CREATIVE & DESIGN TOOLS

Figma

Frames, Auto Layout & Components

Design Systems & Prototyping

Figma & Penpot, Side by Side

Capstone: From Wireframe to Developer Handoff

Philip Osztromok

Generated with Claude

Table of Contents

Figma — 10 Chapters

Vector Graphics covered Illustrator and Inkscape — two file-based applications built around drawing and editing paths. Figma shares that same underlying vector foundation, but it's built on a genuinely different architecture from the ground up: no installed application, no file to save, and multiple people editing the exact same design at the exact same moment.

What Figma Is: A Browser-Based Design Tool

Figma runs primarily in a web browser (a desktop app exists too, but it's essentially a dedicated browser window rather than a fundamentally different application). It's purpose-built for UI/UX and product design — screens, components, interactive prototypes — rather than the print/illustration focus Vector Graphics' own tools lean toward.

Cloud-Native by Default: No File to Save

A Figma design lives on Figma's own servers from the moment it's created. There's no "Save" command in the traditional sense, because there's no local file being written to — every change is continuously synced to the cloud as it happens. This is the same underlying model Google Workspace's own Chapter 1 described for Docs, Sheets, and Slides, applied here to design work instead of documents.

Real-Time Multiplayer: Multiple People, One Design

Multiple people can open the exact same Figma file at the exact same time, each seeing the others' cursors move and edits appear live, the same way Google Workspace's own real-time collaboration works in a shared document. A designer, a teammate reviewing a component, and a stakeholder leaving feedback can all be looking at — and editing — the identical canvas simultaneously, with no separate copies to merge afterward.

Contrasted with Vector Graphics' Own File-Based Model

Illustrator and Inkscape both work with a local file that's opened, edited, and explicitly saved back to disk — one person's own copy at a time, even when a file is later shared with someone else. Figma has no equivalent "the file" at all in that sense; the design simply exists in Figma's own cloud workspace, openable by anyone with permission at any moment, with everyone looking at the same live canvas rather than their own separate copy.

What This Course Actually Teaches

This course covers Figma's own core UI/UX design workflow — frames, auto layout, components, prototyping, collaboration, and developer handoff — alongside Penpot, a free and open-source alternative. Chapter 2 introduces that pairing directly, including an honest note about how lopsided the comparison actually is compared to Illustrator and Inkscape.

ASPECT	ILLUSTRATOR/INKSCAPE (VECTOR GRAPHICS)	FIGMA
Where it runs	Installed desktop application	Web browser (desktop app is a dedicated browser window)
Where the design lives	A local file, opened and saved	Figma's own cloud servers, always synced
Editing together	One person's own copy at a time	Real-time multiplayer, same live canvas
Primary focus	Illustration, print, general vector work	UI/UX and product design

COMING UP IN CHAPTER 2

Figma's real-world dominance in UI/UX design is genuinely more lopsided than Illustrator's own dominance over Inkscape — Chapter 2 says so directly rather than presenting Figma and its open-source alternative, Penpot, as two evenly-matched options.

"NO FILE TO SAVE" DOESN'T MEAN "NO VERSION HISTORY"

It's tempting to assume that since there's no explicit Save command, there's no safety net if something goes wrong or an edit needs undoing after the fact. Figma keeps its own continuous version history automatically in the background — the same underlying safety net Google Workspace's own Version History chapter described for documents, just applied here to design files. Nothing about the lack of a manual Save step means changes are unprotected or unrecoverable.

Hands-On Exercises

EXERCISE 1

A designer used to Illustrator opens Figma for the first time and looks for a "Save" button, growing anxious when they can't find one. Using this chapter's own material, explain why there's no Save command and why this isn't actually a problem.

View solution

EXERCISE 2

A team wants a designer, a developer, and a stakeholder to all review and comment on the exact same design at the same time, without emailing separate file copies back and forth. Using this chapter's own material, explain how Figma's own model solves this directly.

View solution

EXERCISE 3

Explain, using this chapter's own material, why Figma's cloud-native, real-time-multiplayer model is described as a genuinely different architecture from Illustrator and Inkscape's own file-based model, rather than just a different-looking version of the same thing.

View solution

CHAPTER 1 QUICK REFERENCE

- Figma is **browser-based**, purpose-built for **UI/UX and product design**
- Designs live on Figma's own **cloud servers**, continuously synced — there's no local file to manually save
- **Real-time multiplayer** lets multiple people edit the same live canvas simultaneously
- This is a genuinely **different architecture** from Illustrator/Inkscape's own file-based model, not just a different interface
- **Version history** runs automatically in the background — no Save button doesn't mean no safety net

Chapter 1 established Figma's own cloud-native, real-time-multiplayer architecture. This chapter introduces the free, open-source tool this course pairs it with — Penpot — while being upfront about something Vector Graphics never had to say about Illustrator and Inkscape: this comparison is genuinely lopsided.

Figma: The Industry Standard for UI/UX Design

Figma is the dominant tool for UI/UX and product design work across the industry — the default expectation at most design-led companies, startups, and agencies doing digital product work. Its own real-time multiplayer model, established in Chapter 1, became a genuine competitive advantage that reshaped how design teams actually work together, not just a nice extra feature.

Penpot: Free, Open-Source, and Genuinely Figma-Inspired

Penpot is free, open-source, and browser-based, built with a visual language and toolset deliberately similar to Figma's own — frames, components, and a comparable design-to-prototype workflow, running in a browser the same way Figma does. Unlike Inkscape, which developed largely independently of Illustrator's own interface conventions, Penpot was built specifically as an open alternative to Figma, and it shows in how directly its own workflow maps onto Figma's.

