Getting Started with ServiceProof

A walkthrough for the first system you'll build. About a 20-minute read; expect a quiet afternoon to ship the result.

This guide is for the **builder** — the person sitting down at the Studio for the first time with a real system to model. Not the integrator wiring up an API key (that's Integrators/Getting-Started.pdf, about a 5-minute read). Not the developer setting up the platform itself. The **builder** — the tenant admin, ops manager, or solutions architect who has a problem to solve and wants to know what gets pressed in what order.

We're going to build a Purchase Order system together as the worked example. By the end you'll know how to model a domain, where to start, when to import vs. build fresh, how to wire the layers, and how it all connects to the workflows that ultimately do the work in the field.

First — what is this thing you're sitting in front of?

ServiceProof is an **operations orchestration platform**. Not a CRM. Not a system of record. We're deliberately **not the source of truth** for your data — and that posture shapes everything.

Your existing systems (ServiceTitan, your ERP, your CMMS) keep being the truth. Data flows in from them, gets enriched by what humans do in our forms and what workflows in the field collect, and flows back out via webhooks / ServiceBus when our entities dirty. We orchestrate the work that produces it. We don't replace your systems — we operate **on top of them** to give your team the tools they actually need.

Built on **eleven interlocking pillars** (each API-first, multilingual, real-time, and agentic — those are traits of the whole platform, not pillars of their own):

#	Pillar	One-line
1	**Properties**	Typed fields with rules (RegEx, Min/Max, sensitive flags)
2	**Entities**	Composed of properties + parent/child/peer relationships
3	**Context**	**The logic layer — the point.** The Context Engine hydrates an anchor record's related data (parents, children, lookups, scoped reports) into one bundle every other pillar reads with `[[ ]]` tokens. How keyed/related objects come together.
4	**Views**	Live filtered datasets — recall, share, query as-of
5	**Reports**	Cross-entity joins + aggregations + temporal/variance
6	**Forms**	Desktop/tablet/mobile UIs **and** write endpoints for integrators
7	**Menu Management**	Discoverable hierarchy of forms / views / reports / **surfaces** + the agent's tool dictionary
8	**Surfaces**	Tenant-authored multi-canvas pages — Portal + Mobile + RCS — that compose forms / reports / views into branded dashboards, hubs, and detail pages from one `LayoutJson`. Author once, render everywhere.

#	Pillar	One-line
9	**Action Packs**	Tenant-installable, trigger-driven apps + the Action Center. Run on 10 trigger types (OnDemand / Timer / Webhook live; Messaging the payoff); a hybrid coded/recipe executor; every run logged for review / test / simulate. Messaging packs deliver conversational field workflows.
10	**Messaging**	The channel-agnostic delivery layer — WhatsApp (live) / RCS / AMB (scaffolded) / SMS via Infobip. Firehose webhook + tenant-resolution ladder + per-tenant event log + the rich `NeutralSend` render layer + the channel-aware `MENU` command.
11	**Workflows**	**The capstone — the mother of all, the reason the other ten exist.** Authored once, delivered to the field three ways: via **Surfaces** (8), the **Mobile PWA**, and **Rich Messaging** (10). Channel-agnostic runner; handset-first, web/mobile backstop.

> The Surfaces and Menu rows are presented in build-journey order here (you author surfaces, then pin them to the menu); the canonical pillar numbering in `Developers/Platform-Explained.pdf` lists Menu Management as #7 and Surfaces as #8.

Workflows (Pillar 11) are the capstone — the field experiences technicians, dispatchers, and supervisors run to do their jobs. The other ten pillars exist to serve them, and a workflow reaches the field three ways: rendered as a **Surface** (8), in the **Mobile PWA**, and as a conversation over **Rich Messaging** (10 — RBM / Apple Messaging for Business / WhatsApp / SMS). Steps read/write the same entities your Forms use and resolve `[[ ]]` tokens against the **Context** bundle (3).

Now — let's build something.

