[ECP&SG]FE Coding Style Guidelines

Version	Date	People Involved	Description
v1.0	2022-2-17	Dong Xiaocong	Most of them migrated from Chinese version
v1.1	2022-2-22	Dong Xiaocong	Helper functions must be pure, accumulated from comment

Background

The developers who contribute to the same project may come from different backgrounds. They may apply different coding styles to their daily coding which will result in an inconsistent coding style across the project.Therefore, we need to have a unified coding guidelines in our team which is used to **enforce consistent coding style throughout the project**.

Notes: This is a living document, which means it will be updated and edited according to the feedbacks from daily code review activity and regular code review case study.

How to Apply

Most of the guidelines can be enforced by our customized eslint config, while others must be enforced by the code reviewers.

Guidelines

Folder Structure

Here is the example folder structure :

```
project-root/
 src/
  api/
  assets/
  common-styles/
  components/
       App/
       Common/
  constants/
  helpers/
  hocs/
  hooks/
  states/
  App.tsx
  global.css
  index.html
  index.tsx
 typings/
 deploy/
 .babelrc
 .eslintrc.json
 gitlab-ci.yml
 .prettierignore
 commitlint.config.js
 tsconfig.json
 webpack.config.babel.js
```

Each folder/file in the above structure stands for:

- src: source code of the project, all of the non-config code should be placed under this folder
    - api: code used to interact with the backend apis must be placed in this folder
    - assets: if you have any static files in your project, they should be placed in this folder
    - common-styles: since we use styled-component in our project, we will place all of the common styled Components under this folder
    - components: both of the common and business(App) Components live in this folder. Under the App component folder, components will be **`organized according to the path information`**. For example, if you have a path like `/projects/cmdb/application/console` in your application, you will have the following structure under App:

      ```
      components
       App/
            Project/
                  Application/
       Common/
      ```

      We organize components in this way is for helping **`developers locate Component with less effort`**. For example, when user report a bug to the developer, s/he can find the corresponding buggy Component via the path information quickly.
    - constants: a folder holding constants
    - helpers: global helper functions
    - hocs: if you have any global **`high-order components`**, they should be placed here
    - hooks: home for global hooks
    - states: since we use recoil in our project, we will place all of the global states here
    - App.tsx: main application
    - index.tsx: render main application to the html DOM, you may also register your sentry client here
    - index.html: main entry of the application
    - global.css: global css styles
- typings: global typings
- deploy: files related to deployment (Dockerfile, deploy.json, etc).
- .babelrc: babel config
- .eslintrc.json: eslint config
- gitlab-ci.yml: gitlab ci config
- .prettierignore: prettier ignore config
- tsconfig.json: typescript config
- webpack.config.babel.js: webpack config

Naming Convention

A good variable name can help other developers (including yourself in the future) better understand the logic of your code. We have the following rules on naming variables:

Format

Variables except constants must be in **`camelCase`**. For constants, they need to be written in **`uppercase and separated by underscores`**. For example, `THIS_IS_A_CONSTANT` is a constant name while `thisIsAGeneralVariable` is general variable name.

Avoid Abbreviation

Some developers like to use abbreviation as variable names in their code. This will cause the following problem:

- Different developers write different abbreviations for the same word. For example, some people may use `fil` for a file while others use `f` .
- Same developer use different abbreviations for the same word in different places.
- Many developers use abbreviation for time saving, however an ambiguous abbreviation will increase the time other developers spend on understanding it.

To solve the above problem**`, you should avoid abbreviation in your code which means always using the full English words in your variable names no matter how long it is`**. However **`if the abbreviation of a variable is widely used among the developers`**, we can use its abbreviation. For example, `id` is better than `identifier` since all of the developer can easily understand what it means.

Here is the list of allowed abbreviations in our code :

Abbreviation	Full Name
id	identifier
i	index
char	character
cmd	command
cur	current
db	database
diff	difference

Indicate Type in the Name

A good variable name should try to indicate its **`data type`**. For example, here is a list of recommended and non recommended names for an array of `service`:

- service (need to in the plural form to indicates it is an array)
- services
- serviceArray
- serviceList

Avoid Negative Names for Boolean Variables

Names of the boolean variables **`must take the affirmative form`** and **`be grammatically correct`**. For example, here is a list of recommended and non recommended names for a variable used to indicate if a button is disabled:

- notEnabled (should take the affirmative form)
- beDisabled (not grammatically correct)
- isDisable (not grammatically correct)
- isDisabled