Why This Split Is More Lopsided Than Illustrator/Inkscape

Illustrator holds a strong lead over Inkscape, but Inkscape has existed for decades, with a large, mature user base and deep production use in its own right. Penpot is comparatively young and has nowhere near Figma's own market presence, plugin ecosystem, or enterprise adoption. This course names that difference plainly rather than presenting the two as evenly matched — the "one tool is proprietary, one is free" framing established in Raster Editing Fundamentals and Vector Graphics doesn't map onto an equally balanced reality here.

Where Penpot Genuinely Holds Its Own

Despite that gap, Penpot isn't a toy. It's genuinely capable for real UI/UX design work — frames, components, and prototyping all function well — and it offers something Figma doesn't: self-hosting. A team with strict data-residency or privacy requirements can run Penpot entirely on its own infrastructure, a real, practical advantage no amount of Figma's own market dominance changes.

What This Course Actually Teaches

From here forward, this course teaches Figma's own core workflow as the primary focus — frames, auto layout, components, prototyping, collaboration, and developer handoff — with Penpot referenced throughout wherever its own equivalent workflow is worth showing, rather than given fully equal weight the way Photoshop/GIMP and Illustrator/Inkscape were.

ASPECT	FIGMA	PENPOT
Cost	Free tier, paid tiers for teams	Free, open-source
Self-hosting	Not available	Available — full control over infrastructure
Market presence	Dominant industry standard	Small but genuinely growing
Plugin/ecosystem maturity	Large, mature	Smaller, still developing

FRAMES ARE UP NEXT

Chapter 3 introduces Figma's own Frames — a concept both Figma and Penpot share, since Penpot's own workflow was built to closely mirror Figma's, unlike Inkscape's own more independently-evolved interface.

"OPEN-SOURCE ALTERNATIVE" DOESN'T AUTOMATICALLY MEAN "EQUALLY VIABLE CHOICE"

It's tempting to assume every free, open-source alternative to a dominant commercial tool is roughly as viable a choice, the way Inkscape genuinely is next to Illustrator. Penpot is a real, capable tool worth knowing, but pretending it's currently an equally strong industry choice to Figma would be dishonest — the plugin ecosystem, market adoption, and sheer scale of Figma's own usage are genuinely not close. Naming this plainly here doesn't diminish Penpot's own real value, especially its self-hosting capability; it just means the comparison in this course is honestly framed rather than artificially balanced.

Hands-On Exercises

EXERCISE 1

A student who took Vector Graphics assumes the Figma/Penpot relationship must be roughly as balanced as Illustrator/Inkscape's. Using this chapter's own material, explain why this assumption doesn't hold up.

View solution

EXERCISE 2

A company has strict data-residency requirements that prevent them from storing design files on Figma's own servers. Using this chapter's own material, explain what real advantage Penpot offers here that Figma cannot.

View solution

EXERCISE 3

A colleague dismisses Penpot entirely, calling it "not worth learning" since Figma dominates the industry. Using this chapter's own material, explain why this dismissal goes too far.

View solution

CHAPTER 2 QUICK REFERENCE

- **Figma** is the dominant industry standard for UI/UX and product design
- **Penpot** is free, open-source, and deliberately built with a Figma-similar workflow
- This split is **more lopsided** than Illustrator/Inkscape — named plainly, not smoothed over
- Penpot's own genuine advantage is **self-hosting**, something Figma doesn't offer at all
- This course teaches **Figma as the primary focus**, with Penpot referenced throughout rather than given fully equal weight

Chapter 2 introduced Figma and Penpot side by side. This chapter starts working with each tool's own actual building blocks, beginning with Frames — a concept that looks at first like Vector Graphics' own artboards, but does genuinely more.

What a Frame Is: More Than Just a Canvas Boundary

A Frame is a rectangular container that holds other design content — shapes, text, images, components — clipped to its own boundary by default. At first glance it looks exactly like Vector Graphics' own artboards from Chapter 9 of that course: a defined area representing one screen or composition. That similarity is real, but only half the picture.

Frames as Artboards: The Familiar Half

Just like an artboard, a Frame can represent one distinct screen or composition — a mobile login screen, a desktop dashboard view, a component variant — with several Frames laid out side by side on the same canvas, each independently exportable, exactly matching the multi-composition convenience Vector Graphics' own Chapter 9 described.

Frames as Responsive Containers: The Genuinely New Half

Unlike a plain artboard, a Frame can also define how its own contents behave when the Frame itself is resized — constraints controlling whether a child element stays pinned to an edge, scales proportionally, or stretches to fill available space. This responsive-container behavior has no real equivalent in Illustrator or Inkscape's own artboards, which simply mark a fixed area with no built-in resizing logic for what's inside them.

Nesting Frames: Building Hierarchical Layouts

Frames can be nested inside other Frames, building up a layout hierarchy that mirrors how a real interface is actually structured — a card component Frame nested inside a list Frame, nested inside a full screen Frame. This nesting is a genuinely common, everyday pattern in UI design work, reflecting how real interfaces are built from smaller reusable pieces rather than one flat, unstructured canvas.

Frames in Penpot: The Same Dual-Purpose Concept

Penpot's own Frames (called Boards in its own interface) work the same way — a rectangular container doubling as both a distinct composition and a responsive parent for its own contents, following the design decisions Penpot deliberately borrowed from Figma's own model rather than the more independently-evolved approach Inkscape took relative to Illustrator.