The example we're building: a Purchase Order system

We're using POs because they exercise almost every layer of the platform — multi-section forms with parent + line-items, conditional visibility, inline edit, file attachments, integrations to systems-of-record, and workflows in the field. But the same shapes show up across many verticals:

- **HVAC dispatch** — `WorkOrder` ⇄ `WorkOrderVisit` (one work order, many visits) ⇄ `Equipment` (the units serviced) + `Inspection` (per-equipment checklist results) + `Photo` (binary). The dispatcher's portal form lets the office reassign a tech, the field workflow runs the inspection on each unit, the report rolls up "PMs completed this month by tech."
- **Asset-based courier and final-mile delivery** — `Job` (parent, with stop type: bank-bag / medical / legal / standard / LTL / lift-gate) ⇄ `Stop` (many) ⇄ `Package` ⇄ `CustodyEvent` (sealed-bag verified, sender signed, recipient signed, departed). The driver's RCS / mobile workflow branches by stop type — sealed deposit gets seal-number capture and chain-of-custody photos; medical specimen gets temperature log prompts at pickup, in-transit, delivery; LTL pallet gets 4+ angle photos and BOL capture. The dispatch board form shows in-flight runs; the per-customer RCS thread shows the bank compliance officer or hospital lab manager exactly what happened, when, where, and who. **Detention time becomes billable** because it's photo-documented and geofence-validated.
- **Commercial truck service center** — `WorkOrder` ⇄ `Phase` (a list section of dwell-time taps: started / paused-for-parts / resumed / road-test / complete) ⇄ `MobileFleetVisit` (a different child entity for roadside calls) ⇄ `WalkAroundPhoto` ⇄ `Estimate`. Inside the bay, the dwell-time-tap workflow

produces **per-phase metrics no DMS captures** — parts-wait time, customer-approval turnaround, first-time-fix rate. Outside the bay, Mobile Fleet Service techs at 11 PM at a truck stop run the same workflow shape. **Overflow techs and surge contractors get the link in a text** — no install, same data quality.

- **Healthcare wellness checks** — `Patient` ⇄ `Visit` ⇄ `Vital` (a list section of repeating measurements). HIPAA-grade sensitive flags on patient binaries; conditional visibility hides fields based on triage answers.
- **Insurance claims intake** — `Claim` ⇄ `IncidentDetail` ⇄ `Photo` ⇄ `WitnessStatement`. The agent's mobile workflow walks them through the scene; the office form lets the adjuster review and route.

Every one of those is the same five-or-so entities + five-or-so forms + a workflow + a menu. **The shape we're about to build for POs transfers.** Once you've done it once, the next vertical is muscle memory. And the unifying truth across all of them: **the workflow is the measurement instrument**. The taps your field user makes ARE the data. No system needs to be read from. Nothing needs to be integrated.

Here's the real-world problem we're tackling for our example:

> Field technicians at an HVAC company need to drive to Grainger or Hajoca or Rexel for parts during a job. They open a workflow on their phone, scope it to the customer + service location + equipment they're servicing, list every part needed and why, and the platform issues them a PO number. At the counter, they snap a photo of the receipt and walk through each line item marking it received, short, or substituted. The data flows back. The office sees the PO appear instantly, edits notes, marks line items billable, and the changes push back out to ServiceTitan so the invoice can be cut.

That's the round-trip we're modeling. Every choice we make is in service of it.

Stop and think before you build

The single best thing you can do at the Studio is sit for ten minutes BEFORE clicking anything and answer four questions:

1. **What are the things?** (Entities — Customer, ServiceLocation, Equipment, PurchaseOrder, PurchaseOrderLineItem)
2. **What do you need to know about each?** (Properties — PoNumber, Vendor, Status, Quantity, UnitPrice, ReceivedStatus...)
3. **How are the things related?** (Composition — a PO belongs to a Customer + a ServiceLocation + optionally an Equipment item; a PO has many line items)
4. **Who will see this and how?** (Forms — dispatcher's PO board, field workflow, vendor-billable summary, integration write surface)

You don't have to be perfect. You can extend, rename, translate, and reshape every layer later. But if you start clicking before you've thought, you'll spend your afternoon refactoring instead of building.

For our PO system the answers look like this:

```
Customer ──┬── ServiceLocation ──┬── Equipment
           │                     │
           │                     └── PurchaseOrder ─── PurchaseOrderLineItem
(many)
```

```
            │                                          (also linked to Equipment when
scoped)
            └── (optionally — PO without specific equipment, scoped only to the
site)

Properties on PurchaseOrder:
  PoNumber (text, key, required, RegEx for vendor format)
  Vendor (choice — Grainger / Hajoca / Rexel / Other)
  Status (choice — Draft / Issued / Picked Up / Closed)
  TotalEstimated (currency)
  TotalActual (currency)
  Notes (text, multi-line)
  ReceiptImage (binary, sensitive)
  IssuedDate (datetime)
  PickedUpDate (datetime)

Properties on PurchaseOrderLineItem:
  LineNumber (number, key)
  PartNumber (text, required)
  Description (text)
  Quantity (number)
  UnitPrice (currency)
  ExtendedPrice (number, computed downstream)
  ReceivedStatus (choice — Received / Short / Substituted / Backordered)
  Notes (text)
```

That's the shape. Now we click.

