This document aims to outline at a high level the various parts that make up MapHero Native and how they work together.

Repository structure

MapHero Native uses a monolithic repository that houses both core C++ code and code that wraps the C++ core with SDKs for Android, iOS, macOS, Node.js, and Qt. A "monorepo" allows us to:

- Make changes to the core API and SDKs simultaneously, ensuring no platform falls behind.
- Ensure that core changes do not inadvertently break SDK tests.
- Centralize discussions about features and defects that affect multiple platforms.

Core cross-platform C++ code

In the repository, core C++ code is contained in the `include` and `src` directories. The former includes headers that are considered to make up the "public" core C++ API, while the latter includes `.cpp` implementation files and headers that are private to the implementation. Within both directories, files are nested under an `mbgl` directory, which has various subdirectories based on areas of functionality. Both public and private headers therefore can (and should) always be included with the form `#include <mbgl/___/___.hpp>`.

Platform specific code

Code and build scripts belonging to platform SDKs are contained in the `platform` directory, which has subdirectories for each platform.

- `platform/darwin` and `platform/default` directories contain code shared by multiple platform SDKs.
- `platform/ios` and `platform/macos` - the SDKs for Apple's operating systems, forked from https://github.com/mapbox/mapbox-gl-native-ios/commit/a139216 (mapbox hosted both iOS and MacOS SDKs in the same project).
- `platform/android` - Android SDK, forked from https://github.com/mapbox/mapbox-gl-native-android/commit/4c12fb2c.
- `platform/glfw` - GLFW is library to support OpenGL development on the desktop. The code in this directory builds an executable application `mbgl-glfw` for demo/dev/local testing purposes.

Other directories

- `benchmark` contains the performance tests built using https://github.com/google/benchmark/. The code under this directory builds the `mbgl-benchmark-test` executable to execute the benchmark tests.
  - The entry point of this executable is `platform/default/src/mbgl/benchmark/main.cpp -> benchmark/src/mbgl/benchmark/benchmark.cpp`.
  - iOS SDK runs benchmark test through a separate app named `BenchmarkApp`.
  - Android SDK does not have benchmark test.

- `bin` contains the code for tools like `mbgl-cache`, `mbgl-offline`, and `mbgl-render`.
- `expression-test` contains tests for the expression feature in the map style (see more details about expression here.
- `metrics` contains test manifest files and ground truth for graphic comparison based render test.
- `misc` contains protobuf for style, vector tile, and glyphs. It also icons and pictures used in documents.
- `render-test` contains image diff based render tests. These tests verify if the rendering results match with expectations by capturing the rendering results and compare with the groundtruth images in the `metrics` directory.
    - A typical render test run can be triggered by `mbgl-render-test-runner --manifestPath metrics/linux-clang8-release-style.json`. It will by default generate a html file `linux-clang8-release-style.html` with test results visualized and summarized.
- `test` contains unit test for the C++ native code. The tests can be run through `mbgl-test-runner` executable after building the project.
- `vendor`: 3rd party dependencies.

Build system

The MapHero Native build system uses a variety of tools.

Make

GNU Make and a master `Makefile` serves as the coordinating tool for all other build system tools. Most of your interaction with the build system will be via `make` commands. We aim to have a consistent pattern for make targets across platforms: for a given platform, e.g. `ios`, `make ios` builds a reasonable set of end products (though perhaps not all permutations of build options, which can be prohibitively expensive), and `make ios-test` runs a complete test suite for the platform.

Git submodules

Git submodules are used to pull in several dependencies: mason (see below) and several iOS dependencies (though we have plans to phase these out). Initializing these submodules is handled automatically by the necessary `make` targets.

npm

npm is a package manager for Node.js. MapHero Native uses it to pull in several development dependencies that happen to be packaged as node modules -- mainly for testing needs.

Mason

Mason is Mapbox's own cross platform C/C++ package manager. MapHero Native uses mason packages as a source of precompiled binaries for third-party dependencies such as Boost, RapidJSON, and SQLite, and Mapbox's own C++ modules such as earcut.hpp and geojson-vt-cpp. It is also used to obtain a toolchain for Android platform cross-compiliation.

