URADICAL

POCKET REFERENCE

launchd Pocket Reference

Alan Bradley

launchd Pocket Reference

A developer's guide to macOS service management

Alan Bradley

uradical.io

Table of Contents

Chapter 1: What is launchd?

launchd is the first process the macOS kernel starts after boot â€” PID 1. Every process on your Mac, from the Finder to your terminal, is ultimately a descendant of it. It is simultaneously an init system, a service manager, a cron replacement, and a socket activation daemon. Apple introduced it in OS X Tiger (10.4) in 2005, and it has been the sole sanctioned way to run background services on macOS ever since.

Unlike Linux, where init systems vary by distribution â€” SysV, Upstart, and OpenRC remain in active use, and SystemD dominates but is not universal â€” macOS has had one answer to this problem for twenty years. That stability is a feature, but it comes at a cost: launchd is deliberately opaque, Apple documents it minimally, and the tooling has changed under developers' feet more than once without much ceremony.

Key things to know before you start

Everything is a plist. Service definitions are XML property lists â€” the same format macOS uses for application preferences. There is no unit file syntax, no INI-style sections. If you find plist XML verbose, you can write in JSON or use a tool to generate it, but the canonical format is XML and that is what the system reads.

Jobs follow a naming convention. Every launchd job has a `Label` â€” a unique identifier that must not clash with any other job on the system. By convention this follows reverse-DNS notation: `com.yourcompany.servicename` or `io.uradical.devproxy`. This is not enforced by launchd itself, but deviating from it is a reliable way to create collisions with system services.

launchd does not have targets or dependencies. SystemD has a rich dependency graph â€” `Wants`, `Requires`, `After`, `Before`. launchd has

none of this. Services are loaded and started independently. If your service depends on the network being up, the closest native option is `KeepAlive` with a `NetworkState` condition â€” but for most cases you will write retry logic into your script or binary. For developer automation tasks this is rarely a problem. For complex service orchestration, it is a genuine limitation worth knowing upfront.

There are three kinds of job: Launch Agent, system Launch Agent, and Launch Daemon. The distinction matters more than it does in SystemD. User Agents run in your login session â€” they have access to the GUI, the user's environment, and the user's keychain. System Agents run for all users but still within a user session context. Daemons run as root before any user logs in and have no access to the display server at all. Getting this wrong is the single most common source of launchd confusion, and it drives everything: where you put the plist, what the service can access, and how you manage it. Chapter 2 covers all three in full.

`launchctl` **has two personalities.** Before macOS Yosemite (10.10), you managed services with `launchctl load` and `launchctl unload`. Apple replaced this with a new service management framework and new verbs: `bootstrap`, `bootout`, `enable`, `disable`, `kickstart`, `print`. The old verbs still exist but are deprecated and behave differently in subtle ways on modern macOS. This guide uses the modern interface throughout.

Logging requires a little ceremony. launchd does not write your service output to a central journal the way SystemD does with `journalctl`. You specify log file paths in the plist yourself, or you use the unified logging system and `log stream` to tail output in real time. Chapter 7 covers both approaches.

Coming from SystemD

If you have used SystemD on Linux, the mental model transfers about two thirds of the way. The remainder is where the friction lives.

The biggest conceptual shift is **scope over privilege**. In SystemD you think primarily in terms of what user a service runs as. In launchd you think in terms of which session context the job lives in â€” user session, system session, or pre-login daemon context. This is not arbitrary design: it reflects macOS's security architecture, where session boundaries enforce isolation between the display server, the user keychain, and system resources. Understanding why those boundaries exist makes the three-scope model feel logical rather than bureaucratic, and it is where Chapter 2 begins.

Chapter 2: Agents, Daemons, and the Three Scopes

The single most important thing to understand about launchd is that where you put your plist file determines almost everything about how your service behaves. This is not a filing convention â€” it is a declaration of which security context your job runs in, who owns it, what it can access, and how it is managed. Get the location wrong and your service will either fail silently, lack the permissions it needs, or behave differently across user sessions in ways that are difficult to debug.

launchd recognises three distinct scopes, each with its own directory, its own lifetime, and its own constraints.

The three scopes

User Launch Agents â€” ~/Library/LaunchAgents

Jobs placed here belong to the logged-in user and run within that user's session. They start when the user logs in and stop when the user logs out. Because they run inside the session, they have full access to everything the user has access to: the GUI and display server, the user's environment variables, the user's keychain, and user-owned files and sockets.

This is the right scope for the vast majority of developer automation tasks â€” starting a local development server, running a file watcher, scheduling a backup script, keeping a queue worker alive. If you are automating something you would otherwise run in a terminal, a user Launch Agent is almost certainly what you want.

These jobs are owned and managed entirely by the user. No `sudo` is required to load, unload, or inspect them.

System Launch Agents â€” /Library/LaunchAgents

Jobs placed here run for every user who logs into the machine, once per user session. Like user Agents, they run within the user's session context, so they have access to the display server and per-user resources. Unlike user Agents, they are installed system-wide and apply to all users.

This scope is primarily relevant for software installers â€” an application that needs to run a helper process for every user without requiring each user to configure it manually. As a developer managing your own machine, you will rarely use this scope. It requires administrator privileges to install files here.

Launch Daemons â€” /Library/LaunchDaemons

Jobs placed here run as root, before any user logs in, and continue running regardless of whether anyone is logged in at all. They have no access to the display server, no access to any user's keychain, and no access to per-user resources. They are true system services in the Unix sense â€” analogous to `sshd`, `ntpd`, or a database server that should be available the moment the machine boots.

Because they run as root and start before login, Daemons are appropriate for system-level tasks: a network service that must be available pre-login, a privileged helper that user-space processes communicate with via a socket, or a monitoring agent that must survive user logout.

A Daemon can be configured to drop root privileges and run as a specific user and group using the `UserName` and `Group` keys in the plist. These two keys are almost always used together:

```
<key>UserName</key>
<string>_myservice</string>
<key>Group</key>
<string>_myservice</string>
```

The job is still loaded by root and lives in the system context â€” it has no access to any user session â€” but running with least privilege is good practice for any long-lived network-facing service. Chapter 3 covers these keys in full as part of the plist anatomy.

Installing a Daemon always requires `sudo`.

File locations at a glance

The `/System/Library` paths are reserved for Apple. You should never place files there, and on modern macOS with System Integrity Protection enabled, you cannot â€” they are read-only even to root.

The session boundary in practice

The session boundary is not abstract. It has concrete consequences that catch developers out repeatedly.

A Daemon cannot show a UI. If your plist is in `/Library/LaunchDaemons` and your binary tries to open a window, display a notification, or write to the pasteboard, it will fail. There is no display connection available in the daemon context.

A Daemon cannot access the user keychain. If your service needs to read a stored credential, it must either be a user Agent, or it must use a system keychain entry explicitly set to allow system-level access â€” a non-trivial setup.

Environment variables are not inherited from the shell. This applies to both Agents and Daemons, but catches Agent users by surprise. Your `.zshrc`, your `~/.profile`, your `nvm` setup, your Homebrew PATH â€" none of it is present when launchd starts your job. `PATH` in the daemon context is a minimal `/usr/bin:/bin:/usr/sbin:/sbin`. Chapter 6 covers how to handle this correctly.

User Agents do not start until login. If you need a service that is available before any user logs in â€" a VPN daemon, a remote access helper, a database that other system services depend on â€" it must be a Launch Daemon. A user Agent that starts on `RunAtLoad` still only loads when its owner logs in.

Choosing the right scope

The decision comes down to two questions: does the service need access to a user session, and does it need to survive logout or start before login?

When in doubt, start with a user Agent. It requires no elevated privileges, is easy to inspect and debug, and can access everything a terminal session can. Promote to a wider scope only once you have confirmed the service works correctly and understand exactly what permissions it requires.

A note on System Integrity Protection

Since macOS El Capitan (10.11), System Integrity Protection (SIP) restricts what even root can do. The `/System` paths are completely off-limits. More relevantly for developers, SIP also restricts certain launchctl operations on system jobs. If you find that a `launchctl` command that should work is being refused, check whether SIP is the constraint with `csrutil status`. For jobs in `~/Library/LaunchAgents`

and `/Library/LaunchDaemons`, SIP does not interfere with normal management operations.