Step 1 — Entities (start here, always)

Entities are the **objects** you work with. They're tables in the database, but more importantly they're the unit of meaning — a Customer is a thing, a PurchaseOrder is a thing.

You have three ways to add an entity:

Path A — Import from the global catalog

The platform ships a curated library of pre-built entities for common verticals (Field Service, Healthcare, Logistics, etc.). For a PO system you almost certainly want the existing **Customer** + **ServiceLocation** + **Equipment** entities — they already have the right shape, the right composite keys, the right relationships, and they're translated into every language we support.

Import them. You can choose:

- **Linked import** — your tenant tracks the global definition. You can ADD properties locally but you can't remove or rename the linked ones. When the platform updates the global Customer, your tenant gets the update. Best for "I want the standard, I'll just extend a bit."
- **Independent copy** — you take a snapshot at a point in time and own it forever. Full freedom to restructure. Best for "I need this to be mine, I'll diverge."

Path B — Build fresh

For things specific to your business (PurchaseOrder, PurchaseOrderLineItem in our example), build them from scratch. Studio → Entities → New. Give it an InternalName (`PurchaseOrder`, `PurchaseOrderLineItem`), a display Name in your default language, and you're off.

Path C — Already have data? Import a CSV

If you already have a list of Customers in a spreadsheet, you can import the data and let the importer **define the entity AND the properties from the CSV columns** in one shot. Studio → Data Studio → Import. The wizard auto-detects column types (text, number, date, currency, choice when it sees repeated values), lets you tweak before commit, and lands the rows directly. Now you have an entity, properties, AND data in five minutes.

This is one of the platform's most underused tricks. If you have an Excel sheet sitting on your desk RIGHT NOW with the data you care about, the fastest path to a working system is "import it" — extend afterward.

Composition — wire parent / child / peer relationships

Once your entities exist, wire them up. PurchaseOrder belongs-to Customer, belongs-to ServiceLocation, optionally belongs-to Equipment. PurchaseOrderLineItem belongs-to PurchaseOrder. Studio's entity editor has a relationships pane — pick the parent entity, pick how the keys map. The platform handles composite-key inheritance automatically (the line-item entity carries the parent's PoNumber + Customer/Location identity in its key chain so you can never accidentally orphan a line item).

Step 2 — Properties (rules, types, translations)

Every property has a **DataType** — text, number, currency, datetime, date, time, choice, attachment, signature, hierarchy, binary. Pick the right one and the platform handles formatting, validation, storage, language fallback, and rendering on every surface for free.

For each property, you can set:

- **IsRequired** — must have a value
- **IsKey + KeyPosition** — part of the composite natural key (1, 2, or 3)
- **Default value** — pre-filled when adding new
- **RegEx** — a text pattern (`^PO-\d{6}$` for our PoNumber maybe)
- **Min/Max value or length** — numeric range or text length
- **MaxDecimalPlaces** — for currency / number
- **AllowedFileTypes** — for attachments (`image/jpeg,image/png` for ReceiptImage)
- **IsSensitive** — sensitive data segregation; sensitive binaries route through gated proxy URLs
- **Choice options** — for choice fields, the canonical PropertyOption set with localized labels
- **Multi-language display** — Name / Description / placeholder / help text per language

If you imported entities from the global catalog and they came with properties you want to evolve — you can rename them in your tenant's language, add translations, change choice option labels, mark them required where they weren't, add validation rules. Go crazy. Make it yours.

A property the platform knows about can light up everywhere — a Choice property's options become a dropdown automatically; a DateTime property gets a date picker; a Currency property renders with the $ prefix; a Sensitive Binary routes through proxy URLs in any form that surfaces it. **Pick the data type once, get the right behavior across every layer for free.**

Step 3 — Views (live filtered datasets)

Now that you have entities and data, you want to look at slices of it. Open POs. POs from Grainger this month. Line items short on receipt. POs over $1000.

A **View** is a saved filter + column selection over an entity. You build it in the Data Browser — pick filters, columns, sort, grouping — then save it with a name. From now on:

- It's a live dataset, kept fresh as data flows in
- It's a **dropdown source for forms** — pick "PO Vendor" in a form and choose "Grainger" from the dropdown, where the dropdown is a view of your Vendor entity
- It's a **lookup**, e.g., "active customers in Connecticut" for a form's Customer picker
- It's an **API endpoint** — query the view from any integration, get the same shape humans see
- It's a **menu link target** — pin it to your menu so the team can find it
- It's the **basis for reports + forms** — pick a view and build on top

Views also support **as-of querying** — show me what this view looked like last Tuesday. Built-in, no extra setup.