ASPECT	VECTOR GRAPHICS' ARTBOARDS	FIGMA FRAMES / PENPOT BOARDS
Represents one composition	Yes	Yes
Controls how children resize	No	Yes — constraints, and Auto Layout (Ch.4)
Clips content to its own boundary	No, by default	Yes, by default
Can be nested inside another	No	Yes

FRAMES SET THE STAGE FOR CHAPTER 4

A Frame's own constraint settings control basic resizing behavior, but Chapter 4's Auto Layout goes considerably further — automatically arranging and spacing a Frame's own children the way a real responsive component actually needs to behave.

CONTENT CAN GET CLIPPED WITHOUT WARNING

It's tempting to treat a Frame purely as a static artboard boundary, the way Vector Graphics' own artboards work. Because a Frame clips its own contents to its boundary by default, content that extends past a Frame's edge — a shadow, an overlapping decorative element, a child positioned slightly outside — simply disappears from view rather than showing as an overflow the way it might on a plain artboard. Checking a Frame's own clipping/overflow behavior is worth doing whenever something visually "vanishes" unexpectedly near an edge.

Hands-On Exercises

EXERCISE 1

A designer coming from Illustrator resizes a Frame and expects its contents to behave exactly like resizing an artboard in Vector Graphics — nothing about the content itself changing. Using this chapter's own material, explain what's actually different.

View solution

EXERCISE 2

A subtle drop shadow on a button disappears the moment the button is placed near the edge of its parent Frame. Using this chapter's own warning box, explain the likely cause.

View solution

EXERCISE 3

Explain, using this chapter's own material, why nesting a card Frame inside a list Frame inside a screen Frame reflects how a real interface is actually structured, rather than being an arbitrary organizational choice.

View solution

CHAPTER 3 QUICK REFERENCE

- A Frame is a **dual-purpose** concept — a distinct composition, like an artboard, and a responsive container at once
- Frames **clip content to their own boundary** by default, unlike Vector Graphics' own artboards
- Frames can be **nested**, building layout hierarchies that mirror a real interface's own structure
- Penpot's own **Boards** work the same dual-purpose way, since Penpot was built to closely mirror Figma's model
- Frame **constraints** handle basic resizing; Chapter 4's Auto Layout handles genuinely responsive arrangement

Chapter 3 closed with a tip pointing here directly: a Frame's own constraints handle basic resizing, but Auto Layout goes considerably further — automatically arranging and spacing a Frame's own children the way a real responsive component actually needs to behave.

What Auto Layout Actually Does

Applying Auto Layout to a Frame turns it into a live, self-arranging container — its children are automatically stacked, spaced, and resized according to a set of rules, recalculated continuously as content changes, rather than manually repositioned by hand each time something inside it changes size or gets added or removed.

Direction: Horizontal vs. Vertical Stacking

An Auto Layout Frame stacks its children in one direction — horizontally (side by side) or vertically (stacked top to bottom) — with the option to wrap onto multiple rows if content doesn't fit on one line. Switching direction rearranges every child automatically, without needing to manually reposition each one.

Padding and Gap (Spacing Between Items)

Padding sets the space between the Frame's own edge and its contents; gap sets the consistent space between each child element in the stack. Both update live — increasing the gap value pushes every child further apart automatically, rather than requiring each one to be repositioned by hand.

Resizing Behavior: Hug, Fill, Fixed

Each element inside an Auto Layout Frame (and the Frame itself) can be set to Hug contents (shrinking or growing to exactly fit whatever's inside it), Fill container (stretching to take up all available space in its own parent), or Fixed size (staying at an exact set size regardless of its content or parent). A button's own background, for instance, is commonly set to Hug so it automatically resizes if its label text changes length.

Why This Has No Real Equivalent in Illustrator/Inkscape

Illustrator and Inkscape have no equivalent concept at all — resizing a group or changing text in either tool never automatically repositions or resizes sibling elements around it. This absence isn't a missing feature so much as a reflection of what those tools are actually for: static illustration and print work, where a shape's neighbors don't need to dynamically react to it changing. Auto Layout exists specifically because UI design constantly deals with dynamically changing content — different text lengths, varying numbers of list items, different screen sizes — that a static illustration tool was never built to handle automatically.

Auto Layout in Penpot: Flex Layout

Penpot's own equivalent, called Flex Layout, covers the same ground — direction, padding, gap, and per-element sizing behavior — under names deliberately chosen to echo CSS Flexbox terminology (a connection worth noticing directly if the site's own CSS Frameworks or CSS Overview material is already familiar, since the underlying concepts — main axis direction, gap, grow/shrink behavior — map closely onto real CSS flexbox properties).

RESIZING BEHAVIOR	WHAT IT DOES
Hug contents	Shrinks or grows to exactly fit whatever's inside
Fill container	Stretches to take up all available space in its own parent
Fixed size	Stays at an exact set size, regardless of content or parent

THIS MAKES CHAPTER 5'S COMPONENTS ACTUALLY WORK

A component built without Auto Layout breaks visually the moment its label text gets longer or shorter across different instances. Chapter 5's own reusable components depend directly on Auto Layout's own Hug/Fill behavior to resize correctly wherever they're reused.

