The ArcGIS Compendium — Volume F: Apps and Field Operations

ON THIS PAGE

The ArcGIS Compendium — Volume F: Apps and Field Operations

Turning maps into products — Instant Apps, StoryMaps, Dashboards, Experience Builder, and the mobile field apps.

One of eight volumes. Approximately 23023 words. Chapters cross-reference the whole set by number.

In this volume:

Instant Apps, Complete

You have a web map you are proud of. It has good symbology, working pop-ups, and a story to tell. The problem is that a web map by itself is a workbench, not a product. Handing someone a raw Map Viewer link (Chapter 6) means handing them every tool, every panel, and every opportunity to get lost. What most audiences actually need is a focused window onto the map: here is the map, here is what you can do with it, here is nothing else.

That is the entire job of ArcGIS Instant Apps. An Instant App is a ready-made web application that wraps your web map in a purpose-built interface. You do not write any code. You pick a template — a pre-designed app whose behavior is fixed but whose content, colors, and options are yours to set — point it at your map, flip some switches, and publish. The result is a real web application with its own URL, hosted by ArcGIS Online (or ArcGIS Enterprise, if your organization runs its own portal — see Chapter 35), that you can share, embed, and update at will.

This chapter covers the whole system: what the template gallery contains and how to choose from it, how configuration actually works, the express-versus-full setup split, URL parameters, embedding, branding, accessibility, and — just as important — the honest signals that you have outgrown Instant Apps and should move to one of its bigger siblings.

Where Instant Apps sits in the app-builder family

ArcGIS gives you four main no-code ways to turn a map into something shareable, and they divide up cleanly by intent:

BUILDER	BUILT FOR	COVERED IN
Instant Apps	One focused task around one map (or a small set): view, look up, compare, browse	This chapter

BUILDER	BUILT FOR	COVERED IN
StoryMaps	Narrative — text, media, and maps in a scrolling story you read top to bottom	Chapter 27
Dashboards	Live monitoring — charts, gauges, and indicators that update as data changes	Chapter 28
Experience Builder	Everything else — multi-page apps, custom layouts, widgets wired together your way	Chapter 29

The rule of thumb: if you can describe your app's purpose in one sentence that starts with a verb — "let people find their voting district," "let people compare two years of imagery," "let people browse photos attached to inspection points" — Instant Apps almost certainly has a template for that sentence, and it will get you to a published app faster than anything else in the platform. If your description needs the word "and" more than once, keep reading until the last section of this chapter.

Instant Apps is the successor to what Esri long called "configurable apps" or "web app templates." If you encounter older documentation or an aging app in your organization built on Web AppBuilder (a retired predecessor), Instant Apps and Experience Builder are the modern replacements. Everything you build in Instant Apps today rides on Esri's current JavaScript mapping technology, which matters for one practical reason: templates keep receiving performance and accessibility improvements without you rebuilding anything.

The template gallery: what is actually in there

When you create an Instant App — from a web map's item page, from Map Viewer's `Create app` option, or from the Instant Apps home reachable through the app launcher — you land in a gallery of templates. Esri adds, renames, and occasionally retires templates over time, so treat any exhaustive list with suspicion, including one printed in a book. What stays stable is the set of *jobs* the templates do. Learn the jobs and you can navigate any version of the gallery.

Tip: The gallery has a filter and search box, and each template card offers a preview and a description of what it is for. When in doubt, open two or three candidate templates with the same map and spend five minutes in each. Choosing by hands-on feel beats choosing by name every time.

Job one: just show the map

The largest family of templates exists to present a map cleanly, with varying amounts of furniture around it.

Basic is exactly what it sounds like: your map, a title bar, and a small set of optional tools such as a legend (the key that explains what your symbols mean) and a search box. It is the right choice when the map itself is self-explanatory and anything more would be clutter.

Sidebar adds a collapsible panel beside the map that can hold the legend, layer list, editing tools, filters, or details text. This is the workhorse of the whole gallery. If you are unsure which template to pick and your goal is general-purpose viewing with some interactivity, start with Sidebar; it is the closest thing Instant Apps has to a default answer.

Media Map is a minimal, compact frame designed for embedding — a map that will live inside a web page rather than standing alone. It supports handy tricks for embedded use, like limiting navigation so readers scrolling an article do not accidentally get trapped zooming your map.

Atlas and gallery-style templates present a collection of maps or layers rather than a single one, letting viewers switch among related maps in one interface.

There are also templates oriented around 3D scenes — a scene is the 3D counterpart of a web map — for when your content is a cityscape, terrain model, or anything else best seen with elevation. If your subject genuinely benefits from tilting and orbiting, look for the 3D-focused entries in the gallery; otherwise stay in 2D, which is faster and easier for most audiences.

Job two: let people look something up

Lookup templates answer the single most common request public-facing GIS teams get: "let residents type their address and see what applies to them."

Nearby takes a location the viewer provides — typed address or the device's own position — and returns the features within a search distance: the three closest fire stations, all clinics within a few miles, and so on, with directions-friendly details. A "feature," if the term is new, is one row of your data with its geometry: one station, one clinic, one point on the map.

Zone Lookup answers a subtly different question: not "what is near me" but "which area am I inside?" Type an address, get back your council district, school attendance zone, trash pickup day, flood zone. If your data is polygons (closed shapes representing areas) and the user's question is "which one contains my location," Zone Lookup is purpose-built for it.

The distinction matters enough to be worth a table:

YOU WANT VIEWERS TO...	TEMPLATE FAMILY
Find features near a location, ranked by distance	Nearby
Discover which zone/district/area contains a location	Zone Lookup

Job three: compare and contrast

Slider animates a map through time or through a sequence of numeric values — think of a time slider sweeping across years of data, with the map updating as it goes. It suits data with a date or ordered numeric field you want to play like a filmstrip. (For the analytical side of temporal data, see Chapter 20, Space-Time Analysis.)

Compare puts two maps or scenes side by side — or stacked — with their navigation linked, so viewers see the same place in both: before and after a storm, two candidate plans, imagery from two dates. Some templates also offer a swipe tool, which drags a handle across a single map to peel one layer back from another; check the template descriptions and settings for it if the swipe interaction is specifically what you want. Chapter 15 covers the imagery side of these workflows.

Countdown-style and ranking templates highlight the top or bottom features by some value — the ten largest, the five worst — which is a genuinely useful framing when your audience cares about extremes rather than the whole distribution.

Job four: browse rich content

Portfolio bundles multiple items — maps, scenes, apps, images, documents — into one navigable collection. It is the right pick when your deliverable is really several related maps and you do not want to email five links.

Attachment Viewer is built for data whose features carry attachments — photos, PDFs, or other files stored with each feature, commonly collected in the field with the tools covered in Chapter 30. It pairs the map with a gallery of the attachments, so an inspection program's photos become a browsable evidence trail.

Interactive Legend-style templates turn the legend into a filter: click a category in the legend and the map shows only those features. It is a surprisingly effective way to let non-experts explore a categorical map without teaching them anything about filtering.

Chart Viewer displays charts you have authored on your web map's layers alongside the map itself, for audiences who want the numbers and the geography together but do not need a full Dashboard.

Exhibit lets you build a curated slide-by-slide tour of a map — a different extent (the area of the map in view), visible layers, and notes per slide — like a presentation deck where every slide is a live map.

There are further specialists — templates for reporting observations, viewing imagery, comparing insets of non-contiguous places like a state plus its islands, and more — and the gallery describes each one. The categories above cover the decisions you will face ninety-plus percent of the time.

The decision guide in one pass

Ask, in order:

1. **Is the goal a story to read?** Leave Instant Apps; use StoryMaps (Chapter 27).
2. **Is the goal live monitoring with charts and stats?** Use Dashboards (Chapter 28).

3. **Does the viewer bring a location and want an answer about it?** Nearby (what's close) or Zone Lookup (what contains me).
4. **Is the point a difference between two states?** Compare (side by side) or Slider (change over a sequence).
5. **Is the content a collection?** Portfolio for mixed items, Atlas-style for many maps, Attachment Viewer if photos are the star.
6. **Otherwise:** Sidebar if viewers need tools and panels; Basic or Media Map if they just need the map, with Media Map when it will be embedded.

> **Tip:** Treat template choice as close to permanent. Even where the builder offers a way to switch an existing app to another template, only the settings the two templates share carry over — the rest is reconfiguration. Spend the five extra minutes previewing before you commit, especially before you share the URL widely.

Configuration: how the builder actually works

Every Instant App template uses the same configuration environment, so once you have configured one, you can configure them all. When you create an app you give it a title, tags, and a folder — this creates an *item* in your content, the same kind of catalog entry a web map or layer gets — and then you land in the builder: your app previewed live on one side, a panel of settings on the other. Change a setting, watch the preview update.

Express setup versus full setup

The builder opens in **express mode**: a deliberately short list of the settings Esri judges essential for that template — the map, the headline tools, the theme. For many apps, express mode is genuinely all you need, and you can go from blank to published in minutes.

A toggle in the builder (look for an express/full switch in the header of the settings panel) opens **full setup**, which exposes every option the template has. The jump is significant: from a dozen decisions to potentially well over a hundred, organized into categories that are consistent across templates:

- **About** — title, cover or splash information, details panels, header content.
- **Map** (or **Content**) — which web map or scene the app uses, plus map-area choices.
- **Interactivity** — the switches that make up most of your real decisions: legend on or off, layer list, search, home button, zoom controls, bookmarks, filters, measurement, printing, sharing buttons, pop-up behavior, and the template's signature tools (search distance in Nearby, slider settings in Slider, and so on).
- **Theme & Layout** — colors, fonts, logo, tool positions.
- **Languages** — some templates let you supply translated text for the app's own labels so one app serves a multilingual audience.

Two habits make configuration go well. First, work top to bottom once, quickly, in express mode to get a working draft; then switch to full setup and refine only what the draft proves wrong. Second, keep the preview honest: the builder can preview the app at phone, tablet, and desktop widths, and you should check all three before publishing, because a sidebar that is pleasant at desktop width becomes a full-screen takeover on a phone.

> **Watch out:** The app is a *presentation* of your web map, not a copy of it. Symbology, pop-up content, labels, and layer visibility all come from the map itself (Chapters 6, 7, and 9). If the pop-up looks wrong in your app, fix it in Map Viewer, not in the app builder — the app will pick up the change automatically, because it reads the live map every time it loads. The corollary: *anyone* editing that web map is editing your published app, whether they realize it or not.

Drafts, publishing, and the map behind the curtain

The builder saves your work as a **draft**, separate from the published state of the app. Viewers keep seeing the last published version while you experiment; nothing goes live until you press the publish button. This is a genuinely safe workflow — use it. Make bold changes in draft, preview them, and publish when satisfied.

Publishing also runs a check on your app's plumbing, and this is where beginners hit the classic wall: the app is shared with everyone, but the *map* inside it — or a layer inside that map — is not. An app is a stack of items: app, web map, layers (Chapter 10 covers hosted feature layers and their sharing in depth). Every level of the stack must

be shared at least as broadly as the audience you intend. The publish check will usually warn you about mismatches and offer to fix them; take the offer, or your public app will greet the public with a login prompt or a mysteriously empty map.

> **Watch out:** Sharing warnings are easy to click past. If someone outside your organization reports "it asks me to sign in," the cause is almost always an under-shared layer somewhere in the stack — check the sharing level of the app, the map, and each layer, in that order. Chapter 39 (the Troubleshooting Encyclopedia) has the full diagnostic ladder.

URL parameters: one app, many entrances

Every published Instant App has a URL, and most templates accept **URL parameters** — extra instructions appended after a `?` at the end of the address that change how the app opens, without changing the app itself. This is one of the most underused features in the platform.

The exact parameters vary by template (the Instant Apps documentation lists them per template), but the commonly supported ideas include:

- **Open at a place** — center the map on given coordinates or open at a particular zoom level, so a link from your "North District" page opens the app already looking at the north district.
- **Pre-fill a search** — some lookup templates accept a location or address in the URL, so Nearby or Zone Lookup can open with the answer already on screen. Pair this with a form on your website and you have an address-to-answer pipeline with zero code.
- **Pre-select a feature** — open with a particular feature selected and its pop-up showing.
- **Set locale** — open the app's interface in a particular language where supported.

The pattern in every case: one configured app serves many audiences, because each audience gets a link that opens the app in the state that serves them. Before you build three nearly identical apps for three districts, ask whether one app plus three parameterized links does the job.

Tip: Build the parameterized URL, then test it in a private browser window where you are not signed in. That simultaneously verifies the parameter behavior and the sharing stack, from the point of view of the actual audience.

Embedding: putting the app inside your website

Instant Apps are designed to be embedded — placed inside a page on your own website so visitors interact with the map without leaving your site. The mechanism is an **iframe**, a standard HTML element that displays one web page inside another. You do not need to understand HTML deeply: an iframe pointing at the app's URL is a single line of markup, ArcGIS surfaces ready-made embed code in the sharing options for items shared publicly, and whoever runs your website will recognize either. Anywhere you can paste HTML — most content management systems allow this — you can embed an Instant App.

Practical guidance that saves grief:

Choose an embed-friendly template. Media Map exists for this purpose, and Basic embeds well. Templates with sidebars and many panels can work embedded but need generous space; a Sidebar app crushed into a narrow column is misery.