For the PO system, build a few views to start:

- "Open POs" — Status in (Draft, Issued)
- "POs Awaiting Pickup" — Status = Issued
- "POs by Vendor" — grouped by Vendor
- "This Week's Receipts" — Status = Picked Up, PickedUpDate within the last 7 days
- "PO Vendors lookup" — the Vendor entity (or a view of it), used to power the Vendor dropdown on the PO form

---

Step 4 — Reports (the cross-entity shape-maker)

Reports are where ServiceProof flexes. A view shows you slices of one entity. A **report** walks across entities — joins parent / child / peer — and slices, aggregates, filters, and time-travels across the whole graph.

For the PO system, you want reports like:

- **PO spend by Customer** — join PurchaseOrder to Customer, sum TotalActual grouped by Customer, filtered to last quarter
- **Parts received per technician per week** — workflows write the technician, you join PurchaseOrderLineItem to PurchaseOrder to WorkOrderVisit
- **Variance: spend by vendor, weekly** — temporal mode, weekly cadence, see how spend at Grainger trends month-over-month

Reports support:

- **Cross-entity joins** — build a graph, query across it
- **Conjunctive filters** — AND/OR rule trees with deep relationship filters ("customers whose POs in 2026 had at least one line short on receipt")
- **Aggregations** — SUM / COUNT / AVG / MIN / MAX
- **Grouping** — multiple levels
- **Temporal / variance mode** — snapshot the whole report at a cadence (daily / weekly / monthly) and see what changed when. This is the killer feature for trends and sync-back honesty.

Same proc pipeline that powers the Report Builder UI in the portal also powers the Reports REST API. Same primitives, same output shape — humans and integrators read the same data.

Step 5 — Context (bundle the related records — Pillar 3, the point)

You've got entities, the relationships between them, views, and reports. **Context is where you turn that pile of related rows into one coherent thing your forms, workflows, surfaces, emails, and AI agents can all read the same way.**

This is Pillar 3 — and we mean it when we call it *the point of the platform*. Everything else stores or displays data; Context is the layer where business **meaning** lives. You author a **ContextDefinition** once: pick an anchor record (a Purchase Order), then declare what to reach out and grab around it (the Customer it belongs to, the ServiceLocation, the Equipment at that location, its line items). The engine **hydrates** that definition against a specific PO and hands back one JSON **bundle**. Every consumer reads it with `[[ ]]` tokens — `[[Customer.OrgName]]`, `[[ServiceLocation.AddressLine1]]`, `[[Lines[0].PartNumber]]` — and they all resolve identically no matter how your tenant's data is shaped.

The mental model: anchor, then reach

Start from one **anchor**, then reach out. Some things hang off the anchor directly; some hang off *another row you already reached*. That two-level reach is the whole idea:

```
PurchaseOrder (anchor — by PoNumber)
├── Customer            reaches from → the PO               (compositeKey via
[[Po.CustomerNumber]])
├── ServiceLocation     reaches from → the PO               (compositeKey via
[[Po.CustomerNumber]] + [[Po.LocationNumber]])
├── Equipment           reaches from → the ServiceLocation   (child, scope:
[[ServiceLocation]])
├── Lines               reaches from → the PO               (child — the
PurchaseOrderLineItem rows)
└── Vendor              reaches from → the PO               (compositeKey via
[[Po.Vendor]] — the lookup)
```

`Equipment` is the one to notice: it doesn't hang off the Purchase Order, it hangs off the **ServiceLocation** you already pulled in (the equipment *at that site*). In a binding that's `scope: "[[ServiceLocation]]"`. Same shape as "the menu item belongs to the line item, not the order" — once you see it, every domain is the same map.

What the definition looks like

It's one `bindingsJson` tree (the Portal writes this for you — you rarely type it):

```
{
  "anchor": { "name": "Po", "entityRole": "PurchaseOrder", "kind": "rowByKey",
              "identityFields": [ { "field": "PoNumber", "keyPosition": 1 } ] },
  "bindings": [
    { "name": "Customer",        "source": "compositeKey", "entityRole":
"Customer",
      "via": { "key1": "[[Po.CustomerNumber]]" } },
    { "name": "ServiceLocation", "source": "compositeKey", "entityRole":
"ServiceLocation",
      "via": { "key1": "[[Po.CustomerNumber]]", "key2": "[[Po.LocationNumber]]" }
},
    { "name": "Equipment", "source": "child", "entityRole": "Equipment",
      "expects": "many", "scope": "[[ServiceLocation]]" },
    { "name": "Lines", "source": "child", "entityRole": "PurchaseOrderLineItem",
      "expects": "many" }
  ]
}
```

Hydrate it for one PO — `{ "anchorIdentity": { "PoNumber": "PO-000123" } }` — and you get back a bundle carrying `Customer`, `ServiceLocation`, the `Equipment` list, and every `Lines` row, in one round-trip. That bundle is what the receipt email, the dispatcher's surface, and the field workflow all read.

