Backend — Application Entry Point & Build System

Location: src-tauri/src/ and src/

main.rs — Binary Entry Point

The binary entry point is intentionally minimal. It simply calls `base_station_lib::run()`, which lives in `lib.rs`. The only thing of note is the `windows_subsystem = "windows"` attribute on the first line — this suppresses the extra console window that would otherwise appear when launching the app on Windows in release builds. Do not remove it.

lib.rs — Application Bootstrap

`lib.rs` is where the entire Tauri application is configured and started. It does the following in order:

Managed state registration

Three pieces of state are registered with Tauri's state manager so they can be injected into any command via `State<'_>`:

- `RoverState` — the three rover mode booleans (`drive_manual_mode`, `arm_manual_mode`, `pickup_mode`), all wrapped in `Mutex` so they are safe to read and write from async commands
- `DummyStreamHandle` — holds an optional cancellation flag for the dummy simulator
- `RoverAddress` — holds the port the Rover is sending to

> `RoverState` is subject to change, because the rover might become able to drive and move the arm at the same time

Plugin registration

Three official Tauri plugins are loaded:

Plugin	Purpose
`tauri-plugin-fs`	File system access from the frontend
`tauri-plugin-opener`	Open files/URLs in the OS default application
`tauri-plugin-dialog`	Native file picker and dialog boxes

All plugins must be registered in `lib.rs`, `Cargo.toml` and if they require access to anything in `src-tauri/capabilities/default.json`

> AI will often try to get you to add them to `tauri.conf.json` but that is, as far as I have encountered if, incorrect

Command registration

All Tauri commands are registered here via `tauri::generate_handler!`. This is the complete list of commands callable from the frontend via `invoke()`. If you add a new command in any `commands/` file, it must also be added here or it will not be accessible from the frontend.

Setup (startup sequence)

The `.setup()` closure runs once at launch, before any window is shown. It performs these steps in order:

a) GStreamer plugin path Sets the `GST_PLUGIN_PATH` environment variable so GStreamer can find its plugins on Windows. It will look for them at `C:\gstreamer\1.0\msvc_x86_64\bin`. On Linux it can find the plugins automatically.

b) Storage directory creation Calls `ensure_storage_dirs_internal()` to create the `tasks/`, `images/`, and `maps/` subdirectories under the app data directory if they don't already exist.

c) Cache clearing Calls `clear_cache_on_startup()` to wipe any stale cached files from the previous session.

d) GStreamer streaming server Spawns an async task that runs `commands::gstreamer::stream()` for the lifetime of the app. This starts the three GStreamer pipelines and their corresponding MJPEG HTTP servers.

e) UDP service Creates `UdpService` (binding `0.0.0.0:9000`) synchronously using `block_on`. The socket is extracted before the service is moved into Tauri's state manager, so it can be passed to the listener independently.

f) UDP listener Spawns an async task running `network::listener::run_listener()` with the shared socket. This is the loop that receives, decodes, and forwards all incoming rover packets to the frontend.

g) Controller listener Calls `commands::controller::start_controller_listener()`, which spawns an OS thread to poll for gamepad events.

proto.rs — Protobuf Module

This file simply includes the generated Rust code for the `packets` protobuf module:

rust

```
pub mod packets {
    include!(concat!(env!("OUT_DIR"), "/packets.rs"));
}
```

The actual `.proto` source files live in `src-tauri/proto/`. They are compiled at build time by `build.rs` into `packets.rs` in the Cargo `OUT_DIR`. Importing `crate::proto::packets::*` in any Rust file gives access to all the generated message structs.

build.rs — Protobuf Compilation

The build script runs before the Rust compiler and is responsible for compiling all `.proto` files into Rust code. It does this in several steps:

1. Collect proto files Recursively scans the `src-tauri/proto/` directory for `.proto` files. Only files that are inside a `components/` subdirectory are included — this is a deliberate filter to exclude top-level or organisational proto files.

2. Patch proto files Each `.proto` file is copied to inside `src-tauri/generated_proto` and patched to inject `package packets;` after the line `syntax = "proto3";`. This ensures all generated types end up in a single `packets` Rust module, regardless of how the source `.proto` files are organised. The patching is idempotent — it won't inject the package line twice if it is already present.

> This means that the basestation **protobufers are** slightly **different** from the ones embeded and jonny boi (the jestson) uses. Keep this in mind for debugging

3. Compile

- Using **prost** for backend. The patched files are compiled for backend (stored in `src-tauri/target/debug/build/base_station-0f99f7e026ffb091/out/packets.rs`) using `prost_build`. A type attribute is applied globally:

```
config.type_attribute(".", "#[derive(serde::Serialize)]");
```

- Using the plugin **protoc-gen-ts** for frontend. The patched files are compiled for frontend (stored in `src/lib/proto`)

This means every generated message struct and enum automatically derives `serde::Serialize`, so they can be passed directly as Tauri event payloads without any manual wrapper types.

4. protoc The `protoc` compiler binary is sourced from `protoc-bin-vendored`, so no system installation of `protoc` is required.

> If you add new `.proto` files, place them inside a `components/` subdirectory under `src-tauri/proto/` and they will be picked up automatically on the next build.

> If you add new protobufers you must explicitly commit and push them into the github submodule, they will **NOT** sync automatically when you sync the repo.

tauri.conf.json — Application & Security Configuration

Window

The app opens a single window titled `base_station` at 800×600. `devtools` is enabled, meaning the browser DevTools can be opened in development builds.

Content Security Policy

The CSP is configured to be strict by default while allowing the specific localhost ports needed for video:

Directive	What it allows
`default-src`	Only the app itself and Tauri's custom protocol
`script-src`	Self + inline scripts (required by SvelteKit)
`img-src`	App assets + the three MJPEG stream ports (5000, 5001, 5002)
`media-src`	The three MJPEG stream ports
`connect-src`	All localhost ports (for dev server, WebSockets) + `api.ipify.org`

The asset protocol is enabled with scope `$APPDATA/**`, which allows the frontend to read files from the app data directory (e.g. saved maps and images) using the `asset://` protocol.

> **For future developers:** Security gives a lot of problems (and they vary between devices and platforms), if something isn't loading always check if it is because of permissions. Thus far setting security to unrestricted does not fix those issues.

Bundled resources

The `models/*` glob in `bundle.resources` ensures all 3D model files in `src-tauri/models/` are included in the packaged application. This is what `load_model.rs` reads from in release builds.

Cargo.toml — Key Dependencies

Crate	Purpose
`tauri`	Desktop app framework, with `protocol-asset` and `devtools` features
`tokio` (full)	Async runtime for all network and I/O tasks
`prost`	Protobuf encode/decode
`gstreamer` / `gstreamer-app`	Video pipeline
`warp`	MJPEG HTTP server
`gilrs`	Gamepad/controller input
`reqwest` (blocking)	HTTP client used to capture video snapshots
`serde` / `serde_json`	Serialisation for Tauri events and commands
`anyhow`	Ergonomic error handling across async code
`dirs`	Cross-platform system directory paths (cache dir)
`chrono`	Date/time (available for timestamps)
`bytes`	Zero-copy byte buffer for GStreamer frame sharing
`once_cell`	Handling threads of dummy data
`rand`	For generating random numbers (for dummy data)
`socket2`	For creating a socket with custom options
`tobj`	For handling maps with .obj format
`las`	For handling maps with .las format
`nalgebra`	For handling the computing the height map