Size for the container, and test on phones. Set the iframe to fill the width of its container rather than a fixed pixel width, give it a sensible height, and then actually load the page on a phone. Instant Apps themselves are responsive — they rearrange for small screens — but they can only be as big as the box your page gives them.

Tame scroll behavior. An embedded map in the middle of an article can hijack the mouse wheel: the reader scrolls the page, the map zooms instead. Embed-oriented templates offer settings to require an extra action before the map captures scrolling. Turn them on for any app embedded in a scrolling page.

Mind the sharing stack, again. An app embedded in a public website must be shared with everyone, along with its whole stack. An embed that works for you and shows a sign-in box for the world is the sharing problem from the previous section wearing a different hat.

Know the alternative for tighter integration. If you need the map to *react* to the rest of the page — clicking a table row on your site highlights a feature in the map — an iframe cannot do that. That is the border of custom development with the Maps SDK for JavaScript, covered in Chapter 33.

Branding: making it look like yours

Out of the box, an Instant App looks like a clean, neutral Esri product. The Theme & Layout section of full setup is where it becomes recognizably *yours*:

- **Colors.** Templates offer light and dark base themes plus configurable accent colors for headers and buttons. Use your organization's palette, with one caution: the configuration panel is not a substitute for contrast judgment. A brand color that fails against white text fails no matter how official it is (more on this under accessibility).
- **Logo and header.** Add your logo, set the title, and link the logo back to your main website so the app does not become a dead end.
- **Typography.** Font choices are available in many templates. Resist novelty; pick something readable and stop.
- **Shared theme.** Administrators can define an organization-wide **shared theme** — logo and colors set once at the organization level (see Chapter 34) — which Instant Apps can adopt automatically. If you will ever make more than three apps, set the shared theme first and inherit it everywhere; fifty consistently branded apps built by ten different people is the payoff.
- **Cover and details.** Many templates support a splash or cover screen — a first-load panel with your title, an explanation, and a button to enter the app. It is also your natural home for data credits and a "last updated" note, which serious audiences look for (Chapter 5 explains why data provenance deserves the space).

The custom-domain question comes up constantly: the app's URL lives on Esri's domain (or your Enterprise portal's), and making it appear at `maps.yourcity.gov` is a web-hosting and redirect exercise outside ArcGIS itself. Embedding the app in a page on your own domain is the practical route for most organizations.

Accessibility: the quiet strength of Instant Apps

Accessibility means people with disabilities — using screen readers (software that reads the interface aloud), keyboard-only navigation, magnification, or high-contrast settings — can actually use your app. For public agencies this is typically a legal requirement, not a courtesy; many jurisdictions hold government web content to standards derived from the Web Content Accessibility Guidelines (WCAG), the widely adopted international benchmark.

Here is the strategic point: accessibility is one of the strongest arguments *for* Instant Apps over hand-built alternatives. Esri engineers the templates against those standards — keyboard operability, screen-reader labeling, focus handling — and publishes conformance documentation for its products. When you build with a template, you inherit years of that work. When you hand-build an app, you inherit nothing and must earn it all yourself.

Your configuration choices can still help or hurt:

- **Contrast is yours.** The theme colors you pick determine whether text passes contrast standards. Dark text on dark accent bars is the most common self-inflicted failure. Free contrast-checking tools take seconds to use.
- **The map's colors are yours too.** The app can be perfectly accessible while the map inside it is illegible to the roughly one in twelve men with color-vision deficiency. Chapter 4 covers colorblind-safe design; it applies doubly to public apps.
- **Write real text.** Titles, descriptions, splash text, and any alternative text the template lets you attach to images are what a screen-reader user hears. "Map1 final v2" as a title is an accessibility failure of the plainest kind.
- **Do not communicate by color alone.** If the legend distinguishes categories only by hue, add labeling or symbols so the distinction survives being unseen.
- **Prefer fewer, clearer tools.** Every widget you enable is another thing keyboard users must tab through. Express mode's minimalism is quietly an accessibility feature.
- **Test cheaply.** Unplug your mouse and try to use your app with only the Tab, arrow, Enter, and Escape keys. Ten minutes of that teaches more than any checklist.

Tip: If your organization must formally document accessibility, search Esri's site for its accessibility conformance reports rather than asserting compliance yourself — the reports state, per product, what conforms and what has exceptions, and reviewers respect that specificity.

Living with an app: updates, ownership, and hygiene

An Instant App is an item like any other, so the practices from Chapter 34 apply: give it a real title, a summary, a thumbnail, and tags; keep it in a sensible folder; and record in the item description which web map it depends on. Future-you, staring at forty items named "app," will be grateful.

Because the app reads its web map live, routine updates need no app work: new data in the layer appears in the app immediately. Changes to the app's own configuration go through the draft-then-publish cycle. Two lifecycle cautions: deleting or un-sharing the web map (or a layer) breaks every app built on it, and ArcGIS will warn you but not stop you; and when an employee leaves, their items should be transferred, not orphaned — again, Chapter 34.

When Instant Apps runs out of road

Instant Apps trades flexibility for speed, and the trade is explicit. You will know you have hit the wall when you catch yourself wishing for one of these:

A layout the template does not offer. You want the legend *there*, a second panel *here*, tabs across the top. Templates position their furniture for you; if the position is the problem, no amount of configuration fixes it. That is Experience Builder territory (Chapter 29), where layout is yours to design.

More than one page or view. A home page, a map page, an about page, separate views for separate audiences — Instant Apps is structurally one screen. Experience Builder does multi-page natively.

Widgets talking to each other. A chart that filters the map, a list that drives a second map, a survey form beside live results. Cross-component wiring is Experience Builder's

signature feature; in Instant Apps, components interact only in the ways the template anticipated.

Serious at-a-glance analytics. The moment stakeholders ask for gauges, indicators, and live-updating statistics panels, you are describing Dashboards (Chapter 28), not a map app with a chart bolted on.

Narrative flow. If you keep writing longer and longer splash text, the content is telling you it wants to be a StoryMap (Chapter 27).

Behavior no builder offers. Integration with your own systems, custom analysis on click, bespoke interfaces — that is the Maps SDK for JavaScript (Chapter 33) and real development, with all the cost and power that implies.

The graduation path is gentler than it sounds, because everything downstream reuses your work: the same web map, the same layers, the same pop-ups move into Experience Builder or a Dashboard unchanged. Nothing you configured was wasted; you were polishing the map all along, and the map is the asset.

And do not graduate out of reflex. A large share of real-world mapping needs *are* one sentence with one verb, and for those, a template configured in an afternoon — accessible, responsive, branded, and maintained by Esri's engineers while you sleep — is not the compromise option. It is the correct one.

StoryMaps, Complete

A story, in ArcGIS terms, is a scrolling web page that weaves maps, text, images, and video into a single narrative. The reader does not operate it the way they operate a map viewer or a dashboard — they simply scroll, and the page reveals itself in the order you designed. That is the entire trick of ArcGIS StoryMaps, and it is a powerful one: you control the sequence, so you control the argument. A web map hands the reader a pile of evidence and says "explore." A story walks the reader through the evidence one exhibit at a time and says "here is what it means."

This chapter covers both halves of the craft: the mechanics (blocks, sidecars, map actions, tours, swipes, express maps, themes, collections, publishing) and the narrative judgment that decides whether anyone actually reads past your title. If you need a refresher on building the web maps you will place inside a story, that is Chapter 6 (Map Viewer) and Chapter 7 (Styling and Smart Mapping). This chapter assumes those maps already exist — or that you will draw quick ones directly inside the story, which StoryMaps also lets you do.

When a Story Is the Right Tool

StoryMaps sits in a family of app builders, and picking the wrong one wastes weeks. The decision comes down to who controls the experience:

TOOL	READER'S ROLE	BEST FOR
StoryMaps	Scrolls a fixed narrative you authored	Explaining, persuading, reporting, teaching
Dashboards (Chapter 28)	Monitors live indicators at a glance	Operations, metrics, status boards
Instant Apps (Chapter 26)	Explores one map with a focused set of tools	Self-serve lookup and exploration
Experience Builder (Chapter 29)	Whatever you build — fully custom	Multi-page sites, bespoke layouts

If your content has a beginning, a middle, and an end — a flood event, a neighborhood's history, a survey's findings, a proposal you want funded — it is a story. If the reader's question is "what is the current value of X" or "what is near my house," it is not; reach for a dashboard or an Instant App instead.

You create stories from the StoryMaps home (reachable from the app launcher — the grid of dots — in ArcGIS Online, or at storymaps.arcgis.com). One naming trap: storymaps.arcgis.com is the ArcGIS product tied to your organization; storymaps.com is a separate consumer version of StoryMaps with its own accounts and lighter mapping. This chapter is about the former. A story is an item in your organization like any other:

it appears in your content, it has an item page, and it obeys the same sharing rules covered in Chapter 34 (Administration).

The Block Model: How Every Story Is Assembled

Everything in a story is a block — a self-contained unit of content stacked vertically down the page. You add blocks with the plus button that appears when you hover in the body of the story, and the block palette that opens is the complete inventory of what StoryMaps can do. Rearranging a story means dragging blocks up or down; deleting a paragraph means deleting its block. Once you internalize "it's all blocks," the builder stops feeling mysterious.

The palette groups blocks into a few families. Exact names shift between releases, but the families are stable.

Text blocks

Paragraphs, headings, subheadings, bulleted and numbered lists, and pull quotes. Two things matter here beyond typing:

Headings are structure, not decoration. StoryMaps builds the story's optional navigation from your headings — readers can jump between sections, and screen readers use headings to give visually impaired readers a table of contents. Use one heading per real section, in order, and resist styling a paragraph to merely look like a heading.

The narrative text is the story. A common failure is treating text blocks as captions between maps. In a good story the prose carries the argument and the maps serve as evidence; a reader who ignored every map should still be able to follow the thread.

There is also a button block (a styled link, useful for "Read the full report" calls to action) and a separator (a small visual pause between sections — use sparingly; white space usually does the same job more gracefully).

Media blocks

Images, image galleries, video, and audio. You can upload files directly — StoryMaps hosts them with the story — or link to media hosted elsewhere (a YouTube or Vimeo URL for video, for example). Uploaded media travels with the story, which is more

reliable; linked media keeps your storage footprint smaller but breaks if the source disappears.

Every media block accepts a caption and, crucially, alternative text — a short written description used by screen readers and shown if the image fails to load. Fill it in. It takes ten seconds and it is the difference between an accessible story and an exclusionary one.

Media blocks offer size options, typically from small (inline with the text column) up to full-width. Full-width imagery is the signature StoryMaps look, but it is a spice, not a base: a story where every image screams edge-to-edge has no dynamic range left for the moments that deserve it.

> **Tip:** Compress images before uploading. A phone photo can be very large, and a story with dozens of them loads slowly on the exact mobile connections where much of your audience will read it. Resize to roughly screen width and export at moderate quality; nobody will see the difference except in the load time.

Map blocks

A map block places an interactive map into the flow of the story. When you add one, you choose between a web map or scene (a 3D map) you have already built — yours, your organization's, or something shared publicly — and an express map you draw on the spot (more on those shortly). After placing the map you can adjust its extent — the visible area — and toggle layer visibility for this appearance without altering the underlying web map item. The same web map can therefore appear five times in one story, each time zoomed to a different place with different layers on.

You also decide how interactive the map is for readers. The options range from an essentially static map to full pan-and-zoom with a legend. Default to less interactivity than you think you want: inside a scrolling narrative, a fully live map is mostly an opportunity for the reader to scroll-zoom by accident, get lost in the Pacific, and give up.

> **Watch out:** A story does not contain copies of your web maps — it points at them. If you later delete a web map, or edit it for another project, every story using it

changes or breaks. Before publishing something important, consider making a dedicated copy of each web map just for the story, and keep it untouched. Chapter 10 (Hosted Feature Layers) explains the parallel problem one level down, at the layer.

Embeds

An embed block places another web page or web app inside the story: a dashboard, an Instant App, a chart from another service, an audio player, a form. Paste a link and StoryMaps either shows it as an interactive frame or as a clickable card, depending on what the target site allows. Embeds are the escape hatch when StoryMaps lacks a block for what you need — but each one is a small iframe (an embedded browser window), so they cost load time and can look inconsistent with your theme. A story that is mostly embeds is usually a sign you wanted Experience Builder (Chapter 29) all along.

Timelines and other structured blocks

The palette includes a timeline block (dated entries with text and images, rendered as a vertical sequence) and continues to grow release by release. When your content is genuinely chronological — a project history, an event sequence — a timeline block reads better than a list of paragraphs opening with dates.

Express Maps: Quick Maps Drawn Inside the Story

An express map is a lightweight map you sketch directly in the story builder, without leaving StoryMaps and without publishing any layers. When you add a map block and choose to create a new express map, you get a simple drawing environment: drop numbered or plain points, draw lines and areas, add arrows and text annotations, and attach a title, description, and image to any feature so it shows a small pop-up. You can also search for places and add them as points.

Express maps are for annotation, not analysis. The right uses:

- A locator map: "the study area is here."
- A handful of labeled sites: the five parks in your survey, the route of a march, the parcels in question.

