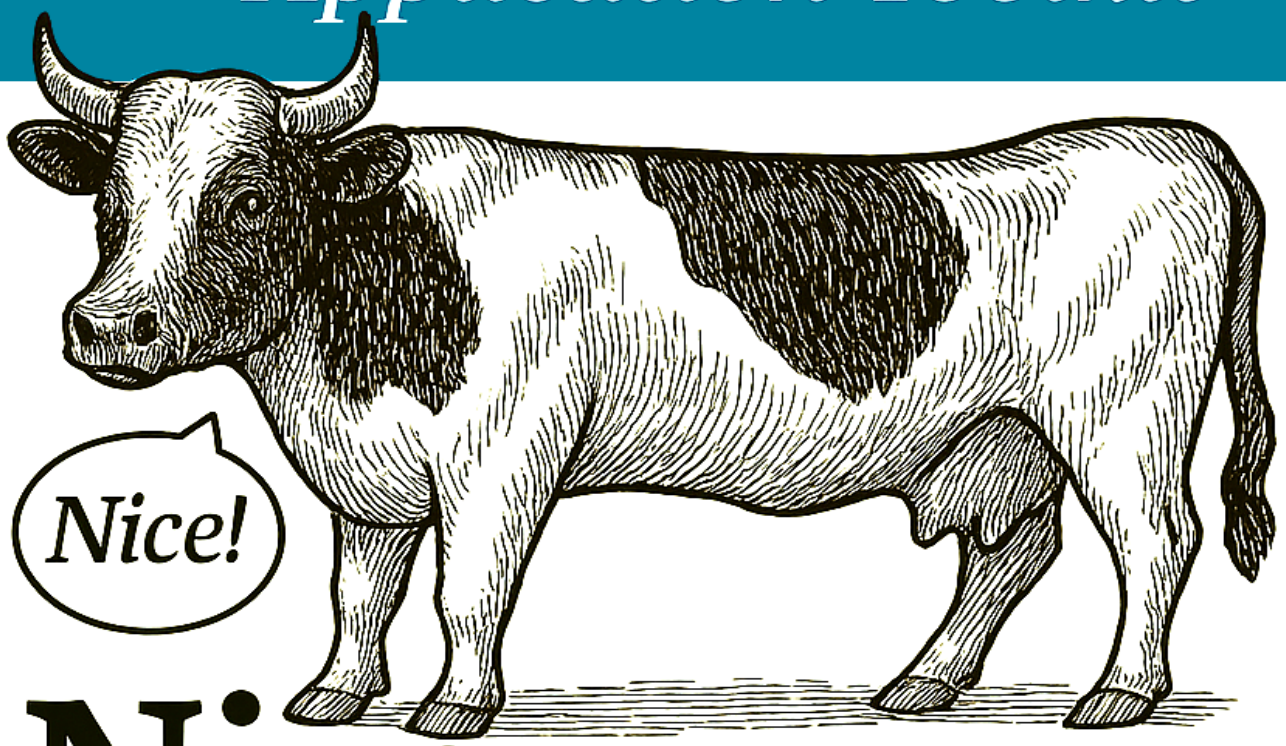
MEAT

Mitt Enhanced Application Toolkit



Nice to Meat You

Luciano Federico Pereira

Table of Contents

Acknowledgments

To everyone who helped shape this project, commit by commit.
To every student who asked the hard questions.
To the community that builds in public and shares the fix before the bug report.

And to those closest to me — Valeria, Lucía, Renata, Joaquín, Alan, Gonzalo, Raúl, thank you for the space to think, the encouragement to finish, and the faith behind the lines of code.

— L.F.P.

Preface

If you're reading this, chances are you've just discovered Meat — or you're still deciding whether it deserves a spot in your toolkit.

This book is a compact guide for developers who want the facts, fast. It’s not here to evangelize reactive programming or reinvent how state works in JavaScript. Instead, it focuses on one goal: to help you write your first 100 meaningful commits with Meat, and build solid mental models along the way.

You’ll find examples before theory, code before prose. There’s no lengthy backstory or poetic metaphors—just what you need to understand the library’s structure, idioms, and behaviors. Like the best tools from that era, Meat favors simplicity, stability, and clarity under load.

Whether you're hacking in a terminal, tweaking a browser app, or piping signals between services, this book is here to get out of your way and get you to the code.

We’ve packed it into 100 pages, each tuned to support your first 100 commits. You’ll find practical signal patterns, rollback strategies, plugin scaffolds, and reactive recipes—enough to start strong and refactor with confidence.

Meat doesn't ask for loyalty. It asks for curiosity.

Now open the editor. We’ll keep the docs clean and the examples sharp.

About the Author

Luciano Federico Pereira is a senior software engineer, mentor, and licensed IT lawyer with over two decades of professional experience. Originally from Buenos Aires and now based in Europe, he has contributed to SaaS platforms, legal systems, and developer bootcamps—bridging product delivery with team growth.

He specializes in building reactive interfaces, mentoring junior engineers, and designing tools that balance clarity with composability. His past roles have spanned engineering, product ownership, and education—giving him a cross-functional approach to collaboration and problem-solving.

This book was self-funded and independently researched to support the developer community and reflect his current active focus: signal-based architecture, plugin-driven composition, and scaled state management.

Contact: lucianopereira@posteo.es
LinkedIn: in/lucianofedericopereira

Commitment: Read This First

This section lives before the introduction, before anything else.

Why? Because it's too useful not to read first.

100% Real No Fake: Like a README.md in a repo you cloned at 2 a.m., only better.

Before you start flipping pages or inspecting signals, take a moment here.

This matrix was built so you don't waste time guessing what page solves your actual problem.

We called it "Commitment" because this book isn't just a reference—it's a developer promise. Every commit range below represents a phase you'll hit as you build with MEAT: from signal basics to DEFCON-grade failsafes.

Commits 0–10: Signal Fundamentals

Pages 22–26

Understand the signal engine: how state flows, reacts, and rolls through your app.

- Core API: `get`, `set`, `subscribe`, `watch`, `once`, `merge()`
- DOM binding, listener teardown

Look Forward to:

- Pages **66–69** → Advanced signal chaining & visibility sentinels
- Page **73** → Snapshot isolation: read-only state derived from those same get/set flows
- Page **97** → Meat Signal Cheatsheet: summary of signal methods and teardown ops
- Page **98** → Signal Patterns: real-world analogies of reactive behavior

Commits 10–30: Composition & Injection

Pages 30–56

Extend MEAT using plugins, diagnostics, and reactive extensions.

- Plugin injection, mutation counter, logger
- Persist state, style UI with signal data
- Input reset + mirror binding

Look Back to:

- Pages **22–26** → You're building on top of the signal engine

Look Forward to:

- Pages **62–69** → Scoped plugin composition + lifecycle
- Page **70** → Circuit breaker logic mirrors your logger-style pattern
- Page **97** → Meat Signal Cheatsheet: plugin and injection references at a glance
- Page **98** → Signal Patterns: intuitive mapping of plugin strategies

Commits 30–60: Sync & Reactive Flows

Pages 57–65

Scale your app's state across tabs, storage, and remote systems.

- Form autosave + offline backoff
- BroadcastChannel and signal syncing
- Devtools overlay + component loader

Look Back to:

- Pages **30–40** → Local input sync sets the foundation for remote federation

Look Forward to:

- Pages **71–73** → WebSocket safety flows echo your tab-to-tab sync logic
- Page **75** → Plugin cleanup strategies mirror teardown in dynamic loaders

- Page **98** → Signal Patterns: mutation & sync metaphors for scale

Commits 60–90: Advanced Signal Patterns

Pages 66–69

Complex signal orchestration for serious builders.

- Scoped plugin injection, visibility sentinel
- Graph watchers, signal pagination
- Timeboxed revert flows

Look Back to:

- Pages **22–26** → Signal methods used recursively in watchers
- Page **54** → Listener teardown pattern resurfaces with advanced guards

Look Forward to:

- Pages **72–73** → GDPR scrub + safety flows are plugin-driven variants of scoped injection
- Page **98** → Signal Patterns: decision-based abstractions for advanced flows

Commits 90–100: DEFCON 1 Safety Flows

Pages 70–73

For mission-critical logic, fail-safe reactivity, and release polish.

- Atomic rollback, GDPR scrub, WebSocket failsafe
- Quiz logic, commit index, final polish

Look Back to:

- Page **53** → Your rollback strategy builds on `meat.safe()`
- Pages **55–56** → State serialization underpins snapshot recovery
- Pages **60–65** → Broadcast & sync flows directly inform WebSocket logic

Problem-solving: Developer-First Architecture

Forged in Commits. Built for Builders. Battle Tested.

Meat isn’t an idea—it’s a response. A response to the mental overhead, boilerplate rituals, and architectural noise that plague everyday development. If you’ve ever had to wrap three layers of logic just to update a key, or explain why your component re-rendered because some context somewhere shifted—Meat is your antidote.

Too many tools try to abstract everything all at once: rendering engines, hydration flows, templating languages, build pipelines. That’s fine for scale, but not every app is a startup IPO. Sometimes you just want to build. Sometimes you need clarity without ceremony. Precision without persuasion. A tool that respects your intelligence—and lets you ship.

Meat was built for that context. The early commits. The real work. The moment where structure is still fluid, but your logic needs to land. It doesn’t care what your view layer is, or whether you’re using JSX, Vue templates, or none of the above. It was designed developer-first. You read the source in a single sitting. You trace every mutation. You hook into events without lifecycle gymnastics. You understand the runtime as you use it.

Because the best tools don’t try to replace you—they amplify what you already know.

Inspired by Mitt. Built to Go Further

Meat didn’t just borrow a name—it inherited a spirit.

Mitt is one of the few libraries that cut through the noise. An event emitter so lean and practical, even developers with wildly different philosophies agree: it’s one of the good ones. No lifecycle jargon. No abstraction layers. Just a clean API that listens and dispatches.

That attitude is in Meat’s DNA. But Meat isn’t a tribute—it’s a leap.

Where Mitt offers signal and response, Meat offers mutation, observation, and extension. It doesn't wrap Mitt—it integrates its core mechanisms and then builds a reactive system around them. Internally, Meat uses the same architectural pattern: flat Map tracking, synchronous dispatch, listener registration with zero side effects. That part is Mitt—pure, fast, and familiar.

But Meat turns the emitter into a reactive engine. Scoped updates, state introspection, rollback strategies, and plugin sandboxing don't exist in Mitt's universe—because Mitt was never meant for that. Meat steps in where Mitt leaves off.

What makes Meat different:

- **Scoped reactivity**: Events aren't just global—they're traceable by key. You listen to specific data flows, not abstract channels.
- **State engine**: Meat binds events to real-time, inspectable data. You don't just emit—you mutate, watch, and serialize.
- **Rollback support**: Snapshots and state history are native. If something goes sideways, you rewind.
- **Plugin surface**: Mitt is intentionally unextendable. Meat embraces controlled expansion—with access gating and conflict detection.
- **Devtools**: Observability is first-class. No need to install anything external—Meat reports what it does, when it does it.

