Senior Liferay DXP 7.4 Interview Preparation Handbook

Detailed Questions, Answers, Architecture and Scenarios

Covers OSGi, MVC, Service Builder, Headless REST, Java, Spring Boot, Hibernate,

React, security, performance, CI/CD, upgrades, testing and leadership.

Prepared for a senior enterprise portal / solution architecture opportunity
November 2026

Contents

1 Liferay DXP 7.4: Platform and Architecture 2

2 OSGi, Declarative Services, and Module Design 3

3 MVC Portlets and Front-End Integration 3

4 Service Builder, Persistence, and Transactions 4

5 Headless REST, API Design, and Integrations 5

6 Java, Jakarta EE, Spring Boot, and Hibernate 5

7 React, JavaScript, TypeScript, HTML5, and CSS 6

8 Authentication, Authorization, and Enterprise Security 7

9 Performance, Scalability, Search, and Caching 7

10 CI/CD, Containers, Upgrades, and Operations 8

11 Testing, Quality, and Troubleshooting 9

12 Enterprise Workflows, Agile, and Architecture Leadership 9

13 Architect-Level Scenario Questions 10

14 Behavioral Interview and Final Revision 11

How to use this handbook

Study each answer as an explanation, not a script. Prepare one real project example for each topic, with architecture choices, trade-offs, failure modes and measurable outcomes. Confirm exact API availability and compatibility against the Liferay DXP 7.4 update or Community Edition GA build used by the employer.

Interview method: definition → design decision → implementation → risks → production example.

1 Liferay DXP 7.4: Platform and Architecture

Q1. Explain Liferay DXP architecture and the request lifecycle.

Liferay DXP is a modular Java digital-experience platform built around an OSGi runtime, portal kernel services, persistence, permissions, search, and a presentation layer. A request first reaches the web server or load balancer, then the servlet container and portal filters. Virtual-host and site resolution establish the company, group, layout, and user context. MVC portlets or headless endpoints invoke application services, which enforce permissions and transactions before accessing persistence or external systems. The response is rendered as a portlet fragment, page, or JSON. In an interview, distinguish portal concerns (identity, sites, permissions, content) from domain concerns that belong in separate modules or services.

Architecture / implementation illustration

```
Browser -> Proxy/TLS -> Tomcat -> Portal filters -> Site/Layout -> Portlet or REST -> Service layer ->
DB/Search/External API
```

Q2. What are the differences among a site, organization, user group, role, and asset library?

A site is a collaboration and content-delivery boundary with pages, content, and membership. An organization models organizational hierarchy and may have an associated site. A user group is a reusable collection of users for assigning membership and personal-site experiences. Roles represent permissions: regular, site, and organization roles have different scopes. Asset libraries support shared content across sites while maintaining governance. Design permissions by resource scope and least privilege rather than assigning administrator roles to simplify development. For multi-tenant systems, remember that a site is not a hard security or infrastructure isolation boundary.

Q3. How do you choose between a portlet, client extension, headless integration, and separate microservice?

Use an MVC portlet when the experience is tightly coupled to portal pages and server-side Liferay services. Use a client extension for supported decoupled UI or integration customization without deploying code into the portal JVM. Use headless APIs when a React or mobile client needs platform capabilities through HTTP. Use a microservice for an independently scaled business domain, long-running computation, or separate release lifecycle. Evaluate transaction boundaries, permission ownership, latency, deployment risk, and operational complexity. Avoid splitting a small CRUD feature into multiple microservices without a measurable reason.

Q4. What changed in Liferay 7.4 that matters for implementation?

Liferay 7.4 emphasizes modern content authoring, fragments, Objects, headless APIs, and modular extension strategies. Exact capabilities and support differ across DXP update levels and Community Edition GA builds; always establish the target build before selecting APIs or features. DXP subscription features and CE availability must not be assumed equivalent. When migrating, inspect release notes and breaking changes, verify module compatibility, rebuild themes, and test customizations against the exact target update.

2 OSGi, Declarative Services, and Module Design

Q5. What is OSGi and how do bundles differ from traditional JARs?

OSGi provides a dynamic module system in which bundles have explicit metadata, class-loading boundaries, and a lifecycle. Unlike a traditional flat classpath, a bundle imports packages and exports only intended APIs. This reduces accidental coupling but exposes version and package-resolution issues. A bundle can be installed yet unresolved because a package import is missing or its version range is incompatible. Prefer narrow API modules, implementation modules, and clear service contracts.

Q6. Explain Declarative Services annotations and component activation.

An @Component class is registered by the DS runtime when its required dependencies are satisfied. @Reference injects an OSGi service, typically by service type and optional target filter. @Activate and @Modified handle initialization and configuration updates; @Deactivate releases resources. A component can be unsatisfied when its reference has no matching provider, even though its bundle is ACTIVE. Diagnose both bundle state and component state rather than treating them as the same thing.

Architecture / implementation illustration

```
@Component(service = GreetingService.class)
public class GreetingServiceImpl implements GreetingService {
@Override public String greet(String name) {
return "Hello, " + name;
}
}
```

Q7. How do you troubleshoot an unresolved or unsatisfied OSGi component?

First check the bundle state and unresolved imports with Gogo shell commands such as lb and diag (availability varies by runtime). Then inspect DS components using scr commands and check logs for unsatisfied references. Compare exported and imported package versions, Gradle dependencies, and bnd metadata. Ensure required provider services are active, and confirm that configuration PIDs and target filters match. Avoid copying dependency JARs into bundles indiscriminately; embedded libraries can cause duplicate classes and class-space conflicts.

Q8. How should a large Liferay solution be modularized?

Separate contracts (API), business implementations (service), persistence generated by Service Builder, web MVC or REST modules, integration adapters, and common utilities only where justified. Export stable interfaces; keep implementation packages private. Apply dependency inversion so web modules call APIs rather than persistence internals. For a multi-team platform, define versioning rules, backward compatibility, module ownership, and release validation. Minimize direct use of portal internal packages that can change between updates.

3 MVC Portlets and Front-End Integration

Q9. Explain the MVC portlet lifecycle and common commands.

A portlet processes action requests for state-changing interactions, render requests for markup, and resource requests for AJAX or file-like responses. In Liferay MVC, MVCRenderCommand, MVCActionCommand, and MVCResourceCommand split these responsibilities. Actions should validate input, enforce permissions, invoke services, and redirect or set render parameters. Resource commands should return explicit content types and appropriate HTTP-like error semantics within the portlet framework. Do not implement business logic in JSP files.

Q10. How do you protect a portlet from CSRF, XSS, and insecure direct object references?

Use portal-generated action URLs and authentication tokens rather than constructing privileged action URLs manually. Escape output in JSP and sanitize user-supplied rich HTML according to the intended content policy. On every server-side resource access, check the caller's permission for the specific entity and action; a hidden button is not authorization. Validate IDs, uploads, and request parameters. Restrict resource commands to appropriate methods and avoid exposing sensitive information in logs or browser responses.

Q11. How can React be integrated with Liferay DXP?

Options include React within a portlet, a supported front-end client extension, or a separately hosted SPA consuming headless APIs. For embedded experiences, mount React into a known DOM root and clean up on navigation or unmount. For decoupled SPAs, define authentication and CSRF behavior, CORS policy, API versioning, error handling, and deployment ownership. Avoid relying on undocumented portal globals. Use