- Arrows and callouts that would be tedious to build as real data: "the flow moved this direction."

The wrong use is anything with more than a few dozen features, anything you will need in another map later, or anything whose attributes matter. Express map features live only inside the story — they are not a hosted layer, you cannot analyze them, style them with smart mapping, or reuse them elsewhere. If the data has a future beyond this one story, create it properly (Chapter 11, Creating Data) and bring it in as a web map.

> **Tip:** Express maps inherit their basemap — the background reference map — from the story's theme, which keeps them visually consistent with everything else. That free coherence is a real design gift — one more reason to use express maps for simple annotation instead of over-engineering a full web map.

Sidecar, In Depth

The sidecar is the block that makes StoryMaps feel like more than a blog with maps, and it is worth learning thoroughly. A sidecar is a horizontal band of the story composed of slides. Each slide pairs a full-screen media panel (a map, image, or video) with a narrative panel (a scrolling column of text and small media). As the reader scrolls, the narrative advances slide by slide while the media panel transforms behind it — the map flies to a new extent, layers switch, the image changes.

This is the scrollytelling pattern you have seen in major newsrooms: the text drives, the visuals respond. It works because it keeps the reader's two jobs — reading and looking — synchronized without asking them to click anything.

Sidecar layouts

When you insert a sidecar, you pick a layout. The names vary slightly across releases, but three shapes persist:

- Docked: the narrative panel is a fixed column beside the full-screen media. The workhorse. Best when each slide carries a real paragraph or two of text.
- Floating: a smaller narrative card floats over the media, which fills the whole screen. Best when the visuals are the point and the text is a short caption per slide.

- Slideshow: horizontal navigation, one screen per slide, advanced by arrows rather than scrolling. Best for a compact, presentation-like sequence; it interrupts the vertical scroll, so use it deliberately.

You can change layout after building, and you can vary the narrative panel's position and size. Inside the narrative panel you can use most ordinary blocks — text, images, buttons — just at a smaller scale.

Building the slide sequence

Each slide's media is set independently: this slide shows web map A at extent one, the next shows the same map at extent two with an extra layer on, the next shows a photograph. When consecutive slides use the same web map, StoryMaps animates the camera between extents, which produces the smooth fly-through effect that makes sidecars feel cinematic. When consecutive slides use different media, the transition is a simple swap.

Design the sequence like shots in a film: establish wide, then cut closer, then show the detail. A sidecar that teleports between unrelated extents at the same zoom level disorients; one that zooms progressively into its subject builds understanding with each slide.

Map actions

Map actions are the sidecar's power feature. A map action is a link or button you place in the narrative panel that, when the reader clicks or taps it, changes the map in the media panel — moving it to a position you chose, toggling layers on or off, or both. The text might read: "Damage was worst in the *eastern district*," where "eastern district" is a map action that pans the map there and lights up the damage layer.

This restores a controlled dose of interactivity inside your authored narrative. The reader is not free to wander, but they can choose to look closer where the text invites them to. You configure a map action by selecting text (or adding an action button) in the narrative panel, then setting up the target map state — extent and layer visibility — much as you configured the slide's map itself.

Two disciplines keep map actions effective. First, make them visibly clickable and verbally explicit ("tap to see the eastern district") — readers do not hover-hunt for

hidden links, especially on phones. Second, keep each action's change legible: one pan, one layer toggle. An action that changes five things at once reads as a glitch, not a reveal.

> **Watch out:** Always scroll your finished sidecar on a phone before publishing. On narrow screens the narrative panel and media stack differently, long slides push the map off-screen, and text that fit comfortably in a desktop column can become a wall. A meaningful share of story readers — often most of them — arrive on mobile, and the phone rendering is the one your work will actually be judged by.

Guided Map Tours

A map tour is a purpose-built block for the pattern "a set of places, presented one at a time." Each tour stop is a location plus media and text — a photo of the site, a paragraph about it. StoryMaps handles the map, the numbering, and the navigation.

Tours come in two broad flavors:

- Guided: the reader moves through stops in the order you set, either by scrolling or with next/previous controls. The map flies from stop to stop. Layout options let you emphasize the media (large photo, small map) or the map (large map, media in a panel) depending on whether the places or the pictures carry the story.
- Explorer: all stops appear at once as a browsable list or grid alongside the map, and the reader picks their own order. This is less a narrative than a catalog — good for "twenty murals downtown," weaker for an argument.

You can build a tour by hand — clicking locations on the map and uploading media per stop — or seed it in bulk, for instance from a batch of photos that carry location information in their metadata; check the tour's start-up options for the bulk-import paths your release offers. Hand-building is fine up to a couple dozen stops; beyond that, maintain the underlying places as a proper layer and think hard about whether a tour is still the right form at all.

The craft question for tours is the same as for any list: does order matter? If your stops build on each other — chronological, upstream-to-downstream, gateway-to-summit — a guided tour converts geography into narrative beautifully. If the stops are

interchangeable, an explorer tour or even a plain web map with good pop-ups (Chapter 9) serves the reader better.

Swipe Blocks

A swipe block puts two maps or two images side by side with a draggable divider between them; the reader sweeps the handle to reveal one or the other. It is the single most persuasive block in the palette when — and only when — your point is a comparison of the same place: before and after the fire, 1950 versus today, plan A versus plan B, imagery versus interpretation.

Setup is simple: choose the left content and the right content. For maps, the two sides stay synchronized as the reader pans and zooms, which is what makes the comparison honest. For the comparison to land, the two sides must match in extent and, as far as possible, in styling — a swipe between a dark-themed map and a pastel one reads as a style change, not a data change. Chapter 4 (Cartographic Design) covers making paired maps visually comparable; Chapter 15 (Imagery and Rasters in Practice) covers sourcing the before/after imagery that swipes so often showcase.

Resist swiping two unrelated maps just because the block looks impressive. A swipe implicitly claims "same place, one variable changed." If that claim is false, the block misleads.

Themes, Typography, and Design

A theme is the story-wide design system: fonts, colors, block styling, and the basemap style that express maps inherit. You set it in the story's `Design` panel, which also controls the cover layout (full-image, side-by-side, or minimal), whether the navigation bar of section headings appears, and other global options.

StoryMaps ships with a handful of built-in themes spanning light, dark, and a few editorial personalities. For many stories, picking the built-in theme that matches your subject's mood — and then leaving it alone — is the correct amount of design.

When your organization has a visual identity to honor, the theme builder lets you construct a custom theme: choose typefaces for headings and body from a curated set (organizations can also make their own brand fonts available), set the accent and background colors, and pick the express-map basemap style. Custom themes are saved

as their own items and can be shared across the organization, which is how a team gets every story looking like the same publication rather than fifteen different hobby blogs.

Typography advice compresses to three rules. One: contrast between heading and body fonts should be deliberate — either clearly different or exactly the same, never almost-matching. Two: your accent color must pass contrast against your background, because it becomes link and button color; a pale accent on white produces invisible map actions. Three: never fight the theme with per-block manual styling. Consistency is what makes a story feel authored; the theme exists so you make each design decision once.

> **Tip:** Choose or build the theme before writing, not after. The theme decides how much text fits comfortably per screen, how images sit against the background, and what your maps should look like to belong. Retrofitting a dark theme onto a story full of light-basemap maps means redoing every map.

Story Structure: What Makes Geographic Storytelling Land

The blocks are easy. The reason most stories fail is structural, so before covering publishing, here is the narrative craft — the part no menu can do for you.

Lead with the point, not the project. The classic institutional opening — "In 2024, the Department initiated a study..." — loses readers in one screen. Open with the finding, the tension, or the human stake: what changed, who is affected, why now. Background belongs later, after the reader has a reason to care. Journalists call the buried version of this mistake "burying the lede," and stories bury it constantly.

One story, one question. A strong story answers a single question the reader could state back: "Why is this river dying?" "Where should the new clinic go?" If your outline answers four questions, you have four stories, or one story and a collection (below). Scope discipline is the difference between a story and a scrolling report.

Maps are evidence, prose is argument. Every map you include should be there because the text just made a claim the map proves. Before adding any map, ask: what will the reader see in three seconds, and what should they conclude? If you cannot answer, cut the map or redesign it (Chapter 4). A story is not a portfolio of your cartography; it is a case, and each exhibit must earn its place.

Vary the rhythm. Long stretches of any one block type numb the reader. The reliable macro-shape: a strong cover, a short text hook, a sidecar that does the heavy explanatory lifting, a swipe or tour at the emotional peak, then a plain-text conclusion with a button pointing at what to do next. Full-width media and sidecars are your loud moments; simple text columns are your quiet ones. Loud only works next to quiet.

End with an action. "Thank you for reading" is a shrug. Tell the reader what to do with what they now understand — comment on the proposal, download the data, visit the site, contact the office — and give the button.

Write the outline before touching the builder. The builder is seductive; you can burn an afternoon perfecting a sidecar transition for a section that should not exist. Outline on paper: the one question, the sequence of claims, the evidence (map, photo, chart) for each claim. Then build.

Collections

A collection is a container item that bundles stories — and other content such as apps and links — into a browsable set with its own cover and navigation. Readers see a gallery or tabbed interface and move between items without returning to your home page.

Collections solve the "four questions" problem honestly: a regional atlas with one story per county, an annual series, a curriculum of lessons, a project hub combining a story, a dashboard, and a survey. You create one from the StoryMaps home alongside new stories, add items in the order you want them browsed, and give the collection its own theme and cover. The collection shares like any other item, and readers need access to the individual items inside it as well — the collection is a wrapper, not a copy.

Keep collections shallow. One level of "here are the six chapters" navigation helps; nesting and sprawl recreate the disorientation stories exist to prevent.

Publishing, Sharing, and Analytics

Draft, publish, and the two-copy model

A story lives as a draft while you edit, autosaving as you work. Nothing is visible to readers until you press `Publish`, at which point StoryMaps takes a snapshot and serves

that snapshot at the story's URL. Afterward, the draft and the published copy are separate: you can keep editing the draft for days — the builder will indicate you have unpublished changes — and readers see the old version until you publish again. This is a feature, not a bug: it means you can rework a live story safely. But it also means the most common publishing confusion is "I fixed the typo and it's still there" — you fixed it in the draft and never republished.

Publishing runs a quick check for problems — most importantly, content inside the story that is shared more restrictively than the story itself.

Sharing, and the container problem

A story shares at the standard levels — private, groups, organization, or everyone (public) — set at publish time or from the item page later; the full model is Chapter 34's territory. The trap specific to stories is that a story is a container of pointers: the story item can be public while a web map inside it is private, in which case readers see your prose wrapped around error messages. And the web map is itself a container — its layers have their own sharing. The whole chain, story to maps to layers, must be at least as open as the story. The publish-time check offers to fix mismatches it finds and will update sharing on the content you own; take the offer, then open the published story in a private browser window while signed out. That signed-out test is the only proof that anonymous readers see what you see.

> **Watch out:** Public stories can also expose more than you intended in the other direction — pop-ups with staff phone numbers, editable layers, precise addresses of vulnerable sites. Before sharing publicly, review every layer the story touches as a stranger would. Chapter 10 covers view layers, which let you publish a safe, read-only, trimmed version of a layer while keeping the master private — the right pattern for public stories built on sensitive data.

Stories can be embedded in other websites, and a published story has a shareable URL you can shorten or QR-code for print. If your organization operates ArcGIS Enterprise (Chapter 35), StoryMaps exists there too, with feature availability trailing ArcGIS Online — check what your version includes before promising a block you saw online.

Analytics awareness

Published stories accumulate view counts, visible to you as the author from the story's management interface and on its item page. Treat the number as a pulse, not a verdict: it counts loads, not reading, and tells you nothing about how far people scrolled or whether the map actions were ever tapped. If a story matters enough to measure properly — a public campaign, a funded deliverable — embed it in a page you instrument with a real web-analytics tool, or at minimum watch whether the call-to-action at the end (that button you included, because you ended with an action) actually gets clicked, using whatever the target of the button can measure. Organization administrators can see broader usage reporting across items; that, and the credit implications of hosting media-heavy stories, are covered in Chapter 34.

Maintenance

A story is live infrastructure. The maps it points at, the layers under them, the embeds it frames — any of these can change or die and quietly rot your story. Put a recurring reminder on your calendar to open each published story you care about, signed out, on a phone, twice a year. Fix what broke, republish, and check the view count while you are there. Stories are the most public thing most GIS practitioners ever make; they deserve at least the maintenance you would give a park sign.

A Pre-Publish Checklist

Run this list before every `Publish`:

- The title states the point, not the project name.
- Every image has alternative text; every map has a purpose you can state in one sentence.
- Sidecars and tours were scrolled end-to-end on a phone.
- Swipe blocks compare the same extent with comparable styling.
- Sharing is consistent down the whole chain — story, maps, layers — verified signed out.
- The final block tells the reader what to do next.
- The draft you are publishing is actually the version you think it is.

Ten minutes of checklist saves the particular embarrassment of a broken story attached to your name at the bottom of a thousand strangers' screens — and when it all works, a

story remains the single best answer ArcGIS has to the oldest question in the field: you made a map, so what?

