hun.sh

Product Specification

Seamless Project Context Switching for Developers

Version 1.1 | February 2026

Table of Contents

1. Executive Summary

hun.sh is a command-line tool designed to eliminate the friction of context switching between development projects. It provides developers with a simple,

intuitive way to switch between projects while automatically managing running processes, capturing logs, and preserving state.

Unlike terminal multiplexers like tmux, hun.sh operates at the **project level** rather than the terminal level. It understands what services a project needs, manages their lifecycle, and provides instant access to logs from anywhere.

Key Value Propositions

- **One Command Switching** — Switch entire development contexts with a single command
- **Zero Port Conflicts** — Automatic process management ensures clean transitions
- **Instant Log Access** — View logs from any project, anywhere, anytime
- **State Preservation** — Remember where you left off in each project
- **Zero Config Start** — Auto-detects project structure and infers services

---

2. Problem Statement

The Daily Reality

Modern developers frequently work across multiple projects throughout a single day. Each project typically requires multiple running processes: frontend dev servers, backend APIs, databases, and supporting services. The current workflow for switching between projects is manual, error-prone, and cognitively expensive.

Current Pain Points

Pain Point	Description
Manual Process Management	Developers must manually stop each service, navigate to a new project directory, and start new services one by one
Port Conflicts	Forgetting to stop a service leads to port conflicts that require debugging and process hunting
Lost Context	When returning to a project, developers must remember what branch they were on, what they were working on, and what services need to run

Pain Point	Description
Scattered Logs	Logs are spread across multiple terminal windows or tabs, making debugging difficult
Cognitive Load	Remembering project-specific commands, ports, and configurations adds unnecessary mental overhead

What Existing Tools Miss

Tool	What It Solves	What It Doesn't Solve
tmux	Terminal session persistence	Project-level orchestration
Docker Compose	Container management	Adds overhead for simple projects
Make / npm scripts	Task running within a project	Cross-project switching
direnv	Environment variables	Process management

There's a gap for a tool that thinks at the project level.

---

3. Product Vision

Core Philosophy

> **"One project at a time, by default. Multiple when you need it. Always in control."**

hun.sh is built on the principle that most developers work on one project at a time, even if they switch between projects frequently throughout the day. The tool optimizes for this reality with **Focus Mode** as the default, while providing a powerful **Multitask Mode** for when parallel execution is needed.

The TUI is the command center. You run `hun` and you're in. No more switching between terminal tabs. This is your cockpit.

Design Principles

Principle	Description
Speed	Switching should take less than 2 seconds. Any slower and developers will work around the tool

Principle	Description
Reliability	Processes must actually stop. Ports must actually free up. No orphaned processes
Visibility	Clear, instant feedback on what's happening. No silent failures
Simplicity	The TUI and five CLI commands cover 95% of use cases. Advanced features are opt-in
Zero Config Default	Works out of the box with sensible defaults. Configuration is optional enhancement
Logs are the star	The TUI maximizes log visibility. Everything else gets out of the way

Two Modes of Operation

Mode	Philosophy	Behavior
Focus Mode (default)	One project at a time, fully	Switching stops the current project and starts the new one. Clean, distraction-free
Multitask Mode	Orchestrate multiple projects	Projects run in parallel with automatic port offsetting. The TUI becomes a multi-project command center

Target Users

Primary: Full-stack developers working on 2-5 active projects who use modern tech stacks (Node.js, Python, Go, etc.) and value clean development environments.

Secondary: Engineering teams who want consistent development environment management across team members.

---

4. Core Concepts

Project

A project is a directory containing a `.hun.yml` configuration file (or auto-detected project markers like `package.json`, `docker-compose.yml`, etc.). Each project has a **unique name** and defines the services required to run it.

Project names must be unique across the registry. If a name collision is detected during `hun init` or `hun add`, the user is prompted to pick a different name. Names can also be set explicitly with `hun init --name <name>`.

Project States

State	Description
Running	Project has services up. Shown in the TUI top bar with a dot indicator
Focused	The running project you're currently looking at in the TUI. Indicated by `●` on the left. Purely a view concept — has no effect on processes
Stopped	Not running. Accessible via the project picker (`[p]`) to start
Known	Registered with hun but not running. Appears in the project picker

Service

A service is a long-running process within a project. Examples include frontend dev servers, backend APIs, databases, and worker processes. Each service has a command, optional port, working directory, and environment variables.

Switch (Focus Mode)

The **primary operation** in Focus Mode. Switching:

1. Stops all services of the current project
2. Starts all services of the target project
3. Changes the working directory
4. Restores previous state

This ensures only one project runs at a time, eliminating port conflicts.

Run (Multitask Mode)

The **primary operation** in Multitask Mode. Running starts a project alongside existing ones with automatic port offset detection. Port offset increments by 1 per project (e.g., project 1 on :3000, project 2 on :3001, project 3 on :3002).

