {

  const response = await axios(url);

  return response.data;

};

const useInfinitePeople = () => {

  const {

    data, fetchNextPage, hasNextPage, fetchPreviousPage, hasPreviousPage,

    isLoading, isFetching, isError, error, isFetchingNextPage,

  } = useInfiniteQuery(

    "sw-people",

    ({ pageParam = initialUrl }) => fetchUrl(pageParam),

    {

      getNextPageParam: (lastPage) => lastPage.next || undefined,

      getPreviousPageParam: (lastPage) => lastPage.previous || undefined,

    }

  );

  return {

    data, fetchNextPage, hasNextPage, fetchPreviousPage, hasPreviousPage,

    isLoading, isFetching, isError, error, isFetchingNextPage };

};

This hook creates an infinite query for characters coming from a server. With the useInfiniteQuery function provided by React Query, on initial call or by paging, it calls the fetchUrl function to get data and concat it to existing data.

use-films.js

const fetchUrl = async (url) => {

  const response = await axios(url);

  return response.data;

};

const fetchUrls = async (urls) => {

  const data = await Promise.all(

    urls.map((url) => {

      const newData = fetchUrl(url);

      return newData;

    })

  );

  return data;

};

const useFilms = () => {

  const character = useSelector((state) => state.filmsData.character);

  const { data, isLoading, isFetching, isError, error, isPreviousData } =

    useQuery(["films", character.name], () => fetchUrls(character.urls));

  return { data, isLoading, isFetching, isError, error, isPreviousData };

};

The useFilms hook first collects character data from the store, then using the useQuery function calls the fetchUrls function. The fetchUrls await all Promises coming from individual calls of fetchUrl, and when they are resolved it returns data through the useQuery function.

use-planet.js

const fetchUrl = async (url) => {

  const response = await axios(url);

  return response.data;

};

const usePlanet = (name, url) => {

  const { data, isLoading, isFetching, isError, error, isPreviousData } =

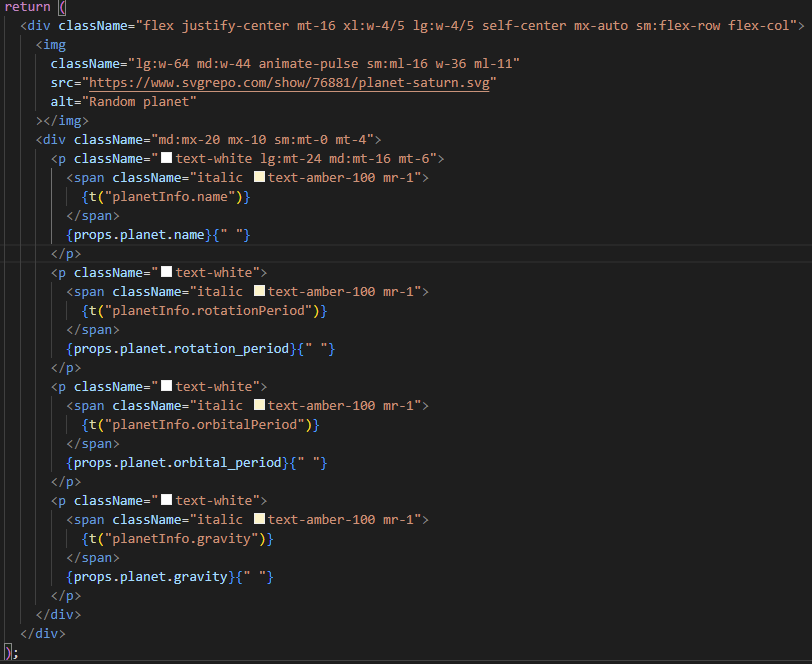
    useQuery(["planet", name], () => fetchUrl(url));

  return { data, isLoading, isFetching, isError, error, isPreviousData };

};

The usePlanet hooks work just like the useFilm hook.

## Styling



Styling is done using Tailwind CSS framework packed with classes like flex, pt-4, text-center, and rotate-90 that can be composed to build any design, directly in markup.

The React Query version of the project is also styled with styled-components.

export const Button = styled.button`

  --tw-bg-opacity: 1;

  background-color: rgb(253 186 116 / var(--tw-bg-opacity));

  width: 2.5rem;

  border-radius: 0.375rem;

  margin-left: 2.5rem;

  margin-right: 2.5rem;

  height: 2.5rem;

`;

export const Container = styled.div`

  align-self: ${(props) => (props.align ? "center" : "auto")};

  margin-bottom: ${(props) => (props.mb ? "2.5rem" : "0")};

  margin-top: ${(props) => (props.mt ? "1rem" : "0")};

`;