They're not competitors. They're complementary. You can use Mitt for project-wide broadcasts and use Meat for scoped state-driven logic. They speak the same language. Same dispatch model. Same respect for developer control. If Mitt is a whisper, Meat is a voice with intent. It's signal + payload + logic + history—all in one reactive layer.

But make no mistake—Meat builds its own world. A world with state, precision, and extension. A toolkit that doesn't just listen—it understands.

Not a framework

Meat is not a framework. It's a reactive toolkit—a deliberate middle ground between micro-libraries like Mitt and macro-frameworks like Redux, Vuex, Zustand, or NanoStores. It was built to solve a recurring pain: managing state and events in JavaScript without bundling rendering logic or opinionated architecture.

Built for Builders

Building tools for people who ship—not just speculate.

Developers aren’t looking for doctrine. They want grip. They want tools that move when the code gets messy and deadlines get loud. What a dev wants is clarity. What a dev needs is unblocked access—to mutate, observe, extend, and ship without friction.

Meat doesn’t sell architecture. It delivers leverage:

- Mutation tools that don’t need reducers or dispatchers.
- Scoped events and wildcards that fire exactly where you need them.
- Plugins that stay boxed unless invited—and clean up after themselves.
- Serialization and rollback built into the runtime, not tacked on.
- Devtools that require zero setup, and answer what changed, when, and why.

No templates. No hydration strategy. No JSX. No context providers. You don’t inherit someone else's stack. You build yours—with total control.

Because abstraction fatigue is real. Because five layers to move data from A to B is four too many. Because developer-first design isn’t just helpful—it’s essential.

Meat doesn’t ask you to believe. It lets you build. Let’s open the payload.

Autonomy Over Abstraction

You control everything in Meat:

- Whether state is mutable or immutable
- How events are scoped, emitted, and intercepted
- What plugins operate, and which APIs they touch
- How you serialize, persist, and inspect state
- Whether you drop it in via CDN, install it with npm, or link it to a DOM structure

Meat is stack-agnostic, extension-friendly, and runtime-clear. It doesn’t push you toward a particular rendering model. It doesn’t wrap your app in magic.

It operates beneath your code—not over it.

No magic wrappers. No lifecycle interference. Meat runs as a runtime—not a religion.

This is what developer-first truly means: No build step. No scaffolding. No hydration pipelines waiting to break. Just predictable logic wired through state, event, and extension—all exposed, all inspectable.

Clarity isn't just a design goal—it’s a runtime trait. You know what `set()` does. You see how `watch()` reacts. You control whether mutations clone or write in place.

Meat doesn't abstract—it amplifies. It hands you primitives that behave like JavaScript should: synchronous, direct, debuggable.

Tools before theory. Mutation before middleware. Events before lifecycle.

And beneath it all, a reactive core engineered to stay simple under pressure. Less ceremony. More signal. Code that commits clean because the runtime doesn’t fight back.

Meat doesn't ask you to believe in architecture. It lets you build with intent.

Engineered for the First 100 Commits

Meat is built for the moment ideas become implementation. When concepts meet keyboards. When architecture is still fluid—but functionality needs to land.

You drop Meat into a project, and within minutes, you’re wiring state, emitting events, and tracing behavior. No scaffolding. No config wizard. Just code that moves when you do.

From `set()` and `watch()` to `subscribe()` and `rollback()`, the primitives are intuitive and reliable. You write logic like you would in plain JavaScript, and Meat sits underneath it like an observant layer—not a dominant one.

It’s not about how much you can do. It’s about how fast you can understand what’s already happening.

Meat vs The Ecosystem

Every tool in the JavaScript ecosystem makes tradeoffs. Some offer deep integrations at the cost of portability. Others strip down for minimalism but leave you wiring critical features by hand. Meat aims for a different balance—precise control without overhead, architecture without assumption.

Redux

- Strong architecture built on reducers, actions, and middleware
- Requires multiple layers to mutate even a single key
- Immutable-only state; mutations must be expressed via reducers
- Plugin support via middleware, but configuration is verbose
- External devtools, powerful but detached
- High learning curve and setup friction
- Bundle size around 20KB; memory cost ~120MB at scale

Vuex

- Seamless integration with Vue’s engine and lifecycle
- Reactive but tightly coupled to Vue templates and context
- State mutations only via commits; heavy framework assumptions
- Plugins and events embedded in Vue’s own system
- Devtools support through Vue ecosystem only

Zustand

- Built for React via hooks and context providers
- Mutable state and direct access feel ergonomic
- Event system absent; relies solely on reactive patterns
- Plugins possible but often handcrafted
- Moderate learning curve; limited portability

NanoStores

- Ultralight (~1KB), with push-based updates
- Multi-framework compatible
- Manual plugin architecture; less built-in support
- Few debugging features unless you build them yourself
- Lean memory footprint (~50MB), minimal setup friction

Meat's Feature Profile

Here's what you get out of the box:

- Bundle size around 3KB
- Setup time under one minute
- Event dispatch latency: ~0.3ms
- Memory footprint with 10k keys: ~40MB
- Direct `set()` and `get()` mutation without ceremony
- Scoped listeners with watch() and once()
- Global subscribe() for broadcast reactivity
- Native serialization and rollback
- Plugin system with sandboxed access and conflict detection
- Devtools and DOM-binding plugins optional, but built-in

Meat's runtime is inspectable, traceable, and small. You grasp the full system in one sitting. That's not a coincidence. That's clarity by design.

All Meat. No Beef.

No drama. No code of conduct. No architecture debates in your pull requests. Just a reactive core that works—then steps aside.

Don't agree with the structure? Extend it. Fork it. Build a plugin. As long as it emits messages, it plays nice. You don't have to rewrite the core to modify the behavior. You don't need to negotiate with the runtime to build on it.

This is precision software with zero fluff. Minimal surface area. Predictable behavior. Honest ergonomics. It doesn't make noise. It makes progress.

Why Meat Wins

- You read and understand the runtime in under 10 minutes
- You mutate state directly—no dispatch indirection
- You observe changes with scoped or global listeners
- You serialize, persist, and rollback without extra libraries
- You extend behavior without violating boundaries

For Use: Not for Praise

Meat doesn’t perform. It doesn’t posture. It doesn’t sell abstraction dressed as innovation. It exists to be used. To move signal from thought to runtime without detour. To act as infrastructure without imprinting opinion.

You don’t have to tweet about it. You don’t have to convert your codebase. You don’t even have to remember its name. You just use it—and it behaves.

It doesn’t compete with your stack. It sharpens it. Quietly. Reliably. Predictably.

Tools like this don’t rise by branding. They rise because the fifth time you use `watch()`, it works exactly as expected. And the tenth time you mutate state without ceremony, you stop asking if the tool will handle it. You know it will.

Meat is runtime built with respect for yours. It earns trust the same way your code does — *by working*.

Meat doesn’t ask for loyalty. It earns trust—quietly, consistently—by staying out of your way and inside your control. It doesn’t challenge your stack. It doesn’t try to replace your tools. It sharpens them.

You don’t adopt Meat. You wield it. Like a favored utility in your terminal, it shows up when things get real. When the abstraction starts leaking. When the framework starts framing your decisions. When all you need is a signal that responds, a state that behaves, and a runtime that doesn’t second-guess your architecture.

Drop it into a vanilla JS project, and it behaves like you wrote it yourself. Pair it with React, Vue, Svelte, whatever—and it won’t object. It doesn’t wrap or inject. It doesn’t bind to a lifecycle. It emits. It reflects. It persists when you tell it to. And it leaves when you unbind.

This isn’t a campaign for developer hearts. It’s a commitment to developer hands. Extensible only when you choose to extend.

Because when deadlines compress, complexity balloons, and tooling gets loud, Meat stays quiet. Reactive where you need it. Transparent where you inspect it. Extensible only when you choose to extend.

Introduction. Nice to Meat You

JavaScript is not short on frameworks. From the ubiquitous React and Vue to micro-libraries like Alpine and Nano, the modern developer has no shortage of architectural opinions competing for mindshare. Yet amid this noise, the simplest tools often fall through the cracks.

Meat, the Mitt Enhanced Application Toolkit, is a lightweight, reactive state manager and event bus with plugin extensibility. It's not trying to reinvent the DOM or abstract away your business logic. It doesn't come with JSX, a virtual DOM, or custom compiler transforms. What it does offer is precision—a well-defined set of methods centered around predictable state updates, scoped event handling, and opt-in mutability. It's roughly 300 lines of code, intentionally so.

This book is a practical guide to understanding Meat's internals, using it effectively, and extending it responsibly. If you want a deeper understanding of how small tools can yield big architectural wins, you're in the right place.

We'll begin by exploring Meat's architectural core: the state object and the eventBus system. From there, we'll build up toward plugin authoring, state serialization, and debugging workflows. You'll learn how to incorporate Meat into real applications with minimal friction—and how to do so without relying on build tools or transpilation. Each concept is grounded in annotated source and concrete examples.

You won't find a toy framework or throwaway experiment here. Meat is opinionated, yes—but those opinions are rooted in runtime clarity and developer autonomy. If your favorite part of Redux was combineReducers, and your least favorite was everything else, you'll probably enjoy what Meat has to offer.

This book assumes you're comfortable with JavaScript and have built at least one client-side application using any framework. You don't need experience with compiler internals or state machines—though you may develop an appreciation for both by the end.

Welcome to Meat. It's not here to replace your stack. It's here to sharpen it.

Why Meat Exists

Meat was created to address a recurring frustration in frontend development: the growing complexity of state and event handling in small to mid-sized applications. While full-featured frameworks offer powerful abstractions, they often demand significant mental overhead, configuration, and runtime costs—especially when the problem at hand is modest.

In many cases, developers simply need a reactive store, scoped events, and a plugin system to extend functionality. That's it. No templates, no JSX, no virtual DOM diffing. **Meat exists to be exactly that: a focused tool that does one thing well without introducing architectural debt.**

The inspiration behind Meat stems from the simplicity of libraries like Mitt—a tiny event emitter—and a desire to apply that clarity to state mutation and data flow. Unlike more elaborate patterns such as Flux or Redux, Meat favors **direct API access** and **observable state transitions**, giving developers both power and visibility.

Meat is:

- **Minimal**: Under 300 lines of code with zero dependencies.
- **Modular**: You can extend it with plugins, but nothing is enforced.
- **Transparent**: Every method and mutation is traceable and inspectable.
- **Flexible**: Mutable or immutable state, runtime configurable.
- **Accessible**: Drop into a page via CDN, or install via npm—no build step required.

The goal isn't to compete with established ecosystems. It's to provide a precision tool for developers who want control without clutter. If your project needs state logic, event orchestration, and plugin extensibility—but not templating engines or hydration strategies—Meat may be the most practical choice.

Core philosophy

Meat is built on three foundational values: **clarity**, **control**, and **minimalism**. Each principle informs the framework's design decisions, internal architecture, and usage patterns. Collectively, they aim to provide developers with tools that are easy to reason about, powerful when needed, and unobtrusive by default.

Clarity

Meat's API is deliberately small and semantically focused. Each method name reflects its intent, and there is no hidden behavior behind abstraction layers. State updates emit events, event handlers execute synchronously, and plugin APIs expose only what's explicitly permitted.

