Liferay Objects Deck

Must-Know Concepts, Explained

Coverage	Core Concepts, Fields & Data Modeling, Business Logic & Automation, UI Integration, APIs & Integration, Scoping & Permissions, Advanced Relationships, Deployment & Lifecycle
Format	Question → Concise answer → comparison / example where useful
Level	Liferay Business/Low-Code Developer to Solution Architect rounds

---

Reflects Liferay DXP 7.4+ / current Liferay DXP Cloud (quarterly release) Objects.
Liferay ships new Objects capabilities every quarter – always confirm feature availability against the release notes for your specific DXP version.

Generated September 18, 2026

1 Core Concepts 2

2 Fields & Data Modeling 4

3 Business Logic & Automation 6

4 UI Integration 8

5 APIs & Integration 10

6 Scoping & Permissions 12

7 Advanced Relationships & Features 14

8 Deployment & Lifecycle 16

Core Concepts

Q1.1

What is Liferay Objects, and what problem does it solve compared to building a custom data model as a traditional OSGi module?

Liferay Objects is a low-code framework, built directly into the Liferay UI, for defining custom business entities (”objects”), their fields, relationships, and behavior – entirely through point-and-click configuration, with no Java code or module deployment required. Before Objects, modeling a new business entity (a ”Ticket,” an ”Insurance Claim,” an ”Equipment Asset”) meant writing a full OSGi service module – Java service builder entities, generated persistence classes, REST endpoints, and a redeploy for every schema change. Objects replaces that entire development cycle with configuration: an administrator or business user defines the object’s fields and relationships in the UI, and Liferay automatically provisions the underlying database table, generates default REST/GraphQL APIs, and builds the admin UI for managing entries – collapsing what used to be a multi-day developer task into a task business analysts can often complete themselves.

Q1.2

What is the difference between a `system object` and a `custom object`, and why does this distinction matter for what you’re allowed to configure?

A **system object** is one of Liferay’s own built-in business entities – Users, Sites, Documents, Blog Entries, and dozens of other objects the platform itself is built on – automatically registered in the Objects framework so they can be extended and integrated with the same tools available to custom objects. A **custom object** is one you define yourself from scratch, with full control over its fields, relationships, and lifecycle. The distinction matters because system objects have restrictions custom objects don’t: you generally cannot delete a system object, rename its existing fields, or change its core structure, since other parts of the platform depend on it – but you *can* extend a system object by adding new custom fields to it, letting you attach business-specific data (e.g. a custom ”Loyalty Tier” field on the User object) without touching Liferay’s own core code.

Q1.3

What is an object definition, and how does it differ from an object entry?

An **object definition** is the schema – the blueprint describing an object’s fields, their types, its relationships to other objects, and its configured behaviors (actions, validation) – created and edited by an administrator in the Objects management UI. An **object entry** is one actual row of data conforming to that schema – a single ”Ticket” record, a single ”Claim” record – created by an end user filling out a form or via the API. This is the same fundamental class-versus-instance distinction found in any data modeling system: you design the object definition once, then end users (or integrations) create, read, update, and delete many individual object entries against that definition over time.

Q1.4

What does it mean to ”publish” an object definition, and why can’t an object be used anywhere in Liferay until this step is completed?

Creating and editing an object definition happens in a **draft** state – fields can be freely added, removed, or reconfigured without any consequence, since no real data exists against it yet. **Publishing** the object definition locks in that schema, provisions the actual underlying database table, generates the object’s default REST and GraphQL APIs, and activates the object as a real, usable application within Liferay – only after publishing does the object appear in the relevant admin panels, become available to reference from display pages and form containers, and start accepting real entries. This draft-then-publish workflow exists specifically to prevent an incomplete or half-configured object from being

exposed to end users or external integrations before it's actually ready.

Q1.5

What are the main places within the Liferay UI where an end user actually interacts with object entries, beyond the dedicated Objects admin panel itself?