Function Name must Start with a VERB

Usually a function is used to perform some kind of action on an object, so we have the following rules on naming functions:

- must start with a verb
- must be **`grammatically correct`**
- must be **`semantically correct`**
- must be **`clear`**

For example, here is a list of recommended and non recommended names for naming a function used to transform data into an array:

- transformation (can not be a noun)
- toObject (should start with a verb)
- transformArray (not semantically correct)
- transformToObject (not clear, should be `array` instead of `object`)
- objectify (not clear)
- transformToArray

Components

Naming

We have the following rules on naming a Component:

- Name of the class and React component must be **`Capitalized`**, like `Folder`, `Watcher`, etc.
- File name of the React component must be Capitalized as well
- For the component used to display information of a resource, they should be named as `resource type` + `component type` . Resource type refers to the type of the resource, like `Service` , `Application` and `Deployment` . Component type refer to the UI type of a component, like `Table` , `List` , `Editor` , `Drawer` , etc.
- For the component used to edit resource, they should be named as `verb` + `resource type` + `component type(optional)` . I have covered resource type and component type on the above rule. A full list of allowed `verb` is:
    - Create: create a resource
    - Update: update a resource
    - Delete: delete a resource

  For example, a Form used to create an application should be named as `CreateApplicationForm`

Props

For the name of a customized event, its props name must start with on . Here is a list of recommended and non recommended names:

- submit (should start with on)
- error (should start with on)
- succeed (should start with on)
- onSubmit
- onError
- onSucceed

As for the event handlers passed to the customized event props, their names must start with `handle` . Here is a list of recommended and non recommended names for event handlers:

- onSubmit (should start with handle)
- handleSubmit
- handleError
- handleSucceed

`Pro tips:` when you are naming a variable, you may ask yourself if the name can be understood by other developers, if not, you had better change it.

Data Management

We have the following rules on how to store the data in your Component:

- If the data is only used by current Component, it should be stored as the state of this Component
- If the data is passed to the children Components as props within two levels, it can be stored as the state of current Component. However, If it is more than 2 levels, you may consider storing it as context to avoid props drilling issue.
- If the data is used by several Components distributed across the page, you may need to use global store like recoil or redux

URL Design

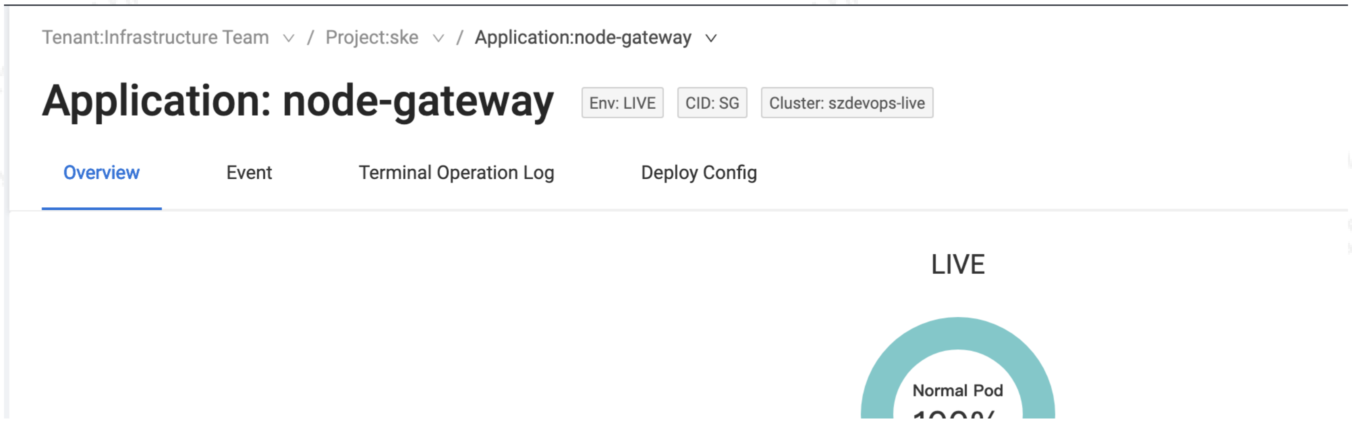
Most of our applications is **`resource-oriented`**, therefore, we recommend you use hierarchical URL structure instead of flat structure. Here is a list of example recommended hierarchical URLs:

- example.com/stores/1/books/1
- example.com/stores/1/books/1/chapters/1
- ...

And here are the corresponding non recommended flat URLs:

- example.com/books/1
- example.com/chapters/1
- ...