Source code transparency is a core concern—Meat is designed to be fully readable in one sitting. Developers who use the framework should feel capable of understanding the entire codebase without needing external tooling or documentation.

Control

Meat favors explicit state transitions over magical reactivity. While reactive updates are supported, they are scoped and predictable. State is either mutable or immutable, depending on configuration. Plugins extend functionality, but cannot override internal behavior unless intentionally permitted.

Event listeners are user-defined and modular. Subscriptions can be scoped to keys, executed once, or globally intercepted. This provides granular control over how your application responds to data changes.

Minimalism

Meat avoids bundling features that are outside its scope. It doesn't handle routing, templating, or UI rendering. Instead, it offers a reactive core that you can pair with your preferred view layer or use standalone in vanilla JavaScript environments.

With no external dependencies and no compilation step, Meat can be included directly via a `<script>` tag or installed via npm. Its footprint is minimal, and its cognitive overhead is intentionally low—allowing developers to focus on their application logic rather than framework intricacies.

By aligning with these principles, Meat aims to serve developers who value architectural simplicity and runtime transparency. It's not the loudest tool in the room. It's the clearest.

What You'll Learn

This book is written for developers who want precise control over state, events, and data flow in JavaScript applications—without relying on large frameworks or toolchains. By working through the chapters, you'll gain a thorough understanding of Meat's architecture, methods, and plugin interface, enabling you to use it effectively in a range of contexts. You'll learn:

- How Meat structures and isolates state, and how to configure it for mutable or immutable updates.
- How the event system works behind the scenes, including scoped listeners, wildcard emitters, and reactivity patterns.
- How to serialize, inspect, and persist application state with minimal overhead.
- How to use built-in plugins for DOM binding and localStorage persistence.
- How to write your own plugins that safely extend Meat's functionality without creating naming conflicts.
- How to debug and audit your application state using Meat's devtools and introspection methods.
- How to incorporate Meat into small JavaScript projects with no build step or transpilation.

Each chapter is self-contained and focused on a specific part of the framework, with annotated code examples and practical advice. By the end of this book, you'll be fluent in the Meat API and equipped to decide when and how to integrate it into your own development workflow.

Meat is not a general-purpose solution, nor is it a teaching tool disguised as a framework. It's a deliberate experiment in clarity and modularity. This book treats it as such—with attention to its design, its limitations, and its possibilities.

> **Open Source**
>
> **MEAT** is an open-source tool, and you can contribute to the project by joining the MEAT GitHub repository.

Core Architecture

Where abstraction melts. And signal takes shape.

If Chapter I was conviction in words, this is metal in motion. Meat isn't a pattern—it's a weapon. A precision tool, sharp in scope, and deadly in delivery. You don't architect it like a palace. You wield it like a blade.

Meat wasn't carved out of architectural purity. It was forged under real-world pressure—where structure is still fluid, but intent must land. That moment when you're building alone, debugging under deadline, or shaping logic you'll have to explain six months from now. Meat exists for that moment. The commits that happen before the debate. The code before the ceremony. It operates in that narrow space where clarity beats convention—and where the runtime must be an ally, not an obstacle.

Its internals reflect clarity at every turn. No hidden lifecycles. No tangled dependencies. No abstracted reactivity trees. What you see is what you control. And what you control responds exactly as expected.

Here's how the forge burns.

The State Engine

Optional immutability. Permanent clarity.

State lives as a flat key-value object. There's no reactivity theater, no nested proxies, no lifecycle negotiations. You own the shape and the behavior.

Mutation style is up to you:

- Immutable mode (`mutable: false`) — default safety. Every update clones the state, maintaining predictability and functional integrity.
- Mutable mode (`mutable: true`) — direct control. Changes happen in-place, perfect for lean flows and fast iterations.

Meat doesn't enforce purity. It offers precision. You choose the metal, you swing the hammer.

Methods to mold and inspect:

- `getState()` — full snapshot.
- `set(key, value)` — mutate and fire scoped event.
- `setState(updates)` — batch apply.
- `merge(obj)` — alias for fluid usage.
- `clear()` / `reset()` — purge or restore.
- `serialize()` — cached JSON string.
- `select(keys)` — filtered slices.
- `has(key)`, `hasChanged(key, value)` — inspection tools.
- `keys()`, `values()` — visibility.
- `find(fn)` — scoped key filtering.
- `dump()` — from persist plugin.
- `inspectKey(key)` — value + watch status.
- `isEmpty()` — forge idle check.

Built-in emissions:

- `meat:update:{key}` — scoped reactivity.
- `meat:update` — full-state broadcast.

Event Bus

Synchronous. Scoped. Predictable. Meat's pub/sub system follows Mitt's discipline but expands it into scoped reactivity and wildcard observability. Built on Map. Dispatch is synchronous, and feedback is traceable.

Listener tools:

- `on(type, fn)` — subscribe to signal.
- `emit(type, payload)` — fire.
- `off(type, fn)` — disconnect.
- `onAny(fn)` — wildcard listener.
- `listeners()` — inspect active channels.
- `isWatched(type)` — scoped presence check.
- `unbindAll()` — global teardown.

Reactivity flows:

- `watch(key, fn)` — scoped to data.
- `once(key, fn)` — single-use and auto-disconnect.
- `subscribe(fn)` — global heartbeat.

Handlers throw? You'll know. `HANDLER_ERROR` and `WILDCARD_ERROR` show up in console—assuming `config.debug: true`.

Plugin System

Extend without bleed. Plugins in Meat don't hijack the runtime—they're scoped, permissioned, and auditable.

Access modes:

- `open` — full API access.
- `restricted` — whitelist-defined surface.
- `locked` — read-only: state, serialization, and config.

You install with use(pluginFn, options), and revoke with unuse(plugin). Plugin conflict warnings show up as `PLUGIN_CLASH`, so nothing gets overwritten in silence.

Built-in Plugins

Useful. Isolated. Honest.

persistPlugin: Adds localStorage persistence and mutation tracking:

- `persist(key)` — save state to disk.
- `load(key)` — load from disk.
- `freeze()` / `thaw()` — toggle mutation mode.
- `lastModified()` — timestamp of change.
- `changedKeys()` — key-level diff log.
- `bindToGlobal(name)` — expose to global scope.
- `configurable()` — log config status.

Hooks into `set()` and `setState()` to track heat.

linkPlugin: Lightweight DOM binding without rendering engine:

- `linkToDOM(selector, attr)` — serialize state into an attribute.
- `unbindDOM()` — removes binding.

Useful for dashboards, visualizers, or zero-framework rendering.

Config + Devtools

Live introspection. No tooling required.

Config lives in `meat.config` and can be mutated directly.

Toggles & Tools:

- `config.mutable` — controls update behavior.
- `config.debug` — enables internal logging.
- `pluginAccess` — scopes plugin reach.
- `debugLog()` — gated internal messaging.
- `warn(code, detail)` — system messages.
- `version` — runtime version label.
- `devtools()` — logs current state as a table.
- `configurable()` — prints active config.

Everything observable. Nothing gated behind extensions.

Runtime Composition

Shipped in ~300 lines. No build step. No scaffolding.

Meat loads via IIFE. No bundler required.

- Exports top-level meat object.
- Registers built-in plugins automatically.
- Can be dropped as script, added via CDN, or bundled by npm.
- `bindToGlobal()` makes it globally reachable for debugging.

There’s no hydration phase. No virtual abstraction. Just real reactive behavior you control, inspect, and extend.

State Management

Mutation is engineered—not excused.

State isn't sacred. And it isn't chaotic. It's just data—shaped, mutated, and observed through direct mechanisms. Meat gives you a clean set of primitives. No proxies. No lifecycle negotiations. No trick renderers. Just honest signal transmission and predictable state behavior that responds at the speed of your intent. This is mutation as infrastructure—not mysticism.

Storing State

Under the hood, Meat maintains a plain object: No magic. No getters. No traps. Just key-value storage wired for visibility.

```
let state = Object.create(null);
```

To read the full store:

```
getState(); // Clones if immutable; returns reference if mutable
```

To inspect specific fields:

```
  get("theme");       // → "dark"
  has("user");        // → true / false
```

Every access is direct. No wrappers. No memoization dance. If you know JavaScript, you know Meat's surface.

Mutating State

Single key updates happen via: `set("theme", "dark")`

Behind the scenes:

```
state = config.mutable
? (state[key] = value, state)
: { ...state, [key]: value };
```

Every mutation triggers:

- A scoped signal: `meat:update:{key}`
- A full snapshot: `meat:update`

Batch updates use either `setState()` or `merge()`:

```
setState({ user: "Luciano", locale: "it-IT" });
```

```
merge({ online: true });
```

You choose mutation mode:

- **Immutable** (`mutable: false`) – changes clone the object, preserving safety and traceability.
- **Mutable** (`mutable: true`) – mutations apply in-place, trading purity for speed.

Switch modes mid-flight with: *(Mutations are real-time, observable, and scoped to intent.)*

- `freeze()`: `config.mutable = false`
- `thaw()`: `config.mutable = true`

Reset and Wipe

To purge the entire store:

- `reset()`: clears all keys and emits global update
- `clear()`: same effect, used directly

```
// Internally:
if (config.mutable) {
    Object.keys(state).forEach(k => delete state[k]);
  } else {
    state = {};
  }
```

Emissions include per-key `meat:update:{key}` with `undefined` and a global `meat:update`. Use it when you rebuild flows, clear form state, or reset reactive infrastructure.

Tracking Mutations

Meat quietly tracks state changes for introspection and audit:

- `lastModified()`: timestamp (ms)
- `changedKeys()`: array of keys mutated since last clear

These logs are updated every time a value shifts—whether by set(), setState(), or plugin-induced updates.This lets you power:

- Undo flows
- Dirty field markers
- Diff persistence
- Custom debugging overlays

It’s not a framework feature. It’s runtime fidelity.

Serializing State

To represent your store as a JSON snapshot: `serialize()`. Returns a cached string unless the reference has shifted. You can use it for:

- `persist()` plugin (localStorage sync)
- `linkToDOM()` plugin (bind to DOM)
- Devtool tables
- Snapshot hashing or syncing

```
// Internal logic:
if (prevRef === state) return cachedString;
  cachedString = JSON.stringify(state);
  prevRef = state;
  return cachedString;
```

No debounce. No middleware. Just memory-aware serialization.

Inspection Tools

Sometimes you need to look inside—not just at the surface.

- `select(["user", "theme"])` → `sliced object`
- `hasChanged("theme", "light")` → `true / false`
- `isEmpty()` → `true if no keys`
- `inspectKey("user")` → `{ value, watched }`

These utilities help shape logic. You can check if a value changed before sending it. Or inspect whether a field is actively watched.

Reactive Composition

Every mutation is accompanied by event emissions. You catch them via:

```
  watch("user", val => {
    renderAvatar(val);
  });
  once("locale", val => {
    loadTranslation(val);
  });
  subscribe(snapshot => {
    updateSidebar(snapshot);
  });
```

All handlers fire synchronously after mutation. You never wait on a render cycle. You never chase stale data.