Dashboards, Complete

A dashboard is a single screen that answers an operational question at a glance: How many crews are in the field right now? Which districts are behind on inspections? Where did the calls come from overnight? ArcGIS Dashboards is Esri's app for building these screens. You assemble a page out of tiles — a map here, a big number there, a chart along the bottom — and every tile draws live from your data, so the answer on screen is the answer right now, not the answer when someone last exported a spreadsheet.

That "at a glance" quality is the whole point, and it is what separates a dashboard from the other app builders in this volume. A StoryMap (Chapter 27) walks a reader through a narrative at the reader's pace. An Instant App (Chapter 26) wraps one map in a focused tool. Experience Builder (Chapter 29) gives you free-form layout for anything. A dashboard is for monitoring: the viewer does not read it so much as check it, often many times a day, and often on a wall-mounted screen where nobody is touching a mouse at all. If your audience needs to *understand* something, reach for a StoryMap. If they need to *watch* something, you are in the right chapter.

Everything a dashboard shows comes from web maps and hosted feature layers — the same items you learned to publish in Chapter 10 (Hosted Feature Layers). A dashboard adds no data of its own; it is a lens. That has a happy consequence: fix the data once and every dashboard pointing at it updates. It also has a sober one: a dashboard can never be better than the layer underneath it, so the schema and quality work from Volume C pays off here directly.

You create a dashboard from your portal content page — look for a `Create app > Dashboards` option, or open the ArcGIS Dashboards app directly from the app launcher. What you get is an empty canvas, a button to add elements, and a header you can title.

Everything else in this chapter is about what to put on that canvas and how to make the pieces talk to each other.

The Anatomy of a Dashboard

A dashboard is a grid of **elements** — the individual tiles. You add one with the add-element control (a `+` button in the edit toolbar), choose its type, point it at a data source, and configure it. Drag edges to resize; drag the whole element to rearrange. Elements can be **stacked** on top of one another so they share a slot and the viewer flips between them with a small pager control, and they can be **grouped** so they move and resize together. Stacking is one of the most useful and least discovered features in the product: a chart and its equivalent table stacked in one slot gives every viewer the format they prefer without spending two slots.

Around the grid sit two optional furniture pieces. The **header** runs across the top and holds the title, a logo if you want one, and — importantly — selectors, which we will meet later. The **side panel** (sometimes labeled as a left panel in the settings) is a collapsible drawer that also holds selectors, useful when you have several filters and do not want to crowd the header.

Each element has its own configuration window with a consistent shape: a **data tab** (what layer or expression feeds it, what statistic to compute, what filters to apply before computing), a display or appearance tab (colors, labels, formatting), and a **general tab** (title, description, and the element's own refresh interval). Once you have configured one element type, the others feel familiar.

> **Tip:** Give every element a real name in its general settings, even elements whose title you hide on screen. When you later wire actions between elements, you will be picking targets from a list, and "Serial Chart (3)" is a miserable thing to debug.

The Element Catalog

Ten element types cover nearly everything a dashboard does. Here is the full tour, with the honest guidance on when each earns its space.

Map

The map element embeds a web map you have already built in Map Viewer (Chapter 6). All the styling, pop-ups, and labels you configured there come along. Inside the dashboard you choose which widgets the embedded map exposes — legend, layer visibility toggle, zoom controls, basemap switcher, search — and each one you enable trades glanceability for interactivity. On a wall display, turn nearly all of them off.

The map is more than a picture: it is the most powerful *source* of actions in the whole system. A map can broadcast its extent (the rectangle of the world currently visible) and its selections to every other element, which is how "the numbers update as I pan" dashboards work. We will build that wiring in the actions section.

Not every dashboard needs a map. If the operational question is "are we on target this week," a map may be decoration. Let the question decide.

Indicator

The indicator is the big number — count of open work orders, sum of gallons pumped, average response minutes. You pick a layer, a statistic (count, sum, minimum, maximum, average, or standard deviation), and optionally a filter so the number describes only the features that matter ("status is Open").

Indicators support a **reference value**: a second number — a fixed target, or another statistic such as yesterday's count — that the main value is compared against. The indicator can then display the difference or ratio and change color or icon conditionally, which is what turns a number into a judgment: not "47" but "47, which is 12 above target, and that is bad, so it is red." A number without a reference forces every viewer to remember what good looks like; a number with one does the remembering for them.

Gauge

A gauge shows one value against a scale, in two flavors. A **progress gauge** fills toward a maximum — percent of inspections complete, reservoir level against capacity. A **meter gauge** looks like a speedometer with a needle and colored bands. Gauges are the most misused element in the catalog: they spend a lot of pixels on one number, and unless the minimum and maximum are genuinely meaningful (a real capacity, a real quota), an indicator says the same thing in a quarter of the space. Use a gauge when "how full" or "how close to the line" is truly the question; use an indicator otherwise.

Serial chart

"Serial" just means data shown along an axis in series — bar charts, column charts, line charts, and area charts all live here. You choose what defines the categories: unique values of a field (bars per district), date grouping (a line per week), or one bar per feature. Serial charts can show multiple series, stack them, and — critically for operations — parse date fields so time-based charts bin correctly by day, week, or month. The serial chart is the workhorse for "how is this trending" and "how do the groups compare," the two questions indicators cannot answer.

Pie chart

The pie shows parts of a whole. It is defensible when there are a handful of categories and the point is "this slice dominates." Beyond roughly five slices, or when the viewer needs to compare slice to slice precisely, a bar chart in a serial chart element communicates faster. Chapter 4 (Cartographic Design) covers why human eyes judge lengths better than angles; the advice transfers directly.

List

The list shows features one per row, formatted with a template you control — you type text and drop in field placeholders in curly braces, so a row might read "{address} — {status}, reported {report_date}". Lists shine for queues: the ten most recent calls, the crews currently checked in, the overdue permits. You can sort by any field and cap how many features display. Recent versions also support advanced formatting with Arcade (Chapter 8), which lets a row's text and colors respond to its data.

Lists are interactive by default: clicking a row can trigger actions on other elements, which makes a list the natural control surface for "pick a thing, see its details."

Table

The table element shows rows and columns like a spreadsheet — either raw feature rows or grouped summary rows (one row per category with statistics). Use a table when viewers genuinely need to scan several attributes side by side or when they will argue with the numbers and want to see the granular records. If a viewer only ever reads one column, that column probably wanted to be a list or a chart.

Details

The details element renders the pop-up of a selected feature — the same pop-up you authored in Chapter 9 (Pop-ups, Fields, and Labels), including its charts and attachments. By itself it shows features one at a time with paging arrows; wired to a map or list selection, it becomes the "inspection panel" of the dashboard: click a feature anywhere, read everything about it here.

Rich text

Rich text is a formatted text block: explanations, legends in prose, links to standard operating procedures, a "how to read this dashboard" note, contact information. It is static, and that is its virtue. Every serious dashboard deserves at least one rich text element saying what the thing is for and when the data updates — the questions every new viewer asks first.

Embedded content

Embedded content displays another web page or document inside the dashboard by URL — a camera feed, an external chart, a form. Its quiet superpower is that the URL can contain field placeholders and be driven by a selected feature, so selecting a station in a list can load that station's camera. Two cautions: the embedded site must allow being framed inside another page (many sites deliberately refuse, and there is nothing Dashboards can do about that), and anything embedded is a performance passenger you no longer control.

ELEMENT	BEST FOR	REACH FOR SOMETHING ELSE WHEN...
Map	Where things are; driving other elements by extent	Location is not part of the question
Indicator	One number, ideally judged against a reference	You need trend or breakdown — use a chart
Gauge	Fullness against a real capacity or quota	The min/max would be arbitrary
Serial chart	Trends over time; comparisons across categories	Only one number matters

ELEMENT	BEST FOR	REACH FOR SOMETHING ELSE WHEN…
Pie chart	Dominance among a few parts of a whole	More than a handful of slices
List	Queues and rosters; a click-to-select control	Viewers need many columns — use a table
Table	Side-by-side attributes; grouped summaries	Only one field is ever read
Details	Full pop-up of the selected feature	Nothing on the dashboard selects features
Rich text	Purpose, instructions, caveats, contacts	You are tempted to write paragraphs nobody will read
Embedded content	External feeds, feature-driven pages	The source forbids embedding or loads slowly

Layer-Driven Elements vs Data Expressions

Every element needs a data source, and there are two fundamentally different kinds.

A **layer-driven element** points straight at a layer in a web map or at a hosted feature layer item. You pick the layer, pick the statistic or grouping, add filters, done. The heavy lifting — counting, summing, grouping — happens on the server that hosts the layer, which is fast, and the configuration is all point-and-click. This should be your default, and for most dashboards it is all you ever need.

A **data expression** is an Arcade script that builds a table for the element to display. Arcade (Chapter 8) is the small scripting language used across ArcGIS; in a data expression, your script fetches one or more layers, reshapes them however you like — joins, calculated categories, custom groupings, combining two layers into one summary — and returns a **FeatureSet**, which is Arcade's word for a table of features. The element then charts or lists that returned table exactly as if it were a layer.

Data expressions exist because click-configuration has limits. Some real examples of questions that need one: "show one row per crew with a count of its open *and* closed jobs side by side" (a pivot the grouped table cannot quite do), "categorize incidents by a rule too complex for a field" (bucketing free-text values into clean categories), "chart a number that combines two different layers" (permits issued divided by staff on duty).

The power is real and so is the cost. A data expression runs in the viewer's browser every time the element refreshes. Simple operations like filters can be translated into efficient queries the server answers, but the reshaping work that makes data expressions worth writing — loops, joins, custom buckets — typically means fetching features into the browser and processing them there. Against a small or pre-summarized layer that is fine; against a huge layer it means shipping large amounts of data to every viewer's browser on every refresh, which is slow for them and expensive for your server. The click-configured path, by contrast, asks the server for just the summary numbers.

> **Watch out:** A data expression that works beautifully against your ten-thousand-row test layer can bring a dashboard to its knees against the million-row production layer — and it will do so on every viewer's machine, on every refresh. Before writing a data expression, ask whether a calculated field in the source layer, a view with a definition query (a filter saved into the layer item itself, so consumers only ever see matching rows — Chapter 10), or a scheduled summary table could pre-answer the question so the dashboard's job stays trivial.

A good rule of thumb: data expressions are for *reshaping*, not for *reducing*. If the script's job is to boil a big layer down to a few numbers, do that reduction upstream in the data instead.

Selectors: Letting Viewers Ask Variations of the Question

A dashboard answers one question, but viewers often want the same question sliced their way: *my* district, *this* week, only the high-priority items. Selectors are the controls that do the slicing, and they live in the header or the side panel rather than in the element grid.

There are three core types:

- **Category selector.** A dropdown, button bar, or list of values. The values can come from the unique values of a field (every district name in the data), from features themselves (pick a specific facility), or from a fixed list you type in. This is the everyday filter: choose a district, everything narrows to it.
- **Number selector.** A slider or input box producing a numeric value or range — "show incidents above this severity," "parcels between these acreages."
- **Date selector.** A date picker or a set of preset ranges (today, last 7 days, this month, a custom range). For any operational data with a timestamp, a date selector is close to mandatory; "compared to what period" is half of every operational question.

A selector does nothing by itself. Each one carries a list of **actions** that say what it filters — you tick the elements that should obey it. A selector can even target another selector, which is how cascading filters work: choose a district, and the facility selector narrows to that district's facilities. This is easy to half-finish: it is very common to see a dashboard where the district selector filters the map and the indicators but someone forgot to tick the chart, so the chart silently keeps showing the whole city. Every viewer who notices trusts the dashboard a little less, and the ones who do not notice walk away misinformed.

Watch out: When you add a new element to a mature dashboard, audit every existing selector and make sure the new element is added to their action lists if it should obey them. A dashboard where some tiles filter and some do not is worse than one with no filters at all, because it looks consistent and is not.

Tip: Selectors can be set to apply a default when the dashboard loads — for example, defaulting the date selector to "last 7 days." Defaults are how you keep the opening view fast and relevant on big datasets: the first thing the server is asked for is a week of data, not all of history.

Actions: Wiring Elements Together

Selectors filtering elements is one instance of a general system. Any interactive element can be a **source** of actions, and most elements can be **targets**. The wiring is configured on the source: open its settings, find the actions section, choose what happens and to which targets.

The main action verbs:

- **Filter** — the target narrows to features matching the source's current selection or extent. The backbone of dashboard interactivity.
- **Set extent / zoom / pan** — a map navigates to the selected feature. Click a row in the list of overnight calls; the map flies there.
- **Flash** — the feature blinks on the map, useful for "which dot is this row" without navigating.
- **Show pop-up** — the map opens the selected feature's pop-up.
- **Follow feature** — for a map showing moving things (vehicles, vessels), the map keeps the selected feature centered as its position updates.

Sources include lists and tables (row click), charts (clicking a bar or slice can filter everything else to that category — a wonderfully discoverable interaction that costs you nothing to enable), selectors, and the map itself. The map is special: it can broadcast on **extent change**, meaning every wired element re-filters to only the features currently in view. Pan to the north side of town and the indicators, charts, and lists all become "north side" numbers. This single wiring choice is what makes a dashboard feel like an instrument instead of a poster, and it is worth building at least once just to internalize it.

