Liferay Core Concepts Deck

Kaleo Workflow, Multi-Tenancy & More, Explained

Coverage	Multi-Tenancy & Instance Architecture, Sites/Organizations/User Groups, Kaleo Workflow Fundamentals, Advanced Kaleo Workflow, Roles & Permissions, Content & Asset Framework, Personalization & Segmentation, Staging & Publishing
Format	Question → Concise answer → comparison / example where useful
Level	Liferay Business Analyst / Administrator to Solution Architect rounds

---

Reflects Liferay DXP 7.4+ / current Liferay DXP Cloud (quarterly release) platform concepts. UI names and default behaviors shift between quarterly releases – always confirm against the release notes for your specific DXP version.

Generated September 20, 2026

1 Multi-Tenancy & Instance Architecture 2

2 Sites, Organizations & User Groups 4

3 Kaleo Workflow Fundamentals 6

4 Advanced Kaleo Workflow 8

5 Roles & Permissions 10

6 Content & Asset Framework 12

7 Personalization & Segmentation 14

8 Staging & Publishing 16

Multi-Tenancy & Instance Architecture

Q1.1

What is a Liferay Virtual Instance, and how does it provide the strongest, most complete level of tenant isolation available within a single Liferay installation?

A Virtual Instance is a completely independent, self-contained portal environment – with its own domain, its own users, its own sites, its own configuration – running within one shared physical Liferay installation and JVM process. Multiple virtual instances share the same underlying codebase, the same running server process, and (depending on configuration) potentially the same physical database, but their actual data and operations remain entirely separate and mutually invisible to each other, exactly as if they were running on completely separate servers. This makes the virtual instance the **strongest** tenant-isolation boundary Liferay offers short of genuinely separate physical installations – appropriate specifically when tenants must be completely, unambiguously isolated from one another with no logical integration or data-sharing between them at all (e.g. hosting entirely unrelated client organizations' portals from one shared platform investment).

Q1.2

What is the hierarchy of tenancy/organizational levels in Liferay – Instance, Company, Site, Organization, and User Group – and how does each successive level narrow the scope of isolation or grouping compared to the one above it?

- **Virtual Instance** – the outermost, strongest isolation boundary (Q1.1); each instance has exactly one associated **Company** record representing that instance's own top-level portal entity.
- **Site** – a collection of pages, content, and applications within a company/instance, used to represent a distinct web presence, community, or team space (a corporate website, an intranet, a partner portal), each with its own membership and permissions.
- **Organization** – a hierarchical grouping of users modeling a real-world structure (a company's departments, regional offices), used primarily for organizing *users* and inheriting permissions down through that hierarchy, rather than primarily for hosting pages/content the way a Site does.
- **User Group** – a simple, typically flat (non-hierarchical) grouping of users, used mainly as a convenient way to bulk-assign the same set of site memberships, page/content permissions, or personalized page layouts to many users at once.

Each level narrows the scope of what's being isolated or grouped, from "an entire independent portal" down to "just a convenient bucket of users to manage together."

Q1.3

What is the difference between choosing Site-level tenancy versus Organization-level tenancy for isolating a set of business units, and what specific factor – whether the units need their own distinct pages/content versus simply needing hierarchical user grouping – should drive that choice?

Site-level tenancy is the right choice when each business unit genuinely needs its *own distinct web presence* – its own pages, its own content, its own look and feel – since Sites are fundamentally the container for pages and applications in Liferay. **Organization**-level tenancy is instead the right choice when the actual need is primarily about representing a *real-world reporting/user hierarchy* (Region $\rightarrow$ Country $\rightarrow$ Office) for the purposes of user management and permission inheritance down that hierarchy, without necessarily needing each level of that hierarchy to have its own

separate set of pages at all. In practice, the two are frequently combined – an Organization structure used to model and manage the user hierarchy, with one or more Sites layered on top to actually host the pages and content those organizational users interact with.

Q1.4

What is the difference between Shared Tables/Shared Database Instance (STSI) and Independent Database/Independent Instance (IDII) as two different underlying database architectures for implementing multi-tenancy, and what trade-off does each represent?