AUTO LAYOUT ISN'T JUST "LAYERS WITH EXTRA PADDING SETTINGS"

It's tempting to think of Auto Layout as a minor convenience on top of ordinary layer stacking, similar to setting consistent spacing manually once. Auto Layout is a live, continuously recalculated arrangement engine, much closer in spirit to a CSS flexbox layout than to a one-time manual arrangement — every time content changes (a longer label, an added list item, a resized parent), the entire arrangement recalculates automatically. Treating it as "just padding" misses that it's actually solving a fundamentally different problem: staying correct as content keeps changing, not just looking right once.

Hands-On Exercises

EXERCISE 1

A button's label text gets translated into a longer language, and the button's own background automatically resizes to fit without the designer touching anything. Using this chapter's own material, explain what setting makes this possible.

View solution

EXERCISE 2

A designer coming from Illustrator asks why resizing a group in that tool never automatically repositions the shapes next to it, while a Figma Auto Layout Frame does exactly that. Using this chapter's own material, explain the difference.

View solution

EXERCISE 3

A colleague describes Auto Layout as "basically just consistent padding you set once." Using this chapter's own warning box, explain what this description is missing.

View solution

CHAPTER 4 QUICK REFERENCE

- Auto Layout turns a Frame into a **live, self-arranging** container, recalculated continuously as content changes
- **Direction, padding, and gap** control stacking orientation and spacing
- **Hug, Fill, and Fixed** control how each element resizes relative to its own content or parent
- Illustrator and Inkscape have **no real equivalent** — a reflection of static illustration vs. dynamic UI content
- Penpot's own **Flex Layout** covers the same ground, with terminology echoing CSS Flexbox

Chapter 4's own tip box pointed here directly: a component depends on Auto Layout to resize correctly wherever it's reused. This chapter covers what a component actually is — a reusable master element, and the live-linked copies of it that make up most of a real design file.

What a Component Actually Is: A Reusable Master

A component is a designated "main" version of a reusable element — a button, a card, a navigation bar — defined once and reused throughout a design. This is the same underlying idea as a Slide Master in Word & PowerPoint or LibreOffice's own Master Slide: one master definition that every reused copy stays connected to, rather than independent, disconnected duplicates.

Instances: Live-Linked Copies

An instance is a copy of a component placed elsewhere in the design, staying live-linked back to its own main component rather than becoming an independent, disconnected duplicate the moment it's placed. A single button component might have dozens of instances scattered across many different screens in the same file, all pointing back to that one shared definition.

Propagating Changes: Editing the Main Component Updates Every Instance

Editing the main component — changing its color, its padding, its icon — automatically updates every instance of it throughout the entire file, instantly. This is the real payoff of the master/instance relationship: a single edit to one button's color, for instance, ripples out to every single button in the design without needing to find and update each one individually.

Overrides: Instance-Specific Differences That Survive an Update

An individual instance can still have its own specific differences — different label text, a swapped icon, a different fill color for one particular case — called overrides. These per-instance differences survive future updates to the main component, so changing the main button's overall padding doesn't wipe out an instance's own custom label text, letting shared structure and per-instance customization coexist.

Variants: Grouping Related Components Into One Switchable Set

Variants group several related components — a button's default, hover, and disabled states, for instance — into one single switchable component set, with a property (like "State") that switches between them. Rather than managing three entirely separate components for three button states, Variants keep them organized as one logical unit with a clean property to switch between.

Components in Penpot: The Same Master/Instance Model

Penpot uses the same main component/instance relationship, including change propagation and per-instance overrides, under closely matching terminology — another place where Penpot's own deliberate similarity to Figma (established back in Chapter 2) makes the underlying concept transfer directly between the two tools.

CONCEPT	WHAT IT DOES
Main component	The single master definition every instance stays linked to
Instance	A live-linked copy of the main component, placed elsewhere
Override	An instance-specific difference that survives future main-component updates
Variant	A group of related components (e.g. button states) switchable via one property

COMPONENTS ARE THE BUILDING BLOCKS OF CHAPTER 6

A design system, covered in Chapter 6, is essentially an organized library of exactly these kinds of components (plus shared colors and text styles), reused consistently across an entire product rather than confined to a single file.

EDITING AN INSTANCE DIRECTLY DOESN'T UPDATE THE OTHERS — AND DETACHING IS PERMANENT

It's tempting to edit a specific instance directly when a change is meant to apply everywhere, assuming it'll somehow propagate back to the main component. Only editing the main component itself propagates a change to every instance — editing an instance directly just creates a per-instance override on that one copy alone, per this chapter's own material. It's equally tempting to "detach" an instance to fully customize it, but detaching permanently severs its live link back to the main component — that instance no longer receives any future updates at all, a one-way action worth confirming is actually needed before using it.

Hands-On Exercises

EXERCISE 1

A designer wants every button across a 40-screen design file to change from blue to green, all at once. Using this chapter's own material, explain the correct way to do this.

View solution

EXERCISE 2

A colleague edits one specific button instance directly, expecting the change to appear on every other button instance too, and is confused when it doesn't. Using this chapter's own warning box, explain what happened.

View solution

EXERCISE 3

Explain, using this chapter's own material, how a button's default, hover, and disabled states are organized using Variants, and why this is preferable to three entirely separate components.

View solution

CHAPTER 5 QUICK REFERENCE

- A **component** is a reusable master element, the same underlying idea as a Slide Master or Master Slide
- An **instance** stays live-linked to its main component — editing the main component updates every instance
- **Overrides** let an instance keep its own specific differences even after the main component updates
- **Variants** group related components (like button states) into one switchable set
- Editing an instance directly only creates an **override** on that one copy; **detaching** permanently severs its live link

Chapter 5's own tip box pointed here directly: a design system is essentially an organized library of components, plus shared colors and text styles, reused consistently across an entire product rather than confined to a single file. This chapter covers exactly that — styles and libraries, the pieces that turn a single file's own components into something a whole team can share.