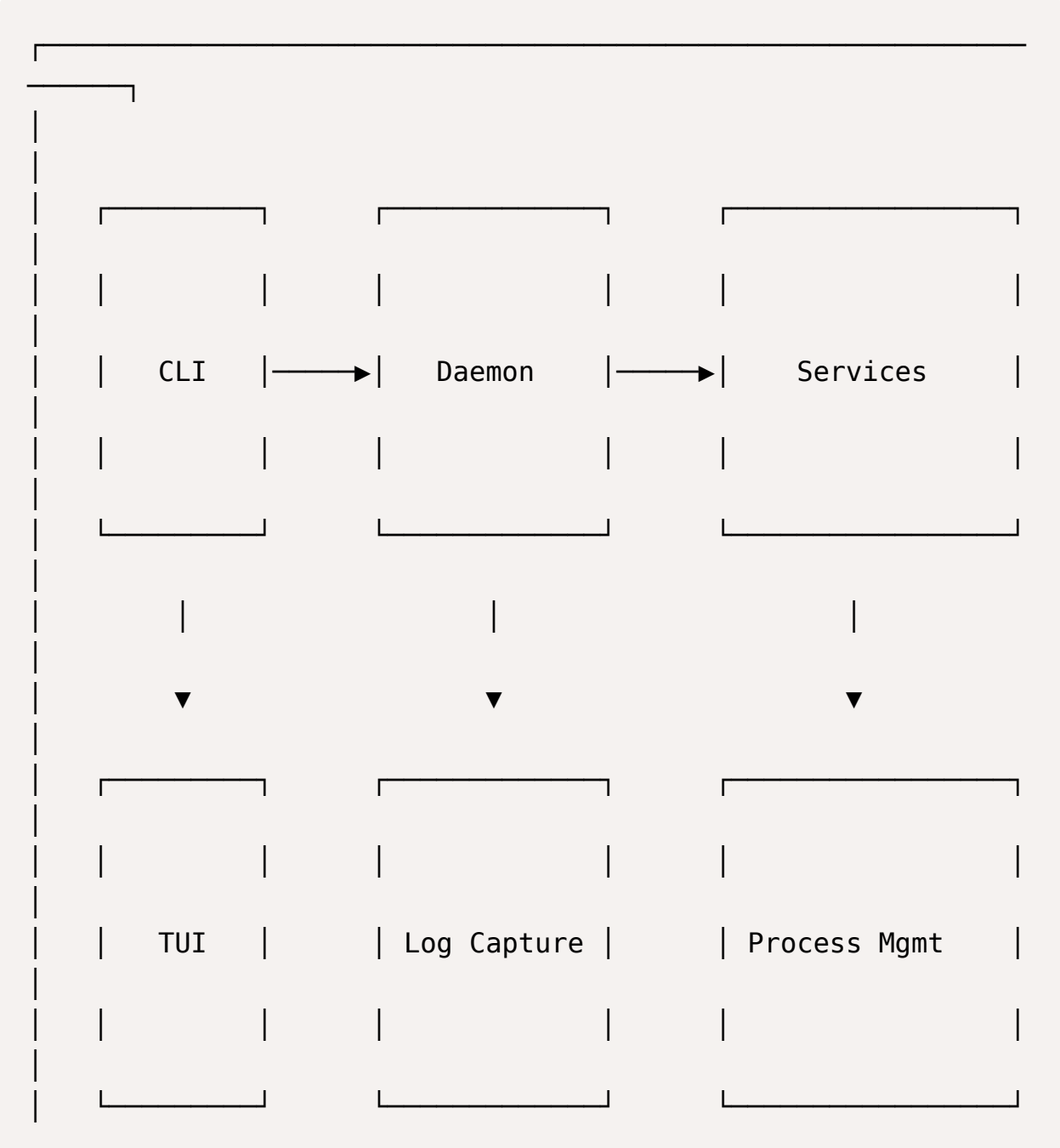
Daemon

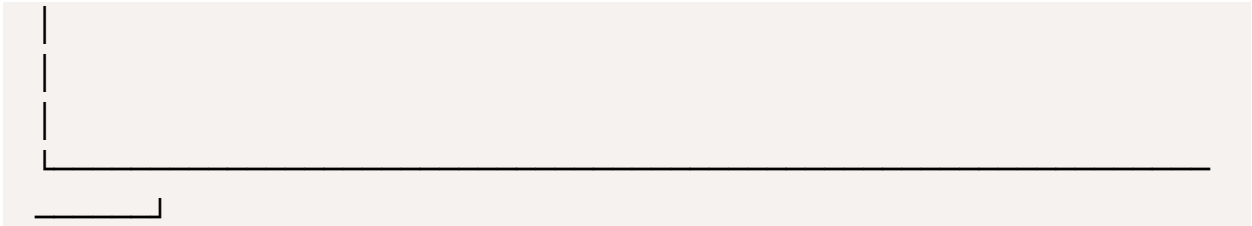
A background process that manages all running services, captures logs (stdout/stderr), exposes an API for the CLI and TUI via Unix socket, and persists across terminal sessions

---

5. Architecture

System Overview





Component Details

Component	Description
CLI	User-facing command interface. Lightweight, fast startup. Communicates with daemon via Unix socket
Daemon	Long-running background process. Manages service lifecycle, captures stdout/stderr, maintains state
TUI	Terminal user interface for log viewing. Built with Bubble Tea. Connects to daemon for real-time logs
Config Parser	Reads `.hun.yml` files and auto-detects project structure when no config exists
State Manager	Persists project state (git branch, notes, last active time) for restoration on switch

File Locations

Path	Purpose
`~/.hun/`	Global hun.sh directory
`~/.hun/config.yml`	Global configuration (hotkeys, defaults)
`~/.hun/state.json`	Active projects, saved states
`~/.hun/daemon.sock`	Unix socket for CLI-daemon communication
`~/.hun/logs/`	Stored log files per project
`<project>/.hun.yml`	Project-specific configuration

6. Command Reference

TUI (Primary Interface)

```
hun                          # Open TUI in Focus Mode (defaul
t)
hun --multi                   # Open TUI in Multitask Mode
```

The TUI is the primary interface. It persists mode between sessions.

CLI Commands (Headless / Agent-Friendly)

These commands work without the TUI, making them suitable for scripts, CI/CD, and AI agents.

Process Management:

```
hun switch <project>         # Focus mode: stop all, start on
e
hun run <project>            # Multitask: start alongside oth
ers (port offset)
hun stop <project>           # Stop specific project
hun stop --all               # Stop all running projects
hun restart <project>:<svc>  # Restart one service in a proje
ct
```

Project Management:

```
hun init                     # Initialize current directory a
s project
hun init --name <name>       # Initialize with explicit name
(avoids collisions)
hun list                     # List all known projects
hun add <path>               # Register an existing project
hun remove <project>         # Unregister a project (doesn't
delete files)
```

Info & Logs:

```
hun status                    # List running projects + servic
es (table output)
hun ports                     # Show port map for all running
services
hun logs <project>:<service> # Dump logs to stdout (pipe-frie
ndly)
hun tail <project>:<service> # Stream logs to stdout (tail -f
style)
hun open <service>            # Open service URL in browser
hun doctor                    # Diagnose common issues
```

CLI vs TUI Behavior

Action	CLI	TUI Focus Mode	TUI Multitask Mode
Start a project	`hun switch` (exclusive) or `hun run` (parallel)	Picker → enter = `hun switch`	Picker → enter = `hun run`
Stop a project	`hun stop <project>`	N/A (switch replaces)	`[s]` stops focused project
Restart service	`hun restart proj:svc`	`[r]` on selected service	`[r]` on selected service
Restart project	`hun stop` • `hun switch`	`[R]` (Shift+R)	`[R]` (Shift+R)

7. Configuration

Project Configuration (.hun.yml)

```
name: hun-sh

services:
  frontend:
    cmd: npm run dev
    cwd: ./frontend
    port: 3000
```

```
    port_env: PORT
    ready: "compiled successfully"

  backend:
    cmd: python main.py
    cwd: ./backend
    port: 8000
    port_env: API_PORT
    env:
      DATABASE_URL: postgres://localhost:5432/hunsh
      DEBUG: "true"
    depends_on:
      - db

  db:
    cmd: docker compose up postgres
    ready: "database system is ready"

hooks:
  pre_start: ./scripts/setup.sh
  post_stop: ./scripts/cleanup.sh

logs:
  max_size: 10MB            # max size per service log file
  max_files: 3              # number of rotated files to kee
p
  retention: 7d             # auto-purge logs older than thi
s
```

Configuration Fields

Field	Description
`name`	Project identifier (used in commands)
`services`	Map of service name to service configuration
`cmd`	Command to run the service

Field	Description
`cwd`	Working directory (relative to project root)
`port`	Primary port the service listens on
`port_env`	Environment variable that controls the port
`ready`	Log pattern indicating service is ready
`env`	Environment variables for the service
`depends_on`	Services that must start first
`hooks`	Scripts to run at lifecycle events
`logs.max_size`	Maximum size per service log file before rotation (default: 10MB)
`logs.max_files`	Number of rotated log files to keep per service (default: 3)
`logs.retention`	Auto-purge log files older than this duration (default: 7d)

Global Configuration (~/.hun/config.yml)

```
# Default behavior
defaults:
  auto_cd: true              # cd to project directory on swi
tch
  show_logs_on_switch: true  # open logs TUI after switching

# Project discovery
scan_dirs:
  - ~/code
  - ~/projects
  - ~/work

# Port management
ports:
  default_offset: 1          # offset per parallel project (+
1, +2, etc.)

# Hotkeys (requires system integration)
hotkeys:
```

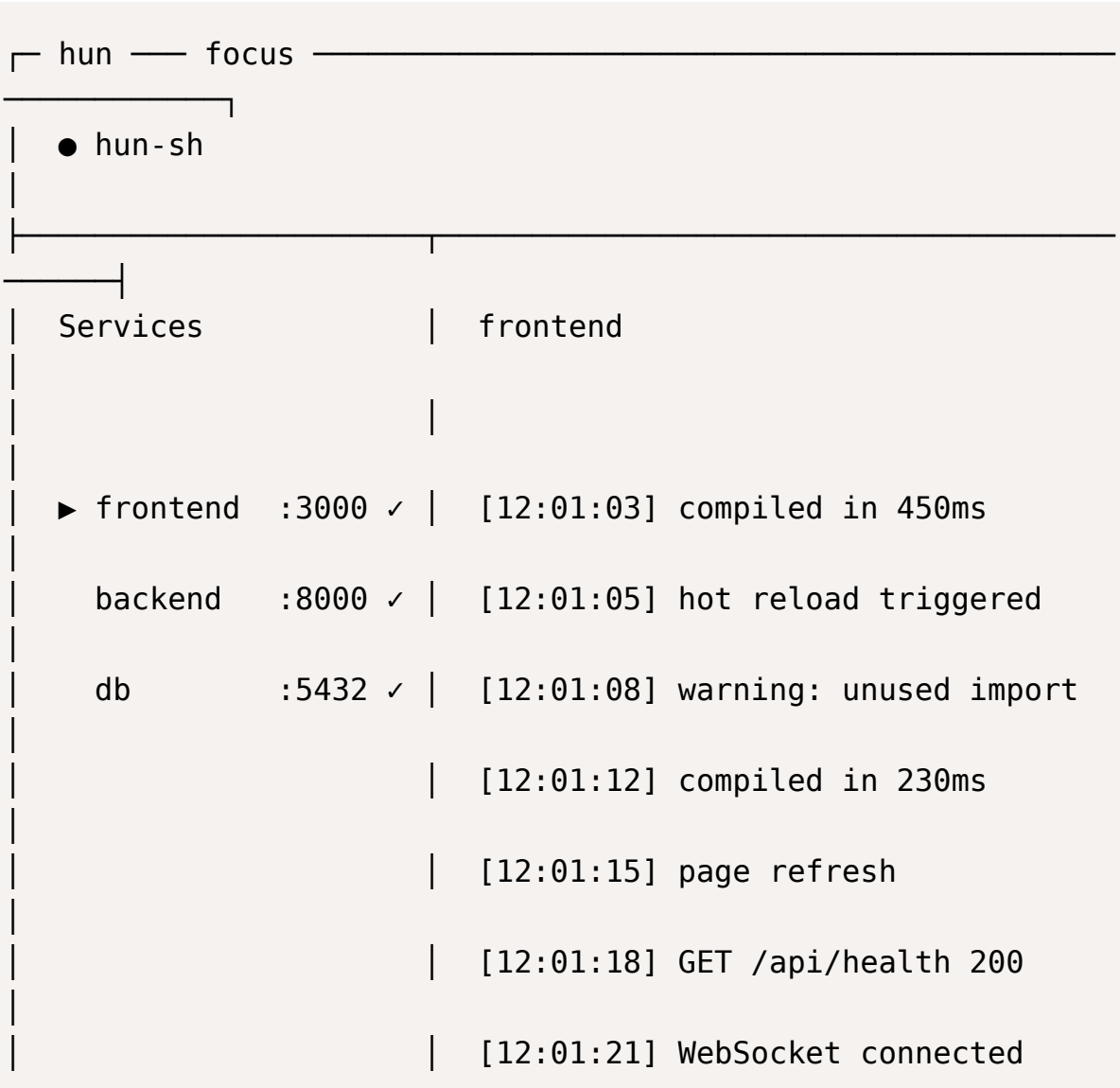
```
  peek: "cmd+shift+l"
  switch: "cmd+shift+p"
```