Once published, an object's entries can be surfaced through several different UI mechanisms depending on the use case: the object's own dedicated **admin panel** entry (a default, auto-generated CRUD UI for managing entries); **form containers** (a page-builder widget that renders a form bound to an object, letting site visitors create or edit entries directly from a page); **display pages** (rendering a single object entry's details, using the same display-page-template mechanism used for web content); and **data sets** (a page-builder widget that renders a filterable, sortable table or list of an object's entries) – together, these let an object's data be woven directly into ordinary Liferay pages without writing any custom frontend code at all, on top of the object's own generated headless APIs for any deeper, custom integration.

Fields & Data Modeling

Q2.1

What are the main categories of field types available when defining an object, and how do you decide between a plain `Text` field and a `Picklist` field for a value with a limited set of options?

Object fields span the common data types you'd expect – Text, Rich Text, Textarea, Integer, Decimal, Boolean, Date, Date & Time, Attachment, and relationship fields (covered below) – plus **Picklist**, which constrains a field's value to a predefined, administrator-managed list of options (e.g. "Open / In Progress / Resolved / Closed" for a ticket's status). The choice between a plain Text field and a Picklist matters for data quality and downstream usability: a Text field lets a user type anything, risking inconsistent values ("open," "Open," "OPEN" all being treated as different strings), while a Picklist guarantees every entry's value is drawn from one controlled, centrally-managed vocabulary – essential whenever a field's value will be used for filtering, reporting, workflow branching, or any other logic that depends on consistent, predictable values.

Q2.2

What is a picklist in Liferay Objects, and why is defining it as a separate, reusable entity – rather than hardcoding the options directly on the field – valuable when the same list of options is needed across multiple objects?

A picklist is a named, independently managed list of key-value options (e.g. a "Priority" picklist with Low/Medium/High/Critical) that can be attached to a Picklist-type field on any object definition. Defining picklists as their own separate, reusable entities – rather than hardcoding a fixed set of options directly onto one specific field – means the exact same controlled vocabulary can be reused across several different objects (e.g. the same "Priority" picklist used on both a "Ticket" object and a "Change Request" object), and updating the picklist's options in one place (adding a new value, relabeling an existing one) automatically propagates to every object that references it, rather than requiring the same edit to be repeated independently on every object that happens to use a similar list.

Q2.3

What are the different relationship types available between objects (`one-to-many`, `many-to-many`), and how do you decide which one models a given real-world relationship correctly?

A **one-to-many** relationship (e.g. one "Customer" has many "Orders") models a relationship where each related entry on the "many" side belongs to exactly one entry on the "one" side, while each "one"-side entry can be associated with any number of "many"-side entries. A **many-to-many** relationship (e.g. "Students" and "Courses," where a student can enroll in several courses and a course has many students) models a relationship with no such single-owner constraint on either side. Choosing correctly matters because it directly determines the underlying data structure Liferay provisions – a one-to-many relationship stores a simple foreign-key-style reference, while a many-to-many relationship requires an implicit join structure – and choosing the wrong type either incorrectly restricts a genuinely many-to-many relationship down to one-to-many, or adds unnecessary complexity to what was actually a simpler one-to-many relationship.

Q2.4

What is a Root Object, and what specific management problem does explicitly configuring one object as the "root" of a set of related objects solve?

When several objects are linked together through relationships (e.g. an "Order" object related to "Order Line Item"

and ”Shipment” objects), a **Root Object** configuration designates one of those objects as the primary, top-level entry point for managing the whole related group – simplifying navigation and administration by letting related child entries be created, viewed, and managed directly from within the root object’s own entry view, rather than requiring an administrator to separately navigate to each related object’s own admin panel and manually cross-reference which entries belong together. This is particularly valuable for object groups that are conceptually always used and thought about together, letting the admin UI reflect that natural parent-child grouping rather than presenting several disconnected, independently-managed object panels.

Q2.5

What is a nested field (via the `nestedFields` query parameter), and how does it let a single API call retrieve a parent object entry together with its related child entries in one response, rather than requiring separate follow-up calls?