What a Design System Actually Is

A design system is a shared collection of styles and components — colors, typography, buttons, cards, navigation elements — used consistently across every screen and every file a team works on, so a product looks and behaves cohesively no matter which designer is working on which part of it.

Color Styles: Naming and Reusing Colors Consistently

A color style saves a specific color under a memorable name — "Primary Brand Blue," "Error Red," "Neutral Gray 400" — so it can be applied consistently by name rather than by re-entering a hex value each time. Updating a color style's own underlying value updates every element using that style throughout the file, the same propagation behavior Chapter 5's own components already established for shape and layout, applied here to color instead.

Text Styles: Consistent Typography Across a File

A text style bundles font, size, weight, and line height together under one named style — "Heading 1," "Body Text," "Caption" — applied consistently wherever that kind of text appears. This keeps typography visually consistent across many screens without needing to manually match font settings by eye each time a new text element is created.

Shared Libraries: Publishing Components Across Multiple Files

A library publishes a file's own components and styles so other files — other screens of the same product, or entirely separate projects — can use them too, staying live-linked back to that shared source the same way an instance stays linked to its own main component. A team's design system typically lives in one dedicated library file, published for every actual product file to draw from.

Keeping a Library Updated: Publishing New Versions

Updating a component or style inside the library file and publishing a new version pushes that update out to every file using the library, the same propagation Chapter 5 already described for a single file's own components — just extended across an entire team's worth of files rather than staying confined to one.

Design Systems in Penpot: Shared Libraries

Penpot supports the same shared-library model — publishing components and styles from one file for other files to draw from — under closely matching terminology, another instance of Penpot's own deliberate similarity to Figma making the underlying concept transfer directly.

SCOPE	WHAT IT COVERS
Local styles/components	Reusable within one single file only
Published library	Reusable across every file a team has access to
Color/text style update	Propagates to every use of that style, in every file using the library
Component update	Propagates to every instance, in every file using the library

THESE SAME STYLES FEED DEVELOPER HANDOFF LATER

Chapter 9's own Dev Mode reads a design's actual color and text style values directly, translating them into the design tokens and code snippets a developer actually needs — which only works cleanly if the design was built from consistent, named styles in the first place, not one-off, unnamed values scattered throughout.

HAVING A DESIGN SYSTEM DOESN'T PREVENT STYLE DRIFT ON ITS OWN

It's tempting to assume that once color styles, text styles, and a shared library exist, consistency is automatically guaranteed from that point forward. A designer working quickly can still pick an unnamed, one-off color or manually set font properties instead of reaching for the established style — a small shortcut that, repeated across a team over time, quietly reintroduces the same inconsistency a design system was built to prevent. The tools enable consistency; keeping to them, chapter after chapter and file after file, is what actually delivers it.

Hands-On Exercises

EXERCISE 1

A company's brand color needs to shift slightly across every screen in a 20-file product. Using this chapter's own material, explain the correct way to make this change once rather than editing each file individually.

View solution

EXERCISE 2

Over several months, a team notices that despite having a published color and text style library, several screens have started using slightly different, unnamed shades of the brand's own primary color. Using this chapter's own warning box, explain how this likely happened.

View solution

EXERCISE 3

Explain, using this chapter's own material, why a color style update and a component update both "propagate," and how that behavior in a shared library differs in scope from the same propagation described for a single file in Chapter 5.

View solution

CHAPTER 6 QUICK REFERENCE

- A **design system** is a shared collection of colors, text styles, and components used consistently across a whole product
- **Color styles and text styles** bundle values under a memorable name, propagating updates everywhere they're used
- A **published library** extends this reuse and propagation across every file a team has access to, not just one
- Design system values feed directly into **Chapter 9's Dev Mode** handoff — consistency here pays off downstream too
- Having the tools available doesn't prevent **style drift** — consistency still requires actually using them every time

Chapter 6 organized components and styles into a shared design system. This chapter puts that system into motion — connecting Frames with interactions so a design can actually be clicked through, simulating what a real, working app would feel like before a single line of code exists.

What Prototyping Actually Does: Simulating a Working App

Prototyping links separate Frames together into a clickable sequence — tapping a button on one screen's Frame jumps to another Frame representing the next screen. Nothing about the underlying design content changes; prototyping simply defines how a viewer moves between the already-designed Frames, simulating the feel of a real app's own navigation.

Connections: Linking One Frame to Another

A connection links one element (a button, a card, an entire Frame) to a destination Frame, drawn directly between the two in Figma's own prototyping view. A single Frame can have several different connections leading out from different elements within it — one button leading to a settings screen, another leading to a profile screen, both starting from the same source Frame.

Triggers: What Starts an Interaction

A trigger defines what causes a connection to fire — On Click/Tap (the most common), While Hovering, On Drag, or After Delay (automatically triggering after a set time, useful for simulating a splash screen transitioning on its own). Choosing the right trigger for a given interaction is what makes a prototype feel like the real, intended behavior rather than an arbitrary click-anywhere simulation.

Actions: What Happens When a Trigger Fires

An action defines what actually happens once a trigger fires — Navigate To (jump to a destination Frame), Open Overlay (show a Frame layered on top, like a modal or dropdown, without fully replacing the current screen), Back (return to the previously viewed Frame), or Scroll To (jump to a specific position within a longer scrolling Frame).

