Frontend Architecture

The project is structured like below. Similar files are grouped together:

```
assets/
  css/
  ...photos
components/
  AddressForms/
    AddAddressForm.js
    EditAddressForm.js
    styles.js ---> CSS-IN-JS
  ...
consts/
  OrderStatus.js
  ProductStatus.js
  UserRole.js
  UserType.js
  ...
navigation/
  AuthRole.js
hooks/
  AddressHooks.js
  AuthHooks.js
  ProductyHooks.js
  ...
pages/
  HomePage/
    HomePage.js
    styles.css ---> CSS-IN-JS
  LoginPage/
    LoginPage.js
    styles.css
  ...
stores/ ---> MobX stores
  AddressStore.js
  CartStore.js
  ...
utils/
  ContractExporter.js
  ReceiptExporter.js
  ...
index.js
```

1. Atomic Design:

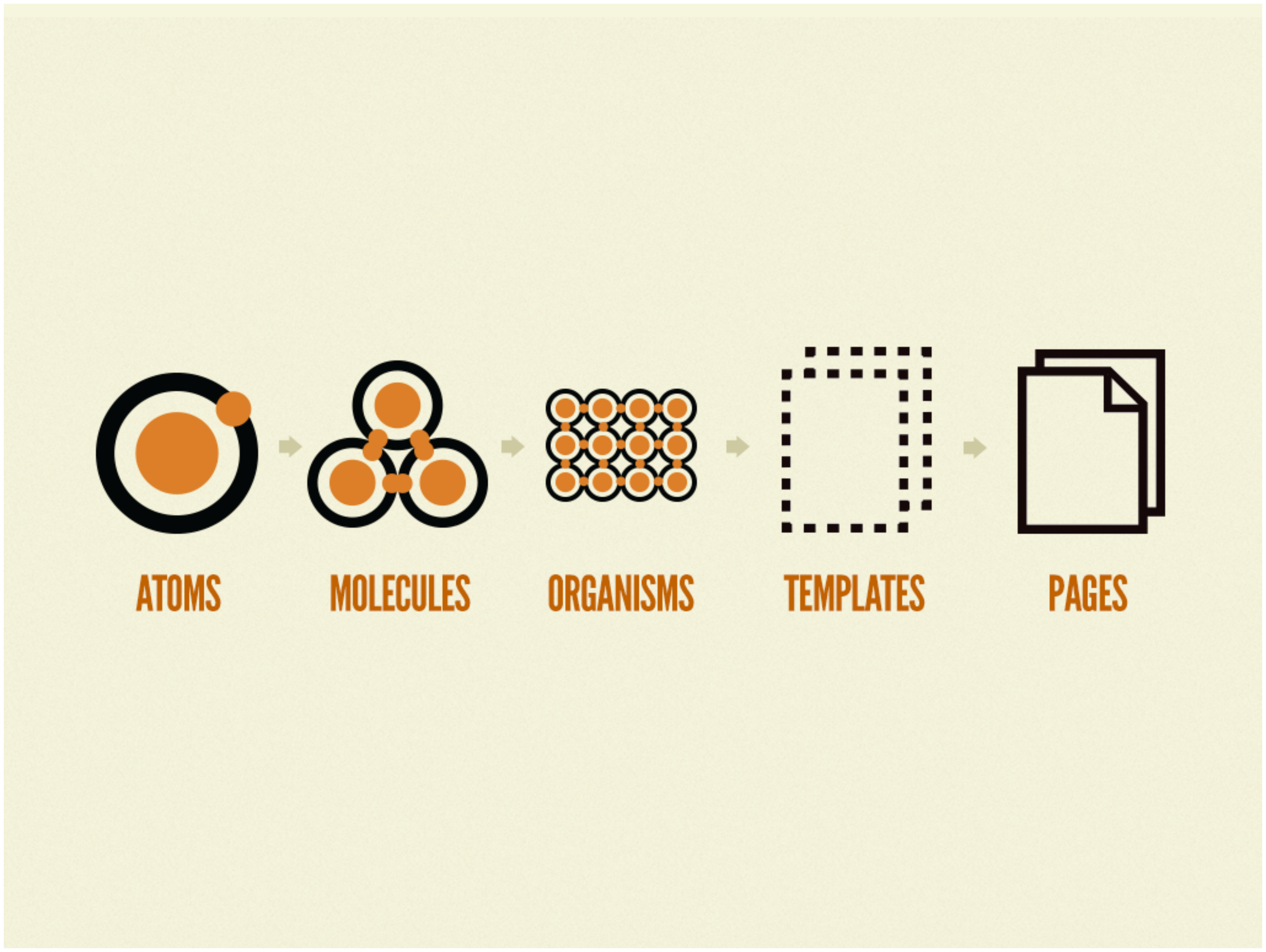
All the UI elements are classified into three main folders: layouts, pages and components. This structure is based on a design methology named Atomic Design.