Two design cautions keep wiring from becoming spaghetti. First, prefer one direction of flow: selectors and the map filter downward into charts and lists; lists select features that the map highlights. When everything filters everything, viewers lose track of why the numbers just changed, and so will you. Second, remember that filters from multiple sources combine — a viewer who picks a district, sets a date range, *and* pans the map is seeing the intersection of all three, and nothing on screen says so explicitly. A rich text element noting "all panels respect the filters above" is cheap insurance.

Mobile Layouts

Dashboards are grid layouts designed for wide screens, and for years they were miserable on phones. Current versions let you author a separate **mobile layout** inside the same dashboard: the desktop layout serves big screens, and when the dashboard detects a phone-sized screen it serves the mobile arrangement instead.

The mobile layout is not a second dashboard — it reuses the same configured elements, so an indicator you fixed once is fixed in both views. What changes is arrangement: the mobile view stacks elements into a vertical, scrollable flow, and you choose which elements appear at all and in what order.

Choose ruthlessly. A phone viewer is standing somewhere checking one thing, not studying a wall of panels. The mobile view of a ten-element dashboard is usually three elements: the headline indicator, the queue list, maybe the map. Put the most-checked number first, because it is the only thing guaranteed to be visible without scrolling. If your field staff are the mobile audience, also consider whether what they actually need is not a dashboard at all but a field app — see Chapter 30 (Field Operations) for that boundary.

Design: What Deserves a KPI

The hardest part of dashboarding is not configuration; it is restraint. Some principles that hold up:

Start from the decision, not the data. A dashboard exists so someone can decide something — dispatch another crew, call a supervisor, do nothing and finish their coffee. Before adding any element, name the decision it informs. Data that informs no decision is decoration, and decoration on a monitoring screen is noise that slows down every future glance. "We have this field, so let's chart it" is how forty-element dashboards happen.

Apply the five-second test. A person who knows the operation should extract the headline state of the world — fine, busy, or on fire — within about five seconds of looking. If they must read, scroll, or click to know whether things are okay, the top of the hierarchy has failed. This is Chapter 4's visual hierarchy applied to numbers instead of maps: biggest and top-left goes to the thing that matters most.

A KPI needs a comparison and a consequence. KPI stands for key performance indicator, and the term earns its "key" only when three things are true: someone acts differently depending on the value, the value is judged against something (a target, a threshold, the same time last week — this is exactly what the indicator's reference value is for), and it moves on a timescale the dashboard's audience can influence. "Total records in the database" fails all three. "Open high-priority work orders older than 24 hours, versus a target of zero" passes all three, and notice how the filter *is* the intelligence — the raw count was never the KPI; the filtered, aged, prioritized count is.

Encode judgment in color, sparingly. Conditional formatting that turns an indicator red past a threshold does the viewer's evaluation for them — that is good. Ten elements in ten saturated colors that mean nothing is the opposite. Reserve strong color for state (good, warning, bad) and keep everything else quiet, so that when something on the screen is red, red is the only thing anyone sees.

> **Tip:** If a stakeholder insists everything is important, build the "everything" content as a stacked layer behind the headline elements or as a second dashboard, and keep the front screen brutal. The five-number front page with a "details" layer behind it satisfies both the executive who wants a glance and the analyst who wants the tables.

Refresh Behavior and the Feeling of Real Time

Nothing in a standard dashboard streams continuously. What feels like "live" is polling: at an interval, an element re-asks its question and repaints. Understanding where those intervals live keeps you from a dashboard that lies by omission.

There are two layers of refresh. First, a **layer refresh interval** can be set on a layer inside the web map (in the layer's settings in Map Viewer — see Chapter 6), which makes the map element re-fetch that layer's features periodically so the dots actually move. Second, each dashboard element has its own **refresh interval** in its general settings, which makes indicators, charts, lists, and tables re-query on schedule. Set only the map's layer refresh and your dots will move while your big number goes stale; set only the elements and the opposite happens. For a monitoring dashboard, set both, deliberately.

Choose intervals honestly. Data that updates nightly gains nothing from a one-minute refresh except server load; refreshing it hourly and *saying so* in a rich text note ("data refreshes nightly; display refreshes hourly") is both cheaper and more trustworthy. Conversely, a dispatch board over data that changes minute-to-minute deserves a short interval — on the layers and elements that need it, not on all of them. Every refresh from every open copy of the dashboard is a query against your layer; a wall display plus thirty open browser tabs, all polling every thirty seconds, is a real load, and it is the classic way a popular dashboard degrades the very services it monitors.

For genuinely continuous feeds — vehicle positions updating every few seconds — Esri has real-time capabilities (stream services and, in larger deployments, the ArcGIS Velocity product) that push updates rather than poll. Those come with their own infrastructure and cost implications; Chapter 35 (ArcGIS Enterprise) sketches when that world applies to you. For most organizations, honest polling on a well-chosen interval is real-time enough.

> **Watch out:** A dashboard has no banner announcing stale data. If the nightly load job silently fails, the dashboard cheerfully shows yesterday's numbers all day with nothing visibly wrong. Two defenses: display a "last updated" timestamp from the data itself (a maximum-of-date-field statistic in a small indicator, or via advanced formatting), and make sure the pipeline that feeds the layer has its own monitoring. A confident-looking screen of wrong numbers is the worst failure mode a dashboard has.

Performance with Big Layers

A dashboard multiplies queries: one screen might fire a dozen at load, and refresh intervals fire them again forever. Against small layers nobody notices. Against layers with hundreds of thousands of features, design choices dominate.

Let the server summarize. Layer-driven statistics ask the server for a few numbers and receive a few numbers. Anything that instead pulls features to the browser — heavy data expressions, lists with no meaningful limits, tables of raw rows — scales with data size. Prefer the summary path wherever one exists.

Index the fields you query. Every filter, category field, and statistic field in your elements should be indexed on the hosted feature layer — an index being the database structure that lets the server find matching rows without scanning the whole table. Chapter 10 covers how; the symptom of a missing index is a dashboard where one element consistently paints seconds after the rest.

Filter early and everywhere. An element-level filter ("last 30 days," "status is Open") shrinks every query that element ever makes. Better still, point the dashboard at a **layer view** with a definition query baked in (Chapter 10 again), so no element *can* accidentally ask for all of history. Defaulted date selectors, as noted earlier, serve the same goal at load time.

Pre-summarize what you check constantly. If the same expensive rollup is computed on every refresh for every viewer, compute it once instead: a scheduled script (Chapter 31 covers the Python tooling) that writes a small summary table on a schedule, with the dashboard reading the small table. The dashboard becomes instant, and your million-row layer is queried once an hour instead of once per viewer per refresh.

Respect the map's own weight. The map element carries all the rendering costs of its web map — too many visible layers, complex symbology, unoptimized drawing all show up here, magnified by refresh. Chapter 10's guidance on layer performance and Chapter 7's on rendering apply verbatim. A dashboard map usually wants fewer layers than an exploration map: it is there to show a situation, not a GIS.

The through-line is that a dashboard should ask small questions frequently, never big questions repeatedly. Get the reduction done upstream — in views, filters, indexes, and summary tables — and the screen itself stays effortless.

From Here

Two chapters continue this thread. Chapter 36 (Worked Project: From Raw Data to Published Dashboard) builds a complete dashboard from scratch with every configuration step shown, and is the natural next read if you learn by doing. Chapter 30 (Field Operations) covers the apps that *create* the live data dashboards love to display — a Survey123 form or Field Maps deployment feeding a dashboard is the classic end-to-end operations pattern, and Chapter 37 walks one of those end to end. Finally, remember that a dashboard is shared like any other item: viewers need access to the

dashboard *and* to every layer it touches, and the mechanics of groups and sharing levels live in Chapter 34 (Administration). Nothing embarrasses a dashboard rollout quite like a wall of "you do not have permission" tiles at the big unveiling — share the whole stack, test with a colleague's account, and then let the screen do its quiet work.

Experience Builder, Complete

Experience Builder is the app builder Esri gives you when the ready-made options run out of room. Instant Apps (Chapter 26) hand you a finished design and let you flip switches. Dashboards (Chapter 28) assume you want charts and gauges around a map. StoryMaps (Chapter 27) assume you want a narrative that scrolls. Experience Builder assumes nothing: you get a blank canvas, a catalog of widgets — self-contained building blocks like a map, a list, a chart, or a button — and a drag-and-drop editor for arranging them into anything from a one-page map viewer to a multi-page mini-website with no map on the front page at all.

That freedom is the product's whole personality, for better and worse. This chapter covers the structure of an experience (pages and windows), how layout and responsive design actually work, the widget catalog, the wiring system that makes widgets talk to each other, multi-source data, theming, publishing, and — because it matters more than any feature — an honest account of when Experience Builder is the right tool and when it is an expensive way to rebuild something an Instant App would have given you in ten minutes.

What an Experience Is

An *experience* is a web app: a single item in your ArcGIS content that anyone with a browser and the right sharing permissions can open. Under the hood it is built on the ArcGIS Maps SDK for JavaScript — the same library professional developers use to hand-code web mapping apps (Chapter 33) — but Experience Builder wraps that library in a visual editor so you never touch code.

You create one from ArcGIS Online through the app launcher (the grid of dots near your profile picture) by choosing Experience Builder, or from your content page via `Create app > Experience Builder`. Either route lands you in a template gallery first; pick anything, including a blank template, and you arrive at the builder itself.

The builder screen has three zones worth naming now, because every instruction in this chapter refers to them:

- **The canvas** in the middle: a live preview of the page you are editing.
- **The left toolbar**: panels for inserting widgets, managing pages, connecting data, and setting the theme.
- **The top bar**: undo/redo, a device-size switcher, and the `Save`, `Preview`, and `Publish` buttons.

An experience is a hierarchy. At the top is the experience itself. It contains one or more **pages**. Pages contain **widgets**, often nested inside layout containers. Floating alongside pages are **windows** — overlays that appear on top of a page. Get this skeleton clear and everything else in the builder makes sense.

Pages and Windows: The Skeleton

Pages

A page is one screen of your app. A simple map explorer needs exactly one. A departmental hub might have a landing page, a map page, a data-table page, and an "about" page, connected by a menu widget so users can move between them.

You manage pages in the page panel on the left toolbar. Each page can be one of two fundamentally different kinds, and the choice shapes everything you build on it:

- **Full-screen pages** fill the browser window exactly, with no scrolling. Widgets are sized relative to the window, so the map stretches to fill whatever screen it is viewed on. This is the right mode for tool-like apps: viewers, editors, dashboard-style layouts where everything should be visible at once.
- **Scrolling pages** behave like a normal web page: content flows top to bottom and the user scrolls through it. This is the right mode for narrative or informational pages —

a landing page with a hero image, some explanatory text, a map partway down, and a footer.

You can mix both kinds in one experience: a scrolling landing page that links to a full-screen map page is one of the most common and most effective patterns.

> **Tip:** Decide full-screen versus scrolling before you build anything on a page, not after. Converting a layout from one mode to the other effectively means rebuilding it, because the sizing logic is different. When in doubt: tools go full-screen, reading material scrolls.

Pages can also be nested into a hierarchy in the page panel — sub-pages under a parent — which menu widgets can render as dropdown navigation. You can set any page as the app's default (the one that loads first), and you can create **links** in the page list that point to external URLs, so your app's menu can include, say, a link back to your organization's main website.

Windows

A window is an overlay: a panel that appears on top of the current page rather than replacing it. Windows are how you build splash screens ("Read this disclaimer before continuing"), confirmation prompts, pop-out help panels, and detail views that appear when a user clicks a button.

Windows live in their own section of the page panel. A window can be set to open automatically when the app loads (the splash-screen pattern, often with an "I agree" button that closes it), or it can be opened by a click — for example, a button widget whose link setting is "open window." Windows come in a couple of flavors: centered dialogs that dim the page behind them, and panels anchored to an edge of the screen. Because a window is just another container, you can put any widgets inside it — text, images, even a small map.

Layout: Containers and Responsive Design

Layout is where new Experience Builder users lose the most time, so it deserves a careful walk-through.

Why containers beat free placement

On a full-screen page, you *can* drag a widget anywhere on the canvas and drop it at absolute coordinates. Resist this. Freely placed widgets are pinned to positions that make sense at your screen size and almost nobody else's. The professional habit is to place widgets inside **layout containers** — special widgets whose only job is to hold and arrange other widgets. The core set:

- **Row** arranges its children side by side, dividing horizontal space among them.
- **Column** stacks its children vertically.
- **Grid** divides space into resizable cells, each holding one widget — the workhorse for dashboard-like full-screen layouts, because users can even drag the cell dividers at runtime if you allow it.
- **Sidebar** splits the screen into a main area and a collapsible side panel, with a built-in handle for users to open and close the panel. The classic "map on the right, list of results on the left" app is one sidebar widget.
- **Fixed panel** holds a widget at a set position and size — useful for a small legend floating over a map — and is the disciplined version of free placement.

Containers nest. A sidebar's panel might contain a column, which contains a search widget above a list widget. Deep nesting is normal; five levels is unremarkable. The builder shows the full nesting as an outline (a tree view of every widget on the page), and that outline is often an easier place to select and rearrange widgets than the canvas, where clicking tends to grab the wrong layer of the sandwich.