Transitions: How the Change Happens

A transition defines how the change between Frames visually happens — an instant cut, a smooth dissolve, or a directional slide/push that mimics how a real mobile app screen often slides in from the side. Choosing a transition that matches the actual platform's own real conventions (a slide for a typical mobile screen push, a dissolve for a subtler overlay) makes the simulation feel meaningfully more realistic.

Prototyping in Penpot: The Same Interaction Model

Penpot supports the same connection/trigger/action/transition model for building clickable prototypes, under closely matching terminology — another place where Penpot's own deliberate similarity to Figma means the underlying workflow transfers directly.

TRIGGER	FIRES WHEN
On Click/Tap	The viewer clicks or taps the element
While Hovering	The cursor hovers over the element
On Drag	The viewer drags the element
After Delay	A set amount of time passes automatically

PROTOTYPES GET SHARED AND REVIEWED IN CHAPTER 8

A finished prototype's own shareable link is exactly what gets used in Chapter 8's own real-time collaboration and comments workflow — stakeholders clicking through a live prototype and leaving comments directly on specific screens as they go.

A PROTOTYPE SIMULATES NAVIGATION — IT DOESN'T RUN REAL LOGIC

It's tempting to expect a prototype to behave like an actual working app, complete with real data, calculations, or conditional logic. A Figma or Penpot prototype only simulates visual navigation between pre-designed Frames — there's no real backend, no actual data processing, and no genuine conditional behavior beyond what's been manually set up as fixed connections. This is an honest, deliberate scope: prototyping exists to validate the look, feel, and flow of an interface before development, not to replace actually building the real, functioning product.

Hands-On Exercises

EXERCISE 1

A designer wants a splash screen to automatically transition to the main app screen after two seconds, with no user interaction needed. Using this chapter's own material, explain which trigger accomplishes this.

View solution

EXERCISE 2

A stakeholder clicks through a prototype and asks why entering a search term doesn't actually filter any real results. Using this chapter's own warning box, explain why this is expected rather than a bug.

View solution

EXERCISE 3

Explain, using this chapter's own material, the difference between the Navigate To and Open Overlay actions, and give an example of a real interface situation each one would fit.

View solution

CHAPTER 7 QUICK REFERENCE

- Prototyping links separate **Frames** into a clickable sequence, simulating a real app's navigation
- A **trigger** (On Click, While Hovering, On Drag, After Delay) defines what starts an interaction
- An **action** (Navigate To, Open Overlay, Back, Scroll To) defines what happens once triggered
- A **transition** defines how the visual change happens — instant, dissolve, or directional slide
- A prototype simulates **navigation only** — no real backend, data, or conditional logic runs underneath it

Chapter 7 built a clickable prototype. This chapter covers what happens once that prototype (or any design file) needs to actually be reviewed by other people — the same real-time, multiplayer collaboration Chapter 1 introduced as Figma's own foundational architecture, put to practical use here.

Multiplayer Cursors: Seeing Where Everyone Else Is Working

Every person currently viewing or editing a file shows up as a labeled cursor moving live on the canvas, the same way Chapter 1 described multiple people editing the exact same design simultaneously. A designer can see exactly where a teammate is currently looking or working, without needing to ask or interrupt.

Comments: Leaving Feedback Directly on the Design

A comment can be pinned to a specific location on the canvas — a particular button, a spot on a layout, a whole Frame — attaching written feedback directly to the exact element it's about, rather than describing it separately in a message elsewhere. Anyone with access to the file can add a comment, reply to an existing one, or tag a specific teammate directly.

Resolving and Threading Comments

A comment thread keeps all replies to one original comment grouped together in order, and marking a thread as resolved clears it from the active view once the feedback has been addressed, keeping a busy file's own comment history organized rather than an ever-growing, undifferentiated pile of open items.

Sharing a Prototype Link for Review

A prototype (Chapter 7) can be shared as a standalone link that lets anyone click through it and leave comments directly on specific screens as they go, without needing full edit access to the underlying file — a natural fit for gathering stakeholder feedback on a design's actual flow, not just its individual screens in isolation.

The Same Underlying Model as Google Workspace

Figma's own multiplayer cursors and live comments are the same real-time collaboration model Google Workspace's own Chapter 2 described for shared documents — multiple people working on one live, shared file, seeing each other's presence and feedback appear instantly, rather than exchanging separate copies and reconciling changes afterward.

Collaboration in Penpot

Penpot supports the same multiplayer cursors and pinned comments, under closely matching terminology, continuing this course's own recurring theme that Penpot's own deliberate similarity to Figma extends well beyond just the drawing tools.

FIGMA FEATURE	GOOGLE WORKSPACE EQUIVALENT (FROM `GWORKSPACE1`)
Multiplayer cursors	Real-time collaboration (multiple cursors, one document)
Pinned comments	Comments pinned to specific text/cells
Resolving a thread	Resolving a comment thread
Shareable prototype link	Sharing a document link for review

THIS SAME SHARED LINK MATTERS AGAIN IN CHAPTER 9

A prototype's own shareable link is exactly what a developer opens to enter Chapter 9's own Dev Mode — the same file, viewed through a different lens built specifically for extracting implementation details rather than leaving design feedback.

FIGMA COMMENTS ARE ANNOTATIONS, NOT PROPOSED EDITS

It's tempting to assume Figma's own comments work like Google Workspace's own Suggesting mode from that course's own Chapter 3 — a formal proposed change that can be explicitly accepted or rejected into the actual document. Figma comments are purely annotations: written feedback pinned to a location, with no mechanism to actually apply, accept, or reject a specific design change directly from the comment itself. Acting on a comment's feedback still means a designer manually going and making the actual change in the design — the comment only communicates what should happen, it doesn't perform it.