IDII gives each tenant (typically each virtual instance) its own genuinely separate database, providing the strongest possible data isolation and eliminating any risk of one tenant's queries or load impacting another's, at the cost of higher operational overhead – more databases to provision, back up, and maintain independently. **STSI** instead has multiple tenants share the same underlying database and tables, with every record tagged (typically via a company/instance ID column) to indicate which tenant it belongs to, and application-level query logic responsible for correctly filtering every query by that tenant identifier – reducing database operational overhead and infrastructure cost, at the cost of a comparatively larger "blast radius" if that tenant-filtering logic is ever implemented incorrectly, and shared resource contention (one tenant's heavy query load can, in principle, impact another tenant sharing the same database). Liferay's own virtual instance architecture defaults toward the STSI model (shared database, tenant-tagged records), though IDII-style separate databases per instance are also achievable with additional configuration.

Q1.5

Why does choosing the "right" level of tenancy (Instance vs. Site vs. Organization vs. User Group) for a given real-world requirement genuinely matter, rather than simply defaulting to the strongest, most isolated option (Virtual Instance) for every scenario?

Each successive level of isolation trades additional administrative and operational overhead for stronger separation – provisioning and maintaining a full virtual instance for every distinct customer, department, or use case, when a simpler Site or Organization would have sufficed, unnecessarily multiplies the number of separately-administered environments (each with its own configuration, its own upgrade/patch application, and often its own user-management overhead) without any genuine corresponding benefit if the underlying requirement never actually demanded that strongest level of isolation in the first place. Liferay's own guidance frames this explicitly as a case-by-case analysis of the actual business requirement – genuinely isolated, unrelated tenant bases justify the Instance level; related communities sharing a common user base but needing distinct web presences fit the Site level; and pure user-hierarchy/permission-inheritance needs, without a corresponding need for genuinely separate content, fit the Organization level – defaulting reflexively to the strongest available isolation option for every scenario generally produces unnecessary long-term administrative burden.

Sites, Organizations & User Groups

Q2.1

What is a Site Template, and how does it let you define a reusable page structure and configuration that many individual sites can be built from, while retaining – or explicitly losing – their connection back to that shared template?

A Site Template defines a reusable starting structure for a site – its pages, layout, and initial configuration – that can be applied when creating any number of new, individual sites, giving each one a consistent, pre-built starting point rather than requiring every new site to be built entirely from scratch. Sites created from a template can optionally remain **linked** to that template, meaning subsequent changes made to the template itself automatically propagate out to every site that was created from it and remains linked – useful for maintaining consistent structure across, for instance, many regional variants of the same overall site design – or a site can be explicitly **unlinked** from its originating template at any point, after which it evolves entirely independently, with no further changes to the template affecting that now-unlinked site at all.

Q2.2

What is the difference between a Regular Organization and a Location in Liferay's organization hierarchy, and what specific structural constraint does designating an organization as a Location impose on it?

Liferay's organization hierarchy supports different organization **types**, most notably Regular Organizations (which can contain further child organizations nested beneath them, building an arbitrarily deep hierarchy) and **Locations** (which represent a genuinely physical or logical "leaf" point in that hierarchy – a specific office, branch, or site – and, reflecting that leaf-node role, cannot themselves have further child organizations nested beneath them). This distinction lets an organization hierarchy correctly model real-world structures where some levels are genuinely further sub-dividable (a Region containing multiple Countries) while other levels represent a true bottom-level unit that doesn't meaningfully subdivide any further (a specific physical branch office) – enforcing that structural constraint directly through the platform's own organization-type configuration, rather than leaving it as an unenforced convention administrators must remember to follow themselves.

Q2.3

Why is a User Group generally the right tool specifically for bulk-applying personalized page layouts or a common set of site memberships to many users at once, rather than using an Organization for that same purpose?

While Organizations *can* also be used for permission and membership management, User Groups are specifically optimized for the comparatively simple, flat use case of "these particular users should all get the same treatment" – bulk-assigning the same site memberships, the same set of personalized/customized pages, or the same permission grants to every member of the group in one action, without needing or implying any hierarchical reporting-structure relationship between those users at all. This makes User Groups the natural fit for groupings that are genuinely about shared *treatment* rather than shared *organizational position* – e.g. "everyone who completed the onboarding training" or "beta testers for the new intranet homepage" – categories that cut across an organization's actual reporting hierarchy rather than corresponding to any single branch of it, which is exactly the kind of grouping an Organization's inherently hierarchical structure isn't well suited to represent directly.

Q2.4

What is the difference between an ”Open,” ”Restricted,” and ”Private” site membership type, and how does this setting control who can discover and join a given site?