By default, fetching an object entry through its REST API returns just that entry’s own fields, with related entries only accessible via a separate call to the relationship’s own dedicated endpoint. Adding the `nestedFields` query parameter to a GET request tells the API to also embed a specified related object’s entries directly within the parent entry’s response – letting a client retrieve a parent entry together with its associated child entries (e.g. an ”Order” together with all of its ”Order Line Item” entries) in a single round trip, rather than first fetching the order and then making a separate follow-up call to the line-items relationship endpoint. This is a standard, valuable performance optimization for any client application that routinely needs a parent entry and its related data together, avoiding the network overhead (and added latency) of multiple sequential API calls for what is conceptually one logical piece of data.

Business Logic & Automation

Q3.1

What is an object action, and how does it let you attach automated business logic to specific events in an object entry's lifecycle without writing a custom OSGi listener module?

An object action is a configurable rule, attached to an object definition, that automatically executes when a specified **trigger** occurs – an entry being created, updated, deleted, or a scheduled/on-demand trigger – optionally filtered by a condition (e.g. only when a "Status" field's value changes to "Approved"). The action itself can send a notification, update a related field, call a webhook, or (for more complex logic) invoke a custom-defined Object Action script. This gives administrators and business analysts a direct, UI-configurable way to wire up common automation patterns – "notify the assigned agent when a new ticket is created," "update the parent order's total when a line item changes" – entirely through configuration, reserving custom code specifically for logic genuinely too complex or specialized for the built-in action triggers and conditions to express.

Q3.2

What is the difference between a field-level validation rule and workflow, as two different mechanisms for controlling how an object entry can change state?

A **validation rule** is a synchronous, immediate check applied at the moment an entry is submitted – rejecting the submission outright if a specific condition isn't met (e.g. requiring an "End Date" field to be later than a "Start Date" field) – ensuring an entry can never even be saved in an invalid state in the first place. **Workflow** instead governs an already-valid entry's progression through a defined sequence of *approval states* over time (e.g. Draft → Pending Review → Approved → Published), typically involving specific people or roles reviewing and explicitly approving or rejecting the entry at each stage, rather than an instantaneous pass/fail check. The two are complementary: validation rules enforce basic data-quality correctness at submission time, while workflow governs the longer-lived business process an already-valid entry must go through before it's considered final.

Q3.3

Can an object definition be configured to use Liferay's workflow engine, and what does enabling workflow on an object actually change about how entries are created and edited?

Yes – an object definition can be configured to require workflow, associating it with one of Liferay's defined workflow processes (built using the same workflow-definition tooling used elsewhere in the platform, such as for web content approval). Once enabled, newly created or edited entries no longer take effect immediately – instead, they enter the configured workflow process in a pending state, becoming visible/active only once they've progressed through and completed that approval sequence, exactly mirroring how workflow-enabled web content requires review and approval before publishing. This lets organizations apply the exact same governance and approval discipline already used for content management to custom business object data – e.g. requiring a manager's sign-off before a new "Expense Report" object entry actually takes effect.

Q3.4

What is object entry versioning, and how does it let you track and, if necessary, revert changes made to an individual entry over time?

Object entry versioning maintains a history of an entry's previous states each time it's modified, rather than simply overwriting the entry's prior values in place – letting an administrator or auditor review exactly what an entry's field values were at any earlier point in its history, see who made a given change and when, and, where supported, revert an entry back to an earlier version if a change turns out to have been mistaken. This mirrors the same versioning

discipline Liferay's content management system already applies to web content and documents, extended specifically to custom object data – particularly valuable for business objects subject to audit or compliance requirements, where being able to reconstruct exactly what a record looked like at a specific past point in time is a genuine, non-optional business requirement.

Q3.5

What is "saving as draft" for an object entry, and how does it differ from an entry that has simply not yet completed its configured workflow approval?

Enabling draft mode on an object (via its form container configuration) lets a user save a genuinely **incomplete**, in-progress entry – one that may be missing required fields or otherwise fail validation – without that entry being rejected outright, explicitly deferring full validation, workflow triggering, and any configured object actions until the entry is later finalized and actually submitted. This is a distinct concept from an entry that has been fully, validly submitted but is simply awaiting workflow approval: a draft entry hasn't yet been considered "submitted" at all in the object's own lifecycle sense, and saving it as a draft deliberately skips the custom validations and triggered actions that a genuine submission would invoke – draft mode exists specifically to let users pause and resume filling out a long or complex form over multiple sessions, without that partial, still-incomplete data being treated as a real, business-meaningful entry yet.