We track mason dependencies for a given platform in the file `platform/<platform>/scripts/configure.sh`. The `configure` script at the root handles sourcing this file during the build, running the appropriate mason commands, and writing configuration settings for the subsequent build to a `config.gypi` in the build output directory.

gyp

gyp is a build system originally created for Chromium and since adopted by Node.js. In MapHero Native it's used to build the core C++ static library, the Node.js platform module, and the shared JNI module for Android.

Platform-specific subsystems

Outside of the core C++ static library, platform SDKs typically rely on platform-native build tooling to complete the job.

- For iOS and macOS this means Xcode and the xcodebuild command line tool.
- For Android, Gradle and Android Studio.
- For Qt, plain CMake is used to generate the build system.

See the relevant platform-specific `README.md` / `INSTALL.md` for details.

Major functional components

Map

Style

The "Style" component of MapHero Native contains an implementation of the MapHero Style Spec, defining what data to draw, the order to draw it in, and how to style the data when drawing it.

In addition to supporting styles loaded from a URL, MapHero Native includes a runtime styling API, which allows users to dynamically modify the current style: add and remove layers, modify layer properties, and so on. As appropriate for a C++ API, the runtime styling API API is *strongly typed*: there are subclasses for each layer type, with correctly-typed accessors for each style property. This results in a large API surface area. Fortunately, this is automated, by generating the API – and the regular portion of the implementation – from the style specification.

The layers API makes a distinction between public API and internal implementation using the `Impl` idiom seen elsewhere in the codebase. Here, it takes the form of parallel class hierarchies:

- `Layer`, `Source`, and their subclasses form the public API. This is the API consumed by SDK bindings.
- `Layer::Impl`, `Source::Impl`, and their subclasses form the internal API. This API is used only by other parts of the core C++ implementation.

For each subclass of `Layer` or source, there's a corresponding `Impl` subclass. For example, `CircleLayer` and `CircleLayer::Impl`. The base `Layer` class holds a reference to the base

`Layer::Impl`, and `CircleLayer` and other subclasses have an `impl()` accessor method that casts this reference to the appropriate subtype.

Immutability

The `Layer::Impl` and `Source::Impl` reference held by `Layer` and `Source` base classes is *immutable*: it's a shared reference to a `const` (read only) pointer. Immutability permits the `Impl` objects to be shared safely between threads when needed -- for example, between the main thread and worker thread that performs computation in the background, or between the main thread and a dedicated renderer thread. See the "Threading" section below for further details on the threading model.

Immutability is an alternative to several other strategies for safe intra-thread communication. One alternative strategy is to insert locks whenever data is shared between threads and at least one thread may be modifying the data. This strategy is prone to problems such as race conditions (if you forget to use a lock), deadlocks (if locks are acquired in conflicting orders), and poor performance due to lock contention. Another strategy is to copy data whenever it's needed by another thread. For complex structures such as a MapHero Style, copying can be an expensive operation. With immutability, the approach is to share data without copying, but ensure that any data so shared cannot be modified by any thread, and that the data is not destroyed until the last thread using it relinquishes its reference.

Immutability is implemented by the `Immutable<T>` template, which acts as a non-nullable shared reference to a `const T`. It has behavior similar to `std::shared_ptr<const T>`, but indicates its intent as an immutable reference that is safe to share between threads.

Immutability is core to the implementation, and yet one of the defining features of Mapbox GL is the ability to manipulate and mutate the style freely at runtime. How can this be possible if everything is immutable? The answer turns on the distinction mentioned in the previous section between public classes such as `Layer` and private implementations such as `Layer::Impl`. In Mapbox GL the following private implementations are immutable, whereas their public counterparts are not:

- **Mutable objects**: `Layer`, `Source`, `Image`, `Light`
- **Immutable objects**: `Layer::Impl`, `Source::Impl`, `Image::Impl`, `Light::Impl`

An instance of a `Layer` subclass such as `CircleLayer` has a reference to an `Immutable<Layer::Impl>`, but is itself mutable -- it has mutating methods such as `setCircleRadius`. Such methods follow a common pattern:

- Create a new instance of the Impl, copied from the existing Impl. This new instance is temporarily mutable.
- Modify this new instance as needed; e.g. set the radius to a new value.
- "Freeze" the new instance by making it immutable, and then assign that immutable reference to be the new Impl for the `Layer`.

