Reasons

1. React tends to bee too unestructured what could become in a mess in terms that there is no order in the folders or the components.
2. Share data between components that are not directly related (parent-child) is a problem because there is no direct communication and you start drilling down with props or involve not related component (pass data through the parents).
3. When components are dependant of its environment (are highly coupled whit its subtree or lift props to its parents) they are less reusable.

Guidelines

Folder Structure

Features - Pages

The project should be orginized following the pattern

```
src/
    lib/
    core/
    features/
    pages/
```

lib

Here are the external dependencies (for example axios). The idea is to wrap the dependencies with a *proxy pattern* and in the project import from `lib` instead of directly. In this way updating the dependencies or replacing them could has a lower impact on the app.

core

```
    core/
        alert-card/
        send-button/
        utils/
```

Here should be the components that are used by the hole app. `utils` functionality could also live here.

features

```
    features/
        feature-a/
            __tests__/
```

```
            state/
            services/
            shared/
            ui/
            index.js
            package.json  // optional for the packages approach
        feature-b/
```

The idea of this pattern is encapsulate all the code related with a main feature. Every external use of a feature should be exported in the `index.js` file, in this way we can defina a public api for a feature.

The `state` folder holds all the state managemen of the feature (there could be the contexts of the feature).

The `ui` holds the all components hierarchy that are directly conected but not the shared components. The components that are shared between diferent components should be placed in the `shared` component.

pages

```
    pages/
        Login.js
        DataFlow.js
        ...
```

Here are the pages of the app the ones that gather a set of feautres. The pages components should be simple components since all the logic should live in the features.

Package main subfolders

This should be discussed.

Package the main subfolder could have some benefits.

- Force the users of a package to just import the components defined in the public API paths.
- Importing from outside is more simple since the you can use just the name of the package instead of the relative path to the folder.
- You can move the package everywhere witout changing any import in the project.

Container - Presentation

```
component-a/
    index.js
    container.js
    presentation.js
    component.test.js
```

The components will be conformed by two files:

1. Container: holds all the local states and the logic of the component.
2. Presentation: holds all `jsx` of the component. This should be stateless and receive all be `props`.
3.

Example:

```
const UsePresentation = ({name, email}) => {
    return (
        <div>
            <h1>Hi {name}!</h1>
            <small>{email}</small>
        </div>
    )
}

const UserContainer = () => {
    const [user, setUser] = useState(null);
    useEffect(() => {
        fetchUser().then(setUser);
    }, []);

    if (!user) return <h2>Loading ...</h2>
    return <UserPresentation name={user.username} email={user.email}/>
}
```

In this way unit test over the Presentation part are simpler because you could test ir isolated with an specific conbination of state (props) without mocking external services.

Also the code becomes more clean since the visual part is separeted from the logic.

Communication

1. Passing data from a parent to a child should be done by props.
2. Passing data from the child to its parent should be done through a callback prop.
3. Passing data between siblings or not directly related components should be done through a state management system.
4. Avoid the use of "lifting state".

Services

1. Place them in the context provider. In this way you could have them centralized and them could be mocked or injected easily.
2. The dependency injection of the service could be a good pattern to help testability and the decoupling