Foundations: Setup, Mindset, and Your Toolchain □□

Before writing a single line of animation code, you need a rock-solid foundation. This chapter establishes the mental models that separate hobbyists from professionals—understanding that great animation isn't about flashy tricks, but about **timing, motion, and meaning** working in harmony. We'll configure your WordPress environment with BricksBuilder and PhpStorm so that debugging is painless, script loading is predictable, and you're never fighting your tools instead of creating.

- 1.1.1 — What “Pro-Level GSAP” Actually Means □
- 1.1.2 — How to Use This Course Page-by-Page (Deliverables per Page) □□
- 1.1.3 — The “Animation Triad”: Timing, Motion, Meaning □□
- 1.2.1 Recommended Local Dev Setup: LocalWP / Docker Options Conceptually □□

1.1.1 — What “Pro-Level GSAP” Actually Means

(Skills checklist + what “good” looks like in real WordPress/Bricks projects)

Pro-level GSAP isn’t about knowing every method; it’s about delivering **motion that feels intentional, resilient, performant, and maintainable**—especially under real constraints: responsive layouts, CMS content changes, caching/minification plugins, popups, query loops, and client edits.

Below is a *practical* checklist you can use to self-assess and to guide what we’ll build throughout the course.

---

1) Motion Design Fundamentals (You’re animating meaning, not pixels)

A pro can explain *why* something moves—not just *how*.

1. **Intent & hierarchy**
    1. You can articulate the purpose of each animation:
        1. *Guidance* (direct attention)
        2. *Feedback* (confirm an action)
        3. *Continuity* (connect states)
        4. *Delight* (brand personality—used sparingly)
    2. You avoid “animation for animation’s sake.” If it doesn’t help, it goes.
2. **Timing literacy**
    1. You intentionally choose durations (not random defaults).
    2. You understand the “feel” difference between:
        1. Quick UI feedback (often ~0.12–0.25s)
        2. Content reveals (often ~0.4–0.9s)
        3. Storytelling sequences (often 1.2s+ total, composed of smaller beats)

3. You can *speed up or slow down* an entire experience consistently (tokens / global defaults).

3. **Easing literacy**
    1. You know what easing communicates:
        1. *Ease out* = arrives gently (common for entrances)
        2. *Ease in* = leaves decisively (common for exits)
        3. *Ease in-out* = smooth travel (common for continuous motion)
    2. You keep easing consistent across a site (design system approach).
4. **Spatial consistency**
    1. Your distances match layout scale (e.g., cards move 12–24px, not 200px).
    2. You avoid moving elements so far they feel disconnected from where they came from.
    3. You consider directionality:
        1. Upward motion often reads as “lifting / appearing”
        2. Downward motion often reads as “dropping / dismissing”
        3. Horizontal motion can imply “navigation / progression”

---

2) GSAP Core Mastery (Fluent with the primitives) □□

A pro is fast because they deeply understand the foundational building blocks.

1. **Tween fluency**
    1. You know when to use:
        1. `gsap.to()` (animate *to* a known end state)
        2. `gsap.from()` (reveal from an initial state without manually setting it in CSS)
        3. `gsap.fromTo()` (when you need exact start and end control)
    2. You understand that `from()` can be tricky if something else sets the starting styles (CSS, inline styles, other scripts).
2. **Property fluency**
    1. You prefer transforms and opacity for performance.
    2. You know the difference between:
        1. `x/y` vs `translateX/translateY`
        2. `autoAlpha` vs `opacity`
            1. `autoAlpha` toggles visibility at `0` (often ideal for entrances/exits)
    3. You recognize “expensive” properties (e.g., layout-triggering ones like `top/left/width/height`) and avoid them when possible.
3. **Stagger fluency**
    1. You can stage lists of elements without writing loops manually.
    2. You can produce different rhythms:
        1. Subtle (small stagger, gentle ease)

    2. Snappy (short duration, tighter stagger)
    3. Dramatic (longer stagger, stronger ease)
4. **Callbacks & lifecycle**
    1. You use callbacks for *integration*:
        1. Toggling classes
        2. Updating ARIA states
        3. Enabling pointer events only after animation completes
    2. You avoid heavy work in `onUpdate` that causes jank.

---

3) Timelines (The “pro multiplier”) □□

Pros default to timelines because they scale.

1. **Composition & readability**
    1. You structure timelines so the sequence reads like a story.
    2. You use labels and the position parameter to avoid “magic delays.”
2. **Control**
    1. You can `play()`, `pause()`, `reverse()`, `restart()`, and set progress.
    2. You can attach a timeline to UI state (menu open/close, modal in/out).
3. **Modularity**
    1. You can write functions that **return timelines**:
        1. One function per component (hero, cards, nav, modal)
        2. No shared global selectors that accidentally target other sections

---

4) ScrollTrigger Competence (Real-world scroll without fragile hacks) □□

Scroll is where “toy demos” often break. Pros make it robust.

1. **Trigger calibration**
    1. You understand `trigger`, `start`, `end`, `toggleActions`, `scrub`, and `pin`.
    2. You use markers for calibration during dev—and remove/disable for production.

2. **Responsiveness**
    1. You implement different scroll behaviors by viewport size using `ScrollTrigger.matchMedia()`.
    2. You anticipate layout changes (fonts, images, accordions) and handle refresh correctly.
3. **Performance**
    1. You batch repeated elements rather than creating a heavy trigger per item when needed.
    2. You avoid massive scrubbed transforms on huge images unless you've tested mobile.

---

5) WordPress + Bricks Integration (Where pros quietly win) □□

In WP/Bricks, "it works on my machine" isn't enough.