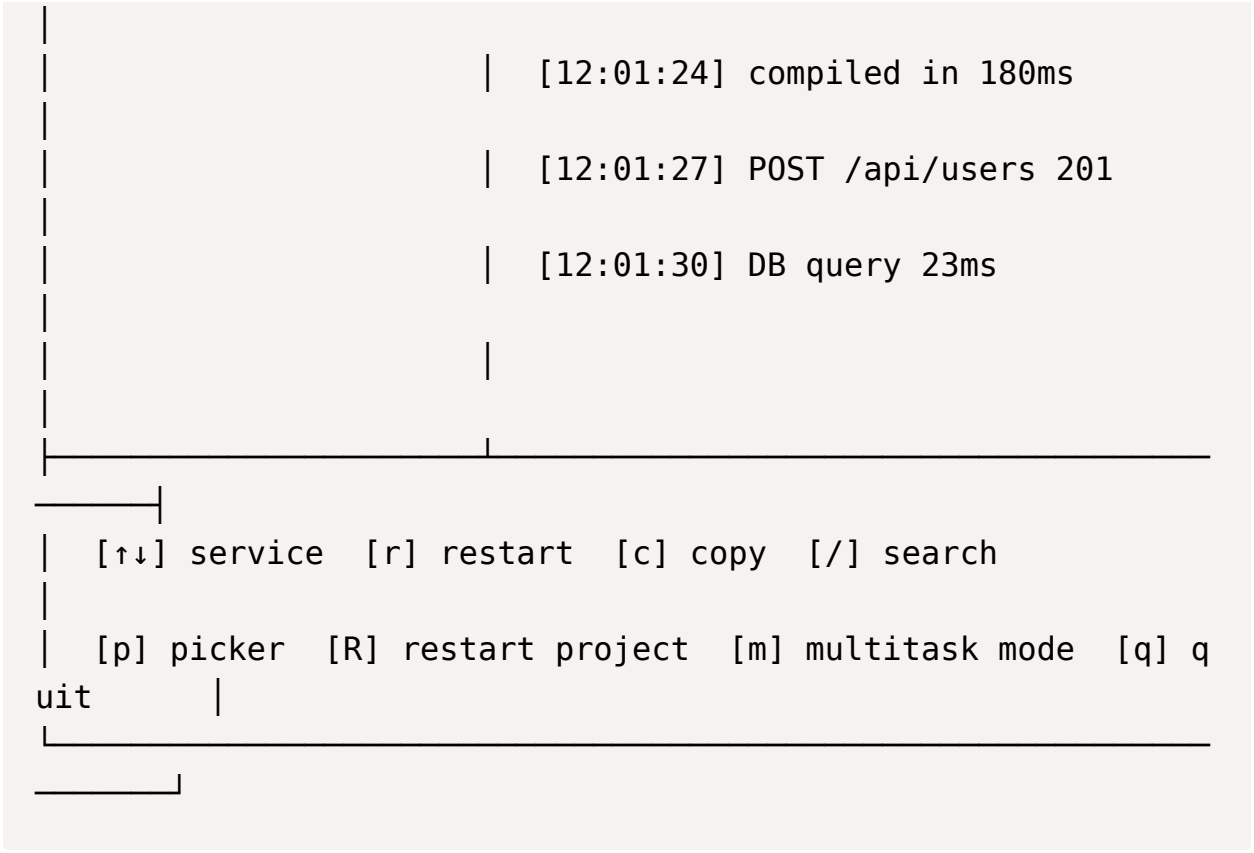
8. User Interface (TUI)

The TUI is the primary interface for hun.sh. It operates in two modes: **Focus Mode** for single-project work and **Multitask Mode** for parallel project orchestration. The TUI connects to the daemon via Unix socket and receives real-time log updates.

Focus Mode (`hun`)

Single project. Clean. Distraction-free. Only one project runs at a time.

