Reporting Issues

This page discusses how to open issues, including the policies and procedures of the Jellyfin project around handling issues.

Issues should **only** detail software bug reports.

All other discussions, including initial troubleshooting, should be directed towards our help channels.

Requesting Features

Please note that feature and enhancement requests should be directed towards our Fider instance for tracking, voting, and reporting. Please keep all feature requests to this page and not GitHub issues.

Searching and Voting

Before opening an issue, please search the existing issues to see if a similar problem or feature request has been reported. Duplicate issues clutter the repository and should be avoided.

If you do find an issue that matches, or closely matches, your issue, please make use of the 👍 reaction to confirm the issue also affects you or that you support the feature request. If you wish, add a comment as well describing your version of the issue or feature use case.

If the existing issue is closed, please read through it to see if the accepted workaround(s) apply to your case. If not, leave a comment and the issue will be reopened. Note that, since PRs go into `dev` first but releases are built from `master`, an issue's fix won't be immediately available in the official sources, but will be included in the next release.

Opening an Issue

Once you're ready to open an issue, please see this page!

Reporting Bugs

When writing an bug issue, please ensure you capture as much relevant detail as possible - this is very important to assist in troubleshooting and triaging/investigating the issue. Some useful elements include:

- How you installed Jellyfin (upgrade or fresh install)
- What platform and operating system you're using (Debian, Arch, Docker, etc.)
- What you were doing that caused the issue to appear
- Any relevant log output

- Any non-standard configurations you use

Bugs should be tagged with `[bug]` at the beginning of their title. This will later be removed by the Jellyfin team when assigning labels. To assist in triaging, if you know which other label(s) should be applied to your issue, please add them after the `[bug]` label.

Bugs should be reproduceable. That is, you should be able to have determined through troubleshooting how to replicate the issue. While one-time bugs shouldn't be ignored, if they're difficult or impossible to reproduce, it's likely very hard to fix them. Please attempt to reproduce the bug before filing the issue, and include the smallest test case you can to demonstrate it.

If you ever need assistance for troubleshooting or opening an issue, please contact the community and we'll try to help you out!

Issue Labels

Jellyfin features a number of issue labels to assist in triaging and managing issues. Users cannot assign these themselves due to GitHub's permissions, but they will be added by an team member during triaging.

Categories

These labels are broad categories for which part of the codebase is affected.

- `backend`: An issue that mainly relates to the server backend code.
- `build`: An issue that mainly relates to the build process.

Criticality

These labels help determine how critical an issue is.

- `regression`: An issue in need of immediate attention due to a regression from the last build.
- `bug`: A bug in the code that affects normal usage.

Management

These labels help assist in managing the project and direction.

- `good first issue`: Something that should be very straightforward to do, and is a great place to get started.
- `help wanted`: An issue that currently has no clear expert within the project and could use outside assistance.
- `roadmap`: A meta-issue related to the future roadmap of the project.
- `investigation`: An investigation-type issue into the codebase.

Pull Requests

These labels apply only to pull requests for administrative purposes.

- `requires testing`: A PR that has not been tested in a live environment yet. Any major backend-affecting PRs should be tested before being merged to avoid regressions.

How to contribute code to Jellyfin

This page details how our repositories are organized, how to get started editing the code and creating your first pull request, and some general procedures around pull requests in Jellyfin.

What should you work on?

There are many projects within the organization to browse through for contributions. Summarized here are the two biggest ones, one for backend devs and another for frontend devs.

- Jellyfin Server: The server portion, built using .NET 6 and C#.
- Jellyfin Web: The main client application built for browsers, but also used in some of our other clients that are just wrappers.

Note that each of the repositories also has its own documentation on how to get started with that project, generally found in the repository README. You can also view the organization source tree to see how some of the bigger projects are structured.