1. **Script loading discipline**
    1. You know where code should live:
        1. Global vs page vs template vs component
    2. You avoid loading the same GSAP init multiple times through templates/popups.
2. **Targeting strategy**
    1. You use stable selectors:
        1. Classes and `data-*` attributes (often best)
        2. Avoid brittle auto-generated IDs or deeply nested selectors
    2. You scope animations per component instance:
        1. If a section repeats, each instance animates independently.
3. **Dynamic DOM awareness**
    1. You handle:
        1. Bricks popups/off-canvas that mount/unmount
        2. Query loops (unknown item count)
        3. AJAX-ish content changes (where applicable)
    2. You defend against missing elements (no console errors on pages where a component doesn't exist).
4. **Compatibility with caching/minification**
    1. You know how to debug when:
        1. Scripts are deferred
        2. Order changes
        3. Minifiers rewrite or bundle unexpectedly

---

6) Maintainability (Code you can ship and revisit) □□

Pros are trusted because their work stays stable over time.

1. **Consistent patterns**
    1. You have a repeatable component pattern:
        1. Find root element
        2. Query children within root
        3. Create timeline/tweens
        4. Return cleanup function (when applicable)
2. **Configuration & “motion tokens”**
    1. You define shared constants for:
        1. Durations (fast/medium/slow)
        2. Eases (standard set)
        3. Distances (small/medium/large)
    2. You can adjust the entire site’s “feel” centrally without editing 40 tweens.
3. **Naming and documentation**
    1. Your code reads like intent, not like a puzzle.
    2. You leave notes where Bricks/WP behavior is non-obvious.

---

7) Accessibility & UX Responsibility ♿

This is non-negotiable at a pro level.

1. **Reduced motion**
    1. You respect `prefers-reduced-motion`:
        1. Provide alternative behavior (fade only, shorter motion, or none)
        2. Don’t punish users with broken layouts if animations are disabled
2. **Keyboard & focus**
    1. You don’t animate in a way that traps or loses focus.
    2. For modals/menus, you synchronize:
        1. Visual state
        2. ARIA state
        3. Focus management
3. **Avoiding “harmful” motion**

1. You avoid aggressive parallax and excessive scrub on large sections unless there’s a strong reason.
2. You consider readability for text animations (no over-kinetic type).

---

8) Performance Engineering (Smooth on real devices) ⚙

Pros test on the devices clients actually have.

1. **Performance intuition**
    1. You understand the rough cost stack:
        1. Layout (often expensive)
        2. Paint (can be expensive)
        3. Composite (usually best-case)
    2. You choose properties and strategies that land you in “composite” territory (transforms/opacity).
2. **Profiling**
    1. You can use DevTools to:
        1. Spot long frames
        2. Identify layout thrashing
        3. Confirm smoothness during scroll
3. **Cleanup**
    1. You kill timelines and ScrollTriggers when they shouldn’t persist.
    2. You avoid duplicate instances after page builder edits, partial reloads, or popup reopens.

---

9) A Quick Self-Assessment (Score yourself) □□

Rate each item **0–2**:

1. **Design intent**
    1. I can explain the purpose of each animation.
2. **Tween fundamentals**
    1. I confidently use `to/from/fromTo`, easing, stagger, callbacks.
3. **Timelines**

    1. I structure sequences with labels/positioning (not delay hacks).
4. **ScrollTrigger**
    1. I can build responsive, calibrated triggers and debug them.
5. **Bricks/WP integration**
    1. My selectors are stable, scoped, and my scripts don’t duplicate.
6. **Accessibility**
    1. I respect reduced motion and keyboard/focus needs.
7. **Performance**
    1. I can profile, optimize, and keep things smooth on mobile.
8. **Maintainability**
    1. My code is modular, reusable, and documented.

Interpretation

1. **0–6**: You can build effects, but they may be fragile or inconsistent.
2. **7–11**: You’re functional; pro habits are emerging.
3. **12–16**: You’re operating professionally—now refine style, systems, and speed.

1.1.2 — How to Use This Course Page-by-Page (Deliverables per Page) □□

This course is designed like a **production checklist**: every page (lesson) ends with **clear deliverables** you can verify in your Bricks/WordPress build. The goal is that you *never* "kind of understand"—you either **shipped the artifact** (code + Bricks setup + notes), or you didn't.

The Page-by-Page Workflow (Repeatable Every Time) □

1) Before You Start: Prepare the "Lesson Sandbox" (5–10 min)

1. **Pick one page to implement** (e.g., `1.1.2`) and commit to finishing it end-to-end.
2. **Choose a controlled location in Bricks**:
   1. A dedicated *Course Sandbox* page (recommended), or
   2. A specific template/section you won't reuse yet.
3. **Disable variables that hide problems** (temporarily):
   1. Cache/minify plugins (or bypass via dev mode)
   2. CDN optimizations
   3. Aggressive script deferring (until your baseline works)

Deliverable: A dedicated Bricks page/template you can safely break and reset.

---

2) Implementation: Build What the Lesson Asks—Nothing More □□

Each lesson page will include:

1. **Goal** (what you're building)
2. **Why it matters** (the mental model)
3. **Bricks implementation** (exact placement + selectors)
4. **Code** (copy/paste + explanation)
5. **Micro-exercises** (small variations that prove understanding)
6. **Troubleshooting** (common failure modes)

Rule: implement *exactly* the described effect first.
Polish (styling, extra features) comes **after** the effect is reliable.

Deliverable: The animation/behavior works exactly as described in the lesson.

---

3) Verification: Prove It Works (Not Just "Looks OK") □□

For every page, you'll verify four categories:

1. **Functional verification**
   1. Does it run on first load?
   2. Does it run after refresh?
   3. Does it run on hard reload (cache bypass)?
2. **Selector verification**
   1. Does it still work if there are *two* instances of the section?
   2. Does it break if the DOM structure changes slightly?
3. **Responsive verification**
   1. Test at least: mobile, tablet, desktop breakpoints
   2. Confirm no overflow, clipping, or unexpected shifts
4. **Accessibility/motion sanity check**
   1. If motion is significant: consider `prefers-reduced-motion`
   2. Ensure no focus traps or invisible-but-clickable overlays

Deliverable: A short checklist (written by you) marking what you tested.