Want wildcards?

```
onAny((type, payload) => {
  logEvent(type, payload);
});
```

Useful for logging, debugging, analytics, or reactive metrics.

Want to undo?

```
const stack = [];
watch("content", val => stack.push(val));
function undo() {
  if (stack.length > 1) {
    stack.pop(); // remove current
    set("content", stack.pop()); // restore previous
  }
}
```

Plugin Shaping

Plugins hook into mutations safely. For example, persistPlugin tracks all changes and lets you do:

- `persist("meatState")` stores serialized JSON
- `load("meatState")` reloads from localStorage

It uses the same mutation tracking: `trackMutation(key, value)`. Combined with `serialize()`, you control snapshot size, timing, and persistence scope.

Want to expose Meat globally for debugging?

- `bindToGlobal()` exposes meat to window
- `configurable()` logs current config flags No ceremony. Just control.

Usage Flow: Manual Control + Reactive Output

A typical composition might look like this:

```
setState({ user: "Luciano", theme: "dark" });
watch("theme", t => {
  applyStyles(t);
});
persist(); // store snapshot
serialize(); // get JSON
console.table(getState()); // devtools
if (hasChanged("theme", "light")) {
  set("theme", "light");
}
reset(); // full teardown
```

Each piece is inspectable. Each hook is manual. No render assumptions. No implicit subscriptions. Meat gives you signal—not scaffolding.

Closing Pulse

Meat's state engine is reactive, inspectable, and shaped with intent. No magic. No mystery. Just mutation that behaves, snapshots that serialize, and events that fire without lifecycle clutter. It's engineered like infrastructure—meant to run, not to perform.

If the previous chapter gave you the blade's composition, this one shows you how it slices in motion. Next we sharpen signal even further: scoped listeners, plugin coordination, and the full orbit of reactivity. Let's keep building with grip.

Events & Reactivity

Where signal moves without ceremony.

Meat's event system is the backbone of its reactive architecture—a minimalist bus that connects mutations to listeners with no lifecycle rituals, no scheduler abstractions, and no framework interference. It's built to behave in real time, from `set()` to `watch()`, from mutation to effect.

Event Bus System

The event bus is powered by a Map. It wires event names to listeners, all invoked synchronously:

- `on(type, fn)` register a listener
- `off(type, fn)` remove listener
- `emit(type, data)` dispatch an event
- `onAny(fn)` global listener for all events
- `listeners()` inspect all active types
- `isWatched(key)` check listener presence

Each mutation in Meat triggers two events:

- `meat:update:{key}` scoped
- `meat:update` full snapshot

Listeners are always called synchronously, in insertion order. No deferrals. No surprises.

Use `onAny()` for logging and instrumentation:

```
onAny((type, payload) => {
  console.log("Event:", type, payload);
});
```

Scoped Reactivity

Meat supports fine-grained observation without lifecycle abstractions:

- `watch("theme", val => applyTheme(val))`
- `once("locale", val => loadLanguage(val))`
- `subscribe(snapshot => syncDashboard(snapshot))`

Each returns a teardown function:

- `const unwatch = watch("user", val => ...)`
- `unwatch()` manual cleanup
- `once()` unbinds itself after firing once.
- `watch()` observes a key.
- `subscribe()` sees all updates.

Initial notification happens on registration. You never “miss” the first state unless you opt out.

Want to know if a key is actively watched? `isWatched("user")`

Synchronous Behavior

All events fire in sync—post-mutation, pre-render. That means you can respond immediately:

```
set("ready", true);
watch("ready", val => {
  if (val) activateView();
});
```

The event system doesn’t wait. It’s designed to mirror runtime flow, not wrap it. This directness keeps logic traceable. No zone.js, no reactivity trees, no “why didn’t this trigger yet?”

Debugging the Flow

Enable `config.debug` to trace internal emissions and behaviors:

```
config.debug = true;
debugLog("listener-registered", { key, fn });
```

Warnings emit via: `warn("HANDLER_ERROR", error)`

Devtools print state clearly: `devtools()` logs state as table

You can inspect active channels:

- `listeners()` → `[ "meat:update:user", ... ]`
- `isWatched("theme")` → `true` / `false`

Pair `onAny()` with external monitors for reactive analytics.

Audit & Intent

Combine with mutation tracking for full visibility:

- `changedKeys()` → ["theme", "user"]
- `lastModified()`→ `timestamp`

Every listener is part of a precise chain. You know what changed. You know which signals fired. You know what responded. No lifecycle guessing. No async drift.

React Without Framework

Reactivity in Meat isn’t tied to renderers, component trees, or lifecycle abstractions. It’s reactive in the original sense: you mutate, it responds. That’s it. You don’t import hooks. You don’t orchestrate hydration. You wire behavior directly.

- `const show = el => el.style.display = "block"`
- `const hide = el => el.style.display = "none"`

```
// Use case: DOM toggle
watch("visible", v => {
    v ? show(panel) : hide(panel);
  });
```

No virtual DOM. No re-render diffing. Just intent flowing from state to output.

```
// Use case: network sync
watch("query", value => {
  fetch(`/search?q=${value}`)
    .then(res => res.json())
    .then(data => set("results", data));
});
```

Reactive without framework. Declarative without syntax tricks. You mutate query, Meat reacts, fetch runs.

```
// Use case: canvas redraw
watch("angle", deg => {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  drawArrow(ctx, deg);
});
```

Mutation drives signal. Signal drives redraw. It’s tight, observable, and purpose-built.

```
// Use case: global listener
onAny((type, payload) => {
  analytics.track(type, payload);
});
```

You log every mutation, every flow, every scoped change—without ceremony. Meat doesn’t care if your app is HTML, SVG, WebGL, JSON over wire, or serverless state sync. It emits. You respond.

```
// Listener cleanup? Manual when you want it:
const unwatch = watch("theme", applyTheme);
unwatch(); // tear down
```

No dependency arrays. No unmount guessing. You control the link—and its teardown. Reactivity here is signal first. It's not react-like. It's reaction—without abstraction. You observe. You mutate. You respond. Every listener runs in sync. Every flow stays traceable. This isn't a framework—it's **a reactive layer you can reason about**.

Closing Signal

Events in Meat behave the way developers expect: predictably, synchronously, observably. No hook fatigue. No framework lock-in. This is reactivity built on clarity—not context. If Chapter III showed how mutation behaves, this chapter shows how signal responds. Meat moves when your code moves. And nothing stands in the way of your intent.

Plugin System

Extensibility without compromise.

Meat supports live augmentation through a plugin model built on simplicity, safety, and clarity. Plugins behave like runtime extensions—but they don't hijack internals. They inherit only what you expose. You control the API surface. They operate in isolation.

This isn't a framework-style plugin system. There's no registry. No lifecycle tax. Just functions with scoped access, wired to behave like tools—not authorities.

Philosophy & Control Scopes

Plugins don't get access to everything by default. That's intentional.

At install, you select an access mode via config:

```
config.pluginAccess = "restricted"; // or "open", "locked"
```

Each mode shapes what the plugin receives:

- `open` — Full access to Meat API. Suitable for trusted internal use.
- `restricted` — Only the methods explicitly whitelisted in config.
- `locked` — Read-only access: `getState()`, `serialize()`, config copy.

This ensures third-party or shared plugins operate within bounds—no surprise mutations, no hidden subscriptions.

Internally, each plugin receives:

```
function createPluginAPI(mode, whitelist) {
  if (mode === "open") return meat;
  if (mode === "restricted") return pick(meat, whitelist);
  if (mode === "locked") return {
    getState,
    serialize,
    config: clone(config)
  };
}
```

Lifecycle

Registering is direct:

```
meat.use(pluginFn, options);
```

Unregistering is surgical:

```
meat.unuse(pluginFn); // calls pluginFn.cleanup() if present
```

Each plugin is a function that receives: `pluginFn(api, options)`

Side effects are opt-in. Cleanup is encouraged:

```
function demoPlugin(api) {
  const stop = api.subscribe(snapshot => log(snapshot));
  demoPlugin.cleanup = () => stop();
}
```

Plugins don't override Meat's internal logic. They only extend via top-level mutation—and every mutation is tracked. No lifecycle traps. No hidden hooks. Just execution you control.

Sandboxing & Isolation

Plugins operate inside sandboxed scopes. They don't get to crawl internals. They don't patch private memory. They behave like guests with a badge—not like root access.

Want to define a scoped method?

```
meat.foo = () => console.log("hello");
```

Meat tracks every top-level addition between use() and unuse(). If anything clashes:

```
console.warn("PLUGIN_CLASH: Method 'foo' already exists");
```

Plugins don't overwrite silently. If they try, you're alerted. It's signal—not sabotage.

Conflict Detection

Before plugin installs:

```
const before = Object.keys(meat);
```

After execution:

```
const after = Object.keys(meat);
```

Any overlaps that weren't explicitly whitelisted emit a warning: `PLUGIN_CLASH`: Method "foo" overwritten.

You fix it by renaming, or by wrapping existing methods safely:

```
const original = meat.set;
meat.set = (key, val) => {
  console.log("Plugin intercepted set");
  original(key, val);
};
```

Plugins don’t compete. They collaborate—if invited.

Adaptive Plugins

Plugins can check their mode and adjust behavior:

```
if (api.config.pluginAccess === "locked") {
  return; // exit quietly
}
```

That makes them portable. A logging plugin can operate in full mode or read-only. A DOM plugin can bail if it’s restricted.

Plugins aren’t monoliths. They’re reactive utilities.

Final Note

Meat’s plugin system isn’t declarative. It’s procedural. You load the plugin. You shape its scope. You control its teardown.

There’s no framework dance. Just behavior you invite.

If Chapter IV shaped signal, Chapter V shapes how external logic folds in—without folding the runtime.

Next up: Built-in Plugins like `persist()`, `linkToDOM()`, and `freeze`/`thaw` extensions that ship out of the box.

Built-in Plugins

Where extension meets utility.

Meat ships with built-in plugins designed for real-world utility—not theory. These plugins operate without introducing hierarchy, rendering dependencies, or lifecycle complexity. Each one extends runtime behavior from a principled, scoped point of view.

Persist Plugin: LocalStorage-powered snapshot saving.

- `persist(key)` save serialized state to disk
- `load(key)` restore from disk
- `freeze()` force immutable mode
- `thaw()` allow mutable mode
- `lastModified()` timestamp of last change
- `changedKeys()` diff since last clear
- `bindToGlobal(name)` exposes meat globally
- `configurable()` print config flags Used for offline sync, hydration flows, and minimal durability features—without reaching for IndexedDB or state frameworks.

Link Plugin: DOM binding without render engine.

- `linkToDOM(selector, attr)` sync state to DOM attribute
- `unbindDOM()` teardown binding

Useful for dashboards, inspection overlays, status bubbles, or pure-data UIs. No virtual DOM assumptions. Just state → markup.

LogState Plugin Console visibility, single method.

- `logState()` console.table of current state

Drop it when you want fast devtools signal without overhead. Especially useful in immutable debug sessions or plugin sandbox inspection.