Hands-On Exercises

EXERCISE 1

A product manager wants to leave feedback on a specific button's color without disrupting the designer's own separate edits happening elsewhere in the same file at the same time. Using this chapter's own material, explain how Figma's own model supports this directly.

View solution

EXERCISE 2

A stakeholder leaves a comment suggesting a button should be blue instead of green, then is confused when the button doesn't change color on its own. Using this chapter's own warning box, explain why this is expected behavior.

View solution

EXERCISE 3

Explain, using this chapter's own material, why Figma's own multiplayer cursors and comments are described as the same underlying model as Google Workspace's own real-time collaboration, rather than a coincidentally similar but unrelated feature.

View solution

CHAPTER 8 QUICK REFERENCE

- **Multiplayer cursors** show everyone currently viewing or editing a file, live, in real time
- **Comments** pin written feedback directly to a specific location on the canvas
- **Resolving** a comment thread clears addressed feedback from the active view
- A shared **prototype link** lets reviewers click through and comment without full edit access
- Comments are **annotations only** — unlike Google Workspace's own Suggesting mode, they don't propose changes that can be directly accepted or rejected

Chapter 8's own tip box pointed here directly: the same shared prototype link used for review is exactly what a developer opens to enter Dev Mode. This chapter covers that developer-facing side of Figma — inspecting exact specs, reading code snippets, and exporting the actual image assets a real build needs.

What Dev Mode Actually Is

Dev Mode is a dedicated view of a Figma file built specifically for developers, surfacing exact measurements, spacing, colors, and code snippets rather than the editing tools a designer would use. It's the same underlying file Chapter 8's collaborators were viewing and commenting on, viewed here through a different lens built for implementation rather than design feedback.

Inspecting a Design: Measurements, Spacing, and Colors

Selecting any element in Dev Mode surfaces its exact pixel dimensions, its distance from neighboring elements, its exact color values, and its font settings — the precise numbers a developer needs to recreate the design accurately, rather than guessing spacing and colors by eye from a static image.

Code Snippets: CSS, iOS, and Android

Dev Mode generates basic code snippets for a selected element — CSS for web, Swift/SwiftUI-oriented values for iOS, and XML/Compose-oriented values for Android — translating the design's own visual properties into a starting-point implementation in whichever platform's language a developer is working in.

Reading Design Tokens Directly From Styles

Chapter 6's own named color and text styles surface directly in Dev Mode as design tokens — a developer inspecting an element sees its actual style name ("Primary Brand Blue," "Heading 1") alongside its raw value, letting a real codebase reference the same named token rather than a disconnected hex code that could quietly drift out of sync with the design over time.

Exporting Assets: Icons and Images at the Right Resolution/Format

Individual assets — icons, illustrations, images — export directly from the design at whatever format and resolution a real build actually needs: SVG for a scalable icon (the same resolution-independence payoff Vector Graphics' own Chapter 9 described), or PNG at multiple fixed pixel densities (1x, 2x, 3x) for platforms that expect raster images sized for different screen densities.

Dev Mode (or Equivalent) in Penpot

Penpot offers its own Inspect functionality covering the same core ground — measurements, spacing, and basic code output — though Figma's own Dev Mode is generally the more mature, actively developed implementation of the two, another honest instance of the lopsided gap Chapter 2 named directly rather than smoothing over.

ASSET TYPE	RECOMMENDED EXPORT FORMAT
Icons (simple, scalable shapes)	SVG — resolution-independent, per Vector Graphics Ch.9
Photos/complex illustrations	PNG/JPG at 1x, 2x, 3x for different screen densities
Colors and text	Design tokens (named styles from Ch.6), not raw hex/font values

THIS CLOSES THE LOOP FOR THE CAPSTONE

Chapter 10's own capstone ends exactly here — a real screen designed, prototyped, and finally handed off through Dev Mode, closing every earlier chapter's own contribution into one complete workflow.

DEV MODE'S CODE SNIPPETS ARE A STARTING POINT, NOT A FINISHED IMPLEMENTATION

It's tempting to treat Dev Mode's generated code snippets as complete, production-ready code that can simply be pasted directly into a real project. These snippets capture a selected element's own visual styling — dimensions, colors, spacing — as a structural starting point, not the whole implementation: real interactivity, actual data, accessibility considerations, and how the element fits into a larger codebase all still need genuine development work on top of what Dev Mode provides. Dev Mode reduces guesswork and back-and-forth, but it doesn't eliminate the need for a developer's own judgment and implementation work.

Hands-On Exercises

EXERCISE 1

A developer needs to know the exact spacing between two elements and the exact hex value of a background color, without asking the designer directly. Using this chapter's own material, explain how Dev Mode provides this.

View solution

EXERCISE 2

A junior developer copies a Dev Mode CSS snippet directly into production and is surprised when the button doesn't actually do anything when clicked. Using this chapter's own warning box, explain why this is expected.

View solution

EXERCISE 3

Explain, using this chapter's own material, why an icon should typically be exported as SVG while a photo should typically be exported as PNG at multiple resolutions, referencing Vector Graphics' own material on resolution independence.