A site’s membership type controls both its visibility and how users can become members: an **Open** site is publicly visible and any user can join it directly, with no approval required. A **Restricted** site is visible (discoverable) to users browsing available sites, but joining requires an administrator’s explicit approval of the user’s membership request, rather than immediate self-service joining. A **Private** site is not even visible to non-members browsing the site directory at all – only users already explicitly added as members (or organizations/user groups granted access) can see or access it. This three-tier model lets site administrators choose the appropriate balance between open self-service community building and tightly controlled, invitation-only access, depending on that specific site’s actual purpose and sensitivity.

Q2.5

How does a user’s effective permissions on a given site typically get assembled from several different sources at once – their direct site membership, any Organization they belong to, any User Group they’re in, and any Roles assigned to them – and why does this layered model require careful administration to avoid unintended access?

A user’s actual, effective set of permissions on a given resource is the cumulative result of *all* the different membership and role assignments that apply to them simultaneously – their direct membership in a site, membership in an Organization that itself has a relationship to that site, membership in a User Group that grants certain permissions, and any Roles (Section 5) directly or indirectly assigned to them through any of these paths – with permissions generally being **additive** (a grant from any one source generally isn’t overridden or narrowed by the absence of that same grant from another source). This layered, additive model is powerful and flexible, but requires genuinely careful administration specifically because it can be surprisingly easy to unintentionally grant a user broader access than intended by combining several individually-reasonable-looking group/role assignments that, in combination, add up to more access than any single assignment alone would suggest – making periodic, holistic permission audits (rather than only ever reviewing one assignment mechanism in isolation) an important operational discipline for any non-trivial Liferay deployment.

Kaleo Workflow Fundamentals

Q3.1

What is Kaleo, and what specific problem does Liferay's built-in workflow engine solve for content and business processes that need review and approval before taking effect?

Kaleo is Liferay's built-in workflow engine, shipped as part of the core platform, providing a general-purpose framework for defining and running multi-step approval processes over any kind of **asset** registered with the platform – web content, blog entries, wiki articles, message board posts, and (as covered in the companion Liferay Objects deck) custom Objects entries can all be routed through a Kaleo-defined workflow. It solves the recurring business need to ensure content or data doesn't take effect (get published, become active) until it has passed through a defined review and approval sequence, involving specific people or roles at each stage – without requiring custom-built approval logic to be separately implemented for every different type of content or entity that happens to need this kind of governance.

Q3.2

What is a workflow definition, and how is it typically authored – as an XML file directly, or through Liferay's graphical Workflow Designer – with each approach's own respective trade-offs?

A workflow definition is the actual, concrete specification of a specific approval process – its nodes, transitions, and configured behaviors – that can then be assigned to govern one or more asset types. It can be authored two ways: directly as a hand-written **XML** file following Kaleo's defined schema (offering full, precise control over every detail, including advanced scripting logic, but requiring familiarity with the XML schema and a more code-centric workflow), or through the graphical **Workflow Designer** (a drag-and-drop visual interface, refreshed with an improved UI/UX in Liferay DXP 7.4 Update 20, letting business analysts and administrators build and modify workflow definitions visually without directly editing XML) – with the Workflow Designer generally the more approachable and now more heavily-invested-in option for most day-to-day workflow authoring, while direct XML editing remains available for cases needing capabilities or precision beyond what the visual designer currently exposes.

Q3.3

What are the core node types available when building a Kaleo workflow definition – State, Task, Condition, Fork, Join, Join XOR, and Timer – and what distinct role does each play in the overall workflow "state machine"?

Kaleo models a workflow as a state machine built from several distinct node types: **State** nodes represent a simple status the asset is in (Created, Approved, Rejected), typically without requiring direct human action of their own. **Task** nodes represent a point requiring actual human interaction – a person must review and act (approve, reject, request changes) before the workflow proceeds. **Condition** nodes evaluate logic (Section 4) to determine which of several possible transitions the workflow should take next, without themselves requiring human interaction. **Fork** and **Join/Join XOR** nodes (Section 4) handle splitting a workflow into multiple simultaneous parallel paths and later recombining them. **Timer** nodes (Section 4) trigger an automatic action after a specified time period has elapsed. Together, these node types provide the building blocks for modeling approval processes ranging from a single-approver rubber stamp to elaborate, multi-stage, multi-reviewer governance processes.

Q3.4

What is a Transition, and how does it connect two nodes together to define the actual path(s) a specific asset can follow as it moves through a workflow definition?