Build it visually — `/context-definitions`

You almost never hand-write that JSON. The Portal builder (Define → Context Definitions, BuilderAdmin) draws the context as a map and writes the JSON for you. Three tabs:

- **Build** — the canvas. An **anchor card** (pick PurchaseOrder + its key), **binding cards grouped by what they reach from** (so Equipment visibly sits under "Reaches from: ServiceLocation"), and **Suggested relationships** chips drawn from your entity graph — one click adds *"the Customer this belongs to"* or *"the line items of this PO."* If a binding can't resolve because two entities aren't linked yet, the card says **"won't resolve yet"** and offers to author the relationship right there, the Data-Studio way. An **entity previewer** lets you peek inside any target and see what a key actually looks like (`CUST-001`, `PO-000123`).
- **Test** — cold-start friendly: it shows you exactly which key it needs, **Find a record** lets you browse your real POs and click one, and it runs immediately — every binding reports ✓ N rows / ⚠ empty / ✗ error, with the bundle JSON and a `[[ ]]` token tester.
- **JSON** — the raw tree with Apply + parse errors, for hand-editing or pasting.

Prefer code (CI, bulk provisioning, an MCP agent)? `POST /api/v1/context` does everything the builder does — see **Integrators/Context-API.pdf**.

Why you build this *before* forms, workflows, and surfaces

Because they all consume it. The dispatcher's **Surface** (Step 8) binds its sections to slots in this bundle. The field **workflow** (Step 7) resolves `[[Customer.OrgName]]` and `[[ServiceLocation.AddressLine1]]` against it so the tech sees who and where without re-keying anything. The receipt **email / RBM thread** drops `[[Po.PoNumber]]` and `[[Customer.OrgName]]` into a template. **Author the context once; every consumer downstream reads the same shape.**

> Wiring synced people (Employees, Techs, Customers) into real platform users — so `[[AssignedTech.Email]]` resolves to someone you can actually text or email — is the **Identity Bridge** side of Context. If your POs get assigned to technicians you sync from an HRIS, read **Integrators/Context-Engine-Guide.pdf** for the flag-and-invite setup.

Step 6 — Forms (the most user-facing layer, for humans AND agents)

This is where it gets serious.

Forms are the **interfaces humans use** to retrieve, view, edit, and add live data — desktop, tablet, mobile. They're also the **write endpoint integrators and AI agents POST against**. One contract, two audiences.

For the PO system we'll build at least three forms:

Form A — `PurchaseOrderForm` (the office-side multi-section form)

The dispatcher's main view. A multi-section form with:

- **Primary section: PurchaseOrder** — fields like PoNumber, Vendor, Status, TotalEstimated, TotalActual, Notes, ReceiptImage. Layouts the dispatcher sees.
- **Detail section: Customer info** — read-only summary of the linked customer (drawn from the Customer entity via the report binding).
- **Detail section: ServiceLocation info** — same.
- **List section: Line Items** — `PurchaseOrderLineItem` as a list section, showing PartNumber + Description + Quantity + UnitPrice + ReceivedStatus. **Inline edit mode** so the dispatcher can flip ReceivedStatus across rows fast — operational throughput. Allow-add for adding a line item; allow-delete (soft) so a wrong row can be undone before save.
- **Conditional visibility** — show the `ReceiptImage` field only when Status = "Picked Up"; show the `Substitution Notes` field only when ReceivedStatus = "Substituted". The form responds to data as the user types.

The same form, **server-validated**, is also the API write endpoint. An integrator POSTing to `/api/v1/forms/PurchaseOrderForm/records` gets identical validation, locked defaults, composite-key inheritance, conditional-visibility-aware required checks. **You don't have to build two systems.**

Form B — `PoLineItemQuickEditForm`

A focused form just for Line Item edits — used by mobile users who only need to update line-item statuses. Filtered + locked to the line items of a single PO via filter-as-lock so the user can't accidentally edit somebody else's PO.