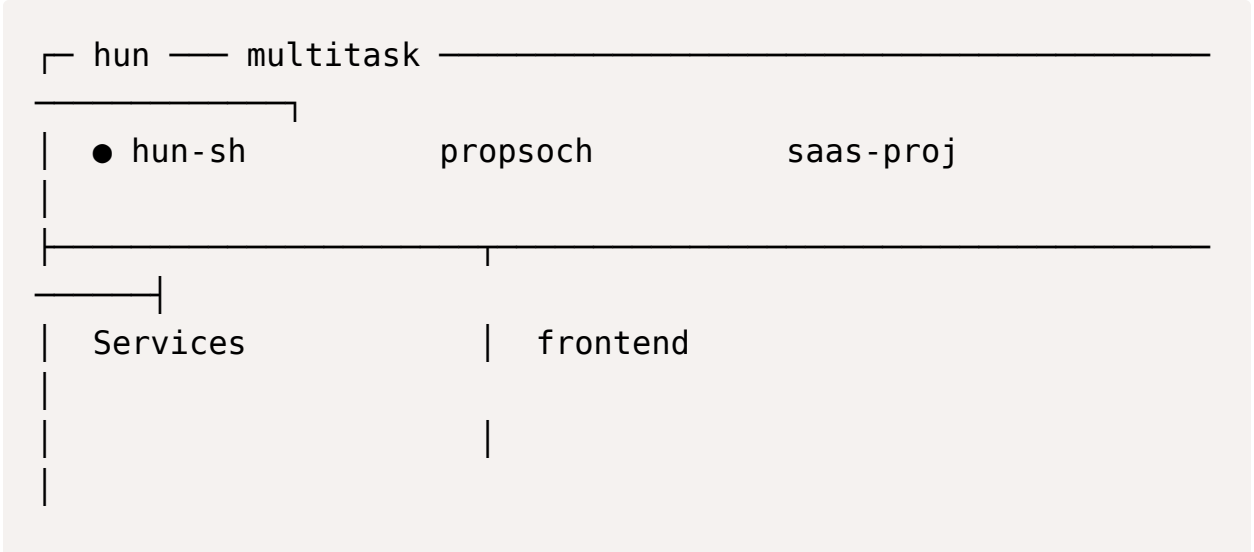


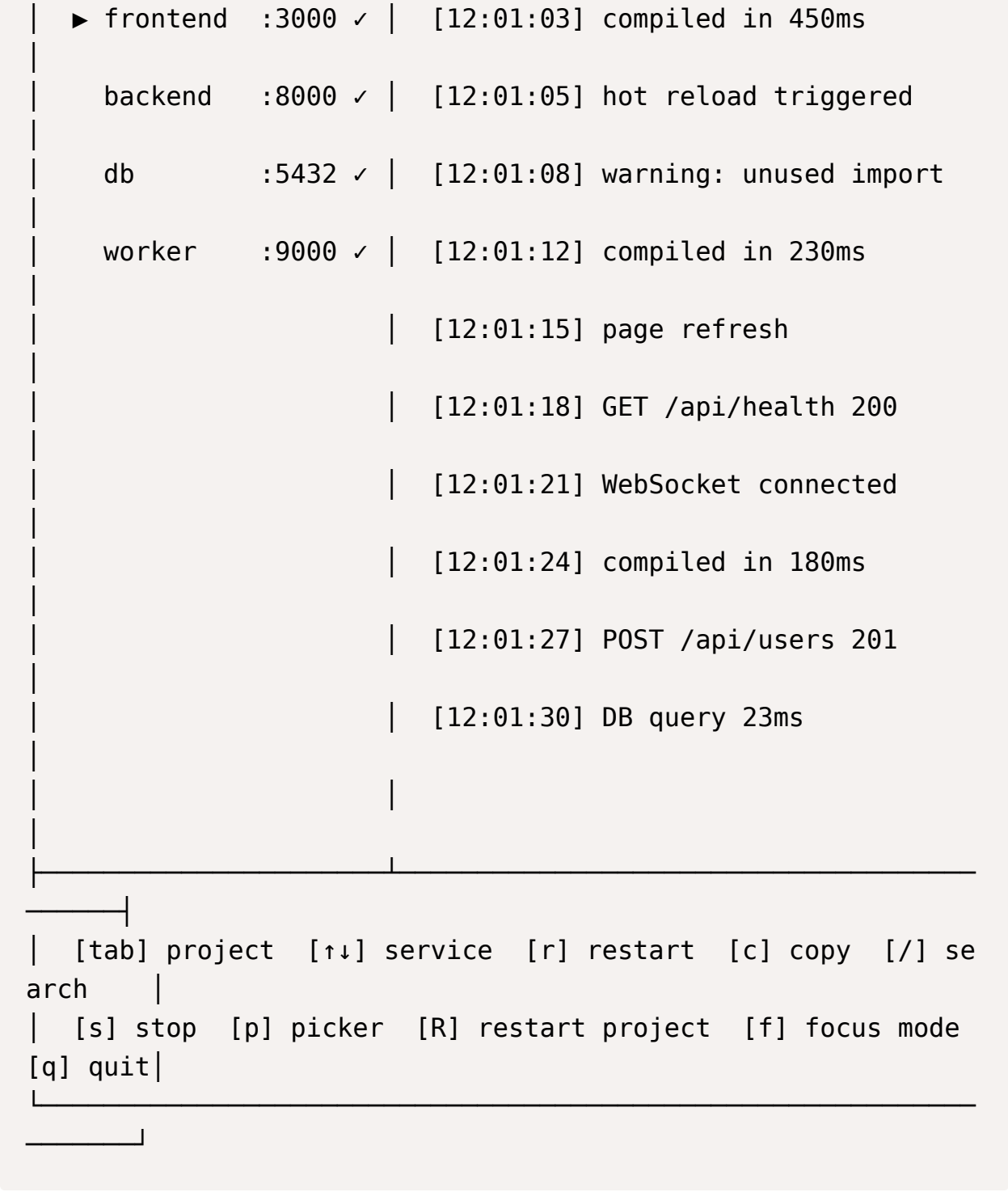


Focus Mode behavior: Top bar shows only the single running project. `[p]` picker uses `hun switch` under the hood — stops current, starts new. No `[tab]` since there's nothing to tab to. `[m]` switches to Multitask Mode.

Multitask Mode (`hun --multi`)

Multiple projects running simultaneously. The orchestrator view.



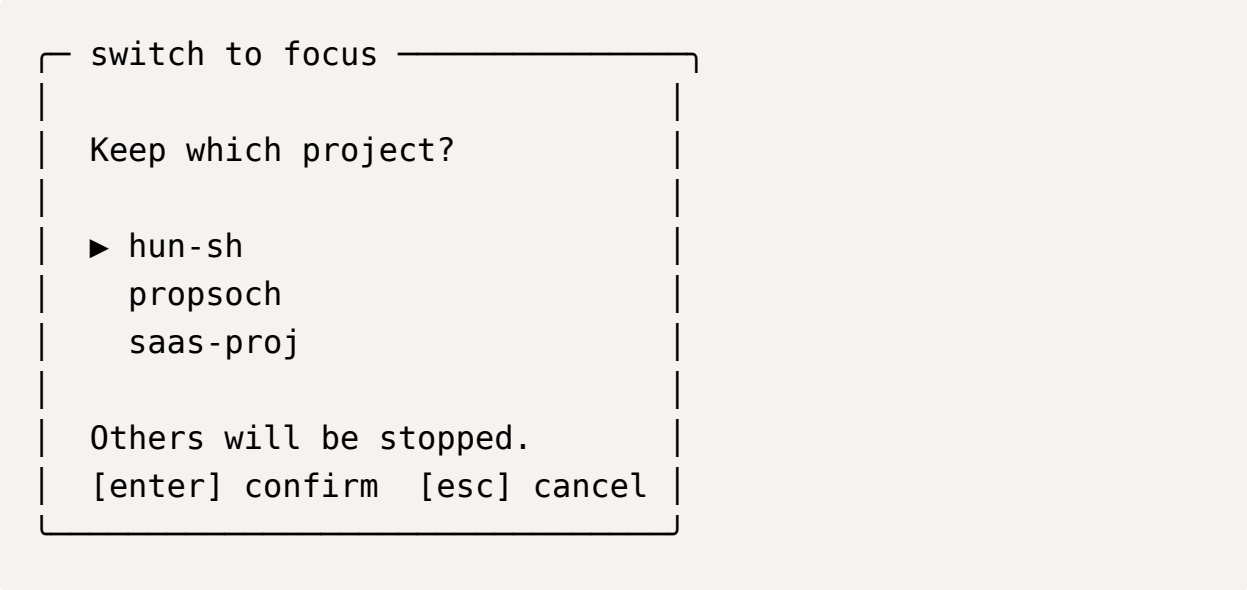


Multitask Mode behavior: Top bar shows all running projects. `●` dot on the left indicates which is focused. `[tab]` cycles focus between running projects (services and logs update). `[p]` picker uses `hun run` under the hood — starts alongside, no stopping. `[s]` stops the focused project and removes it from the top bar. `[f]` switches to Focus Mode.

Mode Switching

Focus → Multitask (`[m]`): Instant. Changes the mode flag. If only one project is running, nothing visible changes until a second is started from the picker.

Multitask → Focus (`[f]`): If multiple projects are running, a prompt asks which to keep:



If only one project is running, switches instantly. Mode persists across TUI sessions in `~/.hun/state.json`.