Plugin logic: Simple. Visual. No config required.

```
meat.logState = () => {
  console.table(meat.getState());
};
```

MeatChronicle Plugin

Time-series, undo, rollback, audit tooling

What It Does

- Tracks every mutation across keys in a bounded history map
- Allows per-key `undo()` restoration
- Supports global `rollbackAll()`
- Wraps side-effect blocks with `safe(fn)` for auto-rollback on error
- Surfaces change logs via `getHistory()`, `logHistory()`, and `historySnapshot()`
- Offers `changedKeys()` to inspect what moved and when, based on the keys touched since the last full reset or snapshot. This lets you analyze field-level activity and trace application behavior over time.

Each key gets a chronological log: `{ value, timestamp, source }`. The source field marks context: "mutation", "undo", "rollback", etc.

Installation:

`meat.use(MeatChronicle, { limit: 100 })`: The "limit" defines how many snapshots per key are retained. Once full, older entries are removed as new ones come in. Useful for:

- Undo/redo in forms
- Async safety nets
- Audit trails
- Telemetry and diagnostics

Real-World Usage

- Undo a key's last change: `meat.undo("theme")`
- Rollback all fields: `meat.rollbackAll()` Protect side effects:

```
meat.safe(() => {
  meat.set("count", null);
  throw new Error("Boom");
});
```

The plugin auto-restores previous state and logs the failure. Inspect timeline: `meat.logHistory("theme")` Access raw entries: `meat.getHistory("theme")` Dump all logs: `meat.historySnapshot()` Check touched keys: `meat.changedKeys()`

Setup Flow

When installed, the plugin watches every mutation:

```
meat.subscribe(snapshot => {
  Object.entries(snapshot).forEach(([key, value]) => {
    push(key, value, "setState");
  });
});
```

Each mutation is pushed to its key's log. If limit exceeded, old entries are dropped.

Rollback logic is simple and scoped:

- `meat.undo(key)` revert one
- `meat.rollbackAll()` revert everything
- `meat.safe(fn)` run with protection

You can also inspect the mutation source and timestamp on every log entry for analytics or diffing.

Framework Plugins

You can build plugins that wire into UI libraries without binding to their internal lifecycles.

Framework plugins don't mutate render engines—they pass signal into them. This keeps reactive behavior decoupled from UI logic. Mutations stay in Meat. Listeners forward state into view.

Meat provides a set of integration plugins that connect its core runtime to popular UI frameworks—without imposing lifecycle, hydration, or reactivity models.

These plugins expose Meat's signal system to the host environment while preserving the principle of external observability.

The plugins don't reimplement Meat's API. They expose it, inject it, or forward it. Behavior stays untouched. Signal stays pure.

Alpine

`meatAlpinePlugin()`:

- Registers MEAT into Alpine's reactive evaluator.
- Updates state via evaluateLater + `effect()`.
- Ideal for embedding Meat in declarative markup.

Angular

`MeatService` (Injectable)

- Provides `get()`, `set()`, `watch()`, `subscribe()` from MEAT core.
- Injectable into Angular components.
- Integrates signal without NG lifecycle coupling.

Astro

`meatAstroPlugin()`

- Adds MEAT state as a serialized payload to window via injectScript.
- Enables hydration-aware use or inspection from client-side scripts.

Next

`meatNextPlugin(app)`

- Attaches MEAT to the app object.
- Minimal wrapper for global access in app components.

Nuxt

`defineNuxtPlugin()`

- Provides MEAT via nuxtApp context.
- Exposes state engine to Vue/Nuxt composition APIs.

Qwik

`meatQwikPlugin()`

- Adds MEAT to Qwik’s provide system.
- Enables reactive signal bridging across islands or loaders.

React

`useMeat(key)`

- Custom hook that subscribes to MEAT and re-renders component.
- Provides runtime-safe access to reactive keys.

Solid

`meatSolidPlugin()`

- Returns MEAT through Solid’s provide pattern.
- Allows components to access and mutate state directly.

Svelte

`meatSveltePlugin(app)`

- Injects MEAT into the Svelte app context.
- Enables direct subscriptions and debug toggling.

Vue

`meatVuePlugin`

- Installs MEAT as $meat on globalProperties.
- Available in all components via this.$meat or Composition API.

> **Plugins serve one job**
> Make Meat available without controlling behavior. Each plugin is separate and scoped.No wrappers. No abstraction chains. Just exposed runtime.This is integration without compromise—and signal you control.

useMeat()

Before we close the chapter on framework plugins, it's worth noting a common utility pattern found across integration folders: `useMeat`.

While the core MEAT runtime doesn't depend on any framework, many plugins ship helper utilities named `useMeat.js` or `useMeat.ts`. These act as reactive bindings, usually wrapping MEAT's `subscribe()` logic into framework-native reactivity.

For example:

Angular `useMeat(key: string)`

- Wraps state access with RxJS BehaviorSubject
- Provides value asObservable() for Angular templates and services

```
export function useMeat(key: string) {
  const subject = new BehaviorSubject(meat.get(key));
  meat.subscribe(key, value => subject.next(value));
  return subject.asObservable();
}
```

React `useMeat(key: string)`

- Uses useState() and useEffect()
- Subscribes to MEAT, triggers local re-render

```
export function useMeat(key) {
  const [value, setValue] = useState(meat.get(key));
  useEffect(() => {
    const unsub = meat.subscribe(key, setValue);
    return unsub;
  }, [key]);
  return value;
}
```

Svelte, Solid, Vue, Qwik, and Nuxt plugins may ship similar useMeat files—but not all are populated.

Important: these utilities are optional.
You don't need them to access MEAT inside a framework. They just smooth the reactive edge if you want plug-and-play syntax.

Debugging & Dev Tools

Where inspection meets truth.

Meat is engineered for runtime clarity. It doesn't abstract mutation—it exposes it. This chapter walks through the debugging surface, devtools integrations, introspection utilities, and runtime diagnostics you can use during development, audit, and reactive reasoning.

Core Debug Controls

- `meat.config.debug = true` Set a global flag to enable internal debug output: This enables console tracing for plugin setup, listener registration, mutation events, and conflict warnings.

Trace warnings manually via:

```
meat.warn("PLUGIN_CLASH", { key: "set" });
```

Use `debugLog()` to emit structured messages:

```
  debugLog("watch-attached", { key, fn });
```

State Inspection

You can inspect the current store with:

```
  console.log(meat.select(["theme"]));    // partial state
  console.log(meat.has("user"));          // key check
  console.log(meat.inspectKey("locale")); // value and watched status
```

Use `logState()` plugin to simplify: `meat.logState()`

Event Visibility

Check which listeners are active:

- `meat.listeners()` → array of active event types
- `meat.isWatched("theme")` → true / false

Wildcard listeners can mirror traffic:

```
meat.onAny((type, payload) => {
  console.log("Event:", type, payload);
});
```

Useful for live logging, testing, and reactive diagnostics.

Mutation Tracking

Audit how state moves over time:

```
meat.changedKeys();      // → array of touched keys
meat.lastModified();     // → timestamp of last change
meat.serialize();        // → JSON snapshot
```

This helps you debug:

- Dirty fields in forms
- Reactive flows in UI
- State-based navigation changes

Devtools Access

Use `meat.devtools()` to print state as a table: `meat.devtools()` → console.table of store

Can be used at breakpoints, or inserted into runtime flows for visibility.

Pair with:

```
console.group("Mutation Audit");
meat.devtools();
console.groupEnd();
```

Plugin Diagnostics

Plugins can introspect mode, config, and exposed API surface:

```
if (meat.config.pluginAccess === "locked") { ... }
```

They can also emit safe logs:

```
meat.logMessage("Plugin initialized");
```

Pair with `MeatChronicle` for mutation journaling and rollback auditing.

Live Debug Scenarios

Example: verifying listener execution

```
meat.watch("ready", val => console.log("READY:", val));

meat.set("ready", true);
```

Expect immediate output, no delay. Listeners are synchronous. Nothing hidden.
Example: confirm plugin conflict

```
meat.set = () => { /* overridden */ };
meat.use(conflictPlugin); // warns on method clash
```

Learn by Doing.

This chapter breaks form. Instead of narrative, it’s raw signal. Dozens of examples—meant to be tested, remixed, broken, and reused. No abstraction, no scaffolding. Just direct hits from the forge.

Basic Wiring

```
meat.set("ready", true);
console.log(meat.get("ready"));

meat.subscribe(state => {
  console.log("Snapshot:", state);
});

meat.watch("theme", t => {
  applyTheme(t);
});

meat.once("user", u => {
  greet(u.name);
});

meat.onAny((type, payload) => {
  console.log("EVENT:", type, payload);
});
```

Batch Changes

```
meat.setState({ theme: "light", locale: "en" });

const profile = {
  name: "Luciano",
  age: 32,
  online: true,
};
meat.merge(profile);
```

DOM Binding

```
meat.linkToDOM("#app", "data-state");
meat.set("theme", "dark");
// <div id="app" data-state='{"theme":"dark"}'></div>
```

Form Sync

```
const input = document.querySelector("#email");
input.addEventListener("input", e => {
  meat.set("email", e.target.value);
  meat.persist();
});

window.addEventListener("load", () => {
  meat.load();
});
```

Undo + Rollback

```
meat.use(MeatChronicle);
meat.set("count", 1);
meat.set("count", 2);
meat.undo("count");         // back to 1
meat.rollbackAll();         // restores all keys
```

Protect Block

```
meat.safe(() => {
  meat.set("score", null);
  throw new Error("Game broke");
});
// score gets rolled back
```

Change Audit

```
meat.set("language", "en");
meat.set("theme", "dark");

console.log(meat.changedKeys());  // ["language", "theme"]
console.log(meat.lastModified()); // timestamp
```

Devtools

```
meat.devtools();     // console.table of state
meat.logState();     // from logStatePlugin
```

Listener Teardown

```
const stop = meat.watch("visible", val => toggle(val));
stop(); // remove listener
```

Plugin: Timestamp Injection

```
function timestampPlugin(api) {
  api.setTimestamp = () => {
    api.set("timestamp", Date.now());
  };
}

meat.use(timestampPlugin);
meat.setTimestamp();
```

Plugin: Mutation Counter

```
function mutationCounter(api) {
  let count = 0;
  api.subscribe(() => count++);
  api.getMutationCount = () => count;
  api.resetCount = () => count = 0;
}

meat.use(mutationCounter);
console.log("Mutations:", meat.getMutationCount());
```

Plugin: Logger

```
function loggerPlugin(api) {
  api.onAny((type, payload) => {
    console.log(`[LOG] ${type}`, payload);
  });
}

meat.use(loggerPlugin);
```

Serialize + Persist

```
const snapshot = meat.serialize();
localStorage.setItem("state", snapshot);
```

Hydrate on Load

```
window.addEventListener("load", () => {
  const data = localStorage.getItem("state");
  if (data) {
    meat.merge(JSON.parse(data));
  }
});
```

Reactive Canvas

```
const canvas = document.getElementById("draw");
const ctx = canvas.getContext("2d");

meat.watch("angle", deg => {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  drawPointer(ctx, deg);
});
```

State-Based Styling

```
meat.watch("theme", theme => {
  document.body.classList.toggle("dark", theme === "dark");
});
```