UI Integration

Q4.1

What is a form container, and how does binding it to an object definition let you build a custom, on-page data-entry form without writing any frontend code?

A form container is a page-builder widget that, once bound to a published object definition, automatically renders an editable form for creating or updating that object's entries directly on an ordinary Liferay page – built entirely through the page builder's drag-and-drop interface, letting an administrator control exactly which fields appear, their order, layout, and labeling, without writing any HTML, CSS, or JavaScript. This is the primary mechanism for exposing object data entry directly to end users (site visitors, portal users) on public or authenticated pages, distinct from the object's own auto-generated back-office admin panel, which is typically reserved for internal administrators managing the underlying data more broadly.

Q4.2

What is a data set (as used in a data set display widget), and how does it let you build a filterable, sortable list or table view of an object's entries directly on a page?

A data set widget renders a configurable table or list view of an object's entries directly on a Liferay page, with built-in support for column selection, sorting, filtering, and pagination – all configured through the page builder's UI rather than custom code. This is the standard way to build, for instance, a public-facing "Browse Open Tickets" page or an internal "All Active Claims" dashboard directly from an object's data, and data sets can be configured to link each row through to that entry's corresponding display page (for viewing full details) or to a form container (for editing) – together, form containers, display pages, and data sets provide the complete set of building blocks needed to construct an entire object-backed application's UI purely through page-builder configuration.

Q4.3

What is a friendly URL for an object entry, and how does configuring the SEO settings on an object definition let each entry be reached through a clean, human-readable, and search-engine-indexable web address?

By default, an object entry's URL is an internal, ID-based address with no inherent readability. Configuring an object's SEO settings (available on object definitions with a display page template configured) lets Liferay automatically assign each entry a **friendly URL** following a configurable pattern – typically incorporating the entry's own title field – e.g. `yoursite.com/products/wireless-headphones` rather than an opaque internal identifier. Beyond simple readability, published entries with a display page template configured become eligible for inclusion in the site's XML sitemap, directly supporting search-engine discoverability – making friendly URL configuration an essential step whenever an object's entries (a product catalog, a knowledge-base article set) are meant to be publicly browsable and discoverable content, not just internally-managed records.

Q4.4

What is in-context translation for object entry text fields, and how does it let a multilingual site's content editors translate a Text, Rich Text, or Textarea field's value directly within the page where it's being edited?

In-context translation lets a user editing an object entry through a form container translate a text-based field's value directly within the same on-page editing experience, rather than needing to navigate away to a separate, dedicated translation management screen to enter each language's version of the text. This streamlines the authoring workflow for multilingual sites specifically built around Objects-backed content (e.g. a multilingual product catalog or knowl-

edge base), letting a single content editor efficiently manage all of an entry's language variants together in one place, directly mirroring the kind of in-context translation experience already familiar from Liferay's web content editing tools.

APIs & Integration

Q5.1

What REST APIs does Liferay automatically generate when you publish an object definition, and what determines the exact URL pattern those generated endpoints use?

Publishing an object definition automatically generates a full set of default REST endpoints for CRUD operations (create, read, update, delete) on that object's entries, without any additional configuration required. The generated endpoints follow the pattern `c/[pluralobjectlabel]` – e.g. an object labeled "Ticket" generates endpoints under `c/tickets` – with the exact base path additionally depending on the object's configured **scope** (Section 6): company-scoped objects use a simpler, top-level path, while site-scoped objects nest their endpoints under a site-specific prefix, since the same object definition's entries are partitioned separately per site in that case.

Q5.2

What is the difference between using an object entry's numeric ID versus its External Reference Code (ERC) when referencing it through the API, and why does Liferay recommend the ERC as the preferred identifier from 2025.Q3 onward?