Chapter 3: Anatomy of a plist

A launchd job is defined entirely by a single property list file. There are no secondary configuration files, no command-line flags passed to launchd itself, no environment to source. Everything launchd needs to know about your job â€” what to run, when to run it, how to restart it, where to log output â€” lives in that one file. Getting comfortable with the plist format and knowing which keys do what is the foundation for everything else in this guide.

The plist format

Property lists come in three flavours: XML, binary, and JSON. launchd reads all three, but XML is the canonical format for service definitions and the one you should use. It is human-readable, diff-friendly, and unambiguous. Binary plists are fine for application preferences but have no place in a file you will be editing and version-controlling.

A minimal valid launchd plist looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>io.uradical.devproxy</string>

  <key>ProgramArguments</key>
  <array>
    <string>/usr/local/bin/devproxy</string>
    <string>--port</string>
    <string>8080</string>
  </array>

  <key>RunAtLoad</key>
```

```
  <true/>
</dict>
</plist>
```

That is enough to define a service that starts when the plist is loaded. Everything else is optional, but most real jobs will need a handful of additional keys.

The plist type system

launchd plists use a small set of XML types. You will encounter all of these:

A common mistake is writing `<bool>true</bool>` â€” that is not valid plist XML. Boolean values are self-closing tags: `<true/>` and `<false/>`.

Keys reference

Identity

`Label` *(string, required)*

The unique identifier for this job. Must be unique across all loaded jobs on the system. Use reverse-DNS notation.

```
<key>Label</key>
<string>io.uradical.devproxy</string>
```

`Disabled` *(boolean)*

If set to `<true/>`, the job will not be loaded automatically when the plist is placed in a launchd directory. The plist is registered with launchd but the job remains inactive until explicitly enabled with `launchctl enable`. You will encounter this key frequently when inspecting system plists â€”

Apple uses it to ship jobs that are installed but not active by default. It is occasionally useful when you want to deploy a plist without it starting immediately.

```
<key>Disabled</key>
<true/>
```

What to run

`ProgramArguments` *(array of strings, required unless* `Program` *is set)*

The command to run, expressed as an array where the first element is the executable path and subsequent elements are arguments. This is the form you will use most often.

```
<key>ProgramArguments</key>
<array>
  <string>/usr/local/bin/devproxy</string>
  <string>--port</string>
  <string>8080</string>
  <string>--config</string>
  <string>/etc/devproxy/config.yaml</string>
</array>
```

Always use absolute paths. launchd does not use your shell's `PATH`, so `devproxy` alone will not be found. Chapter 6 covers the PATH problem in full.

`Program` *(string)*

Use this when you have a binary that takes no arguments and you want a simpler plist. It specifies the executable path directly without the array overhead of `ProgramArguments`. If both `Program` and `ProgramArguments` are present, `Program` is used as the executable path and `ProgramArguments` provides the full `argv` including `argv[0]` â€” a subtle distinction that rarely matters in practice but is worth knowing if you are wrapping a binary that inspects its own process

name.

```
<key>Program</key>
<string>/usr/local/bin/healthcheck</string>
```

When to run

`RunAtLoad` *(boolean)*

Start the job as soon as the plist is loaded â€” at login for Agents, at boot for Daemons. This is the most common way to start a persistent service.

```
<key>RunAtLoad</key>
<true/>
```

`StartInterval` *(integer)*

Run the job repeatedly on a fixed interval, specified in seconds. This is the simplest cron replacement â€” no cron syntax required.

```
<key>StartInterval</key>
<integer>300</integer>
```

This runs the job every five minutes. The timer is wall-clock based, not relative to when the job last finished. If the job is still running when the interval fires, launchd will not start a second instance.

`StartCalendarInterval` *(dict or array of dicts)*

Run the job on a calendar schedule, similar to cron. The dict can contain any combination of `Minute`, `Hour`, `Day`, `Weekday`, and `Month` keys. Omitted keys are treated as wildcards.

Run every day at 02:30:

```
<key>StartCalendarInterval</key>
<dict>
  <key>Hour</key>
  <integer>2</integer>
  <key>Minute</key>
  <integer>30</integer>
</dict>
```

Run every Monday and Thursday at 09:00 by using an array of dicts:

```
<key>StartCalendarInterval</key>
<array>
  <dict>
    <key>Weekday</key>
    <integer>1</integer>
    <key>Hour</key>
    <integer>9</integer>
    <key>Minute</key>
    <integer>0</integer>
  </dict>
  <dict>
    <key>Weekday</key>
    <integer>4</integer>
    <key>Hour</key>
    <integer>9</integer>
    <key>Minute</key>
    <integer>0</integer>
  </dict>
</array>
```

Weekday values: 0 = Sunday, 1 = Monday, through 6 = Saturday. One important behavioural difference from cron: if the Mac is asleep when a scheduled job is due, launchd will run it when the machine wakes. cron silently skips missed runs.

Keeping a service alive

`KeepAlive` *(boolean or dict)*

The simple boolean form restarts the job unconditionally whenever it exits, regardless of exit code.

```
<key>KeepAlive</key>
<true/>
```

The dict form gives you conditional restart behaviour. The most useful conditions:

```
<key>KeepAlive</key>
<dict>
  <!-- Only restart if the job exited non-zero -->
  <key>SuccessfulExit</key>
  <false/>

  <!-- Only keep alive while network is available -->
  <key>NetworkState</key>
  <true/>

  <!-- Only keep alive while a path exists -->
  <key>PathState</key>
  <dict>
    <key>/tmp/devproxy.enable</key>
    <true/>
  </dict>
</dict>
```

`SuccessfulExit` set to `<false/>` means "restart only on failure" â€” useful for services that are expected to exit cleanly when done but should be retried if they crash. `NetworkState` set to `<true/>` means "only run while the network is up" â€” the closest launchd gets to a network dependency.

`ThrottleInterval` *(integer)*

The minimum number of seconds launchd will wait before restarting a job that has exited. Defaults to 10 seconds. If your job is crashing on startup and restarting immediately in a tight loop, launchd will automatically back off, but setting this explicitly gives you control.

```
<key>ThrottleInterval</key>
<integer>30</integer>
```

Environment

`EnvironmentVariables` *(dict)*

Key-value pairs set in the job's environment before launch. This is how you provide `PATH`, secrets, and configuration values that your binary expects to find in the environment.

```
<key>EnvironmentVariables</key>
<dict>
  <key>PATH</key>

<string>/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin</string>
  <key>APP_ENV</key>
  <string>production</string>
  <key>LOG_LEVEL</key>
  <string>info</string>
</dict>
```

`WorkingDirectory` *(string)*

Sets the working directory for the job before it starts. Without this, the working directory is undefined and your service should not rely on relative paths for anything.

```
<key>WorkingDirectory</key>
<string>/var/lib/devproxy</string>
```

Logging

`StandardOutPath` *(string)*

Redirect the job's stdout to a file. The file will be created if it does not exist. The directory must already exist â€” launchd will not create intermediate directories.

```
<key>StandardOutPath</key>
<string>/var/log/devproxy/stdout.log</string>
```

`StandardErrorPath` *(string)*

Redirect stderr to a file. It is common to direct both stdout and stderr to the same file, though separating them can make debugging easier.

```
<key>StandardErrorPath</key>
<string>/var/log/devproxy/stderr.log</string>
```

Privilege management

`UserName` *(string)*

Run the job as this user. Only meaningful for Launch Daemons, which default to root. Has no effect on Launch Agents, which always run as the logged-in user.

`Group` *(string)*

Run the job with this group. Almost always used alongside `UserName`.

```
<key>UserName</key>
<string>_devproxy</string>
<key>Group</key>
<string>_devproxy</string>
```

On-demand and socket activation

`Sockets` *(dict)*

Define one or more sockets that launchd will create and hold open. Rather than starting immediately at load, the job is started on demand when a connection arrives on the socket. This is launchd's socket activation mechanism, equivalent to SystemD's socket units.