What “Deliverables per Page” Actually Means □□

Each page produces *artifacts* you keep. Over time, these become your personal GSAP toolkit.

Core Deliverables (Every Page)

1. **Working demo inside Bricks**
   1. A section/page where the effect is visible
   2. Uses robust selectors (usually classes/data attributes)
2. **Saved code snippet**
   1. Stored in your project in a predictable place (see “Folder pattern” below)
   2. Includes a short header comment: what it does + how to apply
3. **One-liner notes**
   1. What surprised you?
   2. What broke first?
   3. What you’d do differently next time

Optional Deliverables (When the page includes them)

1. **Reusable function** (e.g., `initHeroReveal(rootEl)`)
2. **Mini utility** (e.g., a scoped selector helper)
3. **A variation** (e.g., “same reveal but staggered”)

Standard Folder + Naming Pattern (So You Don’t Lose Anything) □□

Even if you’re not using a bundler yet, use a consistent structure.

1. Create a “course lab” area in your theme or child theme (or a snippets plugin):
   1. `gsap-lab/`
   2. `gsap-lab/pages/`
   3. `gsap-lab/utils/`

2. Name files by lesson number:
    1. `pages/01-01-02-workflow.js`
    2. Later: `pages/04-01-first-tween.js`, etc.
3. Add a short header comment at the top of each file:
    1. What it does
    2. Where it’s used (Bricks page/template)
    3. Dependencies (GSAP core, ScrollTrigger, etc.)

Deliverable: A place where your “lesson code” lives permanently, not in random Bricks textareas.

Bricks Placement Rules (So Scripts Don’t Turn Into Chaos) □□

You’ll repeatedly choose *where* code goes. Use these rules while learning:

1. **During learning (recommended):**
    1. Put lesson JS in **one** predictable place (global or the sandbox page)
    2. Keep selectors scoped to the sandbox section to avoid side effects
2. **When you turn a lesson into a reusable component:**
    1. Move code into a “component init” function
    2. Initialize per section instance (important for query loops/repeated sections)

Deliverable: You can explain *why* your code is placed where it is (not “because it worked once”).

The “Two-Instance Test” (Your Fastest Pro Filter) □□

A huge amount of WordPress/Bricks animation pain comes from accidental global targeting.

Do this after most lessons:

1. Duplicate the animated section in Bricks (same page).
2. Reload.
3. Confirm:
    1. Both instances animate correctly
    2. One instance doesn’t hijack the other’s elements

3. Timing doesn’t “double fire”

If it fails, you learn a key professional habit: **scope your selectors**.

Deliverable: A screenshot or note: “Two-instance test: pass/fail + fix”.

What You Should Send Me After Completing Any Page □□

When you finish a lesson page, reply with:

1. **Where you put the code** (Bricks global/page/element OR theme file)
2. **Your selectors** (classes/data attributes you used)
3. **What worked immediately**
4. **What broke**
5. **One screenshot** (optional but helpful) or a short description

Then I can:

1. Suggest improvements (scoping, maintainability)
2. Flag performance/accessibility risks early
3. Provide a cleaner “component version” of your solution

Your Deliverables for *This* Page (1.1.2) □

1. A written **personal workflow checklist** you will follow for each lesson page (copy the sections above and simplify to your taste).
2. A **sandbox page/template in Bricks** dedicated to course experiments.
3. A **project structure decision**:
   1. Where lesson JS will live *for now*
   2. How you will name and store lesson code snippets
4. A quick **two-instance test habit** added to your checklist.

Quick Micro-Exercise (5 minutes) ✍

Create a checklist you can paste at the top of each future lesson file:

1. Goal:
2. Where installed (Bricks/page/global/file):
3. Selectors used:
4. Tested:
    1. Refresh
    2. Hard reload
    3. Two-instance test
    4. Mobile/tablet/desktop
    5. Reduced motion (if relevant)
5. Notes:

1.1.3 — The “Animation Triad”: Timing, Motion, Meaning □□

Professional-looking animation isn’t primarily about “cool effects”—it’s about making the user *feel* that the interface is responsive, intentional, and clear. This lesson gives you a **repeatable decision framework** you can apply to *any* GSAP animation in WordPress/Bricks.

---

https://codepen.io/editor/ZcarecroW/pen/019da77a-6995-7501-9af8-4866e69b7f1c

---

1) Goal □

By the end of this page you will be able to:

1. **Diagnose** why an animation feels “off” using *Timing / Motion / Meaning*.
2. **Design** an animation intentionally (instead of guessing eases/durations).
3. **Implement** a tiny Bricks sandbox demo that lets you *feel* the triad by toggling settings.
4. Build the habit of writing a **1–2 line “meaning statement”** before animating.

---

2) The Triad (The Mental Model) □□

2.1 **Timing** = *When* and *how fast* things happen □

Timing includes:

1. **Duration**: how long a tween runs.
2. **Delay**: how long before it begins.

3. **Stagger**: spacing between repeated items (lists, cards).
4. **Rhythm**: consistency across the site (so the UI feels coherent).

What timing communicates:

1. *Fast* timing → responsiveness, precision, confidence (common in UI).
2. *Slower* timing → weight, calm, luxury (common in editorial / hero).
3. Inconsistent timing → "cheap" feeling because the site has no rhythm.

Practical rules of thumb (web UI):

1. Micro feedback (hover/press/focus):
   1. Usually **0.08–0.20s**
   2. Should feel instant, not "animated"
2. Component transitions (menu open, accordion, modal):
   1. Usually **0.25–0.60s**
3. Decorative/hero sequences:
   1. Often **0.6–1.2s**, but keep it purposeful

> If you're unsure: start at **0.35s** for UI transitions and adjust.

