Front-end Coding Standards

IGNITE BRILLIANCE

Contents

Coding Standards - Front End

1. Coding Standards

- **Language**: Use ES6+ features where applicable.
- **Framework/Library**: Adhere to the conventions of the framework (e.g., React, Angular, Vue).
- **File Naming**: Use kebab-case for filenames (e.g., user-profile.js, app-header.scss).
- **Formatting**:
- 2 spaces for indentation.
- Use single quotes for strings ('), except in JSX attributes (double quotes ").
- End files with a newline.
- Use single spacing between all tokens, no spaces beyond all tokens on a line.
- **Function Naming**:
    - For exported functions, use functions declaration and not arrow functions (this will help tremendously in debugging)
- Use camelCase for regular functions (e.g., fetchUserData).
- Use PascalCase for React components (e.g., UserProfile).
- Ensure function names are descriptive and convey purpose (e.g., handleFormSubmit, not submit).

2. Code Structure

- Organize files using feature-based or component-based directories.
- Separate concerns:
- **Logic**: Use services/utils/hooks for reusable logic.
- **Styling**: Use CSS modules, styled-components, or SCSS.
- **Constants**: Place reusable values in a constants folder.

3. Styling Guidelines

- Use a CSS preprocessor (e.g., SCSS) or utility-first frameworks (e.g., Tailwind CSS).
- Avoid inline styles (unless dynamically calculated).

4. JavaScript/TypeScript Best Practices

- Always prefer strict typing with TypeScript.
- Prefer TypeScript over JavaScript whenever possible to take advantage of the typing system.
- Avoid **any** type; use generics or unions.
- Avoid **any**; prefer specific types or unknown when necessary.
- Prefer **const** always. Then prefer **const** and **let** over **var**.
- Use **async**/**await** for asynchronous code.
- Ensure functions are modular and reusable.
- Use strict typing (**strict: true** in tsconfig.json).
- Use interfaces (**interface/type**) for object shapes, especially for props or API responses.
- Define types for function parameters and return values explicitly
- Use enums or union types for predefined options
- Handler functions to have the same pattern: **${actionName}Handler(...)**
- Prefer to use default imports from 3rd packages for tree-shaking (can be enforced through eslint rules).

5. Accessibility

- Use semantic HTML (e.g., <button> instead of for buttons).
- Add aria-* attributes where applicable.
- Ensure color contrast meets WCAG standards. TBD
- Test with screen readers. TBD

6. Testing

- Unit testing: Use Jest/Vitest or equivalent (80%+ coverage).
- Component testing: Use React Testing Library/Enzyme or similar.
- End-to-End testing: Use Cypress or Playwright. TBD
- Ensure tests cover:
- Critical user journeys.
- Edge cases.
- Accessibility checks.

7. Version Control

- Use **feature branches** (feature/add-user-profile) and follow proper pull request review processes.
- Use **bugfix branches** (bugfix/user-undefined-error) for errors found in production.
- Write meaningful commit messages that indicate what action was taken to resolve:
    - "Fixed the alignment of the button using flex-box attributes" - preferred, or at least:
    - "fix: resolve button alignment issue"
- Resolve conflicts before merging.
- Follow conventional commits: https://www.conventionalcommits.org/en/v1.0.0/#summary
- We can lint commit message via commitlint: https://commitlint.js.org/
- User **Husky** to trigger some git hooks during the commit or push phases.

8. Documentation

- There should be no reason to write code comments, but for something a little more complex include inline comments.
- Use JSDoc to document all api's.

9. Linting and Formatting

- Use **ESLint** local config.
- Format code with **Prettier** local config.

10. Performance

- Lazy load components and images where possible.
- Use memoization (React.memo, useMemo) for expensive operations/components.
- Minimize use of third-party dependencies. Including further dependencies is a matter of project discussion. Only do so after consulting the team. Consideration is required when impacting the bundle.
- Setup a bundle size budget in the pipeline to track it

11. Security

- Avoid using eval() or inline scripts.
- Sanitize user inputs to prevent XSS, implement a good CSP strategy
- Use HTTPS for all API calls.
- Store sensitive data securely (use environment variables, encrypt storage).

12. Material UI (MUI) Design System (where available)

- **Theming**:
- Use the MUI theme for consistent styling. Define and extend the theme in a central theme.js or theme.ts file.
- Access theme variables (colors, spacing, typography) using the **useTheme** hook or styled utility:

```
const MyComponent = styled('div')(({ theme }) => ({
  padding: theme.spacing(2),
```

```
  color: theme.palette.primary.main,
}));
```

- **Components**:
- Use MUI components whenever possible to maintain consistency.
- Prefer customization via sx prop or theme overrides rather than inline styles.

```
<Button sx={{ margin: 2, color: 'secondary.main' }}>Click Me</Button>
```

- **Typography**:
- Use Typography component for all text to ensure accessibility and consistency.
- Leverage the variant prop instead of manual styles (e.g., variant="h1" for headings).
- **Custom Components**:
- Use styled() or Box with sx for creating custom components.
- Follow the MUI guidelines for composable and reusable design patterns.
- **Icons**:
- Use @mui/icons-material for standard icons instead of custom SVGs unless required.
- Further icons and components can be found: https://icones.js.org/collection/proicons
- Use color="inherit" or theme-based colors for icons.
- **Grid and Layout**:
- Use the MUI Grid component for layout to maintain responsiveness.
- Leverage spacing props (spacing, margin, padding) for consistent spacing:

```
<Grid container spacing={2}>
  <Grid item xs={12} sm={6}>
    <Typography variant="h6">Item 1</Typography>
  </Grid>
  <Grid item xs={12} sm={6}>
    <Typography variant="h6">Item 2</Typography>
  </Grid>
</Grid>
```

- **Accessibility**:
- Use the MUI-provided props (aria-*, role, etc.) to ensure accessible components.
- Test with MUI's built-in support for keyboard navigation and screen readers.
- **Customization**:
- Use the styled() utility or theme overrides for deep customizations.
- Avoid modifying MUI styles directly in global CSS.
-