Two things to note about this process:

- No existing Impls are modified -- only the newly created copy.
- Only one existing reference to an Impl is modified -- the one held by the `Layer` instance being mutated. Any references to the previous Impl held by other threads remain unchanged. They go

on using the previous value for radius until notified by some other means that there has been a change.

So how do things that are holding references to the previous Impl get notified? The answer is **style diffing**. This is the process by which we determine, from one frame to the next, what parts of the style have changed and therefore what needs to be recalculated in response to those changes in order to draw an updated frame. In parallel to style objects such as `Style`, `Layer`, `Source` and so on, Mapbox GL maintains render objects such as `RenderStyle`, `RenderLayer`, `RenderSource` and so on. These render objects:

- are mutable
- may live on either the main thread or an independent rendering thread, depending on the SDK
- contain any and all values calculated from the style objects plus state such as the current zoom level and location -- for example, the set of loaded tiles and the buckets calculated from them

From one frame to the next, these render objects are updated based on the current state of the style objects. In order to permit the render objects to live on a different thread, this state is communicated as immutable Impl references -- in effect, a snapshot of the style state at the time the frame was requested. And it really is a just snapshot -- we don't communicate things like "the radius of this circle layer was changed", "this layer was removed", etc. Instead, that information is reconstructed by `RenderStyle` by comparing the new snapshot to the old snapshot. Immutability allows us to perform this comparison very efficiently:

- If the "before" and "after" of two immutable references refer to the same object, we know that it hasn't changed. If they're different, we know it has changed in some way.
- We can calculate changes in collections of immutable references (a list of layers or sources) using an efficient diff algorithm. We can further improve this efficiency by making the collections themselves immutable, so that we can avoid running the diff algorithm altogether in the common case where the "before" and "after" collections refer to the same immutable object.

One final benefit of this approach of diffing immutable objects: we get "smart" updates between two completely independent styles essentially for free. For example, `RenderStyle` doesn't care if the "before" snapshot is from the Mapbox Light style and the "after" snapshot is from the Mapbox Dark style. It will just calculate the difference between the two snapshots, update render data where necessary, and render the next frame. Since those two styles use the same source data and layer IDs, the result will be a fully automatic, seamless transition between light and dark.

FileSource

Layout

Rendering

Annotations

Threading

At runtime, MapHero Native uses the following threads:

- The "main thread" (or other thread on which a `Map` object is created) handles direct `Map` API requests, owns the active `Style` and its associated objects, and renders the map. Since this thread is usually dispatching events triggered by user input, it's important that these duties not require significant computation or perform blocking I/O that would cause UI jank or hangs.
- Many of the tasks that require significant computation are associated with layout and styling of map features: parsing vector tiles, computing text layout, and generating data buffers to be consumed by OpenGL. This work happens on "worker threads" that are spawned by the main thread, four per `Style` object. The `Style` and its associated objects handle dispatching tasks to the workers, typically on a tile-by-tile basis.
- A "FileSource" thread handles network requests for styles, tiles, and other resources, and I/O on the SQLite database used for offline maps and caching. Requests originate from the main thread, are dispatched by a FileSource internally to its thread, and results are returned to the main thread via an asynchronous callback function. (This is implemented with platform-specific hooks into the native message pump for the requesting thread.)

To minimize data races and invalid memory access, we aim for zero shared memory state between any two threads. Cross-thread communication occurs via one thread requesting the "invocation" of a task on another thread. The parameters for this task should be passed as value objects, so that they are copied, or as ownership-transferring values such as `std::unique_ptr`. (We're not entirely there yet, but that's the goal.) The result (if any) is likewise passed as a value or ownership-transferring parameter to the callback function.

Invoking a task on another thread itself creates an ownership obligation: the responsibility for the work happening on the other thread. This ownership is represented by the invocation method returning `std::unique_ptr<AsyncTask>`. Destroying this object indicates that the result of the task is no longer required: the callback is guaranteed not to be called, and the work on the other thread may be aborted (if doing so is convenient).

All this is implemented by Thread and RunLoop.