CODING STANDARDS SPECIFICATION

Blix - AI Photo Editor

The Spanish Inquisition

June 25, 2023

Contents

Introduction

This is the Coding Standards Document for Blix, a cutting-edge native cross-platform desktop application designed to provide users with a professional and highly capable photo editing experience.

At its core, Blix combines the power of artificial intelligence with an intuitive interface, empowering users to effortlessly enhance and transform their images.

With a unique and innovative blender-like graph, Blix enables users to manipulate images in ways that go beyond conventional editing tools.

This comprehensive coding standards guide outlines the principles and best practices that govern the development of Blix, ensuring the highest standards of code quality, maintainability, and scalability.

By adhering to these standards, our team strives to deliver a consistently exceptional photo editing solution, where precision, efficiency, and professionalism converge.

Join us on this coding journey as we pave the way for a new era in photo editing excellence.

File Structure

```
Blix Photo Editor
├── assets
├── blix-plugins
│   ├── hello-plugin
│   └── input-plugin
├── src
│   ├── electron
│   │   ├── lib
│   │   │   ├── api
│   │   │   ├── commands
│   │   │   ├── core-graph
│   │   │   ├── plugins
│   │   │   │   └── builders
│   │   │   ├── projects
│   │   │   └── tiles
│   │   └── utils
│   ├── frontend
│   │   ├── api
│   │   ├── components
│   │   │   ├── graph
│   │   │   └── projects
│   │   ├── layout
│   │   │   └── tiles
│   │   ├── stores
│   │   └── types
│   └── shared
│       ├── types
│       └── utils
└── tests
    ├── unit_tests
    │   ├── electron
    │   └── frontend
    │       └── components
    ├── integration_tests
    └── end-to-end_tests
```

The Blix repository is split between 3 main folders :

- **assets**
- **src**
- **tests**

The ”asset” folder contains all the core assets that the project will be using to provide the user with a functioning ui and interface.

The ”blix-plugins” folder contains all the plugins that the user has installed to use to edit their images.

The ”src” folder contains all the source code for the project,which is split into 3 folders, electron(backend), frontend(ui), shared(utilities).

The ”tests” folder contains all the unit testing, integration testing and end-to-end testing for the project.

Github Version Control Strategy

Branching

The **Github Flow** branching strategy is used for branching.

The main branch from which all other branches must be created from is the *dev* branch. Blix operates with feature based branching, thus each branch must be created based of what feature it will be implementing.

As these features are completed or milestones are reached, these branches are merged back into the *dev* branch. The *dev* branch is then merged into the *main* branch once a release is ready to be deployed.

The nameing convention for branches is as follows:

- Specific features to be implemented must be prefixed with *feature/* : i.e *feature/palette*
- Core structures to be implemented must be prefixed according to their relevance : i.e *backend/graph*
- miscellaneous branches may be named simply accoridng to their purpose : i.e *dev*, *docs*

Github Version Control Strategy

Commits are handled according to the following rules :

- Commits must be atomic, i.e they must only contain changes related to a single feature or bug fix.
- Commits must be in the present tense, to desbribe what the commit does, not what it did.

Pull requests are handled according to the following rules :

- At least one team member other then the iniatator must review the pull request before it is merged.
- The pull request must be linked to an issue.

- The pull request must be linked to a milestone.
- The pull request must pass all automatic tests.

Issues are handled according to the following rules :

- Issues must be atomic, i.e they must only contain changes related to a single feature or bug fix.
- Issues must describe where the problem is, what the problem is, and how to reproduce the problem.

Merges conflicts are handled according to the following rules :

- The person who created the pull request must ensure that the merge conflicts are resolved.
- The relevant developers must be notified of the merge conflict and resolution.

Coding Conventions

Function conventions

- Function names are to be in camel case, with the first letter being lowercase and the first letter of each subsequent word being uppercase.
- The function names must be descriptive of what the function does.
- The function names must be in the present tense, to describe what the function does, not what it did.
- The function names must not be abbreviated.
- Parameters are formatted according to prettier standards such that each parameter is on a new line.
- Parameters are to be named according to their purpose, not their type.
- Optional parameters are to be at the very end of the parameter list.

Example :

```
1  public addSlider(
2    label: string,
3    min: number,
4    max: number,
5    step: number,
6    defautlVal: number
```

```
  ): NodeUIBuilder {
    this.node.addSlider(label, min, max, step, defautlVal);
    return this;
  }
```

Variable Conventions

- Variable names are to be in camel case, as described in the function conventions.
- Variable names must be descriptive of what the variable should do.
- const is preferred over var and let.
- Variables must never be of type any, unless absolutely neccesary.
- Global variables are discouraged, and should be avoided.
- Variables must be declared with the lowest possible scope.

Example :

```
  private signature: string,
  private name: string,
  private plugin: string,
  private title: string,
  private description: string,
  private icon: string,
  private readonly inputs: InputAnchorInstance[],
  private readonly outputs: OutputAnchorInstance[]
```

Layout conventions

- Code blocks must start with a ”{” on the same line as the declaration and end with a ”}” on a new line.
- Each declaration and statement must be on a new line.

Example :

```
    public instantiate(plugin: string, name: string): NodeBuilder {
        this.nodeBuilder.instantiate(plugin, name);
        return this.nodeBuilder;
      }
```

General Coding Conventions

- All code must be formatted according to prettier standards.
- All code must be linted according to eslint standards.
- All code must be typed according to typescript standards.
- All code must be tested according to jest standards.

Example :

```
private nodesToJSONObject(): NodeToJSON[] {
    const json: NodeToJSON[] = [];
    for (const node in this.nodes) {
      if (!this.nodes.hasOwnProperty(node)) continue;
      json.push(this.nodes[node].toJSONObject());
    }
    return json;
  }
```

Formatting tools

Blix makes use of the following formatting tools to ensure standardization of code and to enforce the coding conventions :

- **Prettier** : Prettier is an opinionated code formatter. It enforces a consistent style by parsing your code and re-printing it with its own rules that take the maximum line length into account, wrapping code when necessary.
- **eslint** : Eslint is a tool for identifying and reporting on patterns found in ECMAScript/-JavaScript code, with the goal of making code more consistent and avoiding bugs.

There are a sizeable number of rules that are enforced by these tools, and they are not all listed here. For a full list of rules enforced by these tools, please refer to the *.eslintrc.js* and *.prettierrc.js* files in the root directory of the project.

In summary these tools are configured to enforce the conventions described above, and to ensure that the code is formatted in a consistent manner.