Autosave Draft

```
meat.watch("draft", val => {
  localStorage.setItem("draft", val);
});
```

Plugin: Visualizer

```
function signalVisualizer(api) {
  api.subscribe(snapshot => {
    document.title = `Keys: ${Object.keys(snapshot).length}`;
  });
}

meat.use(signalVisualizer);
```

Reactive Network

```
meat.watch("query", q => {
  fetch(`/search?q=${q}`)
    .then(res => res.json())
    .then(results => meat.set("results", results));
});
```

Reactive System Time

```
setInterval(() => {
  meat.set("now", Date.now());
}, 1000);
```

Event Audit UI

```
meat.onAny((type, payload) => {
  const el = document.createElement("div");
  el.textContent = `[${type}]: ${JSON.stringify(payload)}`;
  document.body.appendChild(el);
});
```

Reactive Debug Overlay

```
const debug = document.createElement("pre");
document.body.appendChild(debug);

meat.subscribe(state => {
  debug.textContent = JSON.stringify(state, null, 2);
});
```

Live Field Diff

```
const prev = meat.getState();
meat.set("score", 20);
const next = meat.getState();

const diff = Object.keys(next).filter(k => prev[k] !== next[k]);
console.log("Changed:", diff);
```

Reactive Input Binding

```
document.getElementById("message").addEventListener("input", e => {
  meat.set("message", e.target.value);
});

meat.watch("message", m => {
  console.log("Input:", m);
});
```

Full State Dump on Ctrl+D

```
document.addEventListener("keydown", e => {
  if (e.ctrlKey && e.key === "d") {
    meat.devtools();
  }
});
```

Learn by Doing (Advanced)

These examples explore advanced behaviors: orchestration, reactive boundaries, plugin composition, and unconventional control flows. They're meant to challenge assumptions and deepen your signal intuition.

Reactive BroadcastChannel Sync

```
const channel = new BroadcastChannel("meat");

meat.subscribe(snapshot => {
  channel.postMessage(snapshot);
});

channel.onmessage = e => {
  meat.merge(e.data);
};
```

Dynamic Key Watch via Schema

```
const schema = ["user", "theme", "locale"];

schema.forEach(key => {
  meat.watch(key, value => {
    console.log(`Updated ${key}:`, value);
  });
});
```

Chained Signal Effects

```
meat.watch("theme", theme => {
  meat.set("bg", theme === "dark" ? "#000" : "#fff");
});

meat.watch("bg", color => {
  document.body.style.backgroundColor = color;
});
```

Deferred Rollback Queue

```
let queue = [];

meat.safe(() => {
  meat.set("ready", false);
  queue.push("ready");

  meat.set("locked", true);
  queue.push("locked");

  throw new Error("Failed!");
});

queue.forEach(k => console.log("Rolled back:", k));
```

Scoped Plugin Composition

```
function auditPlugin(api) {
  api.onAny((type, payload) => {
    api.set("auditLog", `[${type}] ${JSON.stringify(payload)}`);
  });
}

function notifyPlugin(api) {
  api.watch("auditLog", msg => {
    sendToast(msg);
  });
}

meat.use(auditPlugin);
meat.use(notifyPlugin);
```

Reactive Visibility Sentinel

```
const observer = new IntersectionObserver(entries => {
  entries.forEach(entry => {
    meat.set("visible", entry.isIntersecting);
  });
});

observer.observe(document.getElementById("section"));
```

Plugin Generator Scaffold

```
function createPlugin(name, logic) {
  return api => {
    console.log(`Plugin loaded: ${name}`);
    logic(api);
  };
}

const pingPlugin = createPlugin("Ping", api => {
  api.set("ping", Date.now());
});

meat.use(pingPlugin);
```

Input Mirror + Reset Combo

```
meat.watch("mirror", val => {
  document.getElementById("output").textContent = val;
});

meat.watch("mirror", val => {
  if (val === "reset") meat.reset();
});
```

Devtools + Inspector Dock

```
document.addEventListener("keydown", e => {
  if (e.ctrlKey && e.key === "i") {
    const dock = document.createElement("pre");
    dock.style.position = "fixed";
    dock.style.bottom = "0";
    dock.style.width = "100%";
    document.body.appendChild(dock);

    meat.subscribe(state => {
      dock.textContent = JSON.stringify(state, null, 2);
    });
  }
});
```

Live Mutation Timeline

```
meat.use(MeatChronicle);

const log = document.getElementById("timeline");

meat.watch("events", () => {
  const trail = meat.getHistory("events");
  log.textContent = trail.map(e => `${e.time} → ${e.value}`).join("\n");
});
```

Signal-Driven Component Loader

```
meat.watch("component", id => {
  import(`./components/${id}.js`).then(mod => mod.mount());
});
```

Smart Sync Guard

```
function syncGuard(api) {
  api.watch("session", s => {
    if (!s || !s.token) return;
    api.emit("sync", { token: s.token });
  });
}

meat.use(syncGuard);
```

Advanced signal work isn’t about complexity—it’s about clarity under pressure. Keep it composable. Keep it deliberate. Stay reactive.

Learn by Doing (Ninja)

In this tier, we explore signal architecture with velocity. These patterns aren't just reactive—they're predictive, protective, and composable at runtime.

Multi-Key Proxy Binding

```
function proxyPlugin(api) {
  ["firstName", "lastName"].forEach(key => {
    api.watch(key, () => {
      const full = api.get("firstName") + " " + api.get("lastName");
      api.set("fullName", full);
    });
  });
}

meat.use(proxyPlugin);
```

Timebox + Revert Mechanism

```
function timeboxPlugin(api) {
  api.set("temp", true);
  setTimeout(() => {
    api.undo("temp");
  }, 2000);
}

meat.use(timeboxPlugin);
```

Deep Signal Replication Across Tabs

```
const tabChannel = new BroadcastChannel("meat-sync");

meat.subscribe(snapshot => {
  tabChannel.postMessage(snapshot);
});

tabChannel.onmessage = e => {
  if (e.data._source !== location.href) meat.merge(e.data);
};
```

Conditional Plugin Injection

```
if (window.location.pathname === "/admin") {
  meat.use(adminAuditPlugin);
}
```

Self-Removing Watcher

```
meat.watch("count", val => {
  if (val > 100) {
    console.warn("Too high!");
    return () => false; // teardown immediately
  }
});
```

Reactive Pagination Cursor

```
meat.watch("page", p => {
  fetch(`/api/data?page=${p}`)
    .then(res => res.json())
    .then(items => meat.set("items", items));
});
```

Mutation-Based Trigger Chain

```
meat.watch("step", step => {
  switch (step) {
    case 1: meat.set("message", "Init"); break;
    case 2: meat.set("message", "Running"); break;
    case 3: meat.set("complete", true); break;
  }
});
```

Plugin Triggered by External Signal

```
window.addEventListener("message", e => {
  if (e.data.signal) {
    meat.set("remoteSignal", e.data.signal);
  }
});
```

Recursive Key Graph Explorer

```
function graphExplorer(api) {
  function explore(obj) {
    Object.keys(obj).forEach(k => {
      api.watch(k, val => {
        console.log(`${k}:`, val);
        if (typeof val === "object") explore(val);
      });
    });
  }
  explore(api.getState());
}

meat.use(graphExplorer);
```

Signal fluency isn’t about memorizing tools. It’s about composing movement. These are signal kata—train until they’re reflex.

Learn by Doing (DEFCON 1)

Extreme, high-stakes examples where signal failure has dire consequences.

1. Atomic Transaction Across Multiple Keys

```
meat.safe(() => {
  const { balance, history } = meat.getState().bank;
  const amount = 100;
  meat.set("bank.balance", balance - amount);
  meat.set("bank.history", [...history, { debit: amount }]);
  if (meat.get("bank.balance") < 0) throw new Error("Overdraw!");
});
```

2. Circuit Breaker Plugin

```
function breakerPlugin(api, options = { threshold: 5 }) {
  let failures = 0;
  api.watch("errors", () => {
    failures++;
    if (failures >= options.threshold) {
      api.set("circuitOpen", true);
    }
  });
}
meat.use(breakerPlugin);
```

3. Fail-Safe WebSocket Sync

```
const ws = new WebSocket("wss://example.com/sync");
ws.onmessage = e => meat.merge(JSON.parse(e.data));
meat.onAny((type, payload) => {
  if (ws.readyState === WebSocket.OPEN) {
    ws.send(JSON.stringify({ type, payload }));
  }
});
ws.onerror = () => meat.set("syncError", true);
```

4. Offline Queue with Exponential Backoff

```
const queue = [];
window.addEventListener("offline", () => meat.set("offline", true));
window.addEventListener("online", () => {
  meat.set("offline", false);
  processQueue();
});
function enqueueChange(key, value) {
  queue.push({ key, value });
}
function processQueue(attempt = 1) {
  if (!queue.length) return;
  const item = queue.shift();
  meat.set(item.key, item.value);
  fakeSync(item)
    .then(() => processQueue(1))
    .catch(() => {
      queue.unshift(item);
      setTimeout(() => processQueue(attempt + 1), Math.pow(2, attempt) *
1000);
    });
}
```

5. Collaborative Cursor Sync

```
const channel = new BroadcastChannel("cursor-sync");
meat.watch("cursor", pos => channel.postMessage(pos));
channel.onmessage = e => meat.set("remoteCursor", e.data);
```

6. Dynamic Feature Flags

```
function featureFlagPlugin(api, flagsUrl) {
  fetch(flagsUrl)
    .then(res => res.json())
    .then(flags => api.set("featureFlags", flags));
  api.watch("featureFlags", flags => {
    Object.entries(flags).forEach(([name, enabled]) => {
      api.set(`flags.${name}`, enabled);
    });
  });
}
meat.use(featureFlagPlugin, "https://example.com/flags.json");
```

7. GDPR History Scrub Plugin

```
function scrubPlugin(api, keysToRedact = []) {
  api.use(MeatChronicle);
  api.watch("history", hist => {
    const scrubbed = hist.map(record => {
      keysToRedact.forEach(k => delete record.payload[k]);
      return record;
    });
    api.set("history", scrubbed);
  });
}
meat.use(scrubPlugin, ["email", "ssn"]);
```

8. Snapshot Isolation Read-Only View

```
const snapshot = meat.serialize();
function readOnlyView() {
  const state = JSON.parse(snapshot);
  return state; // safe read without affecting live state
}
```

9. Hierarchical State Partitioning

```
function partitionPlugin(api) {
  const modules = Object.keys(api.getState());
  modules.forEach(mod => {
    api.watch(mod, state => {
      api.set(`partitions.${mod}`, state);
    });
  });
}
meat.use(partitionPlugin);
```

10. On-Demand Component Loader

```
meat.watch("loadComponent", id => {
  import(`./components/${id}.js`)
    .then(mod => mod.mount(meat))
    .catch(err => meat.set("loadError", { id, err }));
});
```

Extending Meat

Where augmentation respects runtime.

Meat's extension model is simple by design—but that simplicity demands discipline. Discipline here means plugins behave without surprises.