2.2 **Motion** = *What path/shape* the movement takes □□

Motion includes:

1. **Properties** you animate:
   1. Prefer `x/y`, `scale`, `rotate`, `opacity/autoAlpha`
   2. Avoid animating `top/left/width/height` unless you know why (layout + jank risk)
2. **Easing** (the velocity curve)
3. **Distance** (how far an element moves)
4. **Direction** (does it match the visual hierarchy and reading direction?)

What motion communicates:

1. Ease-out-ish motion → natural settling (common for UI entering)
2. Overshoot/bounce → playful or "toy-like" (use deliberately)
3. Too much distance → feels like the element is "traveling" instead of "appearing"

A simple motion quality checklist:

1. Does it feel like it has **mass**? (even a tiny bit)
2. Does it start/stop in a way that matches the product tone?
3. Does motion support the layout (not fight it)?

2.3 **Meaning** = *Why* the animation exists

Meaning is the most ignored—and the most “pro”—part.

Meaning includes:

1. **What the user learns** from the animation:
    1. “This opened.”
    2. “This is clickable.”
    3. “This is the next step.”
2. **What it prioritizes**:
    1. Which element is “primary” in the moment?
3. **What it prevents**:
    1. Confusion
    2. Misclicks
    3. Losing context during state changes

Meaning statement (habit): before animating, write:

1. **Trigger:** what causes it? (load / click / scroll)
2. **User benefit:** what clarity/feedback do they get?
3. **Constraint:** what must not happen? (motion sickness, layout shift, delay)

Example:

1. “When the modal opens, the backdrop fades in quickly to signal a new layer, then the dialog rises slightly to confirm focus moved to it—without shifting layout.”

If you can’t write the meaning statement, you’re probably adding motion “because it looks cool” (which is fine in a Dribbble shot, risky in production).

---

3) How the Triad Solves Real Problems (Common “It Feels Off” Diagnoses) □□

3.1 The animation feels **laggy** □□

Usually a **timing** issue (too slow, too much delay), or a **meaning** issue (feedback arrives too late).

Fixes:

1. Reduce delay (often to `0`).
2. Shorten duration by 20–40%.
3. Add a tiny *instant* cue:
    1. e.g. quick `autoAlpha` change first, then movement

3.2 The animation feels **floaty / cheap** □□

Usually a **motion** issue (ease doesn’t match UI), sometimes too much distance.

Fixes:

1. Use a more “UI” ease:
    1. `power2.out`, `power3.out`, `expo.out` (careful—expo can be dramatic)
2. Reduce travel distance:
    1. `y: 24` instead of `y: 80`

3.3 The animation feels **confusing** □□

Almost always a **meaning** issue.

Fixes:

1. Make the state change readable:
    1. Backdrop + dialog separation for overlays
2. Use direction to explain hierarchy:

    1. “New layer” often comes forward/up slightly
3. Don’t animate everything:
    1. Animate the *one* thing that clarifies what changed

3.4 The animation is “fine” but the site feels **inconsistent** □□

A **timing rhythm** problem (durations/eases vary randomly).

Fixes:

1. Define site motion tokens (you’ll formalize later in Chapter 14):
    1. 2–3 durations and 2–3 eases you reuse everywhere

---

4) Bricks Implementation (Sandbox Demo) □□

You’ll build a small section with:

1. A headline + paragraph + button.
2. A “Play” button that runs a reveal animation.
3. A “Mode toggle” that switches between:
    1. **UI mode** (snappy, restrained)
    2. **Hero mode** (slower, more cinematic)

4.1 Bricks structure (what to create)

Create a **Section** (or Container) and add:

1. A wrapper container with class: `triad-demo`
2. Inside it:
    1. Heading (H2) with class: `triad-title`
    2. Text (p) with class: `triad-text`
    3. Button row container with class: `triad-controls`
        1. Button 1: “Play” with class: `triad-play`
        2. Button 2: “Toggle mode” with class: `triad-toggle`

3. (Optional) small label span with class: `triad-mode`

Why these classes?
They're stable, readable, and easy to scope—perfect for WordPress where DOM can change.

4.2 Minimal CSS (Bricks → Page Settings → Custom CSS)

```
.triad-demo {
  max-width: 720px;
  margin: 0 auto;
  padding: 48px 24px;
}

.triad-controls {
  display: flex;
  gap: 12px;
  align-items: center;
  margin-top: 16px;
}

/* Optional: make the “mode” label subtle */
.triad-mode {
  margin-left: 8px;
  font-size: 14px;
  opacity: 0.75;
}
```

