Contributing Guidelines

Welcome, and thank you for your interest in contributing to Lichtblick! We value your contributions and want to make the contributing experience enjoyable and rewarding for you. Here’s how you can get started:

🚀 Getting Started

Please check our README.md and follow the installation steps.

Other useful commands

```
# To launch the storybook
$ yarn storybook
```

```
# Advanced usage: running webpack and electron on different computers (or VMs)
on the same network
$ yarn desktop:serve --host 192.168.xxx.yyy # the address where electron can
reach the webpack dev server
$ yarn dlx electron@22.1.0 .webpack # launch the version of electron for the
current computer's platform
```

```
$ yarn run                  # list available commands
$ yarn lint                 # lint all files
$ yarn test                 # run all tests
$ yarn test:watch           # run tests on changed files
$ yarn test:integration     # run all integration tests
```

🌿 Creating a new branch

To create a branch in this repository, please follow the guidelines below, ensuring that the purpose of each branch is clear and well-defined:

- `feature` : Create this branch when adding new features, modifying existing features, or removing outdated functionality.
- `bugfix` : This branch is for resolving bugs discovered in existing features.
- `hotfix` : Use this for rapidly addressing critical issues. This typically involves implementing a temporary solution that requires immediate attention.
- `test` : This is intended for experimental changes, where the main goal is to explore new ideas or test solutions without addressing a specific issue.

- `docs` : Designate this branch for updates and improvements to documentation, ensuring that information is current and helpful to users.
- `wip` (Work In Progress): Use this for ongoing development that is not yet ready for merging into the main branch.
- `cicd` : Use this for changes into pipeline ci/cd scripts.

Examples

`feature/new-menu-foo`

`test/create-unit-test-for-component-bar`

:label: Version increment

Semantic Versioning has been chosen as our standard method for version increments, which is widely adopted across various software projects. The version format is structured as follows: `<major>.<minor>.<patch>[.<build number>]`.

- *Note: Currently, the version increment process is manual. Developers are required to update the version number in the package.json file manually. An automated pipeline for this task is in development and will be implemented to streamline this process in the future.*

Components

MAJOR: Increasing the major version usually breaks compatibility, allowing developers to remove the deprecated API or rework the existing ones. Users know about it and do not expect a smooth update.

MINOR: Version increment implies adding new functionality without breaking compatibility.

PATCH: Also known as bugfix version that includes fixing security vulnerabilities, etc.

BUILD NUMBER (Optional): Optionally, the build number can be additionally added.

Examples

`1.20.11`

`1.20.11.403`

🌐 Localization

At this time, first-class support for Lichtblick is provided in English only. Localization into other languages is available on a best-effort basis, with translations provided by community volunteers.

Translation support is implemented using `react-i18next`.

Add translations

- We value having *high-quality* translations over having *all* translations for a given component or view. Though every PR must have up-to-date English translations, updating other languages is

completely optional.
- If you update an English translation and cannot provide updated non-English translations, delete the non-English versions in that PR. Optionally, open follow-up PRs to add accurate non-English translations.

Add translations to the `i18n` directory

The `i18n` directory contains translated (localized) strings for all languages supported by Lichtblick.

Translated strings are organized into *namespaces* — e.g. `i18n/[language]/appSettings.ts` contains translations for the app's Settings tab.

Use `useTranslation()` and `t()` to access translated strings

1. Call the `useTranslation(namespace)` hook inside a React component to access strings in a given namespace. The hook returns a function called `t`.
2. Call the `t` function to get the translation for a string.

For example:

```
const { t } = useTranslation("myComponent");
return <p>{t("hello")}</p>;
```

Add localization support to a component

1. Move English strings out of the component code, and into the `i18n` folder. Use a new namespace for logical groups of components or app views.
2. Replace strings hard-coded in source code with calls to the `t()` function. Use `camelCase` for new localization keys.

Before **After**

```
function MyComponent() {
  return <p>Hello!</p>;
}
```

```
function MyComponent() {
  const { t } = useTranslation("myComponent");
  return <p>{t("hello")}</p>;
}
```

```
// i18n/en/myComponent.ts
export const myComponent = {
  hello: "Hello!",
};
```

Complete example

```
// MyComponent.ts

import { useTranslation } from "react-i18next";

function MyComponent(props: Props): React.JSX.Element {
  const { t } = useTranslation("myComponent");

  return <p>{t("hello")}</p>;
}
```

```
// i18n/en/myComponent.ts
export const myComponent = {
  hello: "Hello!",
};

// i18n/en/index.ts
export * from "./myComponent";
```

```
// i18n/zh/myComponent.ts
export const myComponent: Partial<TypeOptions["resources"]["myComponent"]> = {
  hello: "你好！",
};

// i18n/zh/index.ts
export * from "./myComponent";
```

Result:

English	Chinese
`<p>Hello!</p>`	`<p>你好！</p>`