In Atomic Design, atoms are combined into **molecules** and then **molecules** form **organisms**. In our project, in order to keep it simple, modules and organisms are part of **components**. Every single component includes a .tsx file to define how it renders and a .ts file to contain its CSS.

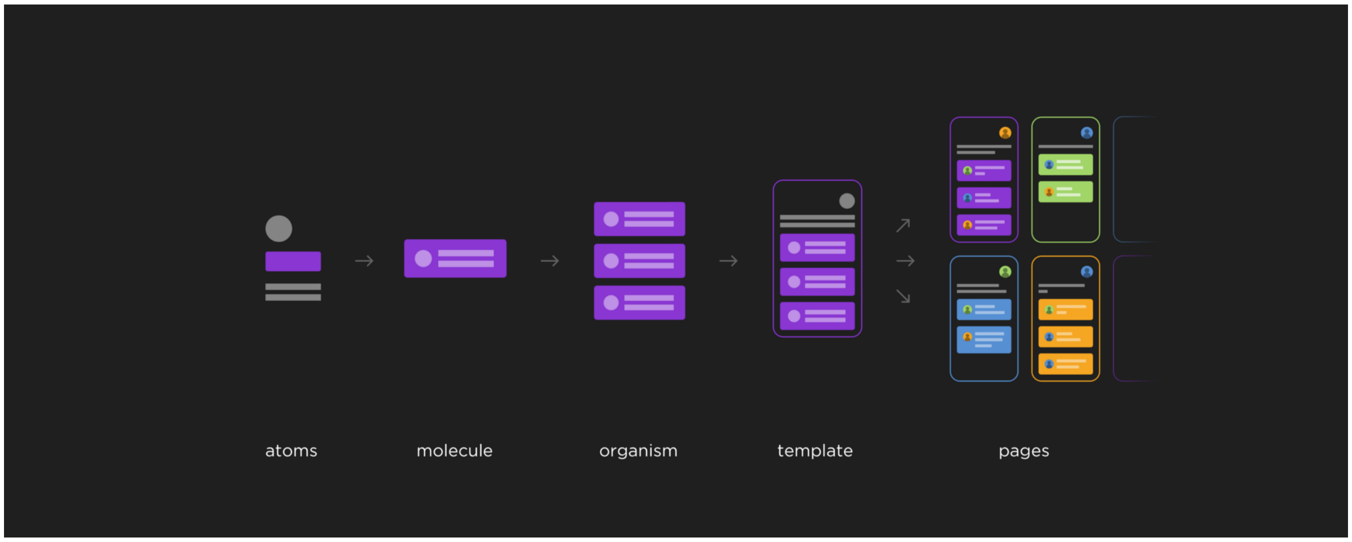
In Atomic Design ebook, Brad Frost states that templates are very concrete and provide context to all these relatively abstract **molecules** and **organisms** (**components** in our project). The frontend project has a similar folder containing all the templates called **layouts**.

Finally, **pages** are specific instances of templates. Like other UI elements, every single page has a resepective CSS file.

In gerenal, the UI elements of the project must be organised based on Atomic Design. Depending on the expansion of the project, **components** may be classified into **molecules** and **organisms** to make the code cleaner.



Atomic Design (Brad Frost, 2013)