The best way to get going on some actual development is to look through the issues list of the associated repository, find an issue you would like to work on, and start hacking! Issues are triaged regularly by the administrative team, and labels assigned that should help you find issues within your skill-set. Once you start working on an issue, please comment on it stating your intent to work on the issue, to avoid unnecessary duplication of work.

Major Issue Types

A list of issue types can be found on the issue guidelines section.

What if there isn't an issue?

If there isn't already an issue dealing with the changes you want to make, please create an issue to track it first, then ensure your PR(s) reference the issue in question. This is especially useful for bugs that are found and then fixed by the author, so both the original issue and the fix can be documented and tracked in detail.

How should you make changes?

Once you've found something you want to work on or improve, the next step is to make your changes in the code, test them, then submit a Pull Request (PR) on GitHub.

For simplicity, all examples assume the developer is operating on Linux with SSH access to GitHub, however the general ideas can be applied to HTTP-based GitHub interfaces, and can be translated to Windows or MacOS.

If you aren't familiar with Git, we recommend the official documentation to understand how this version control system works and how to use it.

Set up your copy of the repo

The first step is to set up a copy of the Git repository of the project you want to contribute to. Jellyfin follows a "fork, feature-branch, and PR" model for contributions.

1. On GitHub, "Fork" the Jellyfin repository you wish to contribute to, to your own user account using the "Fork" button in the relevant repository.

2. Clone your fork to your local machine and enter the directory:

   ```
   git clone git@github.com:yourusername/projectname.git
   cd projectname/
   ```

3. Add the "upstream" remote, which allows you to pull down changes from the main project easily:

   ```
   git remote add upstream git@github.com:jellyfin/projectname.git
   ```

4. To get the `Jellyfin.Server` project to run successfully, checkout both the server, as well as the web client project.

5. Build the Jellyfin Web project with NPM, and copy the location of the resulting `dist` folder.

6. In your `Jellyfin.Server` project add an environment variable named `JELLYFIN_WEB_DIR` with the value set to the full path of your `dist` folder.

You will now be ready to begin building or modifying the project.

Make changes to the repo

Once you have your repository, you can get to work.

1. Rebase your local branches against upstream `master` so you are working off the latest changes:

   ```
   git fetch --all
   git rebase upstream/master
   ```

2. Create a local feature branch off of `master` to make your changes:

   ```
   git checkout -b my-feature master
   ```

3. Make your changes and commits to this local feature branch.
4. Repeat step 1 on your local feature branch once you're done your work, to ensure you have no conflicts with other work done since you stated.
5. Push up your local feature branch to your GitHub fork:

```
git push --set-upstream origin my-feature
```

6. On GitHub, create a new PR against the upstream `master` branch following the advice below.
7. Once your PR is merged, ensure you keep your local branches up-to-date:

```
git fetch --all
git checkout master
git rebase upstream/master
git push -u origin master
```

8. Delete your local feature branch if you no longer need it:

```
git branch -d my-feature
```

CONTRIBUTORS.md

If it's your first time contributing code to a particular repository, please add yourself to the `CONTRIBUTORS.md` file at the bottom of the `Jellyfin Contributors` section. While GitHub does track this, having the written document makes things clearer if the code leaves GitHub and lets everyone quickly see who's worked on the project for copyright or praise!

Official Branches

Feature Branches

From time to time, major projects may come up that require multiple PRs and contributions from multiple people. For these tasks, feature branches specific to the feature should be created, based off of `master`. This helps allow the work to progress without breaking `master` for long periods, and allowing those interested in that particular project the ability to work at their own pace instead of racing to fix a broken feature before the next release. To create a feature branch, please communicate with a Core team member and that can be arranged.

Once the feature a feature branch was created for is ready, it can be merged in one shot into `master` and the feature branch removed. Alternatively, for very-long-lived features, certain "stable" snapshots can be

merged into `master` as required.