A Transition represents a directed connection from one node to another, defining a specific path the workflow can traverse – visually rendered as an arrow connecting two nodes in the Workflow Designer's canvas. A single node can have multiple outgoing transitions (e.g. a Task node might have separate "Approve" and "Reject" transitions leading to two different subsequent nodes), and the specific transition actually taken for a given asset depends on either a direct human decision at a Task node (choosing to approve versus reject) or the evaluated outcome of a Condition node's logic – together, the complete network of nodes and the transitions connecting them fully defines every possible path an asset could take through that workflow definition, from its initial creation through to a final, terminal state.

Q3.5

Walk through the default Single Approver Definition – Liferay's simplest built-in example workflow – and describe what happens as an asset moves through its two task nodes (Review and Update) toward final approval or rejection.

The Single Approver Definition is Liferay's simplest built-in example, intended as an introductory illustration of core workflow concepts: a newly-submitted asset enters an initial "In Review" state, transitioning to a **Review** task node where a designated approver must take action – approving the asset (transitioning it to an Approved end state, at which point it's published/takes effect) or rejecting it, which transitions to a second **Update** task node, returning control to the original author to make revisions before resubmitting the asset back into the Review task once more. This simple, two-task-node cycle (Review $\leftrightarrow$ Update) repeats until the asset is eventually approved, illustrating the basic task-transition-task pattern that more elaborate, multi-approver or multi-stage workflow definitions build further upon.

Advanced Kaleo Workflow

Q4.1

What is a Condition node's evaluator script, and why does needing multiple approvers before an asset is considered fully approved (rather than a single approver) typically require a Condition node to check how many approvals have been recorded so far?

A Condition node uses a script (traditionally written in Groovy) to evaluate some logic against the current state of the workflow and its associated asset, determining which of several possible outgoing transitions should actually be taken. A common, canonical use case is a multi-approver workflow: after each individual approver takes their own approval action at a Task node, a Condition node checks how many total approvals have been recorded so far – if the required threshold (say, two approvals) hasn't yet been reached, the condition transitions the workflow back to the task node for the next required approver; once the threshold is met, it instead transitions to the final Approved state – letting the workflow definition express genuinely conditional, count-dependent branching logic that a simple, unconditional linear sequence of Task nodes alone could not represent.

Q4.2

As of Liferay DXP 2024.Q3, what changed about whether workflow scripting (Groovy-based Condition evaluators and similar) is available by default, and how is it re-enabled if needed?

As of DXP 2024.Q3, workflow scripting capability – including the Groovy-based condition evaluators discussed above – is **disabled by default**, reflecting a broader, platform-wide tightening of default security posture around arbitrary server-side script execution. Administrators who genuinely need scripting capability for their workflow definitions can re-enable it explicitly through **System Settings** → **Script Management**, found under the Security category of the admin console – a deliberate, explicit opt-in step, rather than scripting remaining available by default the way it was in earlier Liferay versions, reflecting the general security principle that powerful, potentially-risky capabilities should require an explicit administrative decision to enable, rather than being available out of the box whether or not they're actually needed.

Q4.3

What is the difference between a Fork node and a Join (or Join XOR) node, and how do they work together to let a workflow route an asset through several genuinely parallel, simultaneous approval paths?

A **Fork** node splits a single incoming workflow path into multiple simultaneous, parallel outgoing paths – useful when several genuinely independent reviewers or approval steps need to happen concurrently, rather than one strictly after another (e.g. simultaneously routing an asset to both a Legal reviewer and a Marketing reviewer, neither of whom needs to wait for the other to finish first). A **Join** node recombines those parallel paths back into one, but only proceeds once *every* one of the parallel paths has completed (all required approvers must finish before the workflow continues) – while a **Join XOR** node instead proceeds as soon as *any one* of the parallel paths completes, without waiting for the others (appropriate when only one of several parallel approval paths actually needs to succeed for the overall step to be considered complete). Together, Fork paired with either Join type is the standard Kaleo pattern for modeling genuinely concurrent, multi-reviewer approval processes.

Q4.4

What is a Task Timer, and what specific operational problem – a task sitting unattended indefinitely because the assigned reviewer forgot about it or is unavailable – does attaching a timer to a Task node solve?

A Task Timer can be attached to a Task node to automatically trigger a configured action – commonly sending a reminder notification, escalating the task to a different or additional reviewer, or automatically transitioning the workflow down some default path – once a specified time period has elapsed without the task being acted upon. This directly addresses a common, entirely realistic operational failure mode: a task assigned to a specific reviewer who is on vacation, has left the organization, or has simply overlooked the pending task in a busy inbox, would otherwise sit indefinitely in limbo with no one aware it's stuck, silently blocking the underlying content or process from ever completing its approval – task timers provide an automatic, built-in safety net against exactly this kind of silent, unattended-task failure mode, without requiring a human to separately, manually monitor every in-flight workflow task for staleness.