4.3 JavaScript (Bricks → Page Settings → Custom Code → Footer Scripts)

> Put it in **Footer** so the DOM exists when this runs (we'll formalize DOM-ready patterns in `2.1.1`)

```
(() => {
  // Scope everything to each .triad-demo instance (so duplicates on the page won’t collide).
  const demos = document.querySelectorAll(".triad-demo");
  if (!demos.length) return;

  demos.forEach((root) => {
    const title = root.querySelector(".triad-title");
    const text = root.querySelector(".triad-text");
    const playBtn = root.querySelector(".triad-play");
    const toggleBtn = root.querySelector(".triad-toggle");
    const modeLabel = root.querySelector(".triad-mode");

    if (!title || !text || !playBtn || !toggleBtn) return;

    // Two presets to FEEL the difference between "UI motion" and "Hero motion".
    const presets = {
      ui: {
        label: "UI mode",
        timing: { duration: 0.38, stagger: 0.06 },
        motion: { y: 18, ease: "power2.out" },
        meaning: "Fast clarity: content appears quickly with minimal travel."
      },
      hero: {
        label: "Hero mode",
        timing: { duration: 0.9, stagger: 0.12 },
        motion: { y: 42, ease: "power3.out" },
        meaning: "Cinematic emphasis: slower, longer travel, more presence."
      }
    };

    let mode = "ui";

    const updateModeUI = () => {
      if (modeLabel) {
        modeLabel.textContent = `${presets[mode].label} — ${presets[mode].meaning}`;
      }
      toggleBtn.setAttribute("aria-pressed", mode === "hero" ? "true" : "false");
    };

    // Build a function that returns a timeline, so it’s easy to reuse.
```

```
    const buildRevealTl = () => {
      const p = presets[mode];

      // Clear any inline transforms/opacity from previous runs (keeps repeat plays
consistent).
      gsap.set([title, text, playBtn], { clearProps: "all" });

      // Set initial state (meaning: they are "not yet here")
      gsap.set([title, text, playBtn], { autoAlpha: 0, y: p.motion.y });

      const tl = gsap.timeline();
      tl.to([title, text, playBtn], {
        autoAlpha: 1,
        y: 0,
        duration: p.timing.duration,
        ease: p.motion.ease,
        stagger: p.timing.stagger
      });

      return tl;
    };

    let activeTl = null;

    playBtn.addEventListener("click", () => {
      // Prevent double-firing overlapping timelines
      if (activeTl) activeTl.kill();
      activeTl = buildRevealTl();
    });

    toggleBtn.addEventListener("click", () => {
      mode = mode === "ui" ? "hero" : "ui";
      updateModeUI();
    });

    // Initial label
    updateModeUI();
  });
})();
```

5) Micro-Exercises (Prove You Understand the Triad) □□

Do these in order—each targets *one* part of the triad.

1. **Timing-only experiment (no motion changes):**
   1. Keep `y` and `ease` the same
   2. Change `duration` from `0.38` → `0.22`
   3. Observe: does it feel more “responsive” or more “harsh”?
2. **Motion-only experiment (same duration):**
   1. Keep duration constant
   2. Change ease:
      1. `power2.out` → `power1.out` (more linear feel)
      2. `power2.out` → `expo.out` (more dramatic snap)
   3. Describe in one sentence what changed emotionally.
3. **Meaning experiment (change what’s animated):**
   1. Animate only the **title** and leave text/button static
   2. Ask: is the state change still clear? What did you lose?
4. **Rhythm experiment (site consistency):**
   1. Pick one duration + ease you like for UI:
      1. e.g. `duration: 0.35`, `ease: power2.out`
   2. Write it down somewhere as your first “motion token”.

---

6) Troubleshooting (Common Failure Modes) □□

1. **Nothing happens when clicking “Play”**
   1. Confirm GSAP is actually loaded on the page:
      1. In DevTools Console, run:

         ```
         typeof gsap
         ```

         It should return `"function"` or `"object"` (not `"undefined"`).
   2. Confirm your class names match exactly:
      1. `.triad-demo`, `.triad-title`, `.triad-text`, `.triad-play`, `.triad-toggle`
2. **Only the first demo works (if you duplicated the section)**

1. This lesson’s code scopes per `.triad-demo` instance—so it *should* work.
2. If it doesn’t, you likely reused IDs or targeted global selectors elsewhere.

3. **Elements jump / flicker**
    1. Ensure the initial state is set *before* animating:
        1. We use `gsap.set(...autoAlpha: 0, y: ...)` for that purpose.
    2. If you have CSS transitions on these elements, remove them (CSS transitions can fight GSAP).
4. **The mode label doesn’t show**
    1. You didn’t add the optional `.triad-mode` element—either add it, or ignore it (the demo still works).

---

7) Verification Checklist (Do This Every Lesson) □

1. **Functional**
    1. Click Play: does it animate?
    2. Click Toggle mode then Play: does it feel noticeably different?
2. **Hard refresh**
    1. Reload with cache bypass (your browser’s hard reload) and retest.
3. **Two-instance test**
    1. Duplicate the `.triad-demo` section in Bricks
    2. Confirm both work independently
4. **Basic accessibility sanity**
    1. Toggle button has `aria-pressed` updating (it does)
    2. Nothing becomes invisible-but-clickable (we use `autoAlpha`, which toggles visibility)

---

8) Your Output (What to Save) □□

1. A short note in your project (or a text file) answering:
    1. **Which preset felt better** for UI?
    2. What duration/ease did you choose as your first “token”?
2. Save this snippet as:
    1. `pages/01-01-03-animation-triad.js`
3. Add a header comment at the top of the file like:
    1. What it does
    2. Where it’s used (Bricks sandbox page)
    3. Dependencies: GSAP core

1.2.1 Recommended Local Dev Setup: LocalWP / Docker Options Conceptually □□

This lesson is about building a **safe, fast, professional development environment** for WordPress animation work with **BricksBuilder, GSAP, and PhpStorm**.

If you want to become *really good* at GSAP on WordPress sites, your setup matters more than most beginners realize. A weak setup causes confusion:

1. You don’t know whether a bug comes from:
    1. your code,
    2. WordPress,
    3. caching,
    4. the host,
    5. or Bricks.
2. You avoid experimenting because breaking a live site feels risky.
3. You waste energy on deployment and environment problems instead of learning animation.

A solid local setup fixes that. □

What you are trying to achieve

Your goal is to create a local WordPress environment where you can:

1. build pages in **BricksBuilder**,
2. write and organize **JavaScript, CSS, and PHP** cleanly,
3. test **GSAP animations** safely,
4. debug quickly in the browser and in PhpStorm,
5. later move your work to staging/live with fewer surprises.

Think of local development as your **animation laboratory** □□

You should be able to:

1. break things without fear,
2. inspect everything,
3. reset quickly,
4. test ideas fast.

The two main local development approaches

For this course, the two main conceptual paths are:

1. **LocalWP**
2. **Docker-based local environment**

Both are valid.
Both can be professional.
But they serve slightly different types of learners and workflows.