The Master Branch

The `master` branch is the primary face of the project and main development branch. Except for emergency release hotfixes, all PRs should target `master`. As a general rule, no PR should break master and all PRs should be tested before merging to ensure this does not occur. We're only human and this is still likely to happen, but you should generally be safe to build off of `master` if you want the latest and greatest version of Jellyfin.

Testing a Pull Request

To test someone else's pull request, you must import the changes to your local repository.

1. Fetch the changes in a pull request and link them to a new local branch:

   ```
   git fetch upstream pull/<PR_ID>/head:my-testing-branch
   ```

   > **NOTE**
   > `<PR_ID>` is pull request number on GitHub.

2. Checkout the new local branch:

   ```
   git checkout my-testing-branch
   ```

3. Perform any testing or build required to test, then return to master and delete the branch:

   ```
   git checkout master
   git branch -D my-testing-branch
   ```

Pull Request Guidelines

When submitting a new PR, please ensure you do the following things. If you haven't, please read How to Write a Git Commit Message as it is a great resource for writing useful commit messages.

- Before submitting a PR, squash "junk" commits together to keep the overall history clean. A single commit should cover a single significant change: avoid squashing all your changes together, especially for large PRs that touch many files, but also don't leave "fixed this", "whoops typo" commits in your branch history as this is needless clutter in the final history of the project.

- Write a good title that quickly describes what has been changed. For example, "Add LDAP support to Jellyfin". As mentioned in How to Write a Git Commit Message, always use the imperative mood, and keep the title short but descriptive. The title will eventually be a changelog entry, so please try to use proper capitalization but no punctuation; note that the Core team may alter titles to better conform to this standard before merging.

- For anything but the most trivial changes that can be described fully in the (short) title, follow the PR template and write a PR body to describe, in as much detail as possible:

    1. Why the changes are being made. Reference specific issues with keywords (`fixes`, `closes`, `addresses`, etc.) if at all possible.

    2. How you approached the issue (if applicable) and briefly describe the changes, especially for large PRs.

- If your pull request isn't finished yet please mark it as a "draft" when you open it. While this tag is in place, the pull request won't be merged, and reviews should remain as comments only. Once you're happy with the final state of your PR, please remove this tag; forgetting to do so might result in your PR being unintentionally ignored as still under active development! Inactive WIPs may occasionally elicit pings from the team inquiring on the status, and closed if there is no response.

- Avoid rebasing and force-pushing to large or complex pull requests if at all possible, and especially after reviews. It forces unnecessary reviews to verify the changes are still okay and build properly.

- Expect review and discussion. If you can't back up your changes with a good description and through review, please reconsider whether it should be done at all. All PRs to `dev` require at least one approving review from an administrative team member, however we welcome and encourage reviews from any contributor, especially if it's in an area you are knowledgeable about. More eyes are always better.

- All PRs require review by at least two team members before being merged into `master`, though reviews from any contributor are welcome! After the second team member review the PR may be merged immediately, or more review or feedback requested explicitly from other contributors if required.

Building and Testing Inside a Docker Container

We need to install all development dependencies and pull down the code inside the container before we can compile and run.

> **NOTE**
>
> Run each command on a separate line. The container we'll test in is named `jftest`. Within Docker, anytime the entrypoint executable is terminated, the session restarts, so just exec into it again to continue. This is also why we explicitly kill it to reload the new version.

Master Branch

```
docker exec -ti jftest bash
apt-get update && apt-get install git gnupg wget apt-transport-https curl autoconf g++ make
libpng-dev gifsicle automake libtool make gcc musl-dev nasm
wget -qO- https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor >
microsoft.asc.gpg && mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/
wget -q https://packages.microsoft.com/config/debian/10/prod.list && mv prod.list
/etc/apt/sources.list.d/microsoft-prod.list
apt-get update && apt-get install dotnet-sdk-6.0 npm
cd /opt && git clone https://github.com/jellyfin/jellyfin.git && git clone
https://github.com/jellyfin/jellyfin-web.git
cd jellyfin/ && DOTNET_CLI_TELEMETRY_OPTOUT=1 dotnet publish --disable-parallel
Jellyfin.Server --configuration Debug --output="/jellyfin" --self-contained --runtime
linux-x64
cd /opt/jellyfin-web && npm install && cp -r /opt/jellyfin-web/dist /jellyfin/jellyfin-web
kill -15 $(pidof jellyfin)
```

Pull Request

First, complete the steps above to setup your container to build the master branch.