> **Tip:** Rename your widgets as you add them. Ten widgets named "Column," "Column 2," and "List 4" become unmanageable fast. A ten-second rename to "Results panel" or "Site map" in the outline pays for itself every time you revisit the app.

On scrolling pages, the container story is simpler: content flows in horizontal bands (often called sections or blocks) stacked down the page, and within each band you use rows and columns as usual.

Two stacking concepts will greet you in templates before you ever build them from scratch. A **Section** is a container that holds several interchangeable *views* — alternate layouts occupying the same footprint — with a views-navigation widget (tabs, arrows, or buttons) flipping between them; that is how a single panel offers "Chart | Table | About" tabs without three separate panels. On scrolling pages, a **screen group** produces the scroll-driven pattern StoryMaps readers know: a media stage that stays put while narrative panels scroll past over it.

Responsive breakpoints: large, medium, small

Every page in an experience has three layouts, one per device class: **large** (desktop), **medium** (tablet), and **small** (phone). The device switcher in the top bar flips the canvas between them.

Here is the mental model that prevents most confusion: the three layouts share the same *widgets* but not the same *arrangement*. When you build on the large layout, the builder auto-generates medium and small arrangements for you. The moment you manually adjust the medium or small layout — move something, resize something — that layout stops mirroring the large one and becomes your responsibility.

Widgets can also be present in one layout and absent from another. When you remove a widget from, say, the small layout, it is not deleted from the app; it moves to a *pending* list — a holding area of widgets that exist in the experience but are not placed on the current screen size. This is a feature, not a bug: a phone layout genuinely should drop the decorative image and the third chart. But it cuts both ways — a widget you add while editing the small layout will sit in the pending list of the large layout until you place it there too.

> **Watch out:** Always check all three device sizes before publishing. The single most common Experience Builder failure in the wild is a beautiful desktop app whose phone layout was auto-generated and never inspected — with widgets overlapping, panels covering the map, or the key button buried. A large share of your users will open the app on a phone. Budget real time for the small layout.

The Widget Catalog

Widgets are the vocabulary of Experience Builder, and the catalog has grown to several dozen, with more added in most releases. You insert them from the widget panel on the left toolbar and configure each one in a settings panel that appears on the right when the widget is selected. Rather than an exhaustive inventory that would be stale within a year, here is the catalog organized by what each category is *for*:

CATEGORY	REPRESENTATIVE WIDGETS	WHAT THEY DO
Map-centric	Map, Legend, Map Layers, Basemap Gallery, Bookmarks, Search, Draw, Measurement, Print, Swipe, Directions	Display a web map or scene and give users the classic map-viewer tools
Data-centric	List, Table, Chart, Filter, Query, Feature Info, Near Me	Show and interrogate the *attributes* behind the map — records, statistics, filtered subsets
Layout	Row, Column, Grid, Sidebar, Fixed Panel, Section, Card, Divider, Widget Controller	Structure the page and organize other widgets
Page content	Text, Image, Button, Embed, Menu, Views Navigation	Narrative, branding, navigation, and embedded outside content
Action-oriented	Button (with link settings), Share, Survey, Edit, Select	Trigger behavior: open windows, link pages, submit forms, edit features

A few of these earn extra commentary.

The **Map widget** is the anchor of most experiences. It displays a web map or web scene (its 3D counterpart) that you built in Map Viewer (Chapter 6) — meaning all your styling, pop-ups, and labels (Chapters 7 and 9) come along for free. One map widget can even hold a second map or scene and let users toggle between them — pairing a 2D map with a 3D scene of the same area is the classic use. Configuration of the map itself still happens in Map Viewer; Experience Builder consumes the finished product. Change the web map, and the experience updates automatically.

The **List widget** repeats a card layout once per record in a data source — one card per fire station, per project, per inspection. You design a single card (a small canvas of its own where you place text, images, and buttons wired to the record's fields) and the widget stamps it out for every feature. Paired with a map through actions (next section), it produces the "click a card, zoom the map" interaction that defines a huge share of real-world experiences.

The **Table widget** shows raw attribute rows, sortable and searchable — the honest spreadsheet view many users secretly want. The **Chart widget** draws bar, line, pie, and scatter charts from a layer's attributes, aggregating on the fly. The **Filter widget** exposes predefined filter conditions as buttons or toggles ("Show only open permits"), while **Query** lets users build a search against attributes or a location and returns the results as a new data set other widgets can display.

The **Widget Controller** deserves a special mention for full-screen apps: it is a toolbar that holds other widgets and opens each one in a floating panel when clicked. Instead of cramming a legend, layer list, basemap gallery, and print tool onto the screen simultaneously, you dock them all in a controller and the user summons one at a time. It is the single best cure for cluttered full-screen layouts.

Two widgets bridge to the rest of the platform: the **Survey widget** embeds a Survey123 form (Chapter 30) so users can submit data without leaving the app, and the **Embed widget** displays any external web content by URL — another app, a video, a chart from a non-Esri tool. The **Edit widget** enables direct feature editing against an editable hosted feature layer, subject to the layer's settings (Chapter 13 covers editing workflows and their guardrails).

If a widget you need does not exist, there is an escape hatch: **Experience Builder Developer Edition**, a downloadable version of the builder that developers run on their own machine to create custom widgets in code, which then appear in the catalog alongside the built-ins. That is squarely developer territory — Chapter 33 sketches the skills involved — but it is worth knowing the ceiling is not the stock catalog.

Wiring Widgets Together: Triggers, Messages, and Data Actions

A pile of widgets becomes an application when the widgets react to each other. Experience Builder has two wiring systems, and understanding the difference saves a

lot of head-scratching.

Message actions: you design the behavior

A **message action** is a rule you configure at design time: *when X happens in widget A, do Y in widget B*. The pieces have names:

- The **trigger** is the event: a record gets selected, the map extent (the visible area) changes, data gets filtered. (Buttons are wired a little differently — a button's click behavior lives in its own link setting, which can open a window, jump to a page, or follow a URL.)
- The **target** is the widget or data source that should respond.
- The **action** is what the target does: zoom to the selection, pan there, flash the feature, apply a filter, show the selection on the map.

You configure message actions in the settings panel of the *triggering* widget, under its action settings — the flow is roughly `select the widget > Action > Add a trigger`, then choose the event, the target, and the action from lists the builder offers. Only sensible combinations are offered: a chart widget knows how to respond to a filter, a map knows how to zoom.

A concrete example, because this is easier shown than defined. Suppose you have a map of libraries and a list widget showing one card per library, and you want clicking a card to zoom the map:

1. Select the List widget and open its action settings.
2. Add a trigger for *record selection changes*.
3. Choose the Map widget as the target.
4. Choose *Zoom to* as the action.

Now add the reverse wiring on the Map widget — when a feature is clicked on the map, select the matching record in the list — and the two widgets feel like one instrument. Add a third rule so that a Filter widget's changes also filter the list, and you have the skeleton of most professional experiences: filter, browse, locate, inspect.

One powerful and non-obvious trigger is *extent changes*: you can make a list or chart show only the features currently visible on the map, so panning the map live-updates

the side panel. Used well, it makes an app feel alive; used carelessly on a large layer, it triggers a query at every pan and the app feels sluggish — see Chapter 10 for why layer performance is mostly won or lost at publishing time.

Data actions: the user picks from a menu

A **data action** is different: it is a menu of operations the builder attaches to data-displaying widgets at runtime, which *users* invoke themselves. Enable data actions on a table widget, and users get a menu on each record or on the whole set with entries like export the data, show the selection on the map, view the record in the table, or calculate statistics. You choose which entries to allow in the widget's settings; the user chooses when to fire them.

The rule of thumb: message actions are choreography you script in advance; data actions are tools you hand the user. Most apps need both — scripted map-to-list linking, plus an export option for the analyst who wants the data in a spreadsheet.

> **Tip:** Actions only run in a live app, not on the editing canvas. Use the `Preview` button constantly while wiring widgets — it opens the experience in a new tab exactly as users will see it. Debugging actions by publishing over and over is slow and pollutes the live app with half-finished states.

Connecting Multiple Data Sources

Everything data-driven in an experience traces back to the **data panel** on the left toolbar, where you register data sources. An experience can hold many, and of different kinds:

- **Web maps and web scenes** — the usual starting point. Adding a web map brings along every layer inside it.
- **Individual feature layers** — hosted layers from your content, layers from the Living Atlas (Chapter 5), or any feature service you can reach by URL, including ones from other organizations' portals and public servers (Chapter 32 explains what those service URLs actually are).
- **Output data sources** — results generated *inside* the app at runtime. A Query widget's result set, for example, registers as a data source of its own, which a list or

chart can then display. This is how you build "search, then explore the results" flows.

The important design consequence: widgets do not have to share a map to share data. A chart on your landing page can summarize the same layer that a map on page two displays, with no map widget on the landing page at all. A single experience can present three different web maps on three pages, plus a standalone table fed straight from a layer. Selection state travels too — select features in one widget and every widget bound to the same data source sees the selection, which is what makes the cross-widget choreography of the previous section possible.

Each data source also supports a design-time **filter** you set in the data panel, so one layer can appear multiple times in the app pre-filtered different ways — "open requests" feeding one list, "closed requests" feeding another — without publishing duplicate layers. If you find yourself doing this heavily, though, read Chapter 10 on hosted feature layer *views*, which solve the same problem at the layer level with better performance and security control.

> **Watch out:** Every data source you add is another set of network requests when the app loads. An experience stitched together from a dozen layers across five servers will be only as fast and as reliable as the slowest, flakiest one — and if a source you do not own gets deleted or its sharing changes, that widget in your app silently breaks. Prefer few, well-managed sources; audit any source you do not control.

Theming and Branding

The theme panel on the left toolbar controls the app's overall look: a set of preset themes (light, dark, and several styled variants), each of which you can then customize — primary and secondary colors, typography, and general surfaces. Change the theme and every widget restyles at once, which is exactly what you want: consistency by default.

Individual widgets can override the theme in their own style settings — background color, borders, padding, and so on. Use overrides sparingly. An app where every widget was styled by hand looks like it, and not in a good way. The professional pattern is: pick

the theme, set the brand colors once, override only where a specific widget genuinely needs emphasis. If your organization has a shared brand, ask your administrator whether an organization-wide shared theme has been configured (Chapter 34 covers organization settings); if so, new apps can inherit it automatically.

Typography deserves one caution: Experience Builder lets you set fonts per text widget, and scrolling pages full of hand-set text drift out of consistency fast. Decide your heading and body styles early, and reuse them. The design principles from Chapter 4 — hierarchy, restraint, contrast — apply to app chrome just as much as to maps.

Templates: Starting Points, Not Prisons

Every experience starts from a template — even "blank" is technically a template. The gallery offers dozens, organized by intent: full-screen map viewers, scrolling one-pagers, multi-page website-style layouts, dashboard-like arrangements, and layouts tuned for particular workflows. Templates differ from Instant Apps in one crucial way: a template is a *starting arrangement*, not a locked design. After you pick one, every widget in it is fully editable — delete half of it, rewire the rest, nothing is off-limits.

The practical guidance: browse templates for the *structure* closest to your goal (number of pages, full-screen versus scrolling, sidebar or not) and ignore the demo content, which you will replace anyway. Starting from the right template saves the hour of container-nesting that a blank canvas costs.

Two lesser-known template facts earn their space here. First, you can generate a template *from your own experience* — build your department's standard app once, save it as a template, and every future app starts from your layout with your theme already applied. Second, templates can be shared within your organization, so a team can standardize without everyone rebuilding the same skeleton. For any organization producing more than a handful of apps, a house template is one of the highest-leverage hours you can spend.

Publishing, Sharing, and the Draft/Published Divide

Experience Builder separates **saving** from **publishing**, and confusing the two is a rite of passage.

`Save` writes your working draft. `Publish` pushes the draft to the live app — the URL other people open. Until you publish, viewers see the *last published* version, no matter how much you have saved since. After the first publish, the builder marks the item when unpublished changes exist, so you can always tell whether the live app is current.

> **Watch out:** "I fixed it and saved, but users still see the old version" is almost always an unpublished draft. Check for the unpublished-changes indicator and publish. The reverse mistake also happens: publishing half-finished work because you wanted to save it. Save early and often; publish deliberately.

Publishing does not make the app visible to anyone by itself — that is governed by the item's **sharing level**, like every other item in ArcGIS: private, shared to groups, shared to the organization, or public. Set it from the share option in the builder or from the item page. One subtlety that generates a disproportionate share of support requests: the app and its data sources are separate items with separate sharing. A public app pointed at a private layer greets the world with login prompts or empty widgets. Before you circulate a link, open it in a private/incognito browser window while signed out — the two-minute test that catches this whole class of failure. Chapter 34 covers sharing mechanics; be aware that widgets backed by premium services — Directions is the usual example — can consume credits each time anyone uses them, including anonymous visitors to a public app. Chapter 10 covers securing the layers themselves.

A published experience has a stable URL you can circulate directly or embed in another website. The URL also accepts parameters that set the initial state — things like which page opens or what the app centers on — useful for linking into a specific view from an email or another app; check the current documentation for the exact parameter names, which have shifted between releases.