Option 1: LocalWP

What LocalWP is

LocalWP is a desktop application designed to make local WordPress development easy.

It handles things like:

1. PHP,
2. MySQL or MariaDB,
3. web server setup,
4. SSL,
5. site creation,
6. local domains.

Instead of manually configuring all those layers, you click a few buttons and get a working WordPress site.

For a beginner, this is often the best choice because it reduces setup friction dramatically.

Why LocalWP is excellent for this course

For learning GSAP with WordPress and Bricks, LocalWP is great because:

1. **Fast setup**
    1. You can create a new site in minutes.
    2. You spend your time learning animation, not server administration.
2. **Good for experimentation**
    1. You can spin up test sites for specific lessons.
    2. You can destroy and recreate environments with low risk.
3. **WordPress-friendly**
    1. It is built around WordPress workflows.
    2. It feels less abstract than Docker for newcomers.
4. **Easy SSL and local domains**
    1. Useful when testing scripts and realistic browser behavior.
    2. Makes your local environment feel closer to a real site.
5. **Lower cognitive load**
    1. As a beginner, you already need to learn GSAP, JavaScript, CSS, Bricks, and WP structure.
    2. LocalWP avoids adding “container orchestration” to that list too early.

When LocalWP is the right choice for you

Choose **LocalWP first** if:

1. you are still learning WordPress internals,
2. you want to focus on **GSAP + Bricks** rather than DevOps,
3. you want the easiest path to a working local site,
4. you work mostly alone on your machine,
5. you value convenience over full environment portability.

For most students starting this course, **this is my recommendation**

Potential downsides of LocalWP

LocalWP is convenient, but it is not perfect.

1. **Less explicit infrastructure knowledge**
    1. A lot is handled for you.
    2. That’s good at first, but it can hide how the environment actually works.
2. **Machine-specific quirks**

    1. Your setup may behave slightly differently from another developer’s.
    2. This matters more in teams.
3. **Less “infrastructure as code”**
    1. Docker setups are often more reproducible.
    2. With LocalWP, your environment is more GUI-managed.
4. **Advanced customization may be less elegant**
    1. You *can* customize things,
    2. but Docker often gives more systematic control.

These are not deal-breakers. They only matter more as your workflow becomes more advanced.

Option 2: Docker

What Docker is

Docker lets you define your development environment in containers.

A container is a packaged runtime environment for part of your app, such as:

1. PHP,
2. database,
3. web server,
4. mail tools,
5. node tooling.

Instead of saying “it works on my machine,” Docker tries to make the machine behavior more predictable by defining the environment in configuration files.

For WordPress, this often means you have services such as:

1. `wordpress` or `php`,
2. `nginx` or `apache`,
3. `mysql` or `mariadb`,
4. optionally `phpmyadmin`,
5. optionally `mailhog`,
6. optionally `node`.

Why Docker is powerful

Docker is attractive because:

1. **Reproducibility**
    1. Team members can use the same environment definition.
    2. Fewer “works for me” problems.
2. **Version control of environment**
    1. Your environment config can live in the project.
    2. That means setup becomes part of the codebase.
3. **Closer to professional engineering workflows**
    1. Useful if you want to work in teams or agencies.
    2. Useful if you deploy to containerized infrastructure.
4. **More explicit setup**
    1. You become more aware of the layers involved.
    2. That deeper knowledge can help debugging.

Why Docker may be harder at first

For a beginner, Docker adds complexity:

1. You may need to understand:
    1. containers,
    2. images,
    3. ports,
    4. volumes,
    5. service names,
    6. networking.
2. When something breaks, the failure may come from:
    1. WordPress,
    2. PHP,
    3. file permissions,
    4. networking,
    5. container config.
3. That can distract from the main goal of this course:
    1. learning **GSAP deeply**,
    2. inside **WordPress and Bricks**.

So Docker is excellent—but often better as a **phase-2 setup**, not necessarily day-1.

My recommendation for *this* course

Here is the practical recommendation:

1. **Start with LocalWP**
    1. It will get you productive fastest.
    2. It reduces noise while you learn core concepts.
2. **Understand Docker conceptually**
    1. So you know what more advanced teams may use.
    2. So you can migrate later if needed.
3. **Move to Docker later only if one of these becomes true**
    1. you work in a team,
    2. you need reproducible onboarding,
    3. you want environment config in version control,
    4. you become comfortable enough that the added complexity is worth it.

So: **LocalWP now, Docker awareness from the start** □□

The professional mindset: environment goals

No matter which tool you choose, a professional local environment should give you these capabilities:

1. **Isolation**
    1. Each project should be able to run without polluting other projects.
    2. Different PHP or plugin setups should not constantly clash.
2. **Repeatability**
    1. You should be able to recreate the setup.
    2. “I forgot what I installed” is a bad sign.
3. **Visibility**
    1. You should be able to inspect logs, files, network requests, and script loading.
    2. Animation debugging depends on visibility.
4. **Speed**
    1. Slow feedback kills experimentation.
    2. GSAP learning requires many tiny test iterations.
5. **Safety**
    1. You should be free to test destructive changes locally.
    2. Never use a live site as your primary learning playground.

A great starter architecture for your learning setup

Here is the setup I recommend for you as a learner:

1. **One main local WordPress site for the course**
    1. Use it as your central learning sandbox.
    2. Install Bricks and create lesson pages there.
2. **Optional extra “throwaway” test sites**
    1. Use these when a lesson is highly experimental.
    2. For example, testing script loading edge cases or plugin conflicts.
3. **A dedicated child theme or code organization strategy**
    1. Even if Bricks lets you inject code in multiple places, keep your logic organized.
    2. We’ll cover that more later.
4. **PhpStorm connected to the project files**
    1. So your local files and editor reflect the actual WordPress site.
    2. This is crucial for scaling beyond tiny snippets.
5. **Browser DevTools always part of the workflow**
    1. GSAP work is visual.
    2. You need the browser open while coding, not just the editor.