```
<key>Sockets</key>
<dict>
  <key>Listeners</key>
  <dict>
    <key>SockServiceName</key>
    <string>8080</string>
    <key>SockType</key>
    <string>stream</string>
  </dict>
</dict>
```

At runtime, your binary retrieves the socket file descriptor by calling `launch_activate_socket()` â€” a C API provided by the macOS SDK that hands back the already-open file descriptor launchd is holding. You do not bind or listen yourself; launchd has done that on your behalf. Chapter 5 covers the full socket activation pattern including a worked example.

A complete annotated example

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <!-- Unique identifier in reverse-DNS format -->
  <key>Label</key>
  <string>io.uradical.devproxy</string>

  <!-- Absolute path to binary, arguments as separate array
elements -->
```

```
  <key>ProgramArguments</key>
  <array>
    <string>/usr/local/bin/devproxy</string>
    <string>--port</string>
    <string>8080</string>
    <string>--config</string>
    <string>/Users/alan/.config/devproxy/config.yaml</string>
  </array>

  <!-- Start immediately when plist is loaded -->
  <key>RunAtLoad</key>
  <true/>

  <!-- Restart on crash but not on clean exit -->
  <key>KeepAlive</key>
  <dict>
    <key>SuccessfulExit</key>
    <false/>
  </dict>

  <!-- Wait at least 15 seconds before restarting -->
  <key>ThrottleInterval</key>
  <integer>15</integer>

  <!-- Provide a sane PATH and runtime config -->
  <key>EnvironmentVariables</key>
  <dict>
    <key>PATH</key>
    <string>/usr/local/bin:/usr/bin:/bin</string>
    <key>APP_ENV</key>
    <string>development</string>
  </dict>

  <!-- Explicit working directory -->
  <key>WorkingDirectory</key>
  <string>/Users/alan/projects/devproxy</string>

  <!-- Capture all output -->
  <key>StandardOutPath</key>

<string>/Users/alan/Library/Logs/devproxy/stdout.log</string>
  <key>StandardErrorPath</key>
```

```
<string>/Users/alan/Library/Logs/devproxy/stderr.log</string>
</dict>
</plist>
```

Common mistakes

Using relative paths anywhere. launchd does not have a shell context. Relative paths in `ProgramArguments` or `WorkingDirectory` will resolve against an undefined working directory and almost certainly fail.

Missing log directories. If `/Users/alan/Library/Logs/devproxy/` does not exist, launchd will not create it and your output will be silently discarded. Create the directory before loading the plist.

Passing arguments as a single string. `<string>/usr/local/bin/devproxy --port 8080</string>` will not work â€” launchd does not invoke a shell to parse the command. Each argument must be a separate `<string>` element in the array.

Using `<bool>true</bool>`**.** Not valid plist XML. Use `<true/>`.

Setting `KeepAlive` **to true on a scheduled job.** If a job has both `StartCalendarInterval` and `KeepAlive <true/>`, launchd will restart the job immediately after it exits rather than waiting for the next scheduled time. Use the conditional `SuccessfulExit` form instead.

Chapter 4: launchctl Command Reference

`launchctl` is the command-line interface to launchd. It is how you load and unload jobs, start and stop them, inspect their state, and diagnose failures. It is also the source of more developer confusion than almost anything else in the launchd ecosystem, primarily because it has two distinct interfaces â€” the legacy interface that most Stack Overflow answers still show, and the modern interface that you should actually be using.

This chapter covers the modern interface exclusively. If you are on macOS Yosemite (10.10) or later â€” which at this point means virtually everyone â€” these are the commands you need.

Services and domains

Before the commands make sense, you need the mental model. The modern `launchctl` organises jobs into **domains**. A domain is a management context â€” it owns a set of jobs and defines the scope in which they run. The domains you will work with as a developer are:

You can find your uid with `id -u`. For most single-user Mac setups it is `501`. A job's full service identifier combines the domain and the label: `gui/501/io.uradical.devproxy`.

The distinction between `gui/<uid>` and `user/<uid>` matters when you are logged in over SSH rather than at the console. An SSH session does not have a GUI session, so `gui/<uid>` does not exist â€” you manage user-scoped jobs through `user/<uid>` instead. On a normal interactive desktop login you will almost always use `gui/<uid>`.

Loading and unloading jobs

launchctl bootstrap

Loads a plist into a domain, registering the job with launchd. This is the modern replacement for `launchctl load`.

```
# Load a user Agent
launchctl bootstrap gui/$(id -u)
~/Library/LaunchAgents/io.uradical.devproxy.plist

# Load a system Daemon (requires sudo)
sudo launchctl bootstrap system
/Library/LaunchDaemons/io.uradical.myservice.plist
```

After bootstrapping, launchd will start the job according to its plist â€” immediately if `RunAtLoad` is set, on schedule if `StartCalendarInterval` is set, or on demand if `Sockets` is set.

launchctl bootout

Removes a job from a domain, stopping it if it is running. The modern replacement for `launchctl unload`.

```
# Unload a user Agent
launchctl bootout gui/$(id -u)
~/Library/LaunchAgents/io.uradical.devproxy.plist

# Unload a system Daemon (requires sudo)
sudo launchctl bootout system
/Library/LaunchDaemons/io.uradical.myservice.plist
```

You can also bootout by service identifier rather than plist path:

```
launchctl bootout gui/$(id -u)/io.uradical.devproxy
```

Reloading after a plist change

launchd does not have a reload or daemon-reload command. When you change a plist you must bootout and bootstrap. During development this becomes repetitive quickly, so it is worth adding a shell function to your `.zshrc`:

```
# Add to ~/.zshrc
function launchctl-reload() {
  local label="$1"
  local plist=~/Library/LaunchAgents/${label}.plist
  launchctl bootout gui/$(id -u)/${label} 2>/dev/null
  launchctl bootstrap gui/$(id -u) "${plist}"
  echo "Reloaded ${label}"
}
```

Usage:

```
launchctl-reload io.uradical.devproxy
```

The `2>/dev/null` on the bootout suppresses the error if the job was not loaded â€” useful when you are iterating on a plist that may not have been successfully bootstrapped yet.

Starting and stopping jobs

launchctl kickstart

Starts a job immediately, regardless of its scheduled trigger. Useful for testing or forcing a run outside the normal schedule.

```
# Start the job
launchctl kickstart gui/$(id -u)/io.uradical.devproxy

# Start and print the job's PID when it launches
launchctl kickstart -p gui/$(id -u)/io.uradical.devproxy
```

```
# Force restart if already running
launchctl kickstart -k gui/$(id -u)/io.uradical.devproxy
```

The `-k` flag kills the currently running instance before starting a new one. Without it, `kickstart` will do nothing if the job is already running.

launchctl kill

Sends a signal to a running job.

```
# Send SIGTERM (graceful shutdown)
launchctl kill SIGTERM gui/$(id -u)/io.uradical.devproxy

# Send SIGINT
launchctl kill SIGINT gui/$(id -u)/io.uradical.devproxy

# Send SIGHUP (conventional reload signal)
launchctl kill SIGHUP gui/$(id -u)/io.uradical.devproxy
```

This is not the same as stopping and unloading â€” the job will be restarted by launchd if `KeepAlive` is set. To stop a job permanently, use `bootout`.

Enabling and disabling jobs

launchctl enable and launchctl disable

These commands control whether a job is permitted to run, independently of whether it is currently loaded. A disabled job will not be bootstrapped automatically at login or boot, even if its plist is in the correct directory.

```
launchctl enable gui/$(id -u)/io.uradical.devproxy
launchctl disable gui/$(id -u)/io.uradical.devproxy
```

The enabled/disabled state is persisted by launchd across reboots in a separate database â€” it is not stored in the plist itself. This means you can deploy a plist with `Disabled <true/>` and later enable it without editing the file. It also means that if you have previously disabled a job, simply bootstrapping it again will not re-enable it â€” you need to explicitly call `enable` first.

This is a common source of confusion: a job that was disabled, then had its plist removed and reinstalled, may still be disabled because the state lives in launchd's database, not the plist.

Inspecting jobs

launchctl list

Lists all loaded jobs in the current user's domain with their PID and last exit status.

```
launchctl list
```

Filter by label:

```
launchctl list | grep uradical
```

Output columns are PID, last exit status, and label. A PID of `-` means the job is not currently running. An exit status of `0` is clean. Non-zero values indicate the job exited with an error â€” or in the case of signal termination, a negative number representing the signal.

launchctl print

The most useful diagnostic command in the modern interface. Prints the full state of a domain or a specific service.

