Movie Library Project

Scope

The “Movie Library” application is fully functional with enough content provided in order to meaningfully interact with it and understand the idea of it. Its main purpose is to be used as an assessment and to demonstrate some main concepts. Further possible improvements are described down below in “Improvements” section.

Technologies used

Front-end

On the front-end side, this application relies mainly on React. On top of it, Redux is used via react-redux and its toolkit. It also uses some generic components from react-bootstrap. Naturally for any front-end development, TypeScript, HTML and CSS are involved too.

Back-end

On the backend-side, the implementation is quite simple, mostly because of time constrains – It relies on NodeJS with an express middleware to handle http calls.

Decision factors

This is mainly driven by the nature of the project. Although code is in early state, it significantly considers high level of scalability and maintainability for both components side and UI visualization side.

Components considerations

Components are divided into following categories:

Stateful

- These components are aware of the application state.
- They usually trigger state changes, “listen” to such, gather information and pass it down to other (mostly stateless) components.
- They should not present any data in the UI.
- Their main purpose is to control the whole flow.
- You can find these under “src\components\stateful” folder

Stateless

- These components only rely on their input properties. An eventual output if needed, is simply forwarded to their consumer. (Small exception could be considered here – for example a “theme” that could be stored in a context, i.e. createContex/useContex)
- They don't have any knowledge about the state. They should not deal with it in any way. (This obviously exclude dealing with their own local non-shared state, i.e. using useState)
- They don’t have any knowledge about the origin of data passed to them, for example where that data comes, how is fetched and/or transformed, where it is stored, etc.

- They must not have any references to a stateful component – if such is needed, it must be extracted in a common location and ideally abstracted too.
- They can have references only to other stateless components or generic components (for the latest, see the notes under "Generic" section)
- You can find these under "src\components\stateless" folder

Generic

This could be considered as a category under stateless components. Generic components are same as stateless with some further restrictions. These restrictions make them extremely reusable and portable.

- They accept only standard props, i.e. primitives and objects. This automatically implies they don't rely on specific interfaces declared and exported somewhere.
- They are very commonly used as wrappers around 3rd implementation. This allows abstraction which ensures easy substitution of the underlying implementation.
- They are usually placed in a common, highly accessible location. For example, high level folder inside the application suit's repo or implemented as node modules

UI considerations

Mobile users

The app is designed with "mobile-first" in mind. The entire layout adjusts properly according to screen size. This drastically lowers the need to create, support and maintain separate code for mobile users. Obviously, such approach is cost effective too.

Buttons order

This is a small detail, yet it shows the good number of thoughts invested into the UI/UX design. The buttons order inside a movie item in the list is far from random – it follows the natural sequence of human actions, like: Check the movie details first, then add it to watch list, mark it as favorite once you watch it and if you like it, then share it, if you want to suggest it to others too.

Non-blocking loading approach

Lots of UIs still block the whole page while something is loading. In most of the cases, this is not needed and must be avoided. For example, when we load the movie details, the popup shows only when loading is completed. Until then, the loading indicator is showed on the top header, which doesn't bother the user, but it informs him. In the meantime, user can still do everything else – UI is fully functional and accessible.

Improvements

Optimizations:

- memorization for complex selectors and data transformers
- state caching (for example already loaded movie details)

Adding persistent layer

Obviously for such type of application, some persistency is needed. No SQL DB might be a better choice, because:

- Schema extensions/modifications might occur (for example adding more details to a movie)
- Search will be kept relatively simple, with no need of complex select queries, views, joins or procedures, which are all typical for traditional RDBMs

CSS preprocessing

For a big project, this is a must. Some sort of CSS pre-processor must be used (i.e. Less/SASS). This ensures following:

- Mixins and tokens (variable definitions) makes the code very easy to change and maintain
- UI/UX team have an excellent visibility and enforcement of the product's visual consistency and branding

A11y

Accessibility must be considered for impaired users. This is also important for regular users, giving them confidence and options to navigate across the app (i.e. "keyboard" users)

I18n/L11n

These two (internationalization and localization) make the product highly re-usable across geographies. This should be seriously considered especially when this is intended to be used in Nord America and also Europe. Not only different language support will be required, but different countries use different formats for many things, like date/time/currency/etc.

Pendo

Consider using "Pendo" or similar approach to "record" and examinate users' actions – how they use the app in general, what do they care most about, etc.

Unit tests & Automation tests

Both of these must exist in the code base. "automation-id" needs to be added for html elements inside relevant components (meaning, the stateless components)

UI improvements

In general, the UI should feel natural and easy to navigate with. The more documentation you need to write on how to use the product, the more improvements your UI needs, most likely. For this particular app, first particular thing that comes into mind would be to replace the action buttons for the movie with something more condensed and easier to grasp – icons for example. Obviously, the whole appearance could be involved too, for instance, by adding more "vivid" content to it, like backgrounds with movies' images or playing trailers.

How to install

Follow below steps to have the application locally installed, up and running. This assumes you have NodeJS already installed on your system. If not, visit https://nodejs.org/en/download and follow the instructions there.

- Download both "VideoPortal" and "VideoPortalBE" folders.
- Open command prompt.
- Go inside "VideoPortalBE" folder and run "npm install". After it finishes, run "node index.js". If that fails, make sure the default port used is not already taken by another app.
- Leave the command prompt running and open new command prompt.
- Go inside "VideoPortal" folder and run "npm install". After it finishes, run "npm run dev".
- Open your browser and hit http://localhost:5173/

Disclaimer

This document along with the “Movie Library” project is intendent to be used as an assessment and only by the “Vention” company - https://vention.io/ .