Suggested folder-level mental model

Even before we get into exact project structure, you should understand the broad layers of your project.

A WordPress site typically involves:

1. **Core WordPress**
    1. The CMS itself.
    2. Usually not where you write your custom animation logic.
2. **Themes**
    1. This is often where your site presentation lives.
    2. Child themes are commonly used for safe customizations.
3. **Plugins**
    1. Third-party functionality.
    2. Sometimes custom behavior can also be placed in a plugin.

4. **Uploads / generated content**
    1. Media, cache artifacts, generated assets.
    2. Usually not where handcrafted source code should live.

For this course, your custom GSAP work will usually belong in one of these places:

1. a **child theme**,
2. a **custom code organization layer** used with Bricks,
3. in some cases a **custom plugin** for reusable site behavior.

Why local development is especially important for animation

Animation work is more sensitive than many other frontend tasks.

A regular content change either appears or does not appear.
An animation can fail in much subtler ways:

1. it runs too early,
2. it runs too late,
3. it fights CSS,
4. it stutters,
5. it causes layout shifts,
6. it works on desktop but not mobile,
7. it breaks after minification,
8. it becomes inaccessible for motion-sensitive users.

That means your environment must help you inspect:

1. **timing**,
2. **layout**,
3. **script loading**,
4. **responsive behavior**,
5. **performance**.

A local setup is where you can observe all of this safely.

Brief JavaScript insight: why environment affects JS learning □□

Since you asked for JS insights along the way, here’s an important one.

JavaScript in WordPress is not just “write code and it runs.”
Your code depends on:

1. **when it loads**,
2. **where it loads**,
3. **whether dependencies loaded first**,
4. **whether the target DOM elements exist yet**,
5. **whether another tool modifies the page after load**.

That matters a lot for GSAP.

For example, conceptually, this can fail:

```
gsap.to(".card", { y: 100 });
```

Why?

1. Maybe GSAP isn’t loaded yet.
2. Maybe `.card` doesn’t exist on that page.
3. Maybe Bricks rendered content differently than you expected.
4. Maybe your script runs before the DOM is ready.
5. Maybe CSS prevents you from seeing the movement clearly.

So your local environment is not just for “hosting WordPress.”
It is the stage on which your JavaScript execution model becomes visible.

Brief CSS insight: why environment affects CSS learning □□

GSAP often animates CSS-related properties such as:

1. `transform`,
2. `opacity`,

3. `x` / `y` equivalents through transforms,
4. scale,
5. rotation.

But CSS can interfere in subtle ways.

If an element already has layout constraints or conflicting styles, your animation may appear broken even when the JavaScript is correct.

For example:

1. a parent may have `overflow: hidden`,
2. a flex or grid layout may influence positioning,
3. an element may already be transformed,
4. transitions may conflict with GSAP,
5. responsive styles may override assumptions.

A local environment helps you inspect **computed styles**, not just your source CSS.

That’s a huge professional skill:
you stop guessing, and start verifying.

LocalWP setup strategy I recommend

You asked for detail, so here is the practical strategy I’d use if I were setting you up for this course.

Site profile

Create **one main site** specifically for this course.

Use it as your GSAP lab.

Suggested characteristics:

1. **Clean WordPress install**
    1. Avoid a bloated starter stack at first.
    2. Less noise means easier debugging.
2. **Bricks installed**
    1. This will be your page-building environment.

    2. Keep your experiments tied to actual Bricks workflows.
3. **Minimal plugin count**
    1. Install only what you need.
    2. Extra plugins increase variables and conflict potential.
4. **Readable local domain**
    1. Something like `gsap-bricks-course.local`
    2. Easy to recognize in browser tabs and tools.
5. **SSL enabled if possible**
    1. This keeps the environment closer to modern real-world browsing conditions.
    2. It also helps avoid mixed-content confusion in some cases.

Plugin philosophy for the learning environment

Keep the plugin list minimal.

A beginner mistake is installing many helper plugins too early. That creates mystery behavior.

For the course environment, use only what supports the learning goal.

Good philosophy:

1. install only what you can explain,
2. remove what you don’t actively need,
3. add tooling deliberately, not emotionally.

Why this matters:

1. animation bugs are hard enough,
2. plugin interactions multiply uncertainty,
3. clean systems teach faster.

Theme philosophy

You should aim for a setup where custom code has a clear home.

Even if Bricks allows code injection in multiple places, your long-term goal is maintainability.

At this stage, the high-level rule is:

1. use Bricks for building,

2. use a structured place for reusable custom code,
3. avoid scattering important logic randomly across pages.

We'll refine this throughout the course.

Docker setup strategy I recommend conceptually

If you go the Docker route later, your mindset should be:

1. define services clearly,
2. keep volumes understandable,
3. make project startup predictable,
4. know where your WordPress files live,
5. make sure PhpStorm edits the real mounted code,
6. keep your `.env` and compose setup readable.

In a typical conceptual Docker WordPress project, you would think in terms of:

1. **application service**
    1. Runs WordPress/PHP.
    2. Serves your site logic.
2. **database service**
    1. Stores WP content and settings.
3. **web server layer**
    1. Sometimes separate, sometimes bundled.
4. **persistent volumes**
    1. Preserve DB and/or project files between runs.
5. **port mapping**
    1. Exposes your local site to the browser.
6. **project-mounted code**
    1. Lets you edit files in PhpStorm and see them in the containerized site.

This is powerful—but it requires comfort with infrastructure vocabulary.

Comparison: LocalWP vs Docker

Here is the practical comparison for your use case.