When Experience Builder Wins — and When It Is Overkill

Now the judgment call the whole chapter has been building toward. Experience Builder sits at the flexible end of a spectrum, and flexibility is a cost you should only pay when you need what it buys.

SITUATION	REACH FOR	WHY
One map, standard tools, done today	Instant Apps (Ch 26)	Configured in minutes, maintained by Esri, hard to make ugly
A story to tell, mostly linear	StoryMaps (Ch 27)	Purpose-built narrative scrolling, better authoring for text
Live operational metrics around a map	Dashboards (Ch 28)	Purpose-built indicators, gauges, and auto-refresh
Multi-page app, custom layout, mixed content, cross-widget interactions	Experience Builder	Nothing else offers the layout and wiring freedom
Requirements beyond any builder	Maps SDK for JavaScript (Ch 33)	Full code, full control, full responsibility

Choose Experience Builder when at least one of these is true: you need **multiple pages**; you need a **layout no template app offers** (a landing page above a map app, side-by-side maps with synchronized lists, a form next to a table next to a chart); you need **widgets talking to each other** in ways you specify; you need **multiple unrelated data sources** in one app; or you need the app to **not look like a GIS viewer** — branded, website-like, map optional.

Choose against it when the honest answer to "what does this app do?" is one sentence an Instant App already covers. The hidden costs of Experience Builder are real:

- **Build time.** A sidebar app that takes ten minutes in Instant Apps takes an afternoon here, once you account for wiring, theming, and three device layouts.
- **Maintenance.** You own the layout. When requirements change, someone has to reopen the builder and understand the nesting and action wiring — which is why renaming widgets and keeping the structure simple matters.
- **Design risk.** Instant Apps cannot be made incoherent; experiences can. Freedom includes the freedom to ship a cluttered, inconsistent app.

- **Responsive burden.** The auto-generated phone layout is a draft, not a deliverable.

A sound habit for any new app request: prototype it as an Instant App first, even if you suspect you will outgrow it. If the Instant App covers the requirement, you are done. If not, you now know *precisely* which gap you are paying Experience Builder's complexity to close — and that clarity will make the experience you build smaller, sharper, and easier to maintain.

For a full worked example that takes raw data through a published app, see Chapter 36; for the field-facing counterpart where Experience Builder often serves as the office-side window into collected data, see Chapters 30 and 37.

Field Operations: Field Maps, Survey123, and QuickCapture

Every field operation, no matter how sophisticated, is trying to solve the same old problem: a person standing somewhere outside knows something, and a person sitting somewhere inside needs to know it too. For most of history the bridge between them was paper — clipboards, inspection forms, hand-annotated map printouts — followed by hours of retyping and a fresh crop of transcription errors. ArcGIS replaces that bridge with three mobile apps that write directly into the same hosted feature layers your maps already read from. The clipboard becomes a phone. The retyping disappears. And the office can watch data arrive on a dashboard while the field crew is still standing in the mud.

The three apps are ArcGIS Field Maps, ArcGIS Survey123, and ArcGIS QuickCapture. They overlap enough to cause confusion and differ enough that picking the wrong one will genuinely hurt. This chapter covers each in depth, explains how to choose, and then walks through the plumbing underneath all three: high-accuracy GPS, offline sync, conflict handling, and the dashboard-backed operations pattern that ties field and office together.

The Shared Architecture

All three apps sit on the same foundation, and understanding it first makes everything else simpler.

At the center is a **hosted feature layer** — a table of geographic records stored in your ArcGIS organization, which Chapter 10 (Hosted Feature Layers) covers in full. The field apps are editors of that layer. When a crew member drops a point, fills in a form, and taps submit, they are adding a row to the same layer your web maps, dashboards, and analysis tools see. There is no export step, no nightly batch, no "field database" separate from the "real database." One layer, many doors into it.

That single fact drives the whole design workflow:

1. **Design the schema first.** Fields, types, domains (pick lists), and attachments are properties of the layer, not the app. Get them right before you touch any app designer — Chapter 12 (Schema Design: Fields, Domains, and Relationships) is the prerequisite reading here, because a well-built domain becomes a well-built dropdown automatically in every field app.
2. **Configure the app experience.** Each app has a web-based designer where you shape what the field user sees: which fields, in what order, with what logic.
3. **Share to a group.** Field users sign in with their ArcGIS accounts, and they see the maps, forms, or projects shared with groups they belong to. Access control is the standard sharing model from Chapter 34 (Administration).
4. **Watch the data arrive.** The office monitors the same layer through Map Viewer, a dashboard, or scheduled reports.

> **Tip:** Resist the urge to open a field app designer on day one. The most common failure mode in field projects is discovering, two weeks into collection, that you need a field you didn't create. Adding a field to a hosted layer mid-project is possible, but reworking forms, retraining crews, and back-filling old records is painful. Spend the extra hour on schema design; it repays itself many times over.

Choosing Between the Three

Here is the honest one-paragraph version: **Field Maps** is a map that you edit — use it when the map itself matters, when crews need to see existing assets and update them.

Survey123 is a form that happens to capture location — use it when the questions matter more than the geography, especially long or logic-heavy questionnaires. **QuickCapture** is a panel of big buttons — use it when speed matters more than detail, like recording observations from a moving vehicle.

The longer version, as a comparison:

	FIELD MAPS	SURVEY123	QUICKCAPTURE
Mental model	An editable map	A smart form	A button board
Primary interface	The map; forms open per feature	The form; map is one question among many	Buttons; one tap records one thing
Best for	Asset inspection, updating existing features, anything where seeing nearby data matters	Structured interviews, inspections with many questions, surveys with branching logic	Rapid observations, windshield surveys, damage tallies, anything done in motion
Editing existing features	Excellent — its core purpose	Creation-first, though the field app's Inbox can open existing records for editing	No — capture only
Form logic	Conditional visibility, required fields, calculated values	The deepest logic of the three: branching, constraints, repeats, calculations	Minimal by design
Related records	Supported through the map's related tables	Supported through repeats	Not the tool for this
Offline	Yes, via offline map areas	Yes, forms and basemaps download to the device	Yes, projects work fully offline

	FIELD MAPS	SURVEY123	QUICKCAPTURE
Speed per record	Moderate	Slowest (deliberately — it asks the most)	Fastest, by a wide margin

Two more decision heuristics that resolve most borderline cases:

Does the crew need to see what's already there? If a technician must find hydrant 4471 among two hundred hydrants and update its status, that is Field Maps — the map is the navigation and the context. If a canvasser knocks on doors and fills out a fresh questionnaire at each one, the existing data barely matters, and Survey123's superior form engine wins.

Is the person stationary while capturing? Filling out a real form requires stopping. If the workflow is "call it out as you drive past" — invasive weeds along a highway, storm damage across a county — nothing beats QuickCapture's one-tap capture, which stamps location, time, and preset attributes without the vehicle slowing down.

And note that this is not an exclusive choice. The apps compose. A common hybrid: crews navigate and manage assets in Field Maps, and a link configured in the Field Maps pop-up launches Survey123 with the asset's ID pre-filled, so the deep inspection form lands in a related table against the right feature. Chapter 37 (Worked Project: A Field Collection Operation End to End) builds exactly this kind of combined operation.

Field Maps in Depth

Field Maps is really two things: a mobile app your crews install from their device's app store, and **Field Maps Designer**, a browser-based configuration tool you open from the app launcher in your ArcGIS organization (the grid icon near your profile). Designer is where all the authoring happens — you point it at a web map, and it lets you shape the forms, offline behavior, and app settings for that map.

The map itself is an ordinary web map, built in Map Viewer exactly as Chapter 6 (Map Viewer: The Complete Reference) describes. Symbology, labels, and pop-ups carry over. What Field Maps Designer adds is the editing experience.

Building Forms

A **form** in Field Maps controls what a field user sees when they create or edit a feature: which attributes appear, in what order, grouped how, with what behavior. Without a configured form, users get a raw list of every editable field in schema order — functional, but hostile.

In Field Maps Designer, select your map, open the form builder for a layer, and you'll find a drag-and-drop canvas. The essential moves:

- **Include only what the field user should touch.** Office-managed fields (asset IDs assigned by another system, QA flags) can stay off the form entirely.
- **Group related fields.** Collapsible groups — "Location Details," "Condition Assessment," "Follow-up" — turn a fifty-field wall into a navigable document.
- **Lean on domains.** Any field with a coded value domain renders as a pick list automatically. Every value a crew can pick instead of type is a typo you never have to clean. This is the payoff of Chapter 12's discipline.
- **Set defaults and calculated values.** Form elements can carry calculated expressions written in Arcade, the expression language covered in Chapter 8 (Arcade: From Zero to Fluent). A calculation can stamp the inspection date, derive a full address from components, or copy a value from the feature the user is editing — filled in live as the form is used.

Conditional Visibility and Required Fields

The two form features that most improve data quality:

Conditional visibility shows or hides a form element based on an Arcade expression that evaluates other answers. The classic pattern: a "Damage type" pick list and "Damage description" text field appear only when "Is damage present?" is set to Yes. Crews doing routine passes see a short form; crews finding problems see the follow-up questions. The expression lives on the form element in Designer — select the element, attach a visibility expression, and write a short Arcade statement that returns true or false.

Required fields refuse submission until answered. Use them sparingly and honestly: mark a field required only if a record genuinely cannot be useful without it. Over-required forms train crews to enter junk placeholder values just to get past the gate — which is worse than a blank, because a blank at least admits it's missing. A powerful

combination is *conditionally required*: required only when visible, so "Damage description" is mandatory exactly when damage was reported.

> **Watch out:** Test every conditional path on an actual phone before deployment, not just in the Designer preview. A form that behaves perfectly in a desktop browser can hide a required field behind a condition that never triggers on the device, leaving crews unable to submit at all. Ten minutes of on-device testing catches what an hour of desktop review misses.

Offline Map Areas

Field work happens where cell coverage doesn't. Field Maps handles this through **offline map areas** — packaged chunks of your map, downloaded to the device, edited locally, and synchronized when connectivity returns.

Prerequisites first, because this is where most offline attempts stall:

- Every editable layer in the map must be a hosted feature layer with **sync enabled** — a setting on the layer's item page, under its editing and offline settings. Sync is what allows a device to check out a local copy and later reconcile changes.
- The basemap must be offline-capable. Esri provides basemaps designed for export; a custom basemap needs its export/download setting enabled. A map area is useless if the reference map under it can't come along.
- The web map itself must be enabled for offline use, a toggle you'll find in Field Maps Designer's offline settings for the map.

With prerequisites met, you have two modes of getting areas onto devices:

Pre-planned map areas, defined by you in Field Maps Designer: draw a rectangle or shape around each work zone, choose the basemap detail level, and let the system package them. Field users then download a ready-made area with one tap. This is the right pattern for organized operations — packaging happens once on the server rather than repeatedly on each device, downloads are faster, and you control exactly what crews carry.

On-demand areas, where the field user outlines their own area on the device before leaving coverage. More flexible, appropriate for unpredictable work, but it depends on the user remembering to do it while they still have signal.

Size the areas thoughtfully. Basemap tiles dominate the download, and detail level is the multiplier: doubling the zoom depth of an area can inflate it dramatically. Give crews the detail they need to do the job and not one level more.

Geofencing Awareness

Field Maps can react to *where the device is*, not just record it. A **geofence** is a virtual boundary drawn around an area; crossing it can trigger an action — a notification on the device, or starting and stopping location sharing. Configured through the map's geofence settings in Field Maps Designer against an area layer you supply, geofences enable patterns like:

- Warning a crew member entering a restricted or hazardous zone.
- Reminding an inspector, on arrival at a site, of an open work item there.
- Switching location sharing on automatically as crews enter a job site, and off as they leave.

Closely related is **location sharing** — the capability where field devices report their positions as a track, letting supervisors see where crews are and where they've been. It's typically licensed separately, and it deserves a candid conversation with your crews before you switch it on. Location tracking that feels like surveillance poisons a field program faster than any technical failure. Frame it around what it does for the crew — safety check-ins, proof of work completed, faster help when something goes wrong — and set retention policies you're prepared to defend.

> **Watch out:** Continuous location features cost battery. A device running an all-day tracked shift with the screen active needs a vehicle charger or battery pack as standard-issue equipment, not an afterthought. Dead devices at 2 p.m. is a predictable failure, so plan for power the way you plan for rain.

High-Accuracy GPS

A phone's built-in GPS is engineered for "which street am I on," not "which side of the pipe am I on." Under open sky, expect positions good to a few meters; under tree canopy or beside buildings, worse. For plenty of work — observations, condition reports, anything symbolized at neighborhood scale — that's fine. For asset mapping where the point needs to be *on* the buried valve, it isn't.

The step up is an **external GNSS receiver** — a dedicated satellite positioning unit that pairs with the phone or tablet over Bluetooth and replaces the device's internal fix. (GNSS, for Global Navigation Satellite System, is the umbrella term covering GPS and its international siblings; more satellites in view means better geometry and better fixes.) The accuracy tiers work roughly like this:

- **Consumer (built-in):** meters of error. Fine for observations and small-scale mapping.
- **Submeter:** an external receiver using correction sources such as free satellite-based augmentation broadcasts, which compensate for atmospheric distortion. Good enough for most utility and asset inventories.
- **Centimeter (RTK):** Real-Time Kinematic positioning, where the receiver gets a live correction stream from a base station or network service — often delivered over the internet via NTRIP, a standard streaming protocol for correction data. This is survey-adjacent territory, with costs and setup complexity to match.