Print everything in the user's GUI domain:

```
launchctl print gui/$(id -u)
```

Print the state of a specific service:

```
launchctl print gui/$(id -u)/io.uradical.devproxy
```

The output for a specific service includes the current state, PID, exit count, last exit status, environment variables as launchd sees them, and the active plist configuration. When a job is misbehaving, `launchctl print` is the first place to look.

launchctl blame

Tells you why a job was started â€” what trigger caused launchd to launch it.

```
launchctl blame gui/$(id -u)/io.uradical.devproxy
```

Useful when a job is starting unexpectedly or you want to confirm that the trigger you configured is actually firing.

System-level commands

launchctl dumpstate

Dumps the full launchd state for all domains to stdout. The output is verbose â€” pipe it through `grep` or redirect to a file. Useful for auditing all running services or hunting for a label conflict.

```
launchctl dumpstate | grep -A 20 "io.uradical"
```

launchctl dumpjpcategory

Dumps the job policy category state. Rarely needed in day-to-day use but occasionally useful when diagnosing jobs that are being throttled or suppressed by App Nap or power management policies.

Legacy commands to avoid

You will encounter these in older guides and Stack Overflow answers. They still work in some contexts but are deprecated and behave inconsistently on modern macOS:

The most important difference is that the legacy commands operate on plists directly and do not understand the domain model. On modern macOS they can silently succeed while actually doing nothing, or produce misleading error messages. Avoid them.

Quick reference

```
# Load a user Agent
launchctl bootstrap gui/$(id -u)
~/Library/LaunchAgents/<label>.plist

# Unload a user Agent
launchctl bootout gui/$(id -u)/<label>

# Reload after plist change
launchctl bootout gui/$(id -u)/<label> 2>/dev/null && \
launchctl bootstrap gui/$(id -u)
~/Library/LaunchAgents/<label>.plist

# Start immediately
launchctl kickstart gui/$(id -u)/<label>
```

```
# Force restart
launchctl kickstart -k gui/$(id -u)/<label>

# Graceful stop (will restart if KeepAlive is set)
launchctl kill SIGTERM gui/$(id -u)/<label>

# Inspect service state
launchctl print gui/$(id -u)/<label>

# List all user jobs with status
launchctl list | grep <label>

# Why did this job start?
launchctl blame gui/$(id -u)/<label>

# Enable a disabled job
launchctl enable gui/$(id -u)/<label>

# Disable a job persistently
launchctl disable gui/$(id -u)/<label>
```

Chapter 5: Common Patterns

The previous chapters covered the building blocks â€” scopes, plist keys, and commands. This chapter puts them together into patterns you will reach for repeatedly. Each pattern is a complete, working plist with enough explanation to adapt it to your own use case.

Pattern 1: Run at login

The simplest pattern. A service that starts when you log in and runs until you log out. No scheduling, no socket activation â€” just a binary that should always be running in your user session.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>io.uradical.devproxy</string>

  <key>ProgramArguments</key>
  <array>
    <string>/usr/local/bin/devproxy</string>
    <string>--port</string>
    <string>8080</string>
  </array>

  <key>RunAtLoad</key>
  <true/>

  <key>KeepAlive</key>
  <true/>

  <key>StandardOutPath</key>

<string>/Users/alan/Library/Logs/devproxy/stdout.log</string>
```

```
  <key>StandardErrorPath</key>

<string>/Users/alan/Library/Logs/devproxy/stderr.log</string>
</dict>
</plist>
```

Save to `~/Library/LaunchAgents/io.uradical.devproxy.plist` and load with:

```
launchctl bootstrap gui/$(id -u)
~/Library/LaunchAgents/io.uradical.devproxy.plist
```

`KeepAlive <true/>` here means launchd will restart the service unconditionally if it exits â€” on crash, on clean exit, on any exit. If your binary is designed to run until explicitly stopped, this is correct. If it exits cleanly under normal conditions, use the conditional form from chapter 3 instead.

Pattern 2: Cron replacement â€” fixed interval

Run a script or binary on a repeating timer. The launchd equivalent of `*/5 * * * *` in cron.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>io.uradical.dbbackup</string>

  <key>ProgramArguments</key>
  <array>
    <string>/usr/local/bin/dbbackup</string>
    <string>--output</string>
    <string>/Users/alan/backups</string>
  </array>
```

```
  <key>StartInterval</key>
  <integer>1800</integer>

  <key>StandardOutPath</key>

<string>/Users/alan/Library/Logs/dbbackup/stdout.log</string>

  <key>StandardErrorPath</key>

<string>/Users/alan/Library/Logs/dbbackup/stderr.log</string>
</dict>
</plist>
```

This runs the backup every thirty minutes. Note the absence of `RunAtLoad` â€" if you want the job to run immediately on load as well as on the interval, add it. Without it the first run waits for the interval to elapse.

Do not combine `StartInterval` with `KeepAlive <true/>`. If your backup script exits cleanly after completing, launchd will immediately restart it rather than waiting for the next interval. If you need crash recovery, use `KeepAlive` with `SuccessfulExit <false/>`.

Pattern 3: Cron replacement â€" calendar schedule

Run a job at a specific time of day or on specific days of the week. More expressive than `StartInterval` for time-of-day scheduling.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>io.uradical.weeklyreport</string>

  <key>ProgramArguments</key>
  <array>
```

```
    <string>/usr/local/bin/generate-report</string>
    <string>--format</string>
    <string>html</string>
    <string>--email</string>
    <string>alan@uradical.io</string>
  </array>

  <key>StartCalendarInterval</key>
  <dict>
    <key>Weekday</key>
    <integer>1</integer>
    <key>Hour</key>
    <integer>8</integer>
    <key>Minute</key>
    <integer>0</integer>
  </dict>

  <key>StandardOutPath</key>

<string>/Users/alan/Library/Logs/weeklyreport/stdout.log</stri
ng>

  <key>StandardErrorPath</key>

<string>/Users/alan/Library/Logs/weeklyreport/stderr.log</stri
ng>
</dict>
</plist>
```

This runs every Monday at 08:00. If the Mac is asleep at that time, the job runs when it wakes â€” a meaningful advantage over cron for laptop users where the machine may not be on at the scheduled time.

Pattern 4: Keep-alive service with crash recovery

A long-running service that should restart on failure but exit cleanly when you explicitly stop it. The right pattern for a local development server, a queue worker, or any service that has a defined shutdown sequence.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>io.uradical.worker</string>

  <key>ProgramArguments</key>
  <array>
    <string>/usr/local/bin/worker</string>
    <string>--queue</string>
    <string>default</string>
    <string>--concurrency</string>
    <string>4</string>
  </array>

  <key>RunAtLoad</key>
  <true/>

  <key>KeepAlive</key>
  <dict>
    <key>SuccessfulExit</key>
    <false/>
  </dict>

  <key>ThrottleInterval</key>
  <integer>10</integer>

  <key>EnvironmentVariables</key>
  <dict>
    <key>PATH</key>
    <string>/usr/local/bin:/usr/bin:/bin</string>
    <key>REDIS_URL</key>
    <string>redis://localhost:6379</string>
  </dict>

  <key>WorkingDirectory</key>
  <string>/Users/alan/projects/worker</string>

  <key>StandardOutPath</key>
  <string>/Users/alan/Library/Logs/worker/stdout.log</string>
```

```
  <key>StandardErrorPath</key>
  <string>/Users/alan/Library/Logs/worker/stderr.log</string>
</dict>
</plist>
```

`SuccessfulExit <false/>` means launchd restarts the worker only if it exits with a non-zero code. A clean exit â€” from a `SIGTERM` handler in your binary or from `launchctl kill SIGTERM` â€” will not trigger a restart. `ThrottleInterval` prevents a tight restart loop if the worker is crashing immediately on startup.