This chapter details how to extend the core safely, write reusable plugins, and clean up responsibly without ghosting the runtime.

Adding Methods with Collision Safety

To extend `meat` without stepping on existing tools, use explicit checks:

```
if (!("logState" in meat)) {
  meat.logState = () => console.table(meat.getState());
} else {
  console.warn("PLUGIN_CLASH: 'logState' already exists.");
}
```

You can also run collision checks across API surface:

```
const before = Object.keys(meat);
// plugin executes...
const after = Object.keys(meat);

after.forEach(k => {
  if (before.includes(k)) {
    console.warn(`PLUGIN_CLASH: Method "${k}" overwritten`);
  }
});
```

Prefix methods to avoid namespace collisions:

```
meat.chronicle_undo = () => { ... };
```

Reusable Utility Plugins

Shape plugins to be portable, scoped, and option-driven.

```
function counterPlugin(api, opts = {}) {
  let count = 0;
  const key = opts.watchKey ?? "count";
  api.watch(key, () => count++);
  api.getCount = () => count;
}
```

Use `meat.use(counterPlugin, { watchKey: "clicks" })` for scoped behavior.

Best practices:

- **Favor Options Over Globals**: Every plugin should accept an options object for configuration. Avoid relying on shared global state or side-effects. Scoped behavior ensures composability and lowers friction when debugging or refactoring.
- **Avoid Hard-Coded Keys**: Reactive keys should be declared via options.key or utility helpers—not buried in constants or assumptions. This prevents collisions and promotes isolation across extensions.
- **Respect `pluginAccess` Modes**: Each plugin receives an access scope: open, locked, or restricted. Never reach outside your sandbox. For example, avoid modifying external state unless granted elevated access.
- **Don’t Mutate Unless Required**: Prefer `get()` and observers over `set()` and side-effectual behavior. Mutations should be opt-in, explicitly tracked, and rollback-safe—especially in shared runtime environments.
- **Design for Reuse, Not Runtime Takeover**: Good plugins do one thing well. Don’t override core logic, duplicate state, or entangle yourself with global listeners. Instead, expose clean APIs and respect the host system’s boundaries.

Plugin Cleanup Strategy

Every plugin should clean up:

```
function demoPlugin(api) {
  const timer = setInterval(() => console.log("tick"), 1000);
  demoPlugin.cleanup = () => {
    clearInterval(timer);
    console.log("Cleaned up demoPlugin");
  };
}
```

Call `meat.unuse(demoPlugin)` and it triggers cleanup.

Use this for:

- Clearing intervals
- Removing DOM listeners
- Unsubscribing from `watch()` or `subscribe()`
- Tearing down resources

No cleanup means potential leaks—especially in SPAs.

Plugin Testing

Test like you ship:

- Attach plugin → observe effect
- Trigger mutations → confirm side effects
- Use `onAny()` to monitor emissions
- Detach with `unuse()` → verify teardown
- Mock flows via `meat.set()` or `meat.setState()`.
- Use `meat.devtools()` mid-test to inspect real-time state.

Meat & Laravel Integration

Full-stack flow with Blade precision

Meat's Laravel integration—via the meat-laravel subproject—brings reactive frontend logic directly into Blade views with minimal ceremony and maximum control.

Instead of manually injecting JavaScript state or wiring custom events, you declare frontend bindings through macros, and Laravel handles the rest.

The result? Blade stays expressive, Meat becomes reactive, and your app becomes a clean, full-stack flow.

Core Features

- **Blade Macros**: `@meatHydrate`, `@meatSync`, `@meatSyncEvent`, `@meatScripts`.
- **Middleware**: `InjectMeatKey` auto-injects HMAC keys across views.
- **MeatHasher**: Encrypts and validates state sync keys.
- **Sync Endpoint**: `/meat-sync` **POST** route for reactive updates.
- **Hydration Bridge**: *meat.js* handles runtime hydration and event binding.

Setup

1. Install meat-laravel
2. Register InjectMeatKey middleware:

```
    ->withMiddleware(function (Middleware $middleware): void {
        $middleware->append(\App\Http\Middleware\InjectMeatKey::class);
    });
```

3. Add MEAT_SECRET to your .env file:

```
    MEAT_SECRET=your-meat-secret
```

4. Publish frontend assets:

```
    php artisan vendor:publish --tag=meat-assets
```

5. Use macros in Blade views

```
{{-- Reactive Binding in Blade --}}
@meatHydrate(['theme' => 'dark', 'username' => $user->name])
@meatSync('username')
@meatSyncEvent('username', \App\Events\UsernameChanged::class)
@meatScripts
```

This setup will hydrate the frontend Meat store, bind the "username" key to reactive mutations, and dispatch a Laravel event when the frontend changes it.

Hydration Bridge Behavior

- Bootstraps Meat state from Blade
- Watches for meat.set() mutations
- Sends HMAC-signed payloads to Laravel
- Dispatches declared backend events
- Verifies sync integrity before processing

Controller Logic Example

```
event(new UsernameChanged($user));
```

This matches the macro’s event reference and completes the signal chain.

Secure Sync Flow

The frontend bridge sends signed payloads to:

```
Route::middleware(['web', VerifyMeatTransaction::class])
     ->post('/meat-sync', MeatSyncController::class)
     ->name('meat.sync');
```

Each payload includes the hashed key generated from:

```
$key = MeatHasher::hash(['field' => 'username']);
```

The backend uses this signature to validate the request.

Examples

Example Page

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Reactive Laravel</title>
  <meta name="csrf-token" content="{{ csrf_token() }}">
</head>
<body>

  <h1>Hello, {{ $user->name }}</h1>

  <input id="message" value="Hi!" />
  <button onclick="document.getElementById('message').value += '
!'">Boost</button>

  @meatHydrate(['message' => 'Hi!'])
  @meatSync('message')
  @meatSyncEvent('message', \App\Events\MessageUpdated::class)
  @meatScripts

</body>
</html>
```

Live Dashboard Pattern

```
@meatHydrate(['status' => 'active', 'notifications' => []])
@meatSync('notifications')
@meatSyncEvent('notifications', \App\Events\NotifyUser::class)
```

Use `meat.watch('notifications', renderToast)` to show updates in real time.

Form Draft Example

```
@meatHydrate(['draft.title' => '', 'draft.body' => ''])
@meatSync('draft.title')
@meatSync('draft.body')
@meatSyncEvent('draft.title', \App\Events\FormFieldUpdated::class)
```

Attach `meat.persist()` on input to autosave with localStorage.

Macros Summary

Macro	Purpose
`@meatHydrate($state)`	Injects initial state as JSON into the Blade template
`@meatSync($key)`	Declares a reactive binding for a given key
`@meatSyncEvent($key, EventClass)`	Links state key to backend event dispatch via handler
`@meatScripts`	Loads the bridge script from `public/vendor/meat/js`

Folder Structure