With a hierarchical structure in your URL, you can easily fetch **`hierarchical data of the current resource and use them in some navigation Components`** like `BreadCrumb`:



Module Importing

Path

Must use absolute imports

Comments

The Purposes of Writing Comments

- help other developers better understand the logic of your code
- use some keywords like **TODO**, **FIXME** to tell yourself or other developers to do something in the future
- tell other developers why you write this piece of code (maybe due to some restriction from backend)

What kinds of code needed to be commented

- common Components, hooks and functions
- incomplete code must be commented with a **TODO** keyword. **`Please don't just put a TODO keyword there without telling others what needs to be done and who is responsible for that`**
- code needed to be improved must be commented with **FIXME** keyword. Like TODO keyword, you also need to explains what needs to be improved
- some special cases like restriction from backend or the framework's bug should also be clarified with comments. For example, if you are writing a piece of code for hacking a react router bug, you can put a comment like `//hack the react router bug: https://linktothebug.com` in your code

Unit Testing

Here are a few guidelines on writing unit tests

When to Write Unit Tests

The following kinds of code **`must`** be unit tested when they are committed:

- common `hook`
- common `hoc`
- common `Component`
- common `helper`

`Notes:` even though unit testing on other kinds of code are not mandatory, we still encourage you to do unit testing when it's possible because it can help you find bugs at earlier stage and act as a document for other developers.

How to Write Unit Tests

For writing unit tests, you can refer to the slide Dong Xiaocong has made

Common User Interactions Handling

There are some rules on handling some common user interactions (UX designer or PM may ignore them in their designs):

Create a Resource

- the button should be in `loading` status while waiting for the backend fulfill our request
- the `submit` button should be in `disabled` status when user's input is incomplete or an error is returned by backend
- should navigate to the detail page of the newly-created resource after the call to the backend api succeeds
- should tell the user his/her request has been fulfilled via antd's `message.success`

Update a Resource

- the button should be in `loading` status while waiting for the backend fulfill our request
- the `submit` button should be in `disabled` status when user's input is incomplete or an error is returned by backend
- should tell the user his/her request has been fulfilled via antd's `message.success`
- should update the details of resource in the page with updated data fetched from backend

Delete a Resource

- the button should be in `loading` status while waiting for the backend fulfill our request
- the `submit` button should be in `disabled` status when user's input is incomplete or an error is returned by backend
- should tell the user his/her request has been fulfilled via antd's `message.success`
- should navigate to the detail page of the deleted resource's parent node after the delete request is fulfilled

Best Practices

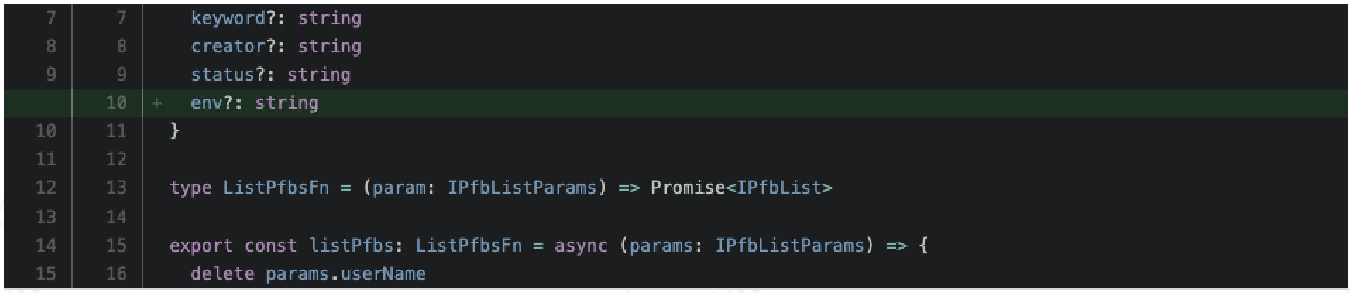
Best practices accumulated from daily code review activities.

Functional Component VS Class Component

Should always use functional components when it is possible

Helper Function Must Not Have Side Effects

Helper function must be pure function which means it can not have any side effect. Example of bad smell code:



```
    keyword?: string
    creator?: string
    status?: string
+   env?: string
  }

  type ListPfbsFn = (param: IPfbListParams) => Promise<IPfbList>

  export const listPfbs: ListPfbsFn = async (params: IPfbListParams) => {
    delete params.userName
```

In the above example, function `listPfbs` delete the `userName` property of params which may result in some unexpected behavior of other functions followed by the call of this one.