Q4.5

What is Workflow Metrics, and how does its reporting dashboard let administrators identify bottlenecks – such as a specific task or reviewer that's consistently slow – across an organization's actual, real-world workflow usage over time?

Workflow Metrics (accessed via Control Panel → Workflow → Metrics) provides reporting and analytics on how workflow processes are actually being used and performing in practice – surfacing data like average time-to-completion for specific tasks, how many assets are currently pending at each stage, and which specific tasks or process steps tend to take unusually long to complete. This turns workflow governance from a purely qualitative, anecdotal concern ("it feels like approvals are slow lately") into something administrators can investigate quantitatively – identifying, for instance, that one specific task node consistently has a much longer average completion time than every other step in the same workflow, pointing toward a specific bottleneck (an overloaded reviewer, an unclear task description causing confusion, or a genuinely mis-designed workflow step) that's actually, measurably slowing down the overall approval process, rather than relying purely on informal impressions of where the process feels slow.

Roles & Permissions

Q5.1

What is the difference between a Regular Role, a Site Role, and an Organization Role in Liferay, and what does each one's scope determine about where a granted permission actually applies?

A **Regular Role** is scoped globally, across the entire portal instance – a user assigned a Regular Role holds that role's permissions everywhere in the instance, regardless of which specific site or organization they're currently interacting with (an "Administrator" role, for instance, is typically a Regular Role). A **Site Role** is scoped specifically to one site at a time – a user can hold a Site Role within one particular site (making them, say, a "Site Content Reviewer" for that specific site) without that same role or its permissions applying to any other site they might also be a member of. An **Organization Role** is analogously scoped to a specific organization, applying only within that organization's own context. This scoping distinction lets an administrator grant exactly the right breadth of permission for a given need – a genuinely instance-wide administrative role versus a role that should only ever apply within one specific site or organizational context.

Q5.2

What is a resource action in Liferay's permission model, and how does the combination of a specific resource type plus a specific action (e.g. "Web Content" plus "Update") form the actual, granular unit that a permission grant applies to?

Every permissible operation in Liferay is modeled as a specific **action** (View, Add, Update, Delete, Permissions, and other operation-specific actions) applicable to a specific **resource** (a resource type like Web Content, or a specific individual resource instance like one particular web content article) – a permission grant is fundamentally a statement of the form "this Role may perform this Action on this Resource (or resource type)." This granular resource-action model is what lets Liferay's permission system express genuinely fine-grained access control – a role might be granted "View" and "Update" actions on Web Content but not "Delete," or might be granted full permissions on one specific site's content but no access at all to another site's – rather than only supporting coarse, all-or-nothing access grants.

Q5.3

What is permission inheritance in the context of Organizations, and how does granting a permission at a parent organization level let that permission automatically flow down to child organizations beneath it in the hierarchy?

When permissions are configured at a parent Organization level, they can be set to automatically **inherit** down through that organization's child organizations beneath it in the hierarchy – a permission granted once at a top-level "Region" organization can automatically apply to every "Country" and "Office" organization nested beneath it, without needing to separately, manually re-grant that same permission at every individual level of the hierarchy. This inheritance behavior is precisely what makes the Organization hierarchy (Section 2) valuable specifically for permission management at scale – letting an administrator manage permissions once, at the appropriate level of a real-world organizational structure, and have that decision correctly and automatically propagate down to every relevant sub-unit, rather than requiring redundant, error-prone manual configuration repeated at every single level of a potentially deep hierarchy.

Q5.4

What is the "Guest" role, and why does understanding exactly what permissions are granted to it matter enormously for correctly securing any content or pages meant to be restricted to authenticated users only?

The **Guest** role represents unauthenticated, anonymous visitors – anyone browsing the site without having logged in. Any permission granted to the Guest role is, by definition, granted to literally *anyone* who can reach the site at all, with no login or credential required whatsoever – making a careless or overly broad Guest-role permission grant one of the most common and most consequential sources of accidental data exposure on any web platform, Liferay included. This is precisely why security reviews of a Liferay deployment should always specifically and carefully audit exactly what the Guest role can see and do – confirming that any genuinely sensitive content, pages, or functionality correctly requires actual authentication, rather than having been inadvertently left accessible to the Guest role through a default or overly permissive configuration.