> **NOTE**
>
> `<PR_ID>` is pull request number on GitHub.

```
docker exec -ti jftest bash
cd /opt/jellyfin
git fetch origin pull/<PR_ID>/head:my-testing-branch
git merge my-testing-branch
dotnet build
kill -15 $(pidof jellyfin)
```

Release Procedure

This document is a guide for the core team, provided publicly to ensure transparency in the release process.

Versioning

Jellyfin uses semantic versioning. All releases will have versions in the `X.Y.Z` format, starting from `10.0.0`. Note however that the `10.Y.Z` release chain represents the "cleanup" of the codebase, so it should be accepted that `10.Y.Z` breaks all compatibility, at some point, with previous Emby-compatible interfaces, and may also break compatibility with previous `10.Y` releases if required for later cleanup work. Our versioning will typically follow the patterns below:

X: Major Versions

- Breaks compatibility with the HTTP or plugin APIs

Y: Minor Versions

- Introduces new features
- Makes minor backwards-compatible API changes

Z: Hotfix Versions

- Critical bug fixes or minor changes

General Release Philosophy

Releases will generally be performed on Sundays "when ready". For Major/Minor releases, the "when ready" is generally quite flexible and is whenever the release is truly ready without major breaking bugs. After a major release, each Sunday the Admin team should review the recently merged PRs and, if backports are required, perform a Hotfix release containing those PRs.

Major Release Procedure

Preparation

1. Testing is ongoing via `master` nightly builds, so `master` should be generally unbroken before proceeding. The version of `master` should already reflect the upcoming major release version (i.e. `X.Y.0`).

2. Once `master` is in a generally stable state after extensive work, announce a "golden nightly" is incoming via the jellyfin-dev Matrix/Riot channel and Reddit.

3. Collect testing information and repeat as needed.

4. Once the release is considered stable and working, announce full PR freeze via the jellyfin-dev Matrix/Riot channel.

5. Allow one further "golden nightly" and at least 48 hours of testing time. Restart this process if major breaking bugs are found.

6. Once all testing is complete and the release remains stable, proceed.

Release Web Client

1. Create a release branch on the jellyfin-web repository via CLI from `master`, named `release-X.Y.z`, where `X` and `Y` are the new version number, and `z` is a literal `z`. Push the new branch to GitHub.

2. Create a GitHub release for the new version, based on the newly-created `release-X.Y.z` branch. The tag should be named `vX.Y.Z` (i.e. `vX.Y.0`) and the release named "Release X.Y.Z". The release body should contain the following link only, replacing the version as required:

   ```
   [Please see the release announcement on the main repository.]
   (https://github.com/jellyfin/jellyfin/releases/tag/vX.Y.Z)
   ```

3. Publish the release.

Release Server

1. Create a release branch on the jellyfin repository via CLI from `master`, named `release-X.Y.z`, where `X` and `Y` are the new version number, and `z` is a literal `z`. Push the new branch to GitHub.

2. Create a GitHub release for the new version, based on the newly-created `release-X.Y.z` branch. The tag should be named `vX.Y.Z` (i.e. `vX.Y.0`) and the release named "Release X.Y.Z". The release body should contain the following components:

   a. A quick top blurb under a `# Jellyfin X.Y.Z` header.

   a. A list of features, including in-line links to Fider if available, under a `## New Features and Major Improvements` header.

   a. A list of known release notes, categorized by the relevant platform (e.g. `[All]` or `[Windows]`), under a `## Important Release Notes` header.

   a. If applicable, a set of release notes/comments about FFmpeg, under a `## FFmpeg` header.

   a. A full changelog, split by repository with `### [repo](https://github.com/jellyfin/repo)` subheaders, under a `## Changelog` header. Each element should be a PR number and the PR title.