Criteria	LocalWP	Docker
Beginner friendliness	Excellent	Moderate to difficult
Setup speed	Very fast	Slower
WordPress focus	Strong	General-purpose
Learning curve	Low	Higher
Team reproducibility	Medium	High
Environment as code	Low	High
Debugging simplicity for beginners	Better	Harder at first
Best for this course start	**Yes**	Later

So if your question is, *“What should I use now?”*
The answer is: **LocalWP**.

Your actual recommended starting stack

Here is the stack I recommend for you right now:

1. **LocalWP**
    1. For local WordPress site management.
2. **WordPress**
    1. Clean install dedicated to course exercises.
3. **BricksBuilder**
    1. Your visual building system.
4. **PhpStorm**
    1. Your code editor and project navigator.
5. **Chrome or Edge DevTools**
    1. For inspecting the DOM, CSS, network, and JS console.
6. **GSAP loaded in a controlled way**
    1. We’ll cover script loading soon.
    2. Don’t worry about the exact loading method yet.

This is enough to begin professionally without overwhelming yourself.

What “good enough” looks like at this stage

At this point, you do **not** need:

1. a perfect enterprise-grade infrastructure,
2. CI/CD pipelines,
3. a containerized multi-service architecture,
4. a headless WordPress setup,
5. advanced deployment automation.

You **do** need:

1. a stable local WordPress site,
2. Bricks working properly,
3. PhpStorm connected to your files,
4. a repeatable workflow for testing code,
5. confidence that you can experiment safely.

That is the professional beginner target.

Common beginner mistakes in local setup

1. Learning directly on a live site

This is one of the biggest mistakes.

Why it’s bad:

1. every experiment carries risk,
2. caching obscures cause and effect,
3. users may see broken states,
4. fear slows learning.

Use local first. Always.

2. Installing too many plugins

People often install optimization, helper, code snippet, animation, CSS, debug, and utility plugins all at once.

Then when something breaks, they don’t know why.

Better approach:

1. start minimal,
2. add one tool at a time,
3. verify behavior after each addition.

---

3. Mixing code locations randomly

A little code in Bricks page settings, a little in a code snippet plugin, a little in the theme, a little in the footer…

This becomes chaos quickly.

Even before we formalize architecture, keep one principle:

1. if code is reusable, give it a reusable home,
2. if code is page-specific, document where it lives,
3. if code becomes important, move it out of ad hoc locations.

---

4. Ignoring file organization because “it’s just a test”

Today’s quick test becomes tomorrow’s production pattern.

Messy local habits often become messy professional habits.

The goal is not perfection—it is **intentional structure**.

---

5. Blaming GSAP for environment problems

Sometimes GSAP is not the issue at all.

The actual problem may be:

1. scripts not loading,
2. DOM not ready,
3. selectors wrong,
4. CSS hiding the effect,
5. cache serving stale files.

A good local setup helps you identify the real cause.

Practical recommendation: your first environment plan

Here is your concrete plan for this course.

1. **Install LocalWP**
    1. Create one main course site.
    2. Give it a clear name.
2. **Install WordPress cleanly**
    1. Avoid importing a giant existing project as your first learning space.
    2. Start with clarity.
3. **Install BricksBuilder**
    1. Confirm it runs properly in the local environment.
    2. Open the builder and make sure page editing works smoothly.
4. **Open the site files in PhpStorm**
    1. We’ll discuss project structure in the next lesson.
    2. For now, the key is making the editor part of your workflow.
5. **Use the browser and DevTools from day one**
    1. Learn to keep Console and Elements tabs nearby.
    2. Animation work without DevTools is guesswork.
6. **Do not over-engineer yet**
    1. Your main job right now is to make the environment stable and understandable.
    2. Not fancy.

Mini mental model: your workflow loop

This is the core loop you will repeat many times throughout the course:

1. change code,
2. refresh page,
3. inspect behavior,
4. verify DOM/CSS/JS state,
5. adjust,
6. test again.

This loop should feel **fast**.

If your environment makes this loop slow or confusing, learning becomes harder than necessary.

Exercise: environment decision checklist

Answer these for yourself:

1. Do I want the fastest path to building GSAP projects in WP right now?
    1. If yes, choose **LocalWP**.
2. Am I currently trying to learn animation and WordPress implementation more than infrastructure engineering?
    1. If yes, choose **LocalWP**.
3. Do I need reproducible team environments and environment config in version control today?
    1. If yes, consider **Docker**.
    2. If not, Docker can wait.
4. Do I clearly understand containers, service mapping, and mounted volumes?
    1. If not, Docker may slow down early progress.

For most learners here, the result will be:
Start with LocalWP.

Implementation target for this lesson

By the end of this page, your intended setup should be:

1. **Primary environment choice**
    1. LocalWP
2. **Primary site purpose**
    1. Dedicated GSAP + Bricks learning sandbox
3. **Editor**
    1. PhpStorm
4. **Builder**
    1. BricksBuilder
5. **Testing approach**
    1. Local first, browser-driven, DevTools-assisted
6. **Advanced environment awareness**
    1. Docker understood conceptually, postponed unless needed

---

Key takeaways

1. **Your local environment is part of your skillset**, not just a technical prerequisite.
2. **LocalWP is the best starting choice** for this course because it removes setup friction.
3. **Docker is valuable**, but usually better once you are comfortable with the core WP + GSAP workflow.
4. **Animation debugging depends heavily on environment quality** because issues often involve timing, CSS, DOM state, and script loading.
5. **A clean, minimal, intentional setup will help you learn faster and more deeply**.

Action steps before moving to 1.2.2

1. Decide on your primary local setup tool:
    1. preferably **LocalWP** for now.
2. Create or plan a dedicated local WordPress site for this course.
3. Commit to this rule:

    1. *I will learn and test animations locally before touching live sites.*
4. Make sure you have:
    1. LocalWP,
    2. PhpStorm,
    3. a modern browser with DevTools,
    4. access to BricksBuilder.