Q5.5

Why does a genuinely comprehensive permission audit need to examine several different assignment mechanisms together – direct user role assignment, role assignment via Organization or User Group membership, and Site membership – rather than checking just one of these in isolation?

As discussed in Section 2, a user's effective permissions are the cumulative, additive result of every applicable grant across all of these different mechanisms simultaneously – a role can be assigned directly to an individual user, inherited by virtue of that user's membership in an Organization or User Group that itself has role assignments, or granted implicitly through Site membership combined with a Site Role. Checking only one of these mechanisms in isolation (auditing only direct user-role assignments, for instance, while ignoring Organization- and User-Group-derived role assignments) gives a genuinely incomplete and potentially seriously misleading picture of a user's *actual* effective access – a thorough, trustworthy permission audit must trace and combine every one of these assignment paths together to correctly determine what a given user can truly see and do across the platform.

Content & Asset Framework

Q6.1

What is the Asset Framework, and how does it provide a single, unified underlying model that lets fundamentally different content types – web content, blog entries, documents, wiki pages, and (per the companion Liferay Objects deck) custom Objects entries – all share common platform capabilities like tagging, categorization, and workflow?

The Asset Framework is Liferay's underlying, unifying abstraction treating every distinct type of content or entity in the platform – web content articles, blog posts, uploaded documents, wiki pages, message board threads, and custom Objects entries – as an "asset," each registered with a common set of shared metadata and capabilities regardless of its own specific, underlying type-particular structure. This unification is precisely what lets platform-wide capabilities like tagging, categorization (Section 6's next questions), Kaleo workflow (Sections 3-4), comments, ratings, and related-content suggestions work *consistently* across every different content type, implemented once in the shared Asset Framework itself, rather than needing to be separately, redundantly reimplemented for each individual content type – exactly the same underlying framework that lets a custom Objects entry (companion Liferay Objects deck) participate in Kaleo workflow using the identical mechanism already used for web content approval.

Q6.2

What is the difference between a Tag and a Category (Vocabulary) as the two main mechanisms the Asset Framework provides for classifying content, and when would you use one over the other?

Tags are simple, free-form, flat keywords that any user with appropriate permission can apply directly to an asset, with no predefined structure or hierarchy constraining what tags exist or how they relate to each other – suited to organic, bottom-up, crowd-sourced-style labeling. **Categories** instead belong to a formally defined **Vocabulary** – an administrator-managed, hierarchical taxonomy (e.g. a "Product Line" vocabulary with categories like Electronics $\rightarrow$ Audio $\rightarrow$ Headphones) that content is classified against in a controlled, consistent, top-down manner. Tags are the right tool for flexible, informal, emergent labeling where you don't want to constrain users to a fixed predefined list; Categories are the right tool whenever consistent, structured, hierarchical classification genuinely matters – for building reliable faceted search/filtering, or for content that must be classified according to a formally agreed, governed taxonomy rather than whatever ad hoc labels happen to occur to individual content authors.

Q6.3

What is a Web Content Structure, and how does it separate a content type's underlying data fields from the Web Content Template that controls how those fields are actually rendered/displayed?

A Web Content Structure defines the actual data fields a given type of web content article holds – a "Press Release" structure might define Title, Body, Publish Date, and Spokesperson fields – directly analogous in spirit to an Objects field definition (companion Liferay Objects deck), but specifically for the web-content-management side of the platform. A separate **Web Content Template** then controls how those structure fields are actually rendered into HTML for display – and, critically, the same underlying structure can be paired with *multiple* different templates (a "compact" template for a homepage teaser listing, a "full" template for the article's own dedicated page), letting the same underlying content data be presented differently in different contexts without needing to duplicate or re-enter the actual content itself for each different presentation.

Q6.4

What is the Asset Publisher widget, and how does its ability to query and display assets based on type, tags, categories, and site scope make it the standard, general-purpose tool for building dynamic, filtered content listings on a page?

The Asset Publisher is a page-builder widget that queries and renders a filtered listing of assets from across the platform's Asset Framework, based on a wide range of configurable selection criteria – asset type (only Blog Entries, or a mix of several types), specific Tags or Categories the assets must be classified with, which site(s) to pull from, and various sorting and display options. This makes it the standard, general-purpose tool for building dynamic content listings – a homepage's "Latest News" section, a category-filtered product listing, a tag-based "related content" block – entirely through configuration, without needing custom development, and directly complementing the more structured, table-oriented Data Set widget (companion Liferay Objects deck) that serves a similar dynamic-listing role specifically for Objects entries.