Pattern 5: Network-dependent service

A service that should only run while the network is available â€” useful for anything that makes outbound connections and has no meaningful offline behaviour.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>io.uradical.syncd</string>

  <key>ProgramArguments</key>
  <array>
    <string>/usr/local/bin/syncd</string>
    <string>--remote</string>
    <string>https://api.uradical.io</string>
  </array>

  <key>RunAtLoad</key>
  <true/>

  <key>KeepAlive</key>
  <dict>
    <key>NetworkState</key>
    <true/>
```

```
  </dict>

  <key>StandardOutPath</key>
  <string>/Users/alan/Library/Logs/syncd/stdout.log</string>

  <key>StandardErrorPath</key>
  <string>/Users/alan/Library/Logs/syncd/stderr.log</string>
</dict>
</plist>
```

`NetworkState <true/>` tells launchd to only keep this job alive while a network interface is up. The job will be stopped when the network goes away and restarted when it returns. Note that "network available" means a network interface is active â€” it does not guarantee that your specific remote endpoint is reachable. Your binary still needs to handle connection failures gracefully.

Pattern 6: Socket activation

An on-demand service that launchd starts only when a connection arrives. The job is not running at all when idle â€” launchd holds the socket open and wakes the job when needed. Useful for services that are infrequently used and do not need to consume resources continuously.

The plist:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>io.uradical.ondemand</string>

  <key>ProgramArguments</key>
  <array>
    <string>/usr/local/bin/ondemand</string>
```

```
  </array>

  <key>Sockets</key>
  <dict>
    <key>Listeners</key>
    <dict>
      <key>SockServiceName</key>
      <string>9090</string>
      <key>SockType</key>
      <string>stream</string>
      <key>SockFamily</key>
      <string>IPv4</string>
    </dict>
  </dict>

  <key>StandardOutPath</key>

<string>/Users/alan/Library/Logs/ondemand/stdout.log</string>

  <key>StandardErrorPath</key>

<string>/Users/alan/Library/Logs/ondemand/stderr.log</string>
</dict>
</plist>
```

Note the absence of `RunAtLoad` â€” without it the job only starts on an incoming connection. The socket is ready immediately after the plist is bootstrapped, but the binary itself is not running.

Retrieving the socket in your binary

Your binary must retrieve the already-open socket file descriptor from launchd rather than binding one itself. The original `launch.h` API was deprecated by Apple and removed from the SDK in Xcode 15. The current approach loads `launch_activate_socket` at runtime via `dlsym`:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <dlfcn.h>
```

```
typedef int (*launch_activate_socket_t)(const char *name, int
**fds, size_t *cnt);

int main(void) {
  launch_activate_socket_t activate =
    dlsym(RTLD_DEFAULT, "launch_activate_socket");

  int *fds = NULL;
  size_t cnt = 0;
  activate("Listeners", &fds, &cnt);
  // fds[0] is your listening socket â€” already bound and
listening
}
```

Socket activation in Go

```
package main

/*
#include <sys/types.h>
#include <sys/socket.h>
#include <dlfcn.h>
#include <stddef.h>

int* activate_socket(const char *name, size_t *cnt) {
    typedef int (*fn_t)(const char*, int**, size_t*);
    fn_t fn = (fn_t)dlsym(RTLD_DEFAULT,
"launch_activate_socket");
    if (!fn) return NULL;
    int *fds = NULL;
    fn(name, &fds, cnt);
    return fds;
}
*/
import "C"
import (
    "net"
    "os"
    "unsafe"
)

func launchdListener(name string) (net.Listener, error) {
```

```
    cname := C.CString(name)
    defer C.free(unsafe.Pointer(cname))
    var cnt C.size_t
    fds := C.activate_socket(cname, &cnt)
    if fds == nil || cnt == 0 {
        return nil, os.ErrNotExist
    }
    fd := *(*C.int)(unsafe.Pointer(fds))
    f := os.NewFile(uintptr(fd), name)
    return net.FileListener(f)
}
```

The socket name `"Listeners"` must match the key in your plist's `Sockets` dict exactly. launchd has already called `bind` and `listen` â€” your binary starts accepting connections immediately.

Pattern 7: Run a shell script

launchd runs binaries directly â€” it does not invoke a shell. If your job is a shell script rather than a compiled binary, you need to invoke the shell explicitly as the program and pass the script as an argument.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>io.uradical.cleanup</string>

  <key>ProgramArguments</key>
  <array>
    <string>/bin/zsh</string>
    <string>/Users/alan/scripts/cleanup.sh</string>
  </array>

  <key>StartCalendarInterval</key>
  <dict>
    <key>Hour</key>
```

```
    <integer>3</integer>
    <key>Minute</key>
    <integer>0</integer>
  </dict>

  <key>StandardOutPath</key>
  <string>/Users/alan/Library/Logs/cleanup/stdout.log</string>

  <key>StandardErrorPath</key>
  <string>/Users/alan/Library/Logs/cleanup/stderr.log</string>
</dict>
</plist>
```

Your script will not have access to anything sourced from `.zshrc` or `.zprofile` â€” the shell launchd invokes is not a login shell and not an interactive shell. If your script depends on tools installed by Homebrew, nvm, rbenv, or similar, either set `PATH` via `EnvironmentVariables` in the plist or source the relevant setup explicitly at the top of the script.

Choosing the right pattern

The network-dependent scheduled job row deserves a note. launchd has no native way to combine `StartCalendarInterval` with a network condition â€” `NetworkState` only works in `KeepAlive`. The practical solution is to let the job run on schedule and perform the network check inside your script or binary, exiting cleanly with code `0` if the network is unavailable. Combined with `KeepAlive SuccessfulExit <false/>`, a clean early exit will not trigger a restart.

Chapter 6: Environment and Paths

If you have ever taken a command that works perfectly in your terminal, put it in a launchd plist, and watched it fail immediately â€” this chapter is why. The environment launchd provides to your job is not your shell environment. It is a minimal, clean slate that knows almost nothing about your user setup. Understanding exactly what is and is not present, and how to supply what is missing, is essential for writing launchd jobs that work reliably.

What launchd provides by default

When launchd starts a job it sets a small number of environment variables automatically. The exact set varies slightly between macOS versions, but you can reliably expect:

That is it. Everything else you rely on in a normal terminal session is absent.

What is missing and why

Your shell startup files â€” `.zshrc`, `.zprofile`, `.bash_profile`, `.bashrc` â€” are not sourced. launchd is not a shell and does not behave like one. It executes your binary directly via `execve`, the same way the kernel executes any process, with no shell intermediary.

This means the following are all absent unless you explicitly provide them:

Homebrew paths. On Apple Silicon, Homebrew installs to `/opt/homebrew/bin`. On Intel it uses `/usr/local/bin`. Neither is in the default launchd `PATH`. Any binary installed via Homebrew â€” `postgresql`, `redis-server`, `python3`, `node`, `ffmpeg` â€” will not be

found unless you add the path explicitly.

Version manager shims. `nvm`, `rbenv`, `pyenv`, `asdf`, and similar tools work by inserting shim directories at the front of your shell `PATH`. None of that is present in a launchd job. If your script calls `node` and expects the `nvm`-managed version, it will either get the system Node or fail with command not found.

Shell functions and aliases. If your script sources or depends on anything defined in your shell configuration, it will not be available.

`GOPATH`**,** `GOROOT`**, and Go toolchain paths.** If you have Go installed via the official installer or Homebrew, the `go` binary is typically at `/usr/local/go/bin/go` or `/opt/homebrew/bin/go`. Neither is in the default PATH.

Application-specific variables. `JAVA_HOME`, `ANDROID_HOME`, `DOCKER_HOST`, `AWS_PROFILE` â€” anything your terminal sets from shell config is absent.

Fixing PATH

The most common fix is setting `PATH` explicitly in `EnvironmentVariables`:

```
<key>EnvironmentVariables</key>
<dict>
  <key>PATH</key>

<string>/opt/homebrew/bin:/opt/homebrew/sbin:/usr/local/bin:/u
sr/bin:/bin:/usr/sbin:/sbin</string>
</dict>
```

If you are on Intel rather than Apple Silicon, replace `/opt/homebrew` with `/usr/local`. If you need to support both architectures in a shared plist, include both paths â€” the non-existent one is harmlessly ignored.