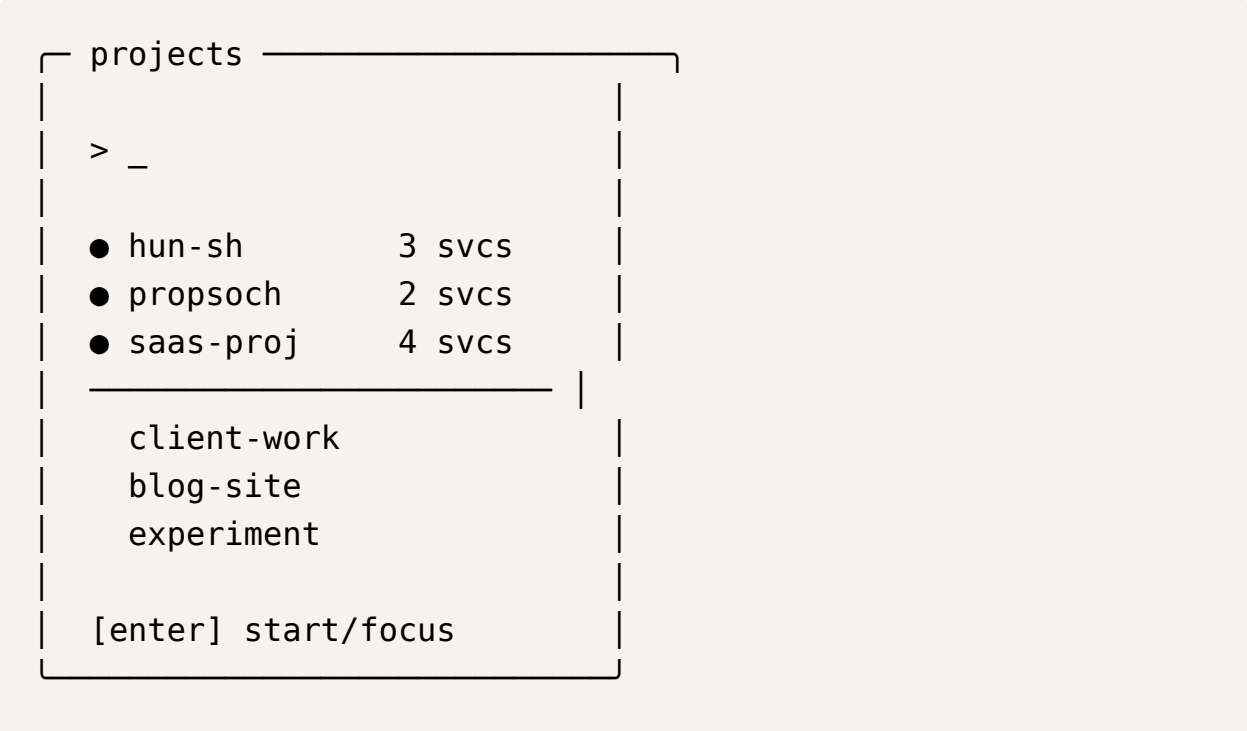
TUI Keybindings

Key	Action	Focus Mode	Multitask Mode
`↑↓`	Select service	✓	✓
`tab`	Cycle focus between running projects	—	✓
`r`	Restart focused service	✓	✓
`R` (Shift+R)	Restart focused project (all services)	✓	✓
`c`	Enter copy mode on logs	✓	✓
`/`	Search / filter logs	✓	✓
`a`	Show combined logs from all services	✓	✓
`p`	Open project picker (fuzzy search)	✓ (switch)	✓ (run)

Key	Action	Focus Mode	Multitask Mode
`s`	Stop focused project	—	✓
`m`	Switch to Multitask Mode	✓	—
`f`	Switch to Focus Mode	—	✓
`q`	Quit TUI (services keep running via daemon)	✓	✓

Project Picker Overlay (`[p]`)

A floating fuzzy search overlay for starting or focusing projects. Appears over the TUI when `[p]` is pressed.



Picker behavior: Running projects shown at top with `●` dot, separated from stopped/known projects below. Fuzzy search filters both sections. `enter` on a running project switches focus to it. `enter` on a stopped project starts it (in Focus Mode: `hun switch`, in Multitask Mode: `hun run`). `esc` dismisses.

---

9. User Journey & Examples

Example 1: First-Time Setup

```
$ cd ~/code/hun-sh
$ hun init
```

```
┌─ hun init ──────────────────────────────────────────────────────────────┐
│                                                                         │
│  Detected project structure:                                            │
│                                                                         │
│  ✓ package.json found                                                   │
│    → npm run dev (port 3000)                                            │
│                                                                         │
│  ✓ backend/main.py found                                                │
│    → python main.py (port 8000)                                         │
│                                                                         │
│  ✓ docker-compose.yml found                                             │
│    → postgres (port 5432)                                               │
│                                                                         │
│  Create .hun.yml with these services? [Y/n] y                           │
│                                                                         │
│  ✓ Created .hun.yml                                                     │
```

```
│
│  ✓ Registered project: hun-sh
│
│
│
│  Run 'hun switch hun-sh' to start
│
│
│
└──────────────────────────────────────────────────────────────────────
──────────┘
```

Example 2: Morning Start

```
$ hun
```

Interactive picker appears. Type "hun" to filter, press enter to select.

```
┌─ hun ────────────────────────────────────────────────────────────────
──────────┐
│
│
│  Switching to hun-sh
│
│
│
│  ▶ Starting services...
│
│    ├ db           :5432  ✓ ready (2.1s)
│
│    ├ backend      :8000  ✓ ready (1.4s)
│
│    └ frontend     :3000  ✓ ready (3.2s)
│
│
```

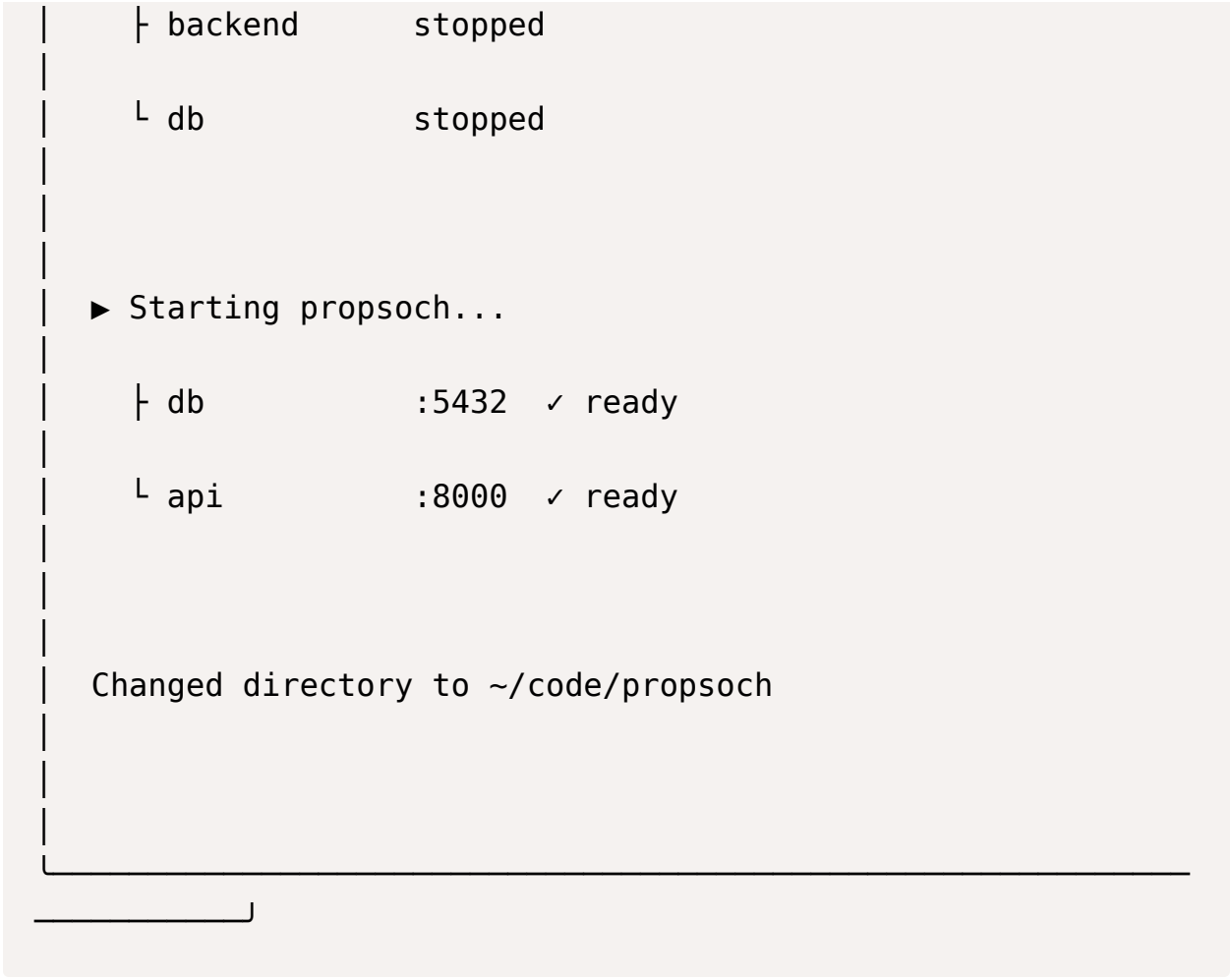
```
│
│  Changed directory to ~/code/hun-sh
│
│
│
│  Previous session:
│
│    Branch: feature/tui-improvements
│
│    Note: "Working on split view layout"
│
│
│
╰──────────────────────────────────────────────────────────────────────
────────────╯
```

Example 3: Context Switch

Currently working on hun-sh, need to fix a bug in propsoch:

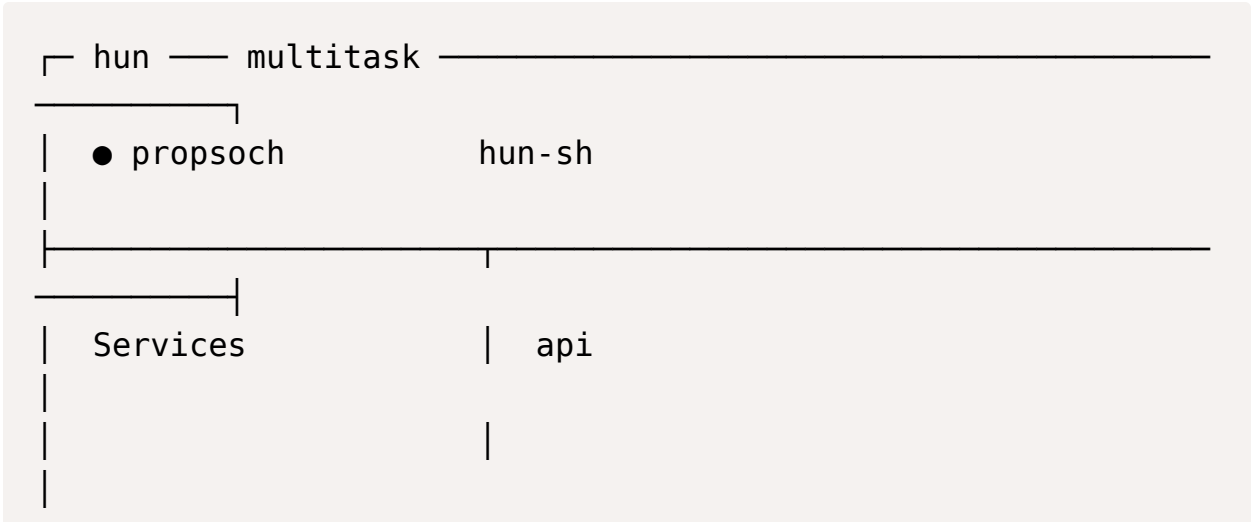
```
$ hun switch propsoch -m "pausing TUI work, will return to sp
lit view"
```

```
╭─ hun ────────────────────────────────────────────────────────────────
────────────╮
│
│
│  Switching to propsoch
│
│
│
│  ⏹ Stopping hun-sh...
│
│    ├ frontend     stopped
│
```



Example 4: Running Multiple Projects (Multitask Mode)

In the TUI, press `[m]` to enter Multitask Mode, then `[p]` to open the picker and start additional projects:



```
│  ▶ api       :8000 ✓ │  [12:01:03] Server started
│
│    db        :5432 ✓ │  [12:01:05] GET /api/health 200
│
│                      │  [12:01:08] POST /api/call 201
│
│                      │
│
├──────────────────────┴──────────────────────────────────
──────────┤
│  [tab] project  [↑↓] service  [r] restart  [s] stop  [q] qu
it      │
└─────────────────────────────────────────────────────────
──────────┘
```

Or from CLI without TUI:

```
$ hun run hun-sh
```

```
┌─ hun run ──────────────────────────────────────────────
──────────┐
│
│
│  Starting hun-sh in parallel
│
│
│
│  ⚠ Port offset applied (+1)
│
│    frontend   :3000 → :3001
│
│    backend    :8000 → :8001
│
│    db         :5432 → :5433
│
```

```
│
│
│  ✓ hun-sh running in parallel
│
│
│
└──────────────────────────────────────────────────────────────────────────────
─────────────┘
```

```
$ hun ports

  propsoch
    api           http://localhost:8000
    db            localhost:5432

  hun-sh (+1)
    frontend      http://localhost:3001
    backend       http://localhost:8001
    db            localhost:5433
```

```
```

Example 5: End of Day

```
$ hun stop --all
```

```
┌─ hun ────────────────────────────────────────────────────────────────────────
──────────────┐
│
│
│  Stopping all projects...
│
│
│
```

```
│  ⏹ propsoch
│
│    ├ api          stopped
│
│    └ db           stopped
│
│
│
│  ⏹ hun-sh
│
│    ├ frontend     stopped
│
│    ├ backend      stopped
│
│    └ db           stopped
│
│
│
│  All projects stopped. See you tomorrow!
│
│
│
└──────────────────────────────────────────────────────────────────────────
──────────┘
```

10. Technical Implementation

Technology Stack

Component	Choice	Rationale
Language	Go	Fast startup, single binary, excellent concurrency
TUI Framework	Bubble Tea + Bubbles + Lip Gloss	Charm ecosystem, composable, beautiful defaults

Component	Choice	Rationale
IPC	Unix domain sockets	Fast, secure CLI-daemon communication
Config Format	YAML	Familiar to developers
State Storage	JSON files	Simple, debuggable
Process Management	Go's os/exec with process groups	Clean termination of process trees

Project Structure

```
hun/
├── cmd/
│   └── hun/
│       └── main.go           # CLI entrypoint
├── internal/
│   ├── cli/                  # Command implementations
│   │   ├── switch.go
│   │   ├── logs.go
│   │   ├── status.go
│   │   └── ...
│   ├── daemon/               # Background daemon
│   │   ├── daemon.go
│   │   ├── process.go
│   │   └── logs.go
│   ├── config/               # Configuration parsing
│   │   ├── project.go
│   │   └── global.go
│   ├── tui/                  # Terminal UI
│   │   ├── logs.go
│   │   ├── picker.go
│   │   └── peek.go
│   └── state/                # State management
│       └── state.go
```

```
├── go.mod
└── go.sum
```

Key Technical Decisions

Decision	Rationale
Single Binary Distribution	No runtime dependencies. Download and run. Critical for adoption
Daemon Auto-Start	CLI automatically starts daemon if not running. Users never manage daemon manually
Process Groups	Services started in process groups, enabling clean termination of entire process trees (important for Node.js which spawns child processes)
Log Ring Buffer	Daemon maintains last 10,000 lines per service in memory for instant TUI access. Older logs written to disk
Log Rotation	Disk logs rotate by size (default 10MB) using lumberjack. Max 3 rotated files per service, 7-day retention. Prevents log bloat over time while preserving recent history
Ready Detection	Services define a 'ready' pattern. Daemon watches stdout before marking service ready
Port Offset +1	Parallel projects get ports offset by +1 per project (project 1: :3000, project 2: :3001, project 3: :3002). Intuitive and easy to remember
TUI Dual Mode	TUI operates in Focus Mode (single project, `hun switch` behavior) or Multitask Mode (parallel projects, `hun run` behavior). Mode persists across sessions

Dependencies

```
// go.mod
module github.com/hun-sh/hun

go 1.22

require (
    github.com/charmbracelet/bubbletea v0.25.0
```

```
    github.com/charmbracelet/bubbles v0.18.0
    github.com/charmbracelet/lipgloss v0.9.1
    github.com/spf13/cobra v1.8.0
    gopkg.in/yaml.v3 v3.0.1
    gopkg.in/natefinished/lumberjack.v2 v2.2.1  // log rotati
on
)
```

11. Daemon Deep Dive

The daemon is the brain of hun.sh. It runs in the background, manages all service processes, captures logs, handles port allocation, and exposes an API for the CLI and TUI to consume. The user never starts or stops the daemon manually — it's fully transparent.

Lifecycle

Auto-start: The first time any `hun` command runs (CLI or TUI), it checks for the daemon via `~/.hun/daemon.sock`. If the socket doesn't exist or the daemon isn't responding, the CLI forks a new daemon process in the background. This happens in under 50ms — the user never notices.

Persistence: The daemon runs independently of any terminal session. Closing the TUI, closing your terminal, even logging out — the daemon keeps running and services stay alive. It only stops when explicitly told to (`hun stop --all`) or when the system shuts down.

Auto-recovery: If the daemon crashes (rare, but possible), the next `hun` command detects the stale socket, cleans it up, and spawns a fresh daemon. The new daemon reads `~/.hun/state.json` to understand what should be running. Services that were managed by the old daemon are detected as orphaned processes and either re-adopted or cleaned up.

Graceful shutdown: When the daemon receives a SIGTERM or SIGINT, it sends SIGTERM to all managed process groups, waits up to 5 seconds for graceful shutdown, then sends SIGKILL to anything still alive. State is persisted before exit.