Q6.5

Why does the Asset Framework's unified tagging/categorization capability provide a meaningful advantage for cross-content-type search and discovery, compared to a hypothetical platform where each content type implemented its own separate, incompatible tagging system?

Because every asset type shares the exact same underlying tagging and categorization mechanism, a single Category or Tag applied consistently across different content types (a "2025-Product-Launch" tag applied to a blog post, a press-release web content article, and a set of supporting documents) enables genuinely **cross-type** discovery and search – a single Asset Publisher configuration, or a single site search query, can surface all content related to that theme regardless of which specific underlying content type each individual piece happens to be. In a hypothetical platform where each content type maintained its own separate, incompatible tagging implementation, achieving this same kind of unified, cross-type content discovery would require custom integration work stitching together each type's own separate tagging system – exactly the kind of redundant, type-specific reimplementation the shared Asset Framework's unified design is specifically intended to avoid.

Personalization & Segmentation

Q7.1

What is a Segment in Liferay, and how does defining membership criteria (based on user profile attributes, behavior, or explicit manual assignment) let content and page experiences be automatically tailored to different groups of site visitors?

A Segment defines a named group of users based on configurable membership criteria – attributes from a user's profile (job title, location, organization membership), their observed behavior on the site, or simple explicit/manual assignment – with Liferay automatically, continuously evaluating which users currently qualify as members of each defined segment as their profile or behavior changes over time. Once defined, segments become the targeting mechanism for personalization (discussed next) – letting content, page sections, or even entire page variations be automatically shown only to visitors who currently qualify as members of a specific segment, without needing to build separate, hand-maintained logic for every individual personalization scenario.

Q7.2

What is personalization (in the specific sense of Liferay's Personalization/Audience Targeting capability), and how does it use defined Segments to show different content, page layout, or even entirely different page variants to different visitors?

Personalization lets administrators and content authors configure a page (or specific sections/fragments within a page) to display different content or behavior depending on which Segment(s) the current visitor belongs to – a homepage banner might show a different call-to-action to visitors segmented as "existing customers" versus "first-time visitors," or an entire page variant might be swapped in wholesale for a specific high-value segment. This capability directly builds on the segment membership evaluation discussed above – Liferay determines, in real time as a page is rendered, which segments the current visitor currently qualifies for, and applies whatever personalization rules have been configured to target those specific segments, without requiring separate custom development for each individual personalization scenario.

Q7.3

What is A/B testing (experimentation) in the context of Liferay's personalization capabilities, and how does it let content authors validate which of several page or content variants actually performs better with real visitors, rather than relying purely on subjective judgment?

A/B testing capability lets a content author define two or more variants of a page or a specific page section, and have Liferay automatically split incoming visitor traffic between those variants (typically randomly, in a configured proportion), tracking a defined success metric (click-through rate, conversion, time on page) for each variant separately over the course of the experiment. This lets an organization make content and layout decisions based on **actual, measured visitor behavior** rather than purely on internal stakeholders' subjective opinions about which variant "looks better" or "should" perform better – directly complementing the segment-based personalization discussed above, since a segment-targeted personalization decision (which specific variant should a given segment see) is itself often best validated and refined through exactly this kind of A/B experimentation before being rolled out as the final, permanent configuration.

Q7.4

Why does building effective personalization and segmentation generally require close collaboration between marketing/content stakeholders (who define the target segments and

desired experiences) and technical/platform teams (who configure the underlying rules and page variants), rather than being purely a one-side responsibility?

Defining a genuinely useful segment (deciding which profile attributes or behaviors actually distinguish a meaningfully different audience worth targeting differently) and deciding what a personalized experience should actually look like for that segment are fundamentally business and content-strategy decisions, typically best made by marketing or content stakeholders who understand the target audience and business goals – but correctly and robustly implementing those decisions as actual working segment-membership rules and page/content variants within Liferay's configuration is a more technical, platform-specific task. This natural division of expertise is precisely why effective use of Liferay's personalization capabilities in practice generally requires ongoing, close collaboration between these two groups, rather than either group attempting to fully own the entire personalization workflow independently of the other – a recurring pattern across most real-world platform capabilities that sit at this same kind of business-strategy-meets-technical-configuration intersection.