```
meat-laravel/
├── Blade/                    // macro definitions
├── Helpers/                  // MeatHasher, config
├── Http/                     // sync controller
├── Middleware/               // InjectMeatKey, verification
├── macros/                   // view injectors
├── public/vendor/meat/js/  // meat.js hydration bridge
└── MeatServiceProvider.php // macro registration
```

Authoring Tips

- All macros are parsed at runtime; order matters.
- Use `Class::class` syntax for events.
- Payloads must be serializable.
- Middleware injects keys automatically—no need to manually share.
- Use `@meatHydrate` early in your view to ensure bridge bootstraps correctly.
- Debug events using `meat.onAny()` in your frontend.

Why This Is Great

- Pure Blade remains untouched by JS framework noise.
- Laravel Events link to frontend mutations automatically.
- Sync verification is cryptographically secure.
- Your frontend responds in real time—with no extra boilerplate.
- Admin panels, dashboards, notification systems, and form workflows gain reactivity with minimal effort.
- No duplication between PHP and JS.
- Your Laravel stack becomes observably full-stack.

When you change state, Meat syncs.
When Meat syncs, Laravel reacts.
When Laravel reacts, your view updates.

All declarative. All in sync. All yours.

Public Interface

This chapter gives you a full overview of Meat’s public interface, plugin behavior, debug utilities, and glossary terms. From top-level state control to plugin lifecycle, everything is here. No scaffolding. No hidden methods. Just the full surface.

State & Mutation API

State & Mutation API	Description
`getState()`	Returns the full reactive state object
`get(key)`	Retrieves the value for a specific key
`set(key, value)`	Updates a key and triggers a mutation
`setState(obj)`	Sets multiple keys using a batch update
`merge(obj)`	Alias for `setState()` with shallow merge semantics
`clear()` / `reset()`	Removes all keys from the reactive store
`serialize()`	Provides a cached JSON string of current state
`select([keys])`	Returns a shallow clone of selected state keys
`has(key)`	Checks if a key exists in the current state
`hasChanged(key, value)`	Compares value against current key for mutation check
`isEmpty()`	Returns `true` if state has no keys
`changedKeys()`	Lists keys that were modified since last snapshot
`lastModified()`	Returns timestamp of the most recent mutation

Event System API

Event System API	Description
`on(type, fn)`	Registers a handler for the given event type
`off(type, fn?)`	Removes a specific handler or all handlers for a type
`emit(type, payload)`	Triggers a custom event with optional payload
`onAny(fn)`	Registers a wildcard listener for all event types
`watch(key, fn)`	Subscribes to changes on a specific key
`once(key, fn)`	Registers a one-time listener for key mutation
`subscribe(fn)`	Observes global snapshot updates
`unbindAll()`	Clears all listeners and subscriptions
`listeners()`	Returns an array of active event types
`has(type)`	Checks if any listener exists for a given type
`isWatched(key)`	Returns true if key has an active watcher

Plugin Lifecycle

Plugin Lifecycle Method	Description
`use(pluginFn, options?)`	Registers a plugin with optional configuration
`unuse(pluginFn)`	Unregisters a plugin and triggers its cleanup callback
`pluginAccess`	Determines scope of access: `open`, `locked`, `restricted`
`createPluginAPI()`	Generates an internal scoped API for safe plugin logic

Plugins never override core logic. They only extend or observe. All mutations tracked.

Built-In Plugins

Plugin	Purpose
`persist(key?)`	Saves current state snapshot to `localStorage`
`load(key?)`	Restores or hydrates state from `localStorage`
`freeze()` / `thaw()`	Enables or disables immutable state mode
`dump()`	Returns a cloned snapshot of the current state
`linkToDOM(selector, attr)`	Binds serialized state to a DOM attribute
`unbindDOM()`	Removes DOM listener created by `linkToDOM()`
`configurable()`	Exposes runtime configuration flags and settings
`devtools()`	Displays current state using `console.table()`
`inspectKey(key)`	Returns the current value and watcher status

MeatChronicle Plugin *(if enabled)*

Method	Description
`undo(key)`	Revert key to its previous value
`rollback(key)`	Alias for `undo(key)`
`rollbackAll()`	Revert all tracked keys to their prior states
`safe(fn)`	Execute block with rollback on error
`getHistory(key)`	Retrieve timeline of all mutations for a key
`clearHistory(key?)`	Clear mutation history (for a key or all)
`logHistory(key)`	Display mutation history as a console.table
`historySnapshot()`	Clone full mutation history for inspection
`logMessage(msg, ctx)`	Log a timestamped debug message with context

logState Plugin *(if enabled)*

Method	Description
`logState()`	Displays the current reactive state as `console.table()`

Config

Config Key	Description
config.debug	Enable debug output
pluginAccess	Defines scope for plugin APIs
Immutable Mode	Controlled by freeze/thaw
Safe Mode	Enabled per plugin via access flag
Chronicle Retention	Set via plugin options { limit }

Glossary of Terms

Term	Definition
State	Core reactive store
Mutation	Any state change
Immutable	Copy-based state behavior
Listener	Function hooked into signal
Event Bus	Internal reactive pub/sub
Plugin	External extension to core logic
Snapshot	Serialized or frozen state
Rollback	Reversion of key to prior value
Sync	Client-server state transmission

Term	Definition
Wildcard Listener	Listener invoked for all events
Plugin Clash	Method overwrite warning

Common Warnings

Warning Code	Description
PLUGIN_CLASH	Plugin tried to overwrite core method
PERSIST_FAIL	localStorage write failure (e.g. quota, blocked)
HANDLER_ERROR	Error occurred inside listener callback
WILDCARD_ERROR	Exception thrown from wildcard onAny listener
CONFLICTING_ACCESS_MODE	Plugin attempted mutation in locked mode
SAFE_EXEC_ERROR	Error caught during safe() execution; rollback

Recent Additions *(validated and included)*

- `setState()` and `merge()` aliases confirmed.
- `select()` for partial access.
- `hasChanged()` for comparison logic.
- `dump()` added for raw snapshot.
- `inspectKey(key)` shows watchers and value.
- `safe(fn)` for transaction rollback.
- `logState()` and `devtools()` for console visibility.
- `Chronicle` audit features fully included.
- `pluginAccess` config confirmed per plugin.
- `linkToDOM` and `unbindDOM` verified.
- `wildcard` and `once` listeners included.
- `createPluginAPI()` internal exposed for reference.

Everything surfaced. Nothing omitted. This reference block reflects every known public method, plugin interface, warning, and glossary item inside Meat’s runtime as of current specification.

Self-Test Your Meat Knowledge

1. *What is the primary purpose of the Meat library?*

☐ **A**. To create animations
☐ **B**. To manage reactive state efficiently
☐ **C**. To handle HTTP requests
☐ **D**. To build UI components

2. *Which method is used to subscribe to state changes in Meat?*

☐ **A**. meat.watch()
☐ **B**. meat.listen()
☐ **C**. meat.observe()
☐ **D**. meat.subscribe()

3. *What does the meat.safe() method do?*

☐ **A**. Ensures state changes are permanent
☐ **B**. Prevents state changes from being logged
☐ **C**. Executes a block of code and rolls back changes if an error occurs
☐ **D**. Locks the state to prevent modifications

4. *How can you persist Meat state across page reloads?*

☐ **A**. Using meat.save() and meat.restore()
☐ **B**. Using meat.serialize() and meat.hydrate()
☐ **C**. Using meat.store() and meat.retrieve()
☐ **D**. Using meat.persist() and meat.load()

5. *Which plugin enables undo and rollback functionality?*

☐ **A**. MeatUndo
☐ **B**. MeatChronicle
☐ **C**. MeatHistory
☐ **D**. MeatRollback

6. *What is the purpose of meat.watch()?*

☐ **A**. To log all state changes
☐ **B**. To execute code when the application starts
☐ **C**. To monitor changes to a specific key in the state
☐ **D**. To prevent changes to a specific key

7. *How can you remove a listener added with meat.watch()?*

☐ **A**. By using meat.unsubscribe()
☐ **B**. By using meat.removeListener()
☐ **C**. By using meat.teardown()
☐ **D**. By calling the returned stop function

8. *What does meat.merge() do?*

☐ **A**. Replaces the current state with a new object
☐ **B**. Combines the current state with a new object
☐ **C**. Splits the state into multiple objects
☐ **D**. Deletes keys from the state

9. *Which method is used to serialize the state for storage?*

☐ **A**. meat.persist()
☐ **B**. meat.save()
☐ **C**. meat.serialize()
☐ **D**. meat.export()

10. *What is the purpose of meat.devtools()?*
☐ **A**. To enable debugging features
☐ **B**. To log all state changes
☐ **C**. To reset the state
☐ **D**. To display the current state in a developer-friendly format

11. *How can you create a custom plugin for Meat?*

☐ **A**. By using the meat.plugin() method
☐ **B**. By defining a function that receives the API object
☐ **C**. By extending the Meat class
☐ **D**. By modifying the core Meat library

12. *What does meat.changedKeys() return?*

☐ **A**. The previous state before changes
☐ **B**. The entire state object
☐ **C**. An array of keys that have been modified
☐ **D**. The keys that are currently being watched

13. *How can you trigger a callback only once for a specific key?*

☐ **A**. Using meat.listen()
☐ **B**. Using meat.once()
☐ **C**. Using meat.watch()
☐ **D**. Using meat.subscribe()

14. *What is the purpose of meat.linkToDOM()?*

☐ **A**. To update the DOM when the state changes
☐ **B**. To remove DOM elements when the state changes
☐ **C**. To bind state changes directly to DOM attributes
☐ **D**. To create DOM elements based on the state

15. *Which method is used to reset the entire state?*

☐ **A**. meat.clear()
☐ **B**. meat.rollbackAll()
☐ **C**. meat.undoAll()
☐ **D**. meat.reset()

Quiz Answers: Self-Test Your Meat Knowledge

1. *What is the primary purpose of the Meat library?*

☐ **A**. To create animations
■ **B**. **To manage reactive state efficiently**.
☐ **C**. To handle HTTP requests
☐ **D**. To build UI components

2. *Which method is used to subscribe to state changes in Meat?*

☐ **A**. meat.watch()
☐ **B**. meat.listen()
☐ **C**. meat.observe()
■ **D**. **meat.subscribe()**.

3. *What does the meat.safe() method do?*

☐ **A**. Ensures state changes are permanent
☐ **B**. Prevents state changes from being logged
■ **C**. **Executes a block of code and rolls back changes if an error occurs**.
☐ **D**. Locks the state to prevent modifications

4. *How can you persist Meat state across page reloads?*

☐ **A**. Using meat.save() and meat.restore()
☐ **B**. Using meat.serialize() and meat.hydrate()
☐ **C**. Using meat.store() and meat.retrieve()
■ **D**. **Using meat.persist() and meat.load()**.

5. *Which plugin enables undo and rollback functionality?*

☐ **A**. MeatUndo
■ **B**. **MeatChronicle**.
☐ **C**. MeatHistory
☐ **D**. MeatRollback

6. *What is the purpose of meat.watch()?*

☐ **A**. To log all state changes
☐ **B**. To execute code when the application starts
■ **C**. **To monitor changes to a specific key in the state**.
☐ **D**. To prevent changes to a specific key

7. *How can you remove a listener added with meat.watch()?*

☐ **A**. By using meat.unsubscribe()
☐ **B**. By using meat.removeListener()
☐ **C**. By using meat.teardown()
■ **D**. **By calling the returned stop function**.

8. *What does meat.merge() do?*

☐ **A**. Replaces the current state with a new object
■ **B**. **Combines the current state with a new object**.
☐ **C**. Splits the state into multiple objects
☐ **D**. Deletes keys from the state

9. *Which method is used to serialize the state for storage?*

☐ **A**. meat.persist()
☐ **B**. meat.save()
■ **C**. **meat.serialize()**.
☐ **D**. meat.export()

10. *What is the purpose of meat.devtools()?*

☐ **A**. To enable debugging features
☐ **B**. To log all state changes
☐ **C**. To reset the state
■ **D**. **To display the current state in a developer-friendly format**.

11. *How can you create a custom plugin for Meat?*

☐ **A**. By using the meat.plugin() method
■ **B**. **By defining a function that receives the API object**.
☐ **C**. By extending the Meat class
☐ **D**. By modifying the core Meat library

12. *What does meat.changedKeys() return?*

☐ **A**. The previous state before changes
☐ **B**. The entire state object
■ **C**. **An array of keys that have been modified**.
☐ **D**. The keys that are currently being watched

13. *How can you trigger a callback only once for a specific key?*

☐ **A**. Using meat.listen()
■ **B**. **Using meat.once()**.
☐ **C**. Using meat.watch()
☐ **D**. Using meat.subscribe()

14. *What is the purpose of meat.linkToDOM()?*

☐ **A**. To update the DOM when the state changes
☐ **B**. To remove DOM elements when the state changes
■ **C**. **To bind state changes directly to DOM attributes**.
☐ **D**. To create DOM elements based on the state

15. *Which method is used to reset the entire state?*

☐ **A**. meat.clear()
☐ **B**. meat.rollbackAll()
☐ **C**. meat.undoAll()
■ **D**. **meat.reset()**.

Meat Signal Cheatsheet

Concept	API or Pattern	Notes
Create a signal	`meat()`	Base constructor
Read value	`get(key)`	Sync read
Write value	`set(key, value)`	Sync update
React to changes	`watch(key, fn)`	Persistent callback
One-time reaction	`once(key, fn)`	Runs once then unbinds
Batch mutations	`merge({...})`	Atomic multi-key updates
Teardown listener	`watch(...).stop()`	Stop a live signal reaction
DOM binding	`linkToDOM(el, key)`	Live data → attribute binding
Plugin injection	`meat.use(plugin)`	Extend signal behavior
Persist state	`persist()` + `load()`	Store and retrieve signal payload
Safe commit	`meat.safe(fn)`	Rollback logic on failure
Audit mutation	`changedKeys()`	Report changed keys
Snapshot state	`snapshot()`	Read-only forked view

Signal Patterns

A quick mapping of MEAT's core signal patterns to practical mental models:

Signal Pattern	Analogy	Mental Model
`watch(key, fn)`	Security camera	Passive observation
`once(key, fn)`	Tripwire trap	One-shot reaction
`set(key, value)`	Manual override	Direct mutation
`merge({...})`	Cargo manifest	Batched updates
`safe(fn)`	Insurance policy	Rollback protection
`persist()` / `load()`	Flash drive	Save & restore state
`changedKeys()`	Audit log	Track modifications
`snapshot()`	Frozen scene	Read-only fork

Use this table to orient thinking as you design reactive flows, plugins, and rollback strategies. Signals aren't just syntax — they're verbs with purpose.

License

This work is released under a Creative Commons Attribution-ShareAlike 4.0 International License.
You are free to copy, modify, remix, and redistribute it—even commercially—as long as you credit the original author and keep derivative works equally open.

This project was self-funded, independently researched, authored, and tested.
Built with signals, not slides.

Learn more: https://creativecommons.org/licenses/by-sa/4.0/

Final Notes

This handbook ships v1.0.

It’s meant to be used, forked, adapted, and taught.
If you find it useful—whether in production, a pull request, or a workshop—consider that a successful release.

Want to say hi, share feedback, or build something together?
lucianopereira@posteo.es
linkedin.com/in/lucianofedericopereira

Thank you for reading.
Now go ship something.