Process Management

Every service is started in its own **process group** using `syscall.SysProcAttr{Setpgid: true}`. This is critical because many dev tools (Node.js, webpack, etc.) spawn child processes. When hun.sh needs to stop a service, it sends SIGTERM to the entire process group (`syscall.Kill(-pid, syscall.SIGTERM)`), ensuring no orphaned child processes.

Process tree for a typical service:

```
hun-daemon (PID 1234)
  └─ process group: frontend
       └─ npm run dev (PID 1235)
            └─ node webpack-dev-server (PID 1236)
                 └─ node worker (PID 1237)

SIGTERM to group → kills 1235, 1236, 1237 cleanly
```

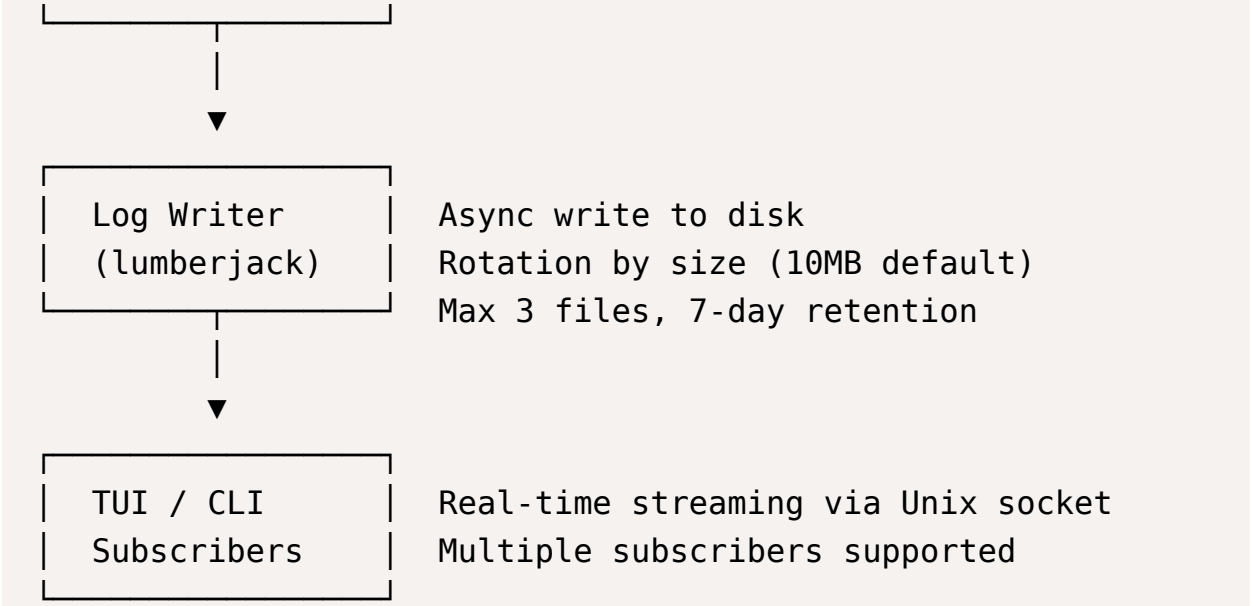
Service startup order: Services with `depends_on` are started in dependency order. The daemon waits for the `ready` pattern in each dependency's stdout before starting the next service. If no `ready` pattern is defined, the daemon waits for the process to be alive for 1 second before proceeding.

Health monitoring: The daemon watches each service process. If a service exits unexpectedly, the daemon marks it as crashed in state and updates the TUI in real-time (the service dot turns red). The daemon does not auto-restart crashed services by default — the developer presses `[r]` in the TUI or runs `hun restart` from the CLI. Auto-restart can be enabled per-service in `.hun.yml` with `restart: on_failure`.

Log Pipeline

Logs flow through a three-stage pipeline:

```
Service stdout/stderr
        │
        ▼
┌──────────────────┐
│  Ring Buffer     │  Last 10,000 lines per service
│  (in-memory)     │  Instant access for TUI
```



Ring buffer: Each service gets a circular buffer of 10,000 lines in memory. When the TUI opens or switches to a service, it reads from the ring buffer for instant display — no disk IO needed. New lines push old lines out of the buffer.

Disk writer: In parallel, logs are written to `~/.hun/logs/<project>/<service>.log` using lumberjack for rotation. This runs on a separate goroutine with a buffered channel so log writing never blocks the service process. Disk logs are the long-term archive; the ring buffer is the fast-access layer.

TUI streaming: The TUI subscribes to log events via the Unix socket. The daemon broadcasts new log lines to all connected subscribers (there could be multiple TUI instances or CLI `hun tail` commands). This uses a simple pub/sub pattern over the socket connection.

Unix Socket API

The daemon exposes a JSON-based API over `~/.hun/daemon.sock`. Both the CLI and TUI are clients of this API.

Core endpoints:

Command	Description	Request
`start`	Start a project's services	`{"action": "start", "project": "hun-sh", "mode": "exclusive" | "parallel"}`

Command	Description	Request
`stop`	Stop a project's services	`{"action": "stop", "project": "hun-sh"}`
`restart`	Restart a specific service	`{"action": "restart", "project": "hun-sh", "service": "frontend"}`
`status`	Get all running projects and services	`{"action": "status"}`
`logs`	Get buffered logs for a service	`{"action": "logs", "project": "hun-sh", "service": "frontend", "lines": 500}`
`subscribe`	Stream real-time logs	`{"action": "subscribe", "project": "hun-sh", "service": "frontend"}`
`ports`	Get port map for all projects	`{"action": "ports"}`

This API is what makes hun.sh agent-friendly. An AI coding agent can connect to the same socket, start/stop projects, read logs, and monitor service health — all without the TUI.

Port Allocation

The daemon manages port allocation centrally. When a project starts:

Exclusive mode (Focus Mode / `hun switch`): Services use their base ports as defined in `.hun.yml`. No offset needed since the previous project was stopped.

Parallel mode (Multitask Mode / `hun run`): The daemon assigns the next available offset. The first project gets offset 0 (base ports), the second gets +1, the third gets +2, and so on. The offset is applied by injecting the `port_env` environment variable with the offset port before starting the service.

```
Project registry:
  propsoch  → offset 0  → api :8000, db :5432
  hun-sh    → offset 1  → frontend :3001, backend :8001, db :
5433
  saas-proj → offset 2  → web :3002, api :8002, worker :9002
```

If a project is stopped and restarted, it gets the lowest available offset — not necessarily the one it had before. The daemon tracks which offsets are in use and recycles them.

State Persistence

The daemon writes state to `~/.hun/state.json` on every significant event (project start, stop, service crash, mode change). This file is the source of truth for recovery.

```
{
  "mode": "multitask",
  "projects": {
    "hun-sh": {
      "status": "running",
      "offset": 1,
      "path": "/Users/sourabh/code/hun-sh",
      "services": {
        "frontend": {"pid": 1235, "port": 3001, "status": "ru
nning"},
        "backend": {"pid": 1240, "port": 8001, "status": "run
ning"},
        "db": {"pid": 1245, "port": 5433, "status": "runnin
g"}
      },
      "git_branch": "feature/tui",
      "last_note": "Working on split view",
      "started_at": "2026-02-12T10:30:00Z"
    }
  },
  "registry": {
    "hun-sh": "/Users/sourabh/code/hun-sh",
    "propsoch": "/Users/sourabh/code/propsoch",
    "client-work": "/Users/sourabh/work/client"
  }
}
```

12. Distribution

Installation Methods

hun.sh is distributed as a **single compiled Go binary** with zero runtime dependencies. The goal is: install in under 30 seconds, works immediately.

Homebrew (macOS & Linux) — Primary:

```
# One-time setup
brew tap hun-sh/tap

# Install
brew install hun
```

Or as a one-liner: `brew install hun-sh/tap/hun`. Once tapped, `brew install hun` and `brew upgrade hun` just work. The long-term goal is to get into Homebrew core so it's just `brew install hun` with no tap needed.

Go install (developers who have Go):

```
go install github.com/hun-sh/hun@latest
```

Requires Go 1.22+. Binary goes to `$GOPATH/bin` which should be in PATH.