To find the exact path of a binary your job needs, use `which` in your terminal:

```
which node
# /opt/homebrew/bin/node

which python3
# /opt/homebrew/bin/python3
```

Using absolute paths directly

For simple jobs that call a single known binary, the cleanest solution is to bypass `PATH` entirely and use the absolute path in `ProgramArguments`:

```
<key>ProgramArguments</key>
<array>
  <string>/opt/homebrew/bin/node</string>
  <string>/Users/alan/scripts/server.js</string>
</array>
```

This is robust, explicit, and immune to PATH changes. The downside is that it encodes a machine-specific path into your plist. For jobs that only run on your own machine that is usually fine. For plists you intend to share or distribute, setting `PATH` in `EnvironmentVariables` is more portable.

Handling version managers

Option 1: Use the absolute path to the version manager's binary directly.

```
# For nvm
ls ~/.nvm/versions/node/v20.11.0/bin/node

# For pyenv
ls ~/.pyenv/versions/3.12.0/bin/python3
```

Use that path directly in `ProgramArguments` or add it to `PATH` in the plist. This is the most reliable approach.

Option 2: Write a wrapper script that sources the version manager.

```
#!/bin/zsh
# ~/scripts/run-server.sh

export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && source "$NVM_DIR/nvm.sh"

exec node /Users/alan/projects/server/index.js
```

Then invoke the wrapper from your plist:

```
<key>ProgramArguments</key>
<array>
  <string>/bin/zsh</string>
  <string>/Users/alan/scripts/run-server.sh</string>
</array>
```

The wrapper approach works but is fragile â€” if the version manager changes its install location or initialisation method, the wrapper breaks silently. Option 1 is preferable where possible.

Providing other environment variables

```
<key>EnvironmentVariables</key>
<dict>
  <key>PATH</key>

<string>/opt/homebrew/bin:/usr/local/bin:/usr/bin:/bin</string
```

```
>

  <key>GOPATH</key>
  <string>/Users/alan/go</string>

  <key>HOME</key>
  <string>/Users/alan</string>

  <key>APP_ENV</key>
  <string>production</string>

  <key>DATABASE_URL</key>
  <string>postgres://localhost:5432/myapp</string>
</dict>
```

`HOME` is set automatically for user Agents but not for Daemons â€” if your Daemon binary reads `HOME` to find config files, set it explicitly.

Inspecting the actual environment

When debugging a job that fails with "command not found" or a missing variable, the fastest diagnostic is to have the job dump its own environment. Note that `>` must be written as `>` in XML plist files â€” the following is how it appears in the actual file:

```
<key>ProgramArguments</key>
<array>
  <string>/bin/zsh</string>
  <string>-c</string>
  <string>env &gt; /tmp/launchd-env.txt</string>
</array>

<key>RunAtLoad</key>
<true/>
```

This writes every environment variable launchd provides to `/tmp/launchd-env.txt`. Load it, inspect the file, then remove the plist once you have what you need.

Alternatively, `launchctl print` shows the environment variables launchd has configured for a loaded job:

```
launchctl print gui/$(id -u)/io.uradical.devproxy
```

Working directory

Without an explicit `WorkingDirectory` key, your job's working directory is undefined. Set it explicitly and use absolute paths for everything else:

```
<key>WorkingDirectory</key>
<string>/Users/alan/projects/myapp</string>
```

A debugging checklist

1. Check `launchctl print` for the environment launchd is providing
2. Confirm the binary path with `which` in your terminal
3. Add the binary's directory to `PATH` in `EnvironmentVariables`, or use the absolute path in `ProgramArguments`
4. Ensure `WorkingDirectory` is set and any paths in your binary are absolute
5. If using a version manager, find the real binary path under the version manager's versions directory
6. If using a shell script, source any required setup at the top of the script rather than relying on `.zshrc`

Chapter 7: Logging and Debugging

A launchd job that fails silently is one of the more frustrating debugging experiences on macOS. There is no central journal, no single command that shows you what went wrong, and no error surfaced to the desktop when a background service crashes. This chapter covers every tool available for understanding what your job is doing, why it failed, and how to get visibility into its behaviour quickly.

The two logging approaches

launchd does not capture your job's output automatically. You have two options for getting that output somewhere useful:

File-based logging â€” redirect stdout and stderr to files via `StandardOutPath` and `StandardErrorPath` in the plist. Simple, portable, always works.

Unified logging â€” write structured log entries to the macOS unified logging system via `os_log` (in C/Swift) or a compatible library, then query with `log stream` or `log show`. More powerful, integrates with Console.app, but requires your binary to emit logs in the right format.

Most developer jobs use file-based logging. Unified logging is worth knowing for more complex or production-grade services.

File-based logging

Setting up log files

```
<key>StandardOutPath</key>
<string>/Users/alan/Library/Logs/myservice/stdout.log</string>

<key>StandardErrorPath</key>
<string>/Users/alan/Library/Logs/myservice/stderr.log</string>
```

`~/Library/Logs/` is the conventional location for user-scoped service logs on macOS and is where Console.app looks by default. For system Daemons, `/var/log/` is conventional.

Two things to do before loading the plist:

```
mkdir -p ~/Library/Logs/myservice
ls -la ~/Library/Logs/myservice
```

launchd will not create missing directories. If the directory does not exist the log path is silently ignored and your output is discarded.

Tailing log files

```
# Follow stdout in real time
tail -f ~/Library/Logs/myservice/stdout.log

# Follow both files simultaneously
tail -f ~/Library/Logs/myservice/stdout.log \
        ~/Library/Logs/myservice/stderr.log

# Last 50 lines of stderr
tail -n 50 ~/Library/Logs/myservice/stderr.log
```

Log rotation

launchd does not rotate log files. For production-grade services, use `newsyslog` by adding a configuration entry:

```
# /etc/newsyslog.d/myservice.conf
/Users/alan/Library/Logs/myservice/stdout.log    644  7  1024
*  GJ
```

This rotates the log when it reaches 1MB, keeps seven rotated copies, compresses them, and sends `SIGHUP` to the process to reopen the log file.

Unified logging

Writing to the unified log

In Swift:

```
import os

let log = Logger(subsystem: "io.uradical.myservice", category:
"main")
log.info("Service started on port \(port)")
log.error("Connection failed: \(error.localizedDescription)")
```

Querying the unified log

```
# Stream log entries from your service in real time
log stream --predicate 'subsystem == "io.uradical.myservice"'

# Stream with timestamps and process info
log stream \
  --predicate 'subsystem == "io.uradical.myservice"' \
  --info \
  --style compact

# Show historical entries from the last hour
log show \
  --predicate 'subsystem == "io.uradical.myservice"' \
  --last 1h

# Show errors only
log show \
  --predicate 'subsystem == "io.uradical.myservice" AND
messageType == error' \
```

```
  --last 24h
```

Filtering launchd's own log entries

```
log show \
  --predicate 'subsystem == "com.apple.launchd" AND
composedMessage CONTAINS "io.uradical.myservice"' \
  --last 1h
```

This is often the fastest way to find out why a job failed to load â€” launchd logs the specific error here before you have had a chance to set up file logging.

---

launchctl diagnostics

launchctl print

```
launchctl print gui/$(id -u)/io.uradical.myservice
```

Key fields to look at:

â€¢ **state** â€” `running`, `waiting`, `throttled`, or `not running`
â€¢ **pid** â€” present if the job is currently running
â€¢ **last exit code** â€” non-zero means failure
â€¢ **exit count** â€” a high number suggests a crash loop
â€¢ **environment** â€” the environment variables launchd is passing to the job
â€¢ **properties** â€” the active plist configuration as launchd parsed it

launchctl blame

```
launchctl blame gui/$(id -u)/io.uradical.myservice
```

Reports what caused the job to be started. Useful when a job is starting at unexpected times.

launchctl error

Translates a numeric error code into a human-readable description:

```
launchctl error 78
# Operation not permitted
```

Common failure modes and how to diagnose them

Job not starting at all

```
launchctl list | grep myservice
```

If it does not appear, check the unified log for launchd errors:

```
log show \
  --predicate 'subsystem == "com.apple.launchd"' \
  --last 10m
```

Validate your plist syntax before loading:

```
plutil -lint
~/Library/LaunchAgents/io.uradical.myservice.plist
```

Common causes: plist XML is malformed, the binary path does not exist, the log directory does not exist, or the job was previously disabled via `launchctl disable`.