An example about Atomic Design

2. React Hooks:

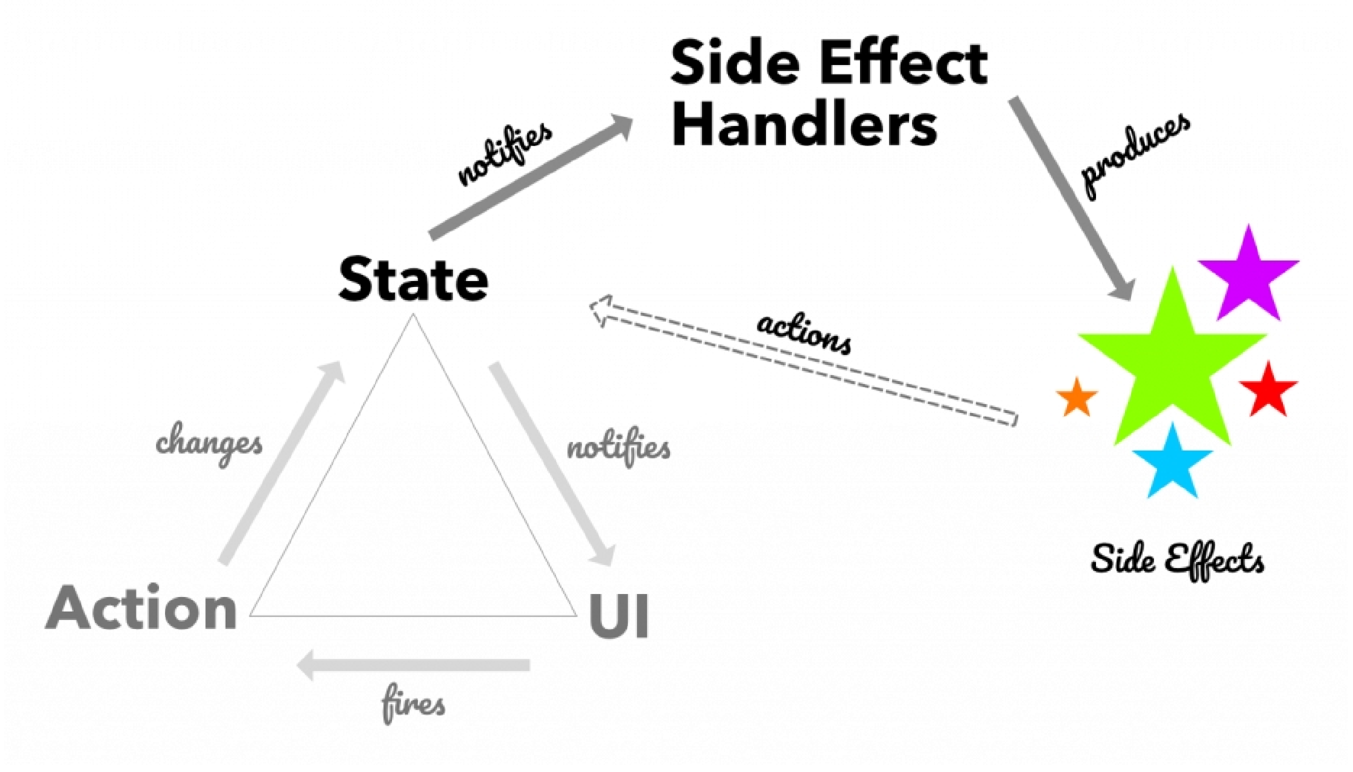
The **hooks** folder contains all the component logic of the application. Extracting component logic into custom hooks (reusuable functions) makes the code cleaner and more managable. In addition, this also makes us easier to test the logic of the application in the future.

You can read about hooks at this link: https://reactjs.org/docs/hooks-intro.html

3. Stores:

The **stores** folder contains observable stores for the React application. The core state management library behind this is MobX. So what is MobX?

“The state is at the epicenter of all things happening in the UI. MobX provides a core building block, called observable, that represents the reactive state of the application” (MobX Guide, Podila and Weststrate, 2018). Mobx has some some conceptual similarities with Redux (another state management library supporting the Flux architecture proposed by Facebook) as far as the data flow is concerned, but that is also where the similarities end.



The side effect model (Podila et al., 2018)

Redux relies on **immutable** state snapshots and reference-comparisions between two state snapshots to check for changes. In contrast, MobX thrives on **mutable** state and uses a granular notification system to track state changes (Podila et al., 2018).

In Redux, we have to do some following things:

- Define `initialState`.
- Identify actions that can be performed to change the state tree. Each action must strictly follow the format `{ type: string, payload: any }`
- Define raw actions to dispatch events.
- Use the `connect` method to wire the React component with the store.
- Side effects must be performed in middleware.
- Must always return a new instance of the state inside every reducer.

In their book, Podila et al. (2018) points out one disadvantage of Redux that is “as the complexity of the feature increases, there is more fragmentation between actions, action-creators, middlewares, reducers and initialState. Not having things co-located also increases the effort needed to develop a crisp mental model of how a feature is put together”.

In MobX world, the developer experience is totally different:

- Define the observable state for the feature in a store class.
- Define actions that will needed to mutate the observable state.
- Side effects (`autorun`, `reaction` and `when`) are defined within the same feature class.
- Use the `observer` API provided by the `mobx-react` package to connect the React components with the observable store.

In general, MobX enables a **more declarive** form of Redux. The workflow associated with Redux is simplified considerably when using MobX. In addition, the advent of a hook named `useReducer` makes Redux no longer impressive for me. In the author’s opinion, combining useReducer with MobX may be considered as the state of the art in 2021.

You can read more about MobX at this link: https://mobx.js.org/README.html

4. Assets:

The **assets** folder just contains CSS, images and fonts used for the project.