Both Field Maps and Survey123 support external receivers: you pair the receiver in the device's Bluetooth settings, then select it as the location provider inside the app's settings, along with an antenna height so positions are corrected from the antenna down to the ground.

Three practices turn good hardware into good data:

Capture GPS metadata. The apps can automatically write fix quality — estimated accuracy, fix type, satellite counts — into designated fields on your layer if those fields exist in the schema; Esri publishes the expected field names, and Field Maps Designer can add the whole set for you. Without metadata, every point in your layer looks equally trustworthy forever; with it, you can audit later and re-visit only the weak fixes.

Set an accuracy threshold. In the app's collection settings, you can require that the current fix be better than a chosen threshold before a point can be captured. Crews then wait the extra seconds for the fix to settle rather than banging in whatever the receiver first reports.

Use averaging for important points. Both apps can average a stream of fixes over a number of seconds into one position, smoothing the jitter inherent in any single reading. Slower per point, meaningfully better for permanent assets.

Remember also that coordinates from any receiver arrive in a specific coordinate system, and high-accuracy work makes datum mismatches visible in ways casual work never does — a transformation subtlety that Chapter 3 (Coordinate Systems and Projections, Complete) explains. If your centimeter receiver and your target layer disagree about the datum, you can pay for centimeter equipment and still land a meter off.

Survey123 in Depth

Survey123 inverts Field Maps' priorities: the form is the whole experience, and the map is demoted to one question within it. That sounds like a limitation until you see what a dedicated form engine buys you — branching logic, validation, repeating sections, and a filling experience polished enough to hand to the general public, since Survey123 forms can also run in a plain web browser with no app install and, if you choose, no sign-in.

Two Designers

Survey123 has two authoring environments, and knowing which you need saves real time.

The **web designer**, reached through the Survey123 website from your app launcher, is drag-and-drop: add questions, set choices, wire up basic logic, publish. It covers the majority of surveys and is where you should start.

Survey123 Connect is a free desktop application for power users. Under the hood, every survey is defined by a spreadsheet standard called **XLSForm** — one row per question, with columns for type, label, logic, and constraints. Connect exposes that spreadsheet directly, which unlocks the full depth of the form engine: advanced calculations, sophisticated constraint expressions, cascading choice lists driven by

external data, and fine control over the generated schema. The trade is that you're editing a spreadsheet, not dragging cards. A sensible path: prototype in the web designer, and graduate to Connect when you hit a wall. (One caution: the two designers are not interchangeable editors of the same survey — once a survey is authored in Connect, keep maintaining it there.)

Question Types

The web designer offers a generous palette. The ones you'll reach for constantly:

QUESTION TYPE	WHAT IT CAPTURES	NOTES
Single line / multiline text	Free text	Use sparingly; every free-text field is future cleanup
Number	Numeric values	Supports range constraints
Single choice / dropdown	One value from a list	Becomes a coded domain in the layer
Multiple choice	Several values from a list	Stored as delimited text — plan analysis accordingly
Date, time, date-time	Temporal values	Can default to "now"
Map (geopoint)	A location	The user's position or a tapped point; lines and areas are also available
Image	Photos	Stored as attachments; can require the camera, disallowing gallery picks
Signature	A drawn signature	Stored as an attachment image
Barcode	Scanned codes	Field app; turns asset tags into instant, typo-free IDs
Rating / Likert	Scaled opinions	For structured assessment questions

QUESTION TYPE	WHAT IT CAPTURES	NOTES
File upload	Documents	For permits, PDFs, supporting material

Survey answers land as fields in a hosted feature layer like any other data — a survey *is* a feature layer with a nice front door.

Logic: Visibility, Constraints, Defaults, Calculations

Four kinds of intelligence make a Survey123 form feel alive:

Visibility rules (called relevance in XLSForm terms) show or hide questions based on earlier answers — the same branching idea as Field Maps' conditional visibility, but composable into deep trees: answer "Commercial" and an entire section of commercial-only questions unfolds, each potentially with its own sub-branches.

Constraints validate an answer before accepting it: a number must fall in a plausible range, a date can't be in the future, an ID must match a pattern. The rejection happens instantly on the device, with a message you write — which beats discovering impossible values during analysis by weeks.

Defaults pre-fill the obvious: today's date, the signed-in user's name, values carried in from a launching link.

Calculations compute answers from other answers — subtotals, derived scores, concatenated labels — updating live as the user types.

The design principle across all four: *move the burden from the person to the form.* Every branch not shown, every error caught at entry, every value computed instead of typed is field time saved and cleanup avoided.

Repeats: Related Records Inside One Form

The standout structural feature is the **repeat** — a section of the form the user can fill multiple times within a single submission. One building inspection, many rooms; one water sample event, many bottles; one household interview, many household members. Press "add another," and the section repeats.

Under the hood, each repeat becomes a **related table** — a child table linked to the parent survey record, the relational pattern from Chapter 12. This matters for analysis: room records live in their own table, joined to their parent inspection, rather than being mashed into "Room1_condition, Room2_condition..." columns. Dashboards and queries can then treat rooms as first-class records.

The practical caveat: repeats have historically been authored in Survey123 Connect rather than the web designer, so a survey needing them is usually your cue to make the jump to XLSForm authoring. Check your version's web designer before assuming either way.

After Submission: Reports and Webhooks

A submitted survey can trigger work automatically. Two mechanisms are worth knowing:

Feature reports turn a submission into a formatted document from a template you design — the inspection report as a polished PDF, generated on demand or in batches. Report generation consumes credits, the organizational currency covered in Chapter 34, so budget accordingly for high-volume operations.

Webhooks are the general-purpose integration hook: when a survey is submitted, Survey123 sends the submission's data as a structured message to a web address you specify — configured in the survey's settings on the Survey123 website. Point that address at an automation platform (Power Automate, Make, Zapier, or a self-hosted tool like n8n) and the possibilities open up:

- Email the on-call supervisor the moment a survey reports a critical defect.
- Create a ticket in your work-order system from a citizen-submitted problem report.
- Post a summary to a team chat channel for every completed inspection.
- Append submissions to an external spreadsheet for a team that lives outside ArcGIS.

The webhook is the escape hatch that makes Survey123 a front door for *any* business process, not just a GIS one.

> **Tip:** Public-facing surveys are one of Survey123's superpowers. Because a survey can run in a browser and be shared with everyone, a citizen-reporting form —

potholes, storm damage, invasive species sightings — costs you nothing extra to stand up. Pair it with a webhook notification and a dashboard, and you've built a lightweight 311 system in an afternoon.

QuickCapture: Speed Above All

QuickCapture strips field collection to its minimum: a screen of large buttons, each representing one thing you might observe. Tap the button; a record is created with the location, the timestamp, the observer, and whatever attributes you pre-assigned to that button. Optionally a photo. That's the entire interaction — no form, no map to fuss with, often no need to look at the screen. Buttons can also start and stop line or polygon capture, tracing a path as the vehicle moves.

You author projects in the browser-based **QuickCapture designer** from your app launcher: lay out buttons in groups, bind each to a target layer, hard-code the attribute values each button writes, and choose options like photo capture or minimum accuracy. Projects download to the mobile app and run fully offline, transmitting records when connectivity allows.

The tool earns its keep wherever the observation rate outruns a form:

- **Windshield surveys:** road condition, signage problems, streetlight outages, called out at driving speed.
- **Rapid damage assessment** after a storm: affected / minor / major / destroyed as four buttons, letting a team triage a neighborhood in minutes and feed the results straight to an emergency dashboard.
- **Field tallies:** wildlife sightings, invasive plants, trail hazards on a hike or aerial pass.

The discipline QuickCapture imposes is deciding, up front, the complete vocabulary of what can be observed — because the field user gets buttons, not blank fields. That constraint is also its gift: every record arrives perfectly categorized, in a fixed schema, with zero typing. When a QuickCapture record needs follow-up detail, the pattern is the same hand-off as before: the rapid record flags the site, and a Field Maps or Survey123 visit does the deep documentation later.

Sync Architecture and Conflict Handling

Everything so far assumes edits flow smoothly from many devices into one layer. Most days they do. Understanding the machinery helps you design for the days they don't.

When a device takes a map offline, the sync-enabled layer hands it a **replica** — a checked-out local copy of the relevant data, tracked on the server. The field user edits the replica: new features, updated attributes, attached photos, deletions. Nothing touches the server until the user syncs.

Sync is a two-way exchange of deltas — just the changes, in both directions. The device uploads its edits; the server applies them and sends back whatever changed centrally since the last sync, so the device's copy freshens too. Attachments ride along, which is why photo-heavy workloads want Wi-Fi syncs; a day of inspection photos is a real upload.

Then the uncomfortable question: what happens when two people edit the *same feature* while both offline? Hosted feature layers resolve this with a blunt rule — **the last sync wins**. There is no merge screen, no conflict dialog; whichever device syncs later silently overwrites the earlier edit to that feature. For wholly separate features (the overwhelmingly common case) there's no conflict at all, and everyone's work interleaves cleanly. But overlapping edits to one feature are a genuine data-loss risk, and the defense is operational, not technical:

- **Partition the work.** Assign crews non-overlapping zones or asset lists, so two people rarely have reason to touch the same feature. Pre-planned offline areas make the partitions tangible.
- **Sync early, sync often.** The window for conflict is the time between syncs. Crews who sync at every coverage opportunity shrink it to nearly nothing. Field Maps can also auto-sync whenever it has connectivity — leave that on unless you have a reason not to.
- **Turn on editor tracking.** A layer setting (on the item page) that stamps every record with who created it and who last edited it, and when. It won't prevent an overwrite, but it makes the history auditable and the accountability clear.
- **Rehearse the failure.** Before launch, have two testers deliberately edit the same feature offline and sync in sequence, so you see exactly what last-wins looks like in

your data and can decide whether your workflow can tolerate it.

> **Watch out:** Never treat a device that hasn't synced as safe. Un-synced edits exist only on that phone — a lost, broken, or wiped device takes them along. End-of-day sync should be as non-negotiable in your field protocol as locking the truck.

One more planning note: schema changes and sync don't mix mid-flight. Adding or altering fields on a layer while devices hold offline replicas is a recipe for failed syncs. Land all edits, then change the schema, then re-download offline areas.

The Dashboard-Backed Field Operation

The pattern that elevates field collection from data entry into an *operation* is closing the loop with a live view for the office. Because field apps write straight into hosted feature layers, and ArcGIS Dashboards (Chapter 28, Dashboards: Complete) reads straight out of them, the two connect with no glue code at all:

1. **The layer** holds the truth — designed once, per Chapter 12.
2. **The field app** — whichever of the three fits — writes to it from the field.
3. **The dashboard** displays it in the office: a map of today's submissions, counters of completed versus outstanding inspections, a bar chart of defect categories, a list of critical findings sorted newest-first, all refreshing as syncs land.

This loop changes behavior on both ends. Supervisors reallocate crews at lunch instead of discovering coverage gaps a week later. Data quality problems surface within hours — a crew misusing a category shows up as a weird spike on a chart while retraining is still cheap. During emergencies, the dashboard *is* the common operating picture: QuickCapture damage tallies flowing onto a screen in the emergency operations center, minutes old.

Extend the pattern as the operation matures. Add a status field with a domain like *Needs visit → In progress → Complete → QA passed*, and the dashboard becomes a workflow tracker; office staff advance records to "QA passed" from Map Viewer while crews drive completion in the field. Add a Survey123 webhook for the critical-severity path so the right person is emailed instantly rather than noticing a dashboard tile. Add

location sharing, where appropriate and agreed, for a safety-oriented view of crew positions. Each addition is configuration, not code.

> **Tip:** Build the dashboard *before* the first real field day, even a crude one, and put it on a screen during your pilot. Nothing motivates a field team like watching their points appear live, and nothing surfaces a broken form faster than a dashboard full of blanks where data should be.

A Pre-Deployment Checklist

Field operations fail in the field, where fixes are expensive. Before real crews collect real data:

- Schema reviewed against Chapter 12's guidance; domains for everything domain-able; GPS metadata and editor-tracking fields in place.
- Forms tested on the actual device models crews will carry, through every conditional branch, including required-field behavior.
- Offline areas downloaded, edited, and synced on a device in airplane mode, end to end — including attachments.
- The two-editors-one-feature conflict rehearsed, and the result judged acceptable or the workflow partitioned.
- Batteries, chargers, and mounts issued; accuracy thresholds set; receivers paired if using external GNSS.
- Dashboard live, showing pilot data; someone named as its watcher.
- A pilot day with one friendly crew, and a real debrief afterward, before the full rollout.

The apps are the visible part of a field operation, but the operations that succeed are the ones designed backward from the data: what the layer must contain, what the dashboard must show, and only then which app puts the right questions in front of the person standing outside who knows the answer. Chapter 37 walks this entire arc as a worked project, from empty layer to closed loop.