Job starts then exits immediately

Check `launchctl print` for a non-zero exit code, then check stderr:

```
tail -n 50 ~/Library/Logs/myservice/stderr.log
```

Run the binary directly in your terminal to reproduce the error:

```
/usr/local/bin/myservice --config /etc/myservice/config.yaml
```

Job in a crash loop

`launchctl print` will show a high exit count and state may show `throttled`. launchd applies exponential backoff when a job crashes repeatedly. Fix the underlying issue then force a restart:

```
launchctl kickstart -k gui/$(id -u)/io.uradical.myservice
```

Permission denied errors

Check `UserName` in your plist and confirm the binary and config files are readable by that user:

```
ls -la /usr/local/bin/myservice
sudo -u _myservice /usr/local/bin/myservice --dry-run
```

On macOS Ventura and later, binaries may also require explicit privacy permissions granted through System Settings > Privacy & Security.

A systematic debugging workflow

```
# 1. Validate the plist
plutil -lint
~/Library/LaunchAgents/io.uradical.myservice.plist

# 2. Check if the job is loaded and its current state
launchctl list | grep myservice
launchctl print gui/$(id -u)/io.uradical.myservice

# 3. Check launchd's own log for load errors
```

```
log show \
  --predicate 'subsystem == "com.apple.launchd" AND
composedMessage CONTAINS "myservice"' \
  --last 30m

# 4. Check stderr for runtime errors
tail -n 100 ~/Library/Logs/myservice/stderr.log

# 5. Translate any unknown exit codes
launchctl error <exit-code>

# 6. Run the binary directly to reproduce the error
interactively
/path/to/binary --your-args

# 7. Dump the job environment to confirm PATH and variables
# (see Chapter 6 for the env dump plist technique)

# 8. Force a restart after applying a fix
launchctl kickstart -k gui/$(id -u)/io.uradical.myservice
```

Console.app

For developers who prefer a GUI, Console.app provides a visual interface to the unified logging system. Open it, select your Mac under Devices, and use the search bar to filter by your subsystem or label. It is particularly useful for correlating log timestamps with system events â€” network changes, sleep/wake cycles, user login â€” when debugging timing-sensitive jobs.

Chapter 8: Real-World Examples

The previous chapters have covered every concept and command you need. This chapter brings them together in three complete, realistic examples â€” each one the kind of job a developer actually needs, built from scratch with the decisions explained along the way.

Example 1: Auto-start a local development server

The scenario. You are working on a Go API that you run locally during development. Currently you start it manually in a terminal every morning, remember to restart it after crashes, and lose the process when you close the terminal. You want it to start automatically at login, restart on crash, and log output somewhere you can check it without interrupting your work.

Choosing the scope. This is your personal development workflow â€” not a system service, not something other users need. User Launch Agent in `~/Library/LaunchAgents`.

The plist.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>io.uradical.localapi</string>

  <key>ProgramArguments</key>
  <array>
    <string>/Users/alan/go/bin/localapi</string>
    <string>--port</string>
    <string>3000</string>
    <string>--config</string>
```

```
    <string>/Users/alan/.config/localapi/config.yaml</string>
  </array>

  <key>RunAtLoad</key>
  <true/>

  <key>KeepAlive</key>
  <dict>
    <key>SuccessfulExit</key>
    <false/>
  </dict>

  <key>ThrottleInterval</key>
  <integer>10</integer>

  <key>EnvironmentVariables</key>
  <dict>
    <key>PATH</key>

<string>/opt/homebrew/bin:/usr/local/go/bin:/usr/bin:/bin</str
ing>
    <key>HOME</key>
    <string>/Users/alan</string>
    <key>APP_ENV</key>
    <string>development</string>
    <key>DATABASE_URL</key>
    <string>postgres://localhost:5432/localapi_dev</string>
  </dict>

  <key>WorkingDirectory</key>
  <string>/Users/alan/projects/localapi</string>

  <key>StandardOutPath</key>

<string>/Users/alan/Library/Logs/localapi/stdout.log</string>

  <key>StandardErrorPath</key>

<string>/Users/alan/Library/Logs/localapi/stderr.log</string>
</dict>
</plist>
```

Setup and loading.

```
mkdir -p ~/Library/Logs/localapi
plutil -lint ~/Library/LaunchAgents/io.uradical.localapi.plist
launchctl bootstrap gui/$(id -u) \
  ~/Library/LaunchAgents/io.uradical.localapi.plist
launchctl list | grep localapi
tail -f ~/Library/Logs/localapi/stdout.log
```

Key decisions explained.

`SuccessfulExit <false/>` means the server restarts on crash but not on a clean shutdown. `ThrottleInterval 10` prevents a tight crash loop during early development. `DATABASE_URL` in `EnvironmentVariables` avoids hardcoding connection strings in the binary.

Day-to-day workflow.

```
# Rebuild and restart after a code change
go build -o ~/go/bin/localapi ./cmd/api
launchctl kickstart -k gui/$(id -u)/io.uradical.localapi

# Check status
launchctl print gui/$(id -u)/io.uradical.localapi

# Stop for the day (will not restart)
launchctl kill SIGTERM gui/$(id -u)/io.uradical.localapi

# Start again
launchctl kickstart gui/$(id -u)/io.uradical.localapi
```

Example 2: Scheduled database backup

The scenario. You run a local PostgreSQL instance for development and want a nightly backup at 02:00. If the Mac is asleep, the backup runs on wake. If it fails you want a desktop notification so you know before you need the backup.

Choosing the scope. The backup runs as your user, accesses your user-owned database, and writes to your home directory. User Launch Agent.

The backup script.

```
#!/bin/zsh
# ~/scripts/pgbackup.sh

set -euo pipefail

BACKUP_DIR="$HOME/backups/postgres"
TIMESTAMP=$(date +%Y%m%d_%H%M%S)
BACKUP_FILE="$BACKUP_DIR/backup_${TIMESTAMP}.dump"

notify() {
  osascript -e "display notification \"$1\" with title
\"pgbackup\""
}

mkdir -p "$BACKUP_DIR"

if ! /opt/homebrew/bin/pg_dump \
  --format=custom \
  --file="$BACKUP_FILE" \
  localapi_dev; then
  notify "Backup FAILED â€” check
~/Library/Logs/pgbackup/stderr.log"
  exit 1
fi

find "$BACKUP_DIR" -name "*.dump" -mtime +14 -delete

echo "Backup complete: $BACKUP_FILE"
notify "Backup complete: backup_${TIMESTAMP}.dump"
```

```
chmod +x ~/scripts/pgbackup.sh
```

The plist.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>io.uradical.pgbackup</string>

  <key>ProgramArguments</key>
  <array>
    <string>/bin/zsh</string>
    <string>/Users/alan/scripts/pgbackup.sh</string>
  </array>

  <key>StartCalendarInterval</key>
  <dict>
    <key>Hour</key>
    <integer>2</integer>
    <key>Minute</key>
    <integer>0</integer>
  </dict>

  <key>EnvironmentVariables</key>
  <dict>
    <key>PATH</key>
    <string>/opt/homebrew/bin:/usr/bin:/bin</string>
    <key>HOME</key>
    <string>/Users/alan</string>
    <key>PGHOST</key>
    <string>localhost</string>
    <key>PGUSER</key>
    <string>alan</string>
  </dict>

  <key>StandardOutPath</key>

<string>/Users/alan/Library/Logs/pgbackup/stdout.log</string>

  <key>StandardErrorPath</key>

<string>/Users/alan/Library/Logs/pgbackup/stderr.log</string>
</dict>
</plist>
```

Setup and loading.

```
mkdir -p ~/Library/Logs/pgbackup
mkdir -p ~/backups/postgres
plutil -lint ~/Library/LaunchAgents/io.uradical.pgbackup.plist
launchctl bootstrap gui/$(id -u) \
  ~/Library/LaunchAgents/io.uradical.pgbackup.plist
```

Testing without waiting for 02:00.

```
launchctl kickstart gui/$(id -u)/io.uradical.pgbackup
tail -f ~/Library/Logs/pgbackup/stdout.log
ls -lh ~/backups/postgres/
```

Key decisions explained.

`set -euo pipefail` means any command failure causes the script to exit non-zero. The explicit `if ! pg_dump` block catches that failure and fires a desktop notification via `osascript`. `osascript` notifications work because this is a user Agent running inside the login session with display server access â€” the same approach would silently fail in a Launch Daemon. No `KeepAlive` â€” this job runs, completes, and exits.