Form C — `OpenPosBoardForm`

A list-only form with no primary record — bound to the "Open POs" view. A board view, all the open POs at a glance, click one to open the detail form.

How forms pair with workflows in the field

The same shape that powers the Portal form also drives the field-side workflow steps. The technician's RBM / WhatsApp / mobile-web workflow reuses the entity properties, validation rules, choice options, and (where relevant) layouts the form designer authored. Build once, run everywhere.

Step 7 — Workflows (where the field meets the data)

Workflows are the field-side companion to your forms. They're built in a separate Workflow Designer but they **read from and write to the same entities**. Steps in a workflow attach to property values: ask the user a question, capture an answer, validate it, branch on it, loop, attach a photo, take a signature.

For the PO system, the field-side workflow looks like this:

1. **Scope step** — pick Customer, ServiceLocation, optionally Equipment (using lookup views populated from your data)
2. **Vendor step** — pick Vendor (Choice property's options drive the question's answers)
3. **Line items loop** — repeating step where the tech adds parts. Each iteration writes a `PurchaseOrderLineItem` row, branching back to allow another or finish.
4. **Issue PO step** — workflow assigns a PoNumber, sets Status = Issued, sends a notification to the dispatcher
5. **(Tech drives to vendor)**
6. **Counter receipt step** — capture photo of the printed receipt (binary attachment)
7. **Receive each item loop** — for each line item, prompt: Received / Short / Substituted / Backordered, optionally add notes. Updates `PurchaseOrderLineItem.ReceivedStatus`.
8. **Finalize step** — sets Status = Picked Up, PickedUpDate = now, sends notification to dispatcher

Every step writes to the entities your forms use. The dispatcher's portal form sees changes live. The system-of-record gets sync-back via webhook when the PO closes.

This is **why the data layer matters** — the workflow doesn't carry its own data model; it operates on the entities you defined. Build the entities right and the workflow inherits the shape.

How the workflow reaches the tech — three delivery paths + Conductor4. Workflows (Pillar 11) are the capstone: authored once, delivered to the field three ways. The runner engine is pure and channel-agnostic, and the PRIMARY target is **rich messaging on the handset** — WhatsApp is live; RCS and Apple Messaging for Business (AMB) are scaffolded; SMS — all via Infobip. When a step can't render in chat (a signature, a date picker, a multi-property entity form), the thread sends a one-tap link, the tech finishes that step on the **Mobile PWA / web runner** backstop, and the run resumes in the thread on the SAME instance — the **hybrid hand-off**. (Surfaces is the third path.) Design messaging-first: picture this PO workflow as a turn-by-turn conversation before a web form. And the way a published workflow actually *reaches a person* is **Conductor4 (Jobs)**: the published workflow becomes a **Workflow Kit** (include/qualify filters + flow gates + on-complete hooks), kits compose into a **Job Set**, the set is **dispatched** against an anchor record (the customer + service location) and **assigned** to a tech, and the tech runs the result — a tracked **Job** — from their worklist (My Jobs). The runner executes; Conductor4 assigns and tracks. See Integrators/Jobs-API.pdf.

Step 8 — Surfaces (the consumer page layer — Portal + Mobile + RCS, one author)

You've got entities, properties, context, views, reports, forms, and workflows in place. **Surfaces are how all of that shows up to your team and your customers** — without you writing a line of Razor or React.

A **Surface** is a tenant-authored multi-canvas page. You author one `LayoutJson` describing the sections that make up a page; a universal renderer brings it to life on **HD desktop (Portal)**, **mobile PWA**, and (soon) **RCS / AMB card carousels** — sharing the same authored content, the same data bindings, the same theme, the same multilingual strings.

This is Pillar 8 — the destination layer where every investment you made in entities, forms, views, and reports finally meets your users.

What's in a Surface

- **35 section kinds** across 5 categories:
  - **Display + info** — card / tile / tiles / table / kanban / timeline / pivot / gauge / chart / map / hero-banner / text / metrics-strip
  - **Embed + compose** — embed-form / embed-report / embed-view / embed-surface (yes, surfaces can compose other surfaces) / tabs
  - **Interactive + flow** — action-bar / breadcrumbs / picker-strip / shortcut-strip / search-picker
  - **Channel-aware** (the field-tech magic) — signature / camera / geo-checkin / scan (QR / barcode / VIN / asset tag) / voice
  - **Status + feedback** — status-card / error-card / success-card
- **Five anchor sources** — how the surface picks the row of data it renders against:
  - `currentUser` — "My jobs", "My time card" — no identity from URL
  - `route` — `/s/AcmeJobDetail?Key1=WO-5001` — per-row detail surfaces
  - `search` — picker first, then context loads — "Find a customer / asset / job"
  - `parameter` — declared `SurfaceParameter` values — time-travel dashboards
  - `none` — static or compute-only
- **Context Engine bundle bindings** — every section can bind to a slot in your tenant's hydrated `ContextDefinition` bundle (Customer + their Locations + their Equipment + their Open POs — all in one JSON, all live)
- **Theme tokens** — your tenant's brand colors flow through via `--theme-*` CSS variables on every canvas

Why this matters for the PO system

For our worked example you'd author **at least three surfaces**:

- `PoBoard` — a dispatcher's home Surface anchored on `currentUser`. Hero banner + metrics-strip ("Open POs: 14", "Awaiting Pickup: 6", "This Week's Spend: $8,420") + `tiles` bound to the "Open POs" view + `embed-report` showing PO Spend by Vendor. Lands at `/s/PoBoard` on Portal AND `/mobile/s/PoBoard` on the PWA. Same authored payload, two canvases.

- `PoDetail` — route-anchored on the PoNumber. Card section showing PO header, embed-form pointing at `PurchaseOrderForm` for inline edit, list of line items, and a `signature` + `camera` section for the receipt capture (field techs hit this on mobile; office staff hit the same surface on Portal and see the upload-from-file affordance instead).
- `VendorHub` — search-anchored ("Find a vendor") then renders the vendor's POs, spend trend chart, and outstanding receipts.

Each one wires to the data you've already built. **You're not modeling anything new** — you're composing existing pieces into the user-facing pages your team will live in every day.

How to author

Three paths:

1. **Portal SurfaceBuilder UX** — three-pane workbench (palette / canvas / inspector). Drag a section kind onto the canvas, bind it to a bundle slot, save.
2. **POST `/api/v1/surfaces`** — the typed REST API. Author from a script, from CI, from anything that can POST JSON.
3. **MCP / Copilot agent** — Wave 12 ships 9 surface-authoring tools so an AI agent can stand up a Customer 360 or Dispatch Board on your behalf from a natural-language prompt. The agent uses the same `/validate` envelope the Portal UI does — drafts get back `{ valid, issues[] }` before save, so the agent can self-correct.

The MCP path is the integrator multiplier. **An agent that can author surfaces is an agent that can build the team's daily-workflow UI** — the kind of leverage that turns "we need a dashboard" from a sprint into a five-minute conversation.

What to read next on Surfaces

- **Integrators/Surfaces-Guide.pdf** — the full 35-kind vocabulary, the 5 anchor sources, `LayoutJson` schema, the validation envelope, and a 10-recipe cookbook from real surfaces. Start with §12 (5-minute getting started) and §13 (cookbook).
- **Integrators/Surface-Builder-Guide.pdf** — Portal UX walkthrough for tenant admins authoring through the workbench.
- **Integrators/Context-API.pdf** — the Context Engine: how to author a `ContextDefinition` (visually at `/context-definitions` or over REST) and hydrate it into the bundle surfaces bind to. Read this BEFORE authoring a surface against an entity you've never bundled before.
- **Integrators/Context-Engine-Guide.pdf** — the identity-bridge + tenant-setup side: how synced Employee/Customer/Tech records become real platform users your tokens resolve against.

Step 9 — Menu Management (make it findable + agent-discoverable)

Now your team needs to actually find the stuff you built. The Menu Manager is where you organize forms / views / reports into a logical hierarchy:

```
Operations
├── PO Dashboard              → surface (PoBoard)        ← Pillar 8 page
├── Purchase Orders
```

```
│   ├── Open POs              → form (OpenPosBoardForm)
│   ├── Awaiting Pickup       → view (POs Awaiting Pickup)
│   ├── This Week's Receipts  → view (This Week's Receipts)
│   └── PO Spend Report       → report (PO spend by Customer)
├── Vendor Hub                → surface (VendorHub)      ← Pillar 8 page
├── Quick Edit
│   └── Line Item Status      → form (PoLineItemQuickEditForm)
└── ...
```

Menu items can target **forms, views, reports, OR surfaces** — Pillar 8 surfaces drop into the same hierarchy as everything else. The menu hierarchy is also the **agent's tool dictionary** — when an MCP agent connects to your tenant, the menu defines what tools it knows about and is allowed to use. You control what surfaces the agent can act on by curating the menu hierarchy.

Step 10 — Storage mapping (we store indexes, not your binaries — unless you want us to)

ServiceProof's binary storage is **provider-agnostic**. We support six providers out of the box (Azure Blob, S3, Google Drive, Local, FTP/SFTP, plus 8 stub providers ready to flesh out). The binary itself can live wherever you want it to live; we store the index, the metadata, and the access controls.

For the PO system's `ReceiptImage` property, you have choices:

- **Tenant default storage** — we store the image in our default storage source (per-tenant Azure Blob, say)
- **Customer-managed storage** — you point us at your S3 bucket; we write through to it
- **Existing storage source** — you already have receipts in a folder somewhere; we hook up the storage source and we read indexes against your data

That last one is the underrated move. **You may already have years of receipt images in a folder.** Hook up that storage source, point the entity's binary property at it, and now your PO system has full historical context without copying anything.

Storage providers cascade and append: each tenant has its own credentials chain, sensitive binaries route through gated proxy URLs (the URL never reveals the underlying storage), credentials are encrypted at rest in `Tenant.SettingsJSON`. See STORAGE.pdf for the full architecture.

You're done. Now iterate.

That's the journey. Properties → entities → relationships → views → reports → **context** → forms → workflows → **surfaces** → menu → storage. Ten steps, each one earning the ones above it.

Reality: you won't do them in this exact order. You'll bounce. You'll import a CSV first and discover halfway through that you need a child entity you didn't think of. You'll build a form and realize a property needs a RegEx rule. You'll add a column to a view and realize it would make a great report. **That's normal.** The platform is built so every layer is editable and the others adapt automatically. Add a property, every form binding sees it. Change a view's filter, every dropdown using it stays current. Edit a form's layout, every API client gets the new validation rules.

What to read next

If you're a...	Read
Integrator wiring an API key	Integrators/Getting-Started.pdf
Builder who just wants to model schema	Integrators/Entities-API.pdf
Builder who has data already	Integrators/Entity-Data-API.pdf (also: try the Studio's CSV importer)
Builder making operational dashboards	Integrators/Entity-Reports-API.pdf
Builder making forms	Integrators/Forms-API.pdf
Builder authoring multi-canvas pages (Pillar 8)	**Integrators/Surfaces-Guide.pdf** — 35 kinds, 5 anchor sources, full `LayoutJson` schema + 10-recipe cookbook. Also Surface-Builder-Guide.pdf for the Portal authoring UX.
Builder bringing related records into one bundle (Context Engine)	**Integrators/Context-API.pdf** — author a context visually at `/context-definitions` or over REST, then hydrate it; the bundle every surface / workflow / email reads with `[[ ]]` tokens.
Builder delivering workflows to the handset (Pillars 9 + 10)	**Integrators/Action-Packs-API.pdf** (trigger-driven apps + Action Center) + **Integrators/Messaging-API.pdf** / Conversational-Messaging.pdf (send → await → poll → end over WhatsApp).
Builder dispatching + assigning workflows as tracked jobs (Conductor4)	**Integrators/Jobs-API.pdf** — Workflow Kit → Job Set → dispatch + assign → worklist. The runner executes; Conductor4 assigns + tracks.
Builder delivering workflows to the handset (the runner + the loop)	**Integrators/Conversational-Messaging.pdf** + **Integrators/Messaging-API.pdf** — rich messaging delivery (WhatsApp live; RCS/AMB scaffolded), the send → await → poll → end loop.
Builder on the Mobile PWA runner (the backstop + hybrid hand-off)	**Integrators/MOBILE.pdf** — the `/mobile/*` PWA runner that catches steps chat can't render, then resumes the run in the thread.
Developer onboarding to the platform	Developers/Architecture.pdf, Developers/Platform-Explained.pdf

The core philosophical reminder

ServiceProof is **not the source of truth**. We orchestrate the work that produces the truth. Your data flows in from where it lives, gets enriched and validated by the eleven pillars — the foundation layers above (Properties, Entities, **Context**, Views, Reports, Forms, Menus) plus **Surfaces** (Pillar 8, the multi-canvas page layer you just built in Step 8), **Action Packs** (Pillar 9, tenant-installable trigger-driven apps + the Action Center — see `docs/claude/ACTION-PACKS.pdf`), and **Messaging** (Pillar 10, the channel-agnostic WhatsApp / RCS / AMB / SMS delivery layer — see `docs/claude/MESSAGING.pdf`) — all serving **Workflows** (Pillar 11, the capstone, delivered via Surfaces / Mobile / Rich Messaging) — and flows back out. The point is the workflows

in the field, and the office-side companions that support them. Action Packs and Messaging are how those workflows reach the handset.

Go build something. We'll be here.