Every object entry has an internal, auto-assigned numeric ID, generated by Liferay and generally meaningless outside the specific Liferay instance that created it. An **External Reference Code** is instead a client-assignable, human-meaningful identifier that an external system can set explicitly when creating an entry (e.g. matching a corresponding record's ID in an external CRM or ERP system) – and, since Liferay's official guidance from 2025.Q3 onward, the ERC is the **preferred** identifier for API calls specifically because it remains stable and predictable across environments (a numeric ID assigned in a development environment will generally differ from the ID the same logical record receives in production, since IDs are typically not guaranteed to be identical across separate database instances) – making the ERC essential for building integrations and automated deployment/promotion pipelines that need to reliably reference the "same" logical record across multiple separate Liferay environments.

Q5.3

What is the difference between using Liferay's REST APIs versus its GraphQL API for managing object entries, and when would you reach for GraphQL specifically?

The REST APIs (generated automatically per object, as discussed above) expose one endpoint per object, following standard REST CRUD conventions – straightforward and directly discoverable, but potentially requiring multiple separate calls (or the `nestedFields` parameter workaround, Section 2) to assemble data spanning several related objects. The **GraphQL** API instead lets a client specify, in a single request, exactly which fields and related-object data it needs across potentially several objects at once, returning precisely that shape of data in one round trip – particularly valuable for frontend applications (a custom-element client extension, for instance, as covered in the companion Liferay Client Extensions deck) that need to efficiently assemble a complex view spanning multiple related objects without either over-fetching unneeded data or making many sequential REST calls.

Q5.4

What is the Batch API, and what problem does it solve when you need to create, update, or delete a large number of object entries at once?

The Batch API lets a client submit many create, update, or delete operations for an object's entries in a single API call, rather than making one separate HTTP request per individual entry. This is essential for bulk data operations – an initial data migration importing thousands of records from a legacy system, or a periodic bulk-sync job pulling updated records from an external system – where making one individual REST call per record would be prohibitively

slow and would generate an enormous, unnecessary amount of network and server overhead compared to batching the same operations together into one (or a small number of) larger requests.

Q5.5

What is the difference between managing object entries via the REST/GraphQL APIs at runtime versus using a Site Initializer or Configuration as Code to define an object's structure itself as part of an automated deployment?

The REST and GraphQL APIs (and the Batch API) operate on **entries** – the actual data records conforming to an already-published object definition – suited to runtime integration, data sync, and bulk data loading. Defining the **object definition itself** (its fields, relationships, and configuration) as part of a reproducible, version-controlled deployment is instead typically handled through Liferay's **Configuration as Code** and **Site Initializer** mechanisms (covered in depth in the companion Liferay Client Extensions deck), which let an object's entire schema be defined in source-controlled files and deployed automatically and consistently across environments – ensuring the same object structure exists identically in development, staging, and production, rather than relying on an administrator manually, and potentially inconsistently, recreating the same object definition by hand in each environment through the UI.

Scoping & Permissions

Q6.1

What is the difference between a `Company`-scoped object and a `Site`-scoped object, and how does this choice affect where an object's entries are visible and how its data is partitioned?

A **Company**-scoped object's entries are shared globally across the entire Liferay instance, visible and manageable regardless of which specific site a user is currently in – appropriate for genuinely global business data not tied to any one particular site (a company-wide product catalog, a shared list of business partners). A **Site**-scoped object's entries, by contrast, are partitioned separately per site – the same object definition can be reused across multiple sites, but each site sees and manages only its own independent set of entries, with no visibility into another site's entries by default – appropriate for data that's naturally specific to one site's own context (a site-specific event calendar, or site-specific support tickets in a multi-tenant portal setup). Choosing the correct scope up front matters because it directly determines the generated REST API's URL structure (Section 5) and cannot generally be changed after an object has already been published and populated with real data.

Q6.2

How does permission configuration work for an object definition, and what specific actions (viewing, adding, updating, deleting entries) can be independently controlled per role?