Staging & Publishing

Q8.1

What is Liferay Staging, and what problem does it solve for organizations that need to review and approve a batch of content and structural changes together before they all go live simultaneously?

Staging provides a separate, working copy of a site's content, pages, and configuration where changes can be made, reviewed, and refined *without* those in-progress changes being visible on the live, production site at all – only once a staged set of changes is explicitly **published** does it actually take effect on the live site, all at once, as one coordinated batch. This solves a genuine, recurring organizational need: a marketing team preparing an entire new campaign's worth of new pages, updated content, and revised navigation wants to build, review, and coordinate all of those interrelated changes together as a complete, cohesive unit, and only make the whole batch live simultaneously once everything is ready and approved – rather than each individual change going live piecemeal, the instant it's saved, potentially leaving the live site in an inconsistent, half-updated state during the preparation period.

Q8.2

What is the difference between Local Staging and Remote Staging, and what specific infrastructure topology does Remote Staging require that Local Staging does not?

Local Staging runs the staging environment within the *same* Liferay instance as the live site – a separate staged version of the site exists alongside the live version, both managed by the same running Liferay installation, with publishing simply promoting the staged changes into the live version within that same instance. **Remote Staging** instead involves *two genuinely separate* Liferay installations – a dedicated staging server (potentially with its own separate infrastructure entirely) and a separate live/production server – with publishing transmitting the staged changes across the network from the staging installation to the live installation. Remote Staging requires the additional infrastructure of maintaining a genuinely separate staging server environment, but offers stronger isolation (staging activity and load has zero impact on the live production server's own performance) and is the more common choice for larger, more security- or performance-conscious production deployments.

Q8.3

Why does publishing a batch of staged changes to a live site sometimes surface a conflict – and what typically causes such a conflict when the same piece of content has also been directly modified on the live site while staging was in progress?

If a specific piece of content or configuration is modified *both* within the staging environment *and* directly on the live site during the same period staging was in progress (e.g. an urgent live-site fix made directly to a page that a separate, longer-running staging effort was also independently modifying), publishing the staged changes can surface a genuine conflict – Liferay cannot automatically, safely determine which of the two independently-made versions of that same content should actually win. Correctly handling this generally requires an administrator to manually review and resolve the specific conflict, deciding which version's changes should ultimately be kept (or how to merge them) before the publish can safely proceed – a good practical reason many organizations establish clear operational conventions about which specific environment (staging versus live) is the authoritative source of truth for ongoing edits during any period when a staging effort is actively in progress, specifically to minimize how often this kind of conflicting-parallel-edit scenario arises in the first place.

Q8.4

How does Kaleo Workflow (Sections 3-4) typically integrate with the Staging and publishing process, and why might an organization configure workflow specifically on the staging environment rather than (or in addition to) the live site?

An organization can configure Kaleo workflow to govern content changes made *within* the staging environment itself – requiring, for instance, a content reviewer's sign-off on each individual staged change before that change is even eligible to be included in the next published batch – layering the granular, per-asset approval discipline of workflow on top of staging's own separate, batch-level "publish the whole reviewed set together" control. This combination lets an organization enforce both levels of governance simultaneously: individual changes are vetted one at a time as they're made (via workflow), while the overall, coordinated timing of when a complete, reviewed batch of changes actually goes live to real visitors remains a separate, deliberate publishing decision (via staging) – rather than needing to choose between only one of these two complementary governance mechanisms.

Q8.5

Why is testing the actual publish process itself – not just reviewing staged content in isolation – an important practice before relying on staging for a genuinely critical, high-stakes content release?

Reviewing staged content by simply browsing the staging environment confirms the *content itself* looks correct in isolation, but doesn't necessarily exercise or validate the separate, distinct mechanics of the *publish operation itself* – how a large, complex batch of interrelated changes actually gets applied to the live site, how any conflicts (discussed above) get surfaced and handled, and how long the publish operation itself actually takes to complete for a batch of realistic size and complexity. For a genuinely critical, high-visibility release (a major campaign launch, a significant site redesign), it's good practice to actually perform a full test publish – ideally against a realistic staging copy of the production environment, rather than only ever reviewing the staged content by eye – specifically to validate that the publish mechanics themselves will behave as expected, rather than discovering an unexpected conflict, an unexpectedly long publish duration, or some other publish-specific issue for the very first time during the actual, live, time-sensitive production release itself.

End of deck – see the companion Liferay Objects, Client Extensions, and Clustering decks for the rest of the platform.