3. Publish the release.

4. Wait for builds to complete.

5. Announce the new release in the jellyfin-announce Matrix/Riot channel and anywhere else required (e.g. Reddit, etc.).

Hotfix Release Procedure

1. During normal work on the `master` branch, select PRs suitable for backporting by tagging them with the `stable-backport` label during the PR lifecycle. All PRs will target `master` and thus bugfixes for the stable release must include this label to be included.

2. Collect the list of merged `stable-backport` PRs from all relevant repositories.

3. For each repository, perform stable branch reconciliation for the relevant PRs:

    1. For each PR slated for backport:

        1. Grab the *merge commit* hash for the PR from `master` branch.

        2. Cherry-pick the merge commit into the `release-x.y.z` branch via: `git cherry-pick -sx -m1 <merge-commit-hash>`.

        3. Fix any merge conflicts, generally keeping what's in the merge. If there are significant merge conflicts, this likely indicates that the fix is too large for backporting.

        4. Finalize the cherry-pick via: `git add` and `git commit -v`.

    2. For the main jellyfin repository, bump the version of the repository to the new hotfix version with the `bump_version` script and commit the result with the message "Bump version for X.Y.Z".

    3. Push the updated release branch to GitHub.

Web Client

1. Create a GitHub release for the new version, based on the relevant `release-X.Y.z` branch. The tag should be named `vX.Y.Z` and the release named "Release X.Y.Z". The release body should contain the following link only, replacing the version as required:

```
[Please see the release announcement on the main repository.]
(https://github.com/jellyfin/jellyfin/releases/tag/vX.Y.Z)
```

2. Publish the release on GitHub and the archive repository.

Server

1. Create a GitHub release for the new version, based on the relevant `release-X.Y.z` branch. The tag should be named `vX.Y.Z` and the release named "Release X.Y.Z". The release body should contain the following components:

   a. A quick top blurb under a `# Jellyfin X.Y.Z` header.

   a. A list of known release notes, categorized by the relevant platform (e.g. `[All]` or `[Windows]`), under a `## Important Release Notes` header.

   a. If applicable, a set of release notes/comments about FFmpeg, under a `## FFmpeg` header.

   a. A full changelog, split by repository with `### [repo](https://github.com/jellyfin/repo)` subheaders, under a `## Changelog` header. Each element should be a PR number and the PR title.

2. Publish the release.

3. Wait for builds to complete.

4. Announce the new release in the jellyfin-announce channel and anywhere else as required.

Source Tree

Jellyfin is a maze of clients, plugins, and other useful projects. These source trees can serve as an excellent tool to inform new developers about the structure of several projects.

