Frontend Style Guide

Generally we follow the Airbnb React Style Guide.

Table of Contents

- Frontend Style Guide
  - Table of Contents
  - Basic rules
    - Naming conventions
      - Use `PascalCase` for:
        - Typescript class names
        - Types and interfaces
        - Enums
      - Use `camelCase` for:
        - Functions
        - Methods
        - Variables
        - React state and properties
        - Emotion class names
      - Use `ALL_CAPS` for constants.
      - Use BEM convention for SASS styles.
    - Typing
    - File and directory naming conventions
    - Code organization
      - Exports
    - Comments
    - Linting
  - React
    - Props
      - Name callback props and handlers with an "on" prefix.
      - React Component definitions
      - React Component constructor
      - React Component defaultProps
  - State management
  - Proposal for removing or replacing Angular dependencies

Basic rules

- Try to keep files small and focused.
- Break large components up into sub-components.
- Use spaces for indentation.

Naming conventions

Use `PascalCase` for:

Typescript class names

```
// bad
class dataLink {
  //...
}

// good
class DataLink {
  //...
}
```

Types and interfaces

```
// bad
interface buttonProps {
  //...
}
// bad
interface button_props {
  //...
}
// bad
interface IButtonProps {
  //...
}

// good
interface ButtonProps {
  //...
}

// bad
type requestInfo = ...
// bad
type request_info = ...

// good
type RequestInfo = ...
```

Enums

```
// bad
enum buttonVariant {
 //...
}

// good
enum ButtonVariant {
```

```
  //...
}
```

Use `camelCase` for:

Functions

```
// bad
const CalculatePercentage = () => { ... }
// bad
const calculate_percentage = () => { ... }

// good
const calculatePercentage = () => { ... }
```

Methods

```
class DateCalculator {
  // bad
  CalculateTimeRange () {...}
}
class DateCalculator {
  // bad
  calculate_time_range () {...}
}

class DateCalculator {
  // good
  calculateTimeRange () {...}
}
```

Variables

```
// bad
const QueryTargets = [];
// bad
const query_targets = [];

// good
const queryTargets = [];
```

React state and properties

```
interface ModalState {
  // bad
  IsActive: boolean;
  // bad
  is_active: boolean;

  // good
```

```
  isActive: boolean;
}
```

Emotion class names

```
const getStyles  = = () => ({
  // bad
  ElementWrapper: css`...`,
  // bad
  ["element-wrapper"]: css`...`,

  // good
  elementWrapper: css`...`,
});
```

Use `ALL_CAPS` for constants.

```
// bad
const constantValue = "This string won't change";
// bad
const constant_value = "This string won't change";

// good
const CONSTANT_VALUE = "This string won't change";
```

Use BEM convention for SASS styles.

SASS styles are deprecated. Please migrate to Emotion whenever you need to modify SASS styles.

Typing

In general, you should let Typescript infer the types so that there's no need to explicitly define type for each variable.

There are some exceptions to this:

```
// Typescript needs to know type of arrays or objects otherwise it would infer it as
array of any

// bad
const stringArray = [];

// good
const stringArray: string[] = [];
```

Specify function return types explicitly in new code. This improves readability by being able to tell what a function returns just by looking at the signature. It also prevents errors when a function's return type is broader than expected by the author.

> ***Note:*** *We don't have linting for this enabled because of lots of old code that needs to be fixed first.*

```
// bad
function transform(value?: string) {
  if (!value) {
    return undefined;
  }
  return applyTransform(value);
}

// good
function transform(value?: string): TransformedValue | undefined {
  if (!value) {
    return undefined;
  }
  return applyTransform(value);
}
```

File and directory naming conventions

Name files according to the primary export:

- When the primary export is a class or React component, use PascalCase.
- When the primary export is a function, use camelCase.

For files exporting multiple utility functions, use the name that describes the responsibility of grouped utilities. For example, a file exporting math utilities should be named `math.ts`.

- Use `constants.ts` for files exporting constants.
- Use `actions.ts` for files exporting Redux actions.
- Use `reducers.ts` Redux reducers.
- Use `*.test.ts(x)` for test files.
- Use kebab case for directory names: lowercase, words delimited by hyphen (`-`). For example, `features/new-important-feature/utils.ts`.

Code organization

Organize your code in a directory that encloses feature code:

- Put Redux state and domain logic code in `state` directory (i.e. `features/my-feature/state/actions.ts`).
- Put React components in `components` directory (i.e. `features/my-feature/components/ButtonPeopleDreamOf.tsx`).
- Put test files next to the test subject.
- Put containers (pages) in feature root (i.e. `features/my-feature/DashboardPage.tsx`).
- Put API function calls that isn't a redux thunk in an `api.ts` file within the same directory.
- Subcomponents can live in the component folders. Small component do not need their own folder.
- Component SASS styles should live in the same folder as component code.

For code that needs to be used by external plugin:

- Put components and types in `@grafana/ui`.
- Put data models and data utilities in `@grafana/data`.
- Put runtime services interfaces in `@grafana/runtime`.

Exports

- Use named exports for all code you want to export from a file.
- Use declaration exports (i.e. `export const foo = ...`).
- Avoid using default exports (for example, `export default foo`).
- Export only the code that is meant to be used outside the module.

Comments

- Use TSDoc comments to document your code.
- Use react-docgen comments (`/** ... */`) for props documentation.
- Use inline comments for comments inside functions, classes etc.
- Please try to follow the code comment guidelines when adding comments.

Linting

Linting is performed using @grafana/eslint-config.

React

Use the following conventions when implementing React components:

Props

Name callback props and handlers with an "on" prefix.

```
// bad
handleChange = () => {

};

render() {
  return (
    <MyComponent changed={this.handleChange} />
  );
}

// good
onChange = () => {

};

render() {
  return (
    <MyComponent onChange={this.onChange} />
  );
}
```

React Component definitions

```
// bad
export class YourClass extends PureComponent { ... }

// good
export class YourClass extends PureComponent<{},{}> { ... }
```

React Component constructor

```
// bad
constructor(props) {...}

// good
constructor(props: Props) {...}
```

React Component defaultProps

```
// bad
static defaultProps = { ... }

// good
static defaultProps: Partial<Props> = { ... }
```

How to declare functional components

We recommend using named regular functions when creating a new react functional component.

```
export function Component(props: Props): ReactElement { ... }
```

State management

- Don't mutate state in reducers or thunks.
- Use `createSlice` . See Redux Toolkit for more details.
- Use `reducerTester` to test reducers. See Redux framework for more details.
- Use state selectors to access state instead of accessing state directly.