Direct download (universal):

```
curl -fsSL https://hun.sh/install.sh | sh
```

The install script detects OS and architecture (macOS arm64/amd64, Linux arm64/amd64), downloads the correct binary from GitHub Releases, and places it in `/usr/local/bin`.

Homebrew Tap Setup

hun.sh uses a dedicated Homebrew tap at `hun-sh/homebrew-tap` on GitHub. The formula:

```
class Hun < Formula
  desc "Seamless project context switching for developers"
  homepage "https://hun.sh"
```

```
  version "1.0.0"

  on_macos do
    if Hardware::CPU.arm?
      url "https://github.com/hun-sh/hun/releases/download/v
1.0.0/hun_darwin_arm64.tar.gz"
      sha256 "..."
    else
      url "https://github.com/hun-sh/hun/releases/download/v
1.0.0/hun_darwin_amd64.tar.gz"
      sha256 "..."
    end
  end

  on_linux do
    if Hardware::CPU.arm?
      url "https://github.com/hun-sh/hun/releases/download/v
1.0.0/hun_linux_arm64.tar.gz"
      sha256 "..."
    else
      url "https://github.com/hun-sh/hun/releases/download/v
1.0.0/hun_linux_amd64.tar.gz"
      sha256 "..."
    end
  end

  def install
    bin.install "hun"
  end

  test do
    assert_match "hun.sh", shell_output("#{bin}/hun --versio
n")
  end
end
```

Release Pipeline

Releases are automated via **GoReleaser** + **GitHub Actions**. When a new tag is pushed (`v1.0.0`), the pipeline:

1. **Builds** — Cross-compiles for macOS (arm64, amd64) and Linux (arm64, amd64) using `GOOS` and `GOARCH`
2. **Archives** — Creates `.tar.gz` for each platform with the binary and a LICENSE file
3. **Checksums** — Generates SHA256 checksums for all archives
4. **GitHub Release** — Creates a release with changelog (auto-generated from commit messages) and uploads all archives
5. **Homebrew** — Automatically updates the Homebrew formula in `hun-sh/homebrew-tap` with new version, URLs, and checksums
6. **Install script** — The `hun.sh/install.sh` script always points to the latest release

GoReleaser config (`.goreleaser.yml`):

```
builds:
  - main: ./cmd/hun
    binary: hun
    goos:
      - darwin
      - linux
    goarch:
      - amd64
      - arm64
    ldflags:
      - -s -w
      - -X main.version={{.Version}}
      - -X main.commit={{.ShortCommit}}

archives:
  - format: tar.gz
    name_template: "hun_{{ .Os }}_{{ .Arch }}"
```

```
brews:
  - repository:
      owner: hun-sh
      name: homebrew-tap
    homepage: "https://hun.sh"
    description: "Seamless project context switching for deve
lopers"
    install: |
      bin.install "hun"

checksum:
  name_template: checksums.txt

changelog:
  sort: asc
  filters:
    exclude:
      - "^docs:"
      - "^chore:"
```

Update Strategy

Homebrew users: `brew upgrade hun` pulls the latest version. The TUI can show a subtle notification when a new version is available (checks GitHub API once per day, non-blocking).

Direct install users: `hun update` re-runs the install script to fetch the latest binary. Or users can re-run the curl command.

Version check: `hun --version` shows the current version, commit hash, and build date. `hun doctor` includes a version check that compares against the latest GitHub release.

---

13. Roadmap

Phase 1: Foundation (v0.1 - v0.3)

Version	Features
v0.1	Config parsing, basic process start/stop, single project
v0.2	Daemon mode, log capture, multiple services
v0.3	Switch command, state preservation, project registry

Phase 2: User Experience (v0.4 - v0.6)

Version	Features
v0.4	TUI log viewer with service switching
v0.5	Interactive project picker with fuzzy search
v0.6	Background run with port offsetting, hun peek

Phase 3: Polish (v0.7 - v1.0)

Version	Features
v0.7	Auto-detection (package.json, docker-compose, etc.)
v0.8	Ready detection, service dependencies
v0.9	Global hotkey integration, hun doctor
v1.0	Public release, documentation, website

Future Considerations

- Web UI on localhost for browser-based log viewing
- VS Code / Cursor extension for IDE integration
- Team features: shared configurations, environment sync
- Cloud sync: project configs across machines
- Plugin system for custom service types

---

14. Success Metrics

User Experience Metrics

Metric	Target
Switch Time	< 2 seconds from command to all services ready
CLI Startup	< 100ms for any command to begin execution
Memory Usage	< 50MB for daemon with 5 active projects
Log Latency	< 50ms from service output to TUI display

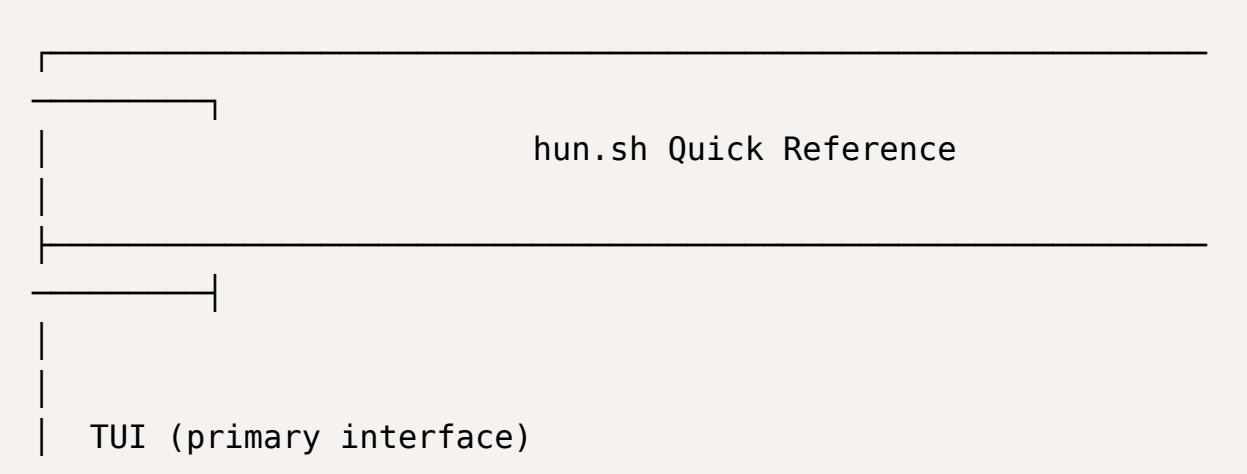
Adoption Metrics

Metric	Target
GitHub Stars	1,000 in first 3 months
Weekly Active Users	500 in first 3 months
Retention	40% weekly retention after first use
NPS	> 50 from user surveys

Quality Metrics

Metric	Target
Process Cleanup	100% - no orphaned processes after stop
Crash Recovery	Daemon auto-restarts, preserves state
Cross-Platform	Works on macOS, Linux (Windows later)
Test Coverage	> 80% for core functionality

Appendix: Quick Reference Card



```
│
│    hun                       open TUI (Focus Mode)
│
│    hun --multi               open TUI (Multitask Mode)
│
│
│
│  TUI KEYBINDINGS
│
│    ↑↓          select service       tab       cycle project
s         │
│    r           restart service       R         restart proj
ect        │
│    c           copy mode             /         search logs
│
│    p           project picker        a         all logs
│
│    s           stop project          m/f       switch mode
│
│    q           quit TUI
│
│
│
│  CLI (headless / agents)
│
│    hun switch <project>     stop all, start one (focus)
│
│    hun run <project>        start alongside others (multita
sk)       │
│    hun stop <project>       stop specific project
│
│    hun stop --all           stop everything
│
│    hun restart proj:svc     restart one service
│
│
```

```
  │
  │  INFO
  │
  │    hun status               all projects overview
  │
  │    hun ports                port map
  │
  │    hun logs proj:svc        dump logs to stdout
  │
  │    hun tail proj:svc        stream logs
  │
  │
  │
  │  SETUP
  │
  │    hun init                 initialize current directory
  │
  │    hun init --name <n>      init with explicit name
  │
  │    hun add <path>           register existing project
  │
  │    hun list                 all known projects
  │
  │
  │
  └──────────────────────────────────────────────────────────────────────
  ──────────┘
```

This specification is a living document. As development progresses and user feedback is gathered, priorities and features may evolve. The core philosophy — one project at a time by default, multiple when you need it, always in control — remains the north star.