Jellyfin Server

1. .ci: `Azure Pipelines Build definitions`
2. DvdLib: `DVD Anaylzer`
3. Emby.Dlna: `DLNA support for the server`
    - Profiles: `DLNA Profiles for clients`
4. Emby.Drawing: `image processor managing the image encoder and image cache paths`
5. Emby.Naming: `parsers for the media filenames`
6. Emby.Notifications: `listening for events and sending the associated notification`
7. Emby.Photos: `metadata provider for photos`
8. Emby.Server.Implementations: `main implementations of the interfaces`
    - ScheduledTasks: `all scheduled tasks can be found here`
9. Jellyfin.Api: `Jellyfin API`
    - Controller: `API controllers answering the Jellyfin API requests`
    - Helpers:
        - MediaInfoHelper.cs: `logic for the stream builder that determines method of playback such as Direct Play or Transcoding`
10. Jellyfin.Data: `models used in the Entity Framework Core Database schema`
11. Jellyfin.Drawing.Skia: `image manipulation like resizing images, making image collages`
12. Jellyfin.Networking: `managing network interaces and settings`
13. Jellyfin.Server.Implementations: `like Emby.Server.Implementations, implementations using the EF Core Database`
14. Jellyfin.Server: `main server project that starts the whole server`
15. MediaBrowser.Common: `common methods used throughout the server`
16. MediaBrowser.Controller: `interface definitions`
17. MediaBrowser.LocalMetadata: `metadata provider and saver for local images, local Collections and Playlists`
18. MediaBrowser.MediaEncoding: `managing ffmpeg while interacting with the media files`
19. MediaBrowser.Model: `defining models used throughout the server`
20. MediaBrowser.Providers: `managing multiple metadata sources`
21. MediaBrowser.XbmcMetadata: `metadata provider and saver for local .nfo files`
22. RSSDP: RSSDP library, `including custom changes, for the Simple Service Discovery (SSDP) protocol`
23. apiclient: `files used for generating the axios API client`
24. deployment: `files used while building Jellyfin for different plattforms`
25. tests: `multiple Unit Test projects testing Jellyfin functionality`

26. Dockerfile.* `Dockerfiles defining the Jellyfin Docker image`

Web Client

1. src:
    - assets: `images, styles, splash screens, and any other static assets`
        - css: `all global stylesheets used throughout the client`
        - img: `images for things like device icons and logos`
        - splash: `progressive web apps will show these splash screens`
    - components: `custom elements used for different sections of the user interface`
        - playerstats.js: `display playback info in browsers and other clients that include the web source`
    - controllers: `scripts that handle the logic for different pages`
    - elements: `custom UI components that are used globally such as buttons or menus`
    - legacy: `currently used for all polyfills and scripts related to backwards compatibility`
    - libraries: `dependencies that we eventually want to remove and include during the build step`
    - scripts: `any script that isn't tied to a UI element or page but rather general functionality`
    - strings: `translations for the entire interface`
    - themes: `custom and bundled themes can be found here in their own directories`

Android

1. res:
    - android:
2. src:
    - NativeShell:
        - res:
        - src:
            - RemotePlayerService.java: `handles the notification tile that can control playback`
        - www:
    - cordova:

Android TV

1. app:
    - src:
        - main:
            - java/org/jellyfin/androidtv:
                - constant: `constants/enums`
                - data:

        - compat: `classes ported from old apiclient to maintain compatibility in the app (Deprecated should be replaced/removed)`
        - eventhandling: `API webservice event handling`
        - model: `various data models`
        - querying: `extensions to the querying package in the apiclient (Should probably be replaced/removed)`
        - repository: `data repositories for shared access`
      - di: `dependency injection modules`
      - integration: `Android TV homescreen channel integrations`
      - preference: `interface for Android shared preferences`
      - ui:
        - browsing: `views for browsing items (rows, grids, etc.)`
        - home: `home screen views`
        - itemdetail: `item detail views`
        - itemhandling: `BaseItem views`
        - livetv: `live TV views`
        - playback: `media player views`
        - preference: `app preferences/settings views`
        - presentation: `presenters from MVP architecture`
        - search: `search views`
        - shared: `shared code for UI classes`
        - startup: `authentication views`
      - util: `various utilities`
    - res: `Android resource files for XML layouts, translations, images, etc.`

Kodi

1. jellyfin_kodi
   - database: `manipulating the local Jellyfin sqlite database`
   - dialogs: `code behind popup menus for user interaction`
   - entrypoint: `main add-on settings page`
   - helper: `small helper functions, mostly formatting or reused functions`
   - jellyfin: `interacting with the server`
   - objects:
     - kodi: `handling local Kodi media types and database`
2. resources:
   - language: `string files for localization`
   - skins: `design of popup menus for user interaction`

Branding

Usage of the Jellyfin name

You are free to use the Jellyfin name to promote your project, with some restrictions:

- Do not use the Jellyfin name in a way that would make the average user think you are associated with the project, unless permission was given by the Project Leader or Leadership Team.
- Only include the Jellyfin name in your project's name in a way that makes it clear you are not affiliated with the Jellyfin project, and to indicate compatibility with Jellyfin (For example *Awesome Client for Jellyfin*).
- Do not use the Jellyfin name in any context that promotes, allows or encourages piracy.
- Do not wrongfully claim to be part of the Jellyfin team.