Example 3: On-demand local reverse proxy

The scenario. You are developing a service that must be accessed over HTTPS locally â€” perhaps for webhooks from a third-party service that requires HTTPS callbacks. You want a lightweight reverse proxy that terminates TLS locally and forwards to your development server, starting only when something connects.

Choosing the scope. Developer tooling, user session, no elevated privileges needed. User Launch Agent with socket activation.

The tool. This example uses Caddy â€” a widely used web server and reverse proxy with native socket activation support.

```
brew install caddy
```

The Caddy configuration.

```
# ~/.config/caddy/localproxy.caddy
{
  auto_https off
}

localhost:8443 {
  tls /Users/alan/.config/caddy/localhost.crt \
      /Users/alan/.config/caddy/localhost.key
  reverse_proxy localhost:3000
}
```

Generate a locally-trusted certificate with `mkcert`:

```
brew install mkcert
mkcert -install
mkdir -p ~/.config/caddy
mkcert \
  -cert-file ~/.config/caddy/localhost.crt \
  -key-file  ~/.config/caddy/localhost.key \
  localhost 127.0.0.1
```

The plist.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>io.uradical.localproxy</string>

  <key>ProgramArguments</key>
  <array>
    <string>/opt/homebrew/bin/caddy</string>
    <string>run</string>
    <string>--config</string>

<string>/Users/alan/.config/caddy/localproxy.caddy</string>
```

```
  </array>

  <key>Sockets</key>
  <dict>
    <key>Listeners</key>
    <dict>
      <key>SockServiceName</key>
      <string>8443</string>
      <key>SockType</key>
      <string>stream</string>
      <key>SockFamily</key>
      <string>IPv4</string>
    </dict>
  </dict>

  <key>EnvironmentVariables</key>
  <dict>
    <key>PATH</key>
    <string>/opt/homebrew/bin:/usr/bin:/bin</string>
    <key>HOME</key>
    <string>/Users/alan</string>
  </dict>

  <key>StandardOutPath</key>

<string>/Users/alan/Library/Logs/localproxy/stdout.log</string
>

  <key>StandardErrorPath</key>

<string>/Users/alan/Library/Logs/localproxy/stderr.log</string
>
</dict>
</plist>
```

Setup and loading.

```
mkdir -p ~/Library/Logs/localproxy
plutil -lint
~/Library/LaunchAgents/io.uradical.localproxy.plist
launchctl bootstrap gui/$(id -u) \
  ~/Library/LaunchAgents/io.uradical.localproxy.plist
launchctl print gui/$(id -u)/io.uradical.localproxy
```

Key decisions explained.

No `RunAtLoad` â€” Caddy only starts when a connection arrives on port 8443. launchd holds the socket open continuously so the port is always reserved. Caddy is a real, named tool with well-documented socket activation behaviour â€” predictable and tested. For production use on a shared machine you would move this to a Launch Daemon and drop privileges, but for a developer laptop a user Agent is simpler and sufficient.

What these examples have in common

All three use absolute paths throughout â€” in `ProgramArguments`, in `EnvironmentVariables`, in script paths. No relative paths appear anywhere.

All three create their log directories before loading the plist. This is a setup step that is easy to forget and hard to diagnose when missed.

All three validate with `plutil -lint` before loading. This catches XML errors before launchd sees the file.

All three use `KeepAlive` selectively and deliberately â€” the API server uses crash recovery, the backup job uses none, the proxy uses socket activation. The choice reflects what each service actually needs rather than a default.

None of them rely on the shell environment being present. Each plist is self-contained: every path, variable, and configuration value it needs is declared explicitly.

These are not just rules for launchd â€” they are the habits that make any background service debuggable, reliable, and maintainable over time. Chapter 9 compresses all of that into a single quick reference you can reach for without re-reading the guide.

Chapter 9: Quick Reference

File locations

plist skeleton

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>

  <key>Label</key>
  <string>io.yourname.servicename</string>

  <key>ProgramArguments</key>
  <array>
    <string>/absolute/path/to/binary</string>
    <string>--flag</string>
    <string>value</string>
  </array>

  <key>RunAtLoad</key>
  <true/>

  <key>KeepAlive</key>
  <dict>
    <key>SuccessfulExit</key>
    <false/>
  </dict>

  <key>ThrottleInterval</key>
  <integer>10</integer>

  <key>EnvironmentVariables</key>
  <dict>
```

```
    <key>PATH</key>

<string>/opt/homebrew/bin:/usr/local/bin:/usr/bin:/bin</string
>
  </dict>

  <key>WorkingDirectory</key>
  <string>/absolute/path/to/workdir</string>

  <key>StandardOutPath</key>

<string>/Users/you/Library/Logs/servicename/stdout.log</string
>

  <key>StandardErrorPath</key>

<string>/Users/you/Library/Logs/servicename/stderr.log</string
>

</dict>
</plist>
```

Key plist keys

Identity

What to run

When to run

Restart behaviour

Environment

Logging

On-demand

StartCalendarInterval values

Omitted keys are wildcards. Missed runs fire on next wake.

---

launchctl commands

Load and unload

```
# Load a user Agent
launchctl bootstrap gui/$(id -u)
~/Library/LaunchAgents/<label>.plist

# Load a system Daemon
sudo launchctl bootstrap system
/Library/LaunchDaemons/<label>.plist

# Unload by path
launchctl bootout gui/$(id -u)
~/Library/LaunchAgents/<label>.plist

# Unload by identifier
launchctl bootout gui/$(id -u)/<label>

# Reload after plist change
launchctl bootout gui/$(id -u)/<label> 2>/dev/null && \
launchctl bootstrap gui/$(id -u)
~/Library/LaunchAgents/<label>.plist
```

Start and stop

```
# Start immediately
launchctl kickstart gui/$(id -u)/<label>

# Start and print PID
launchctl kickstart -p gui/$(id -u)/<label>

# Force restart if already running
launchctl kickstart -k gui/$(id -u)/<label>

# Send signal
launchctl kill SIGTERM gui/$(id -u)/<label>
launchctl kill SIGHUP  gui/$(id -u)/<label>
launchctl kill SIGINT  gui/$(id -u)/<label>
```

Enable and disable

```
launchctl enable  gui/$(id -u)/<label>
launchctl disable gui/$(id -u)/<label>
```

Inspect

```
# List all user jobs with PID and exit status
launchctl list

# Filter by name
launchctl list | grep <label>

# Full service state
launchctl print gui/$(id -u)/<label>

# Why did this job last start?
launchctl blame gui/$(id -u)/<label>

# Translate an exit code
launchctl error <code>

# Full launchd state dump
```

```
launchctl dumpstate | grep -A 20 "<label>"
```

Domains

Logging and diagnostics

```
# Validate plist before loading
plutil -lint ~/Library/LaunchAgents/<label>.plist

# Tail log files
tail -f ~/Library/Logs/<service>/stdout.log
tail -f ~/Library/Logs/<service>/stderr.log

# Stream unified log for your service
log stream --predicate 'subsystem == "io.yourname.service"'

# Show last hour of unified log
log show \
  --predicate 'subsystem == "io.yourname.service"' \
  --last 1h

# Show launchd errors for your job
log show \
  --predicate 'subsystem == "com.apple.launchd" AND
composedMessage CONTAINS "<label>"' \
  --last 30m
```

Debugging checklist

1. `plutil -lint` â€” validate plist XML before loading
2. `launchctl list | grep <label>` â€” confirm job is loaded
3. `launchctl print gui/$(id -u)/<label>` â€” check state, exit code, exit count

4. Check stderr log for runtime errors
5. `launchctl error <code>` â€” translate non-zero exit codes
6. Run the binary directly in terminal to reproduce errors interactively
7. Dump the job environment to confirm PATH and variables are correct
8. `launchctl kickstart -k` after applying a fix

Common mistakes

Legacy command mapping

Shell helper function

```
# Add to ~/.zshrc
function launchctl-reload() {
  local label="$1"
  local plist=~/Library/LaunchAgents/${label}.plist
  launchctl bootout gui/$(id -u)/${label} 2>/dev/null
  launchctl bootstrap gui/$(id -u) "${plist}"
  echo "Reloaded ${label}"
}
```