View solution

CHAPTER 9 QUICK REFERENCE

- **Dev Mode** is a developer-facing view of the same file, surfacing exact specs instead of editing tools
- **Inspect** reveals exact measurements, spacing, colors, and font values for any selected element
- Code snippets cover **CSS, iOS, and Android**, translating design properties into a starting-point implementation
- Named styles from Chapter 6 surface as **design tokens** in Dev Mode, not disconnected raw values
- Export **SVG for icons**, **PNG at multiple densities** for photos — and treat code snippets as a starting point, not finished code

Every prior chapter introduced one concept at a time. This capstone designs a single real app screen end to end, applying each of those concepts in sequence, with Figma's own workflow and Penpot's own equivalent workflow shown side by side at every step — closing the course the way it began, treating both tools as genuinely capable implementations of the same underlying discipline, even given the lopsided gap Chapter 2 named honestly.

The Scenario

A mobile app needs a Settings screen — a scrollable list of toggleable options, each row following the same visual pattern, ending in a working prototype reviewed by stakeholders and handed off to a developer ready to build it.

Step 1 — Building the Screen's Frame (Chapter 3)

Create a Frame at a standard mobile device size, giving the whole screen a defined, exportable boundary — the same dual-purpose container Chapter 3 described, doubling here as both this composition's own canvas and the responsive parent everything else in this capstone gets built inside.

Step 2 — Laying Out Content with Auto Layout (Chapter 4)

Apply Auto Layout to the screen Frame, stacking rows vertically with consistent padding and gap, and set the settings list itself to Fill the available width while each row Hugs its own content height — exactly the resizing behavior Chapter 4 described, ensuring the layout holds together correctly regardless of how many settings rows actually end up in the list.

Step 3 — Building a Reusable Settings-Row Component (Chapter 5)

Build one settings row as a main component, with a toggle switch, then create Variants for its On and Off states. Every actual row placed in the list becomes an instance of this one component, so any future styling change to the row only ever needs to happen once, in the main component.

Step 4 — Applying the Design System's Styles (Chapter 6)

Apply the shared library's own color styles (for the toggle's on/off states, background, and dividers) and text styles (for each row's label) rather than one-off values — keeping this screen visually consistent with every other screen already built from the same design system.

Step 5 — Prototyping the Screen's Interactions (Chapter 7)

Connect each toggle row with an On Click trigger that switches its own Variant between On and Off states, and connect a "Notifications" row to Navigate To a dedicated notifications sub-screen — giving stakeholders something genuinely clickable rather than a static image to imagine interacting with.

Step 6 — Sharing for Review and Comments (Chapter 8)

Share the prototype's own link with the product team, who click through it live, leaving pinned comments directly on specific rows — a request to reorder two settings, a note about a label's own wording — each addressed by editing the actual design, then marking the resolved threads as done.

Step 7 — Handing Off to a Developer via Dev Mode (Chapter 9)

Once revisions are complete, a developer opens the same file in Dev Mode, inspecting exact spacing and reading the design tokens behind each color and text style, then exports any icons used in the settings rows as SVG — the final, concrete deliverable this capstone was building toward the whole time.

CAPSTONE STEP	CHAPTER IT DRAWS FROM
Step 1 — Screen Frame	Chapter 3
Step 2 — Auto Layout	Chapter 4
Step 3 — Reusable component & Variants	Chapter 5
Step 4 — Design system styles	Chapter 6
Step 5 — Prototyping interactions	Chapter 7
Step 6 — Review and comments	Chapter 8
Step 7 — Dev Mode handoff	Chapter 9

THE SAME SHAPE AS VECTOR GRAPHICS' OWN CAPSTONE

One real design worked end to end, with both tools' own equivalent workflow shown side by side at each step, is the same capstone shape Vector Graphics and Raster Editing Fundamentals both used — fitting for a comparative, two-tool course, even given the more lopsided real-world gap between Figma and Penpot that Chapter 2 named directly.

EVERY TECHNIQUE FROM THIS COURSE GETS USED HERE, NOT JUST MENTIONED

A single component with Variants (Step 3), driven by design-system styles rather than one-off values (Step 4), and connected with real interactions rather than a static image (Step 5) is what makes Step 6's own review genuinely useful and Step 7's own handoff genuinely accurate — this capstone only works because every earlier chapter's own habit was actually followed, not skipped for the sake of finishing faster.

Hands-On Exercises

EXERCISE 1

In this capstone's own Step 5, a stakeholder asks why the toggle switches are described as changing "Variant," rather than simply saying the toggle's color changed. Using Chapter 5's own material, explain the distinction.

View solution

EXERCISE 2

A stakeholder's comment from Step 6 asks to reorder two settings rows. Explain why this request is straightforward given how Step 3's component was built, referencing the specific chapter that made this possible.

View solution

EXERCISE 3

Explain why this capstone's own shape — one screen, worked through step by step with both tools' equivalent techniques shown side by side — matches Vector Graphics' and Raster Editing Fundamentals' own capstone shape, even though Chapter 2 named a genuinely more lopsided gap between Figma and Penpot than either of those courses' own tool pairings.

View solution

COURSE COMPLETE

Figma — 10 of 10 chapters complete. Advanced Compositing & Retouching and Blender remain outstanding in the wider Creative & Design Tools subject.

CHAPTER 10 QUICK REFERENCE

- One real app screen, worked **end to end**, with both Figma's and Penpot's own workflows shown side by side at each step
- Every step draws directly from a specific earlier chapter — **nothing new is introduced** in this capstone itself
- A component with **Variants**, driven by **design-system styles**, is what makes review and handoff actually work smoothly
- The finished screen goes from **wireframe to a working prototype to a developer-ready Dev Mode handoff** in one continuous workflow