Writing Style

As a general rule, Jellyfin should always be capitalized, but language, file, or system conventions trump Jellyfin naming conventions.

Specific examples include:

- Writing referring to the project in the abstract should use capitalized `Jellyfin` at all times. `I contribute to Jellyfin and you should too!`
- C# class and project names, including their files and directories, should use capitalized `Jellyfin` as required by the C# case standards (camelCase or PascalCase). `Jellyfin.LiveTv`, `Jellyfin.sln`
- Other code elements, where the code formatting or style requires lowercase, should use lowercase `jellyfin`. `jellyfinWebComponentsBowerPath`
- The Git repository and non-C# files inside of it should use lowercase `jellyfin` for convenience on case-sensitive filesystems. `build-jellyfin.ps1`
- The final output binary, initscrips, and package names should use lowercase `jellyfin` for similar reasons as above. `jellyfin.dll`, `jellyfin_3.5.2-1_all.deb`, `jellyfin.zip`
- Configuration directories can use either depending on operating system conventions. `/var/lib/jellyfin`, `AppData/Jellyfin`
- The logo has no strict rules for capitalization, the style is dependent on aesthetics and font choice.

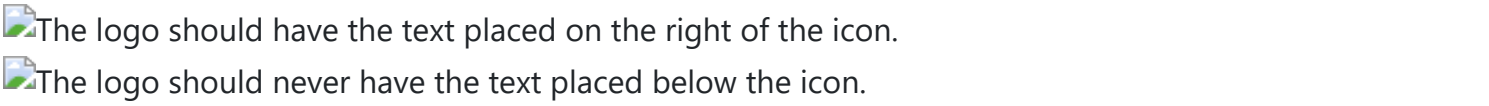
Icons and Other Assets

All iconography and other resources can be found in the jellyfin-ux repository.

- Icons
- Banners
- Fonts

Logo

When using the full version of the logo, the text should only be placed to the right of the icon.



The design for the logo uses a gradient for the infill, and if the non-transparent logo is chosen there is an optional background color.

- Gradient Start: `#AA5CC3`
- Gradient End: `#00A4DC`
- Background Colour: `#000B25`

Theme

- Background Colour: `#101010`
- Accent Colour: `#00A4DC`

Fonts

The banner uses the Quicksand font.

Style Guides

This section documents the code style used for the different languages used by Jellyfin.

If the language you're looking for doesn't have a style guide yet, respect the style of the surrounding code in the files you are editing.

JavaScript

Filenames

Filenames must be camel case and may not include underscores (_) or dashes (-). The filename's extensions must be `.js`.

File Structure

All files must abide by the following general structure, with JSDoc comments being optional but preferred.

```
/**
 * This module documents the structure used by Javascript code in Jellyfin.
 *
 * @module path/to/this/module
 */

import module from 'dependency';
import { myFunction, myClass } from 'dependency/submodule';
import 'otherDependency';

/**
 * Defines a non-exported function, accessible only from this module.
 *
 * @param {Object} argument - The argument to pass to the function.
 * @returns {Int|null} The resulting object from the function.
 */
function privateFunction (argument) {
    // Code ommitted
}

export publicFunction (argument) {
    // Code ommitted
}

export default { publicFunction }
```

Miscellaneous

File Encoding

All files must be encoded in UTF-8 and use LF line endings when committed.

Non-ASCII Characters

For printable characters, use the actual Unicode character directly in your code.

For non-printable characters, use the hexadecimal or Unicode escape.