Each object definition has its own configurable set of permissions, following Liferay's standard role-based permission model – an administrator can independently grant or restrict specific roles' ability to **view**, **add**, **update**, and **delete** entries for that object, exactly as with any other permissioned resource in Liferay (web content, documents, and so on). This lets an organization precisely control, for instance, that ordinary site members can create and view their own support ticket entries, while only members of a "Support Agent" role can update or resolve any ticket – with permissions configurable at the object-definition level (governing the object as a whole) and, in more advanced configurations, potentially refined further to control access to specific individual entries based on ownership or other business logic.

Q6.3

What does it mean for an object field to be marked "required," and how does this interact with the draft-saving feature (Section 3) discussed earlier?

Marking a field as required means Liferay's standard validation will reject any attempt to submit an entry that leaves that field empty – a basic, built-in data-quality guarantee applied automatically without needing a custom validation rule. As noted in Section 3, when draft mode is enabled for an object's form container, saving a draft deliberately bypasses this required-field validation (along with other custom validations), specifically to let a user save genuinely incomplete, in-progress work – meaning "required" should be understood as governing what's needed for a *final, submitted* entry to be considered valid, not as an absolute constraint enforced at every possible save action against that object.

Q6.4

Why does careful object scope and permission planning matter especially for objects extending a system object (Section 1), where the underlying entity may already have its own separate, pre-existing permission model?

When you add custom fields to a system object (extending, say, the User object with additional business-specific fields), those new custom fields inherit and interact with the system object's own *existing*, often more elaborate permission model – one that predates and is independent of the Objects framework itself, since system objects are core

platform entities with their own long-established access-control rules. This means extending a system object requires understanding not just the Objects framework's own permission configuration, but also how that configuration interacts with the underlying system object's pre-existing permission checks – a genuinely more involved planning exercise than configuring permissions on a purely custom object created entirely from scratch within the Objects framework, where no pre-existing permission model needs to be reconciled with.

Advanced Relationships & Features

Q7.1

What API endpoints does Liferay automatically generate for a defined relationship between two objects, and what does each one (GET, PUT, DELETE) let a client do with that relationship?

For each defined relationship between two objects, Liferay generates a dedicated set of relationship-specific endpoints: a **GET** endpoint returning the related entries for a given parent entry, a **PUT** endpoint for creating or updating an association between two specific entries (relating them together), and a **DELETE** endpoint for disassociating (unlinking) two previously-related entries without necessarily deleting either entry itself. This gives a client fine-grained, explicit control over managing relationships independently from managing the entries themselves – letting an integration, for instance, link an existing "Order" entry to an existing "Customer" entry via the relationship's PUT endpoint, without needing to re-submit or modify either entry's own core field data in the process.

Q7.2

What are aggregation terms in the context of Object REST APIs, and how do they let a client request faceted, aggregated summary data (like counts grouped by a field's value) directly from the API rather than computing it client-side?

Aggregation terms let a client specify one of an object's fields as **facet** criteria in a GET request, causing the API response to include aggregated summary data – most commonly, a count of matching entries grouped by each distinct value of that field – computed server-side as part of the query itself. This is valuable for building dashboard or filter-sidebar UIs (e.g. "127 Open, 43 In Progress, 812 Closed" ticket counts shown alongside a filterable ticket list) without needing the client application to separately fetch every single matching entry and manually tally the counts itself – letting the database do the aggregation work it's already well-suited for, rather than shifting that computational burden onto the client.

Q7.3

Why does defining a mandatory (required) relationship between two objects change how a client must structure its create/update API calls, compared to an optional relationship?

When a relationship between two objects is configured as **mandatory**, the API enforces that every new entry on the "many" side of that relationship must specify its related parent entry at creation time – the create call must include an identifier for the related entry (by ID, by external reference code, or via a `nestedFields` specification, following a defined priority order when more than one identifier is supplied) as part of the request, or the creation will be rejected outright. This is a meaningfully different API contract than an *optional* relationship, where a new entry can be created with no related-entry reference at all and the relationship established later, if ever – making it important for integration developers to check an object's relationship configuration before assuming a given related-entry reference is genuinely optional in their API calls.

Q7.4

What is the general trade-off between modeling a complex business domain as several separate, related objects versus a smaller number of objects with more fields each – and what guidance does normalized relational data modeling offer here?

This is fundamentally the same normalization trade-off familiar from traditional relational database design, simply expressed through the Objects framework's configuration UI instead of raw SQL DDL: splitting a domain into several smaller, clearly-scoped related objects (an "Order" object separate from an "Order Line Item" object, rather than

cramming a variable number of line items into a single sprawling ”Order” object’s own fields) generally produces a cleaner, more maintainable, and more queryable data model, avoids data duplication, and lets each object’s relationships, permissions, and lifecycle be independently configured – at the cost of needing to think through and correctly configure the relationships (Section 2) connecting those separate objects together, and needing more API calls (or judicious use of `nestedFields`, Section 2) to assemble a complete picture spanning several related objects at once.

Deployment & Lifecycle

Q8.1

What is Liferay Publications, and how does its capability to carry object-related customizations along during an upgrade address a historically painful part of Liferay upgrade projects?

Publications is Liferay's mechanism for packaging up a set of configuration changes – object definitions, site structure, and other customizations – made within a given environment, so they can be reliably promoted and reapplied across other environments (from a development sandbox through staging to production) as a single, coherent, trackable unit, rather than requiring manual, error-prone reconfiguration in each environment separately. A historically painful aspect of Liferay upgrades was that customizations made in an older version could become "Outdated" and require manual rework after upgrading – recent Liferay releases specifically improved Publications to instead let customizations made *before* an upgrade remain directly usable afterward, meaningfully reducing the manual rework, support burden, and risk historically associated with major version upgrades for organizations with substantial Objects-based customization already in production.

Q8.2

Why does modifying an already-published object definition's existing fields (renaming a field, changing its type) require more caution than modifying a draft object definition that has no entries yet?

Once an object definition has been published and real entries have been created against it, its fields are directly tied to an actual, populated database schema and to any existing entries, API integrations, display pages, and form containers already built around that specific structure. Renaming or changing the type of an existing field on a published object risks breaking existing data (a type change may not cleanly convert existing values), breaking any external integrations that reference the field by its original name, and breaking any UI components (form containers, data sets, display page templates) configured to reference that specific field – which is precisely why such changes to a published, populated object generally require careful planning, thorough testing in a non-production environment first, and often a deliberate, considered migration strategy, quite unlike the free, consequence-free editing possible on a draft object definition with no real data or dependent integrations yet.

Q8.3

What is the general recommended workflow for developing and rolling out a new object definition across a typical multi-environment Liferay project (development, staging, production)?

A typical, recommended workflow designs and iterates on the object definition (fields, relationships, actions, validation) in a lower, non-production **development** environment first, where mistakes are cheap and no real business data is at risk; once the design stabilizes, it's promoted to a **staging** environment (via Publications or Configuration as Code, Section 5) for broader stakeholder review and integration testing against realistic, staging-representative data and connected systems; and only once fully validated is it finally promoted to **production**, ideally using the same automated promotion mechanism used for the staging promotion, to ensure the production object definition is genuinely identical to what was already tested in staging, rather than being manually, and potentially inconsistently, recreated by hand directly in the production environment.

Q8.4

Why should an integration or automated deployment pipeline generally prefer referencing objects and fields by their stable technical name, rather than by their (potentially later-changed) display label?

Every object and field has both a **display label** (the human-readable name shown in the UI, which administrators may reasonably want to rename over time as terminology evolves – e.g. relabeling ”Ticket” to ”Support Request”) and an underlying, more stable technical name used internally and in API references. Building integrations, custom validation logic, or automated deployment scripts that reference objects/fields by their display label creates a fragile dependency that would silently break the moment someone renames that label for purely cosmetic, UI-facing reasons – exactly the kind of stability concern that also motivates preferring External Reference Codes over potentially environment-specific numeric IDs (Section 5) – reinforcing a general pattern throughout Objects development: prefer stable, technical identifiers for anything programmatic or environment-spanning, and reserve display labels purely for what end users actually see in the UI.

End of deck – see the companion Liferay Client Extensions deck for frontend/theming integration.