# RFC: bulletproof-lua

# RFC: bulletproof-lua

## Summary

We propose a standard structure for our Lua code, based on the ideas from bulletproof-react: code is grouped by what it is *for*, split into a few fixed layers, with clear rules about what may import what.

Two parts. **Features** get one folder each, with a fixed internal layout. **Domain** is a new layer below features that holds the site's entities — what a match is, what a tournament is — so features can read them without reaching into each other.

We start with one pilot feature (Squads), learn from it, and grow the structure from evidence rather than defining it all up front.

## What problem is this solving?

Today, the code for one feature is spread across the codebase. For Squads, the logic lives in `Squad/`, the display code in `Widget/Squad/`, and shared bits in a `Squad/Utils.lua` grab-bag that mixes parsing, storage, and display concerns. There is also no rule about what may import what: display code imports business logic, and modules reach into the internals of other features.

This causes real problems:

- **Hard to find things.** A contributor who wants to change the squad table has to know to look in three different places.
- **Hard to change things.** Because everything imports everything, a small change can break code far away. People (reasonably) become afraid to refactor.
- **Hard to test things.** Logic that is tangled with database calls and display code cannot be unit tested on its own.
- **Quality drifts apart.** A review of our six major features showed a clear pattern: the features that already follow a layered structure (Standings, TeamParticipants) have the highest code quality and are the most testable; the ones that don't (Squads, PrizePool) carry the most debt.

**And features have nowhere to get shared data from.** Many features need to read matches or tournament data. With no layer that owns those things, each one improvises:

- 11 files outside the match code query `match2` records straight from the database, bypassing the match model entirely. `PrizePool/Import` and

`Standings/Parse/Lpdb` do both — they use the model *and* hand-roll their own queries.
- 14 places still read raw `tournament_*` page variables by string key, even though a shared reader already exists.
- `Squad/Auto` hand-rolls a 13-field copy of the transfer record shape, which silently drifts from the real thing.

These look like different bugs. They are one missing layer.

## Background

Our codebase is organized half by feature (`Squad/`, `PrizePool/`) and half by kind of code (`Widget/`), but sometimes feature code is unorganized entirely (Match2). Recently rebuilt features (Standings, TeamParticipants) already follow an informal layered style — parse, logic, storage, and display in separate modules — and they turned out to be the easiest to test and maintain. This RFC turns that informal style into an explicit, shared convention.

bulletproof-react is a widely used open-source guide for structuring React projects. We are not adopting React — we are adopting its structural ideas, which are not tied to any framework: group code by feature, give each feature a public entry point, keep imports flowing in one direction, and avoid "utils" dumping grounds.

We add one thing bulletproof-react does not prescribe: a **domain layer**. That is expected for Liquipedia Lua, because our modules are not pure frontend the way a React app is — they also own the site's data model. Without a domain layer, "group by feature" pushes features into importing each other, which is the tangle we already have.

### Authors

- Rikard Blixt

### Audience

Everyone who writes or reviews Lua modules for Liquipedia — staff developers and volunteer contributors alike. No prior knowledge of bulletproof-react is needed; this document explains everything used from it.

### Glossary

- **Feature** — one user-facing capability and all its code, e.g. Squads, PrizePool.
- **Domain** — the layer holding the site's entities and their rules: what a match is, what a tournament is. No display, no wiki-specific behaviour.
- **Entity** — a thing the whole site talks about: match, tournament, opponent, team, person, placement.

- **Layer** — a group of modules with one job (e.g. "storage" or "display").
- **Entry point** — the one module of a feature (`Custom.lua`) that per-wiki code and templates are allowed to import. Other features never import it.
- **Import direction** — which layer may `Lua.import` which. "Downward only" means a layer may only import layers below it, never above.
- **Adapter** — a small module that translates old input formats to the new code, kept separate so the main code stays clean.

# Overview

## Three layers

```
None
features    user-facing capabilities              Bracket, PrizePool, Squads,
Infobox
domain      the site's entities and their rules   Match, Tournament, Opponent,
Squad
shared      generic tools, no Liquipedia          Array, Table, Logic, Date
```

**One rule: imports only point down.** `features → domain → shared`. Which means:

- A feature may import domain and shared.
- Domain may import shared, and other domain entities in a declared order. Never a feature.
- Shared imports nothing but shared.

Only `domain` is a new place — `shared` is mostly where things already are, so this is not a big move of existing files.

Per-wiki code is not a layer. It stays as it is today: a parallel tree (`lua/wikis/<wiki>/`) that plugs into each feature's `Custom` slot, plus per-wiki `Info.lua` config. Per-wiki code may only be accessed by features, not by domain or shared.

## Feature folder layout

Every feature gets one folder with the same layout:

```
None
<Feature>/
├── Custom.lua        entry point (overridable) — the ONLY module outsiders import
├── Controller.lua    orchestration: parse → derive → store → render
```

```
├── Components/       display (widgets); knows nothing about parsing or storage
├── Api/              this feature's own reads/writes: LPDB queries, persistence
├── Lib/              pure logic for this feature; no I/O, no widgets
├── Types.lua         enums, maps, type definitions; data only
└── LegacyAdapter.lua old-format support (deleted when no longer needed)
```

With two rules:

1. **Imports only point down** the list: `Custom → Controller → {Components, Api, Lib} → Types → domain → shared`. Never up, never sideways into another feature's internals.
2. **Outside code imports only `Custom`** (the entry point) — "outside code" here means everything that is not within the same feature folder.

We pilot this on **Squads** (smallest feature, clearest problems, only 4 wikis with customizations), then split `MatchGroup/Util` along the domain line, then define `domain/` from what those two produced.

### Non-goals

- **No rewrites.** We move and split code; we do not change what it does.
- **No visible changes.** Readers and editors see the same output; templates keep the same parameters.
- **No new widget system.** Components keep using the existing Widget system.
- **Not touching shared libraries** (`Array`, `Table`, `Class`, …) — they stay where they are until the pattern is proven. "Shared" describes them; it is not a relocation.
- **Not designing `domain/` up front.** We do not write a full entity catalogue and then fill it in. Two real extractions define the layer; the rest follows as needed.
- **Not restructuring every feature now.** Additional features are out of scope until the pattern is proven.

## Detailed design/discussion

**Why these layers?** Each layer answers one question:

- `Types` — *what are the concepts?* Enums like squad status, display-name maps. Data only, and internal to the feature — if another feature needs a type, it belongs in domain.
- `Lib` — *what are the rules?* Pure functions: given input, return output. No database, no HTML. This is where the valuable logic lives (e.g. turning a player's transfer history into squad stints), and pure functions are the easiest thing to unit test.

- `Api` — *where does data come from and go?* Every LPDB query and write, every cache. Keeping I/O in one layer means the rest of the feature can be tested without a database.
- `Components` — *what does it look like?* Widgets that receive ready-made data and render it. They never compute, query, or store.
- `Controller` — *in what order does it happen?* A thin module that calls the layers: parse input (Lib), fetch data (Api), compute (Lib), store (Api), render (Components).
- `Custom` — *how does this wiki differ?* Per-wiki wiring and overrides, at the top where they can see everything, instead of buried inside the core.
- `domain` — *what is a match / tournament / placement?* Shared by every feature that displays them.

**Why one entry point?** Today `Squad/Utils` is imported from seven different places — including `Widget/EmptyPagePreview/Team.lua`, which is a different feature reaching straight into Squad's internals. That means those internals are frozen: any change risks breaking an unknown consumer. When outsiders may only import `Custom`, the inside of a feature becomes safe to improve.

**Rules so `domain/` does not become the next grab-bag.** This is the real risk — a domain layer that accretes everything is worse than what we have now:

1. **One entity per module.** If you cannot name the entity, it is not domain.
2. **No rendering.** No widgets, no HTML, no `mw.html`.
3. **No global config reads.** Domain functions take per-game rules as *parameters* rather than reaching for `Info.config`. This is what makes them unit-testable.

**Alternatives considered:**

- *Do nothing.* The quality gap between our layered and unlayered features keeps growing.
- *Feature folders only, no domain layer.* This was the first draft of this RFC. It fails on the cases above: with no domain layer, either features import each other, or every feature that owns data has to publish an API for the others — and we found no consistent rule for which. The 11 raw `match2` queries are what "no domain layer" looks like in practice.
- *Put shared data code in the shared libraries.* Then the shared layer starts knowing about brackets and tournaments, and everything that uses shared drags that along. Shared is for tools that do not know what Liquipedia is.
- *Organize by kind* (all widgets together, all storage together). This is what `Widget/` does today; it optimizes for the file tree looking tidy, not for the way people actually work, which is "I want to change the squad table" — one feature, all layers.
- *Full rewrites.* Highest risk, longest time without shippable results. Moving code in small steps keeps every commit releasable.

**Enforcement.** Rules that live only in reviewers' heads decay. We add a small CI check that reads `Lua.import(...)` calls and fails a PR that (a) imports a feature's internals from outside, (b) imports upward within a feature, (c) imports a feature from domain or shared, or (d) imports one feature from another — including its entry point. Lua has no ready-made tool for this, but a short script over the repo is enough.

## Integration with existing features

- **Templates are unaffected.** On-wiki templates invoke entry points (`Squad/Custom`, `PrizePool/Custom`), which keep their names and behavior.
- **Features that read matches.** Standings, PrizePool, TournamentStructure, GameSummary, MvpTable, NextMatch and Ratings all need match data today, and get it inconsistently. After the `MatchGroup/Util` split they read `domain/Match` — the match model and bracket rules move there, while bracket drawing stays in the bracket feature.
- **Page variables.** Features also communicate through wiki page variables (e.g. the infobox publishes tournament data that PrizePool reads). A shared reader already exists (`Tournament.partialTournamentFromContext`) but is only partly adopted. It becomes `domain/Tournament`, gains the fields people still read raw (`startdate`, `mode`), and the raw reads are deleted.
- **Existing shared entities.** `Opponent`, `TeamTemplate` and `Tournament` are already domain entities living at the top level. They move into `domain/` as-is when convenient; no behaviour change.

## Open questions

1. **Naming:** `Lib` (feature-internal pure logic) and `domain` (shared entities) are now different things, so this is no longer one-or-the-other.
2. **How strict on day one?** Do we block PRs on the CI import check immediately, or run it as a warning for a transition period? **Blocklist (or just a folder check if part of the new folder structure)**
3. **The `Widget/` tree long-term:** feature components move into feature folders — do truly shared widgets (tables, icons) stay in `Widget/`, and where is the line? **Components folder on top level**
4. **Where does the domain line fall for brackets?** Bracket *topology* (how losers drop, how placements resolve) is domain by the test above. Bracket *coordinates* are used both for drawing and by PrizePool's placement import.

## Rollout plan

Small steps, each independently releasable, nothing moves until tests cover it. Note that `domain/` is created in Phase 2 — *after* two real extractions show what belongs in it.

- **Phase 0 — agree.** This RFC is discussed and accepted; the structure guide is added to the docs.
- **Phase 1 — two samples.**
  - *Squads pilot:*
    1. Write the test. Today's squad spec is 57 lines plus golden snapshots - not enough to protect an extraction this big.
    2. Extract `Squad/Types` (removes the widgets→Utils dependency).
    3. Split `Squad/Utils` into `Lib/Parse`, `Lib/Columns`, `Api/Store`.
    4. Extract the transfer-history logic from `Squad/Auto` (706 lines — 47% of the whole feature) into `Lib/History` + `Api/TransferHistory`. Expect this to turn out domain-shaped: it answers "was this person on the team, and when".
    5. Move `Widget/Squad/*` into `Squad/Components/*`.
  - *`MatchGroup/Util` split:* separate the match model (reading records into matches, opponents, games, match ids) from bracket drawing. Low risk, the extracted functions are pure.
- **Phase 2 — create `domain/`.** Move those two extractions in. The layer is now defined by real content, not speculation. Add the CI import check in **warn-only** mode, so we get signal without blocking on a convention we may still adjust.
- **Phase 3 — evaluate.** Did the pilot stay shippable at every step? Did contributors find the new layout easier? Adjust the convention if needed, then switch the CI check from warning to blocking.
- **Phase 4 — migrate consumers opportunistically**, against the finish lines in the testing plan below.
- **Phase 5 — PrizePool**, which by then is largely deleting code that domain owns. Then other features when they need significant work anyway.

Squads is the right pilot because it is small (~1,500 lines), shows every problem the structure fixes, and has only 4 per-wiki customizations.

## Testing plan

- **Before moving anything:** characterization tests. Current coverage on Squads is thin, so writing this safety net is the first real task. Extend the spec and add one for the transfer-history logic.
- **After extraction:** the pure `Lib/` and domain modules get real unit tests (busted specs) — this is the payoff; today that logic is untestable because it is tangled with LPDB calls. Target: the history state machine, column logic, and the match model fully covered. The match code is our least-tested major feature, and its model is what every other feature depends on.
- **Measurable finish lines.** These are countable today, so progress is not a matter of opinion:

    - Raw `lpdb('match2')` queries outside the match domain. Target: **0**
    - Raw `varDefault('tournament_*')` reads outside the tournament domain. Target: **0**
    - Hand-rolled copies of another entity's record shape. Target: **0**.
- **Continuously:** the CI import check keeps the structure from decaying, and the existing full test suite runs on every PR as usual.

## Documentation plan

- This RFC records the decision and reasoning.
- A short **"How to structure a feature"** guide in `docs/`, written for contributors: the three layers, the folder layout, the import rules, the domain-or-feature question, and a worked example.
- Per-feature architecture notes as features are restructured (current state → target → migration).
- `CONTRIBUTING.md` gets a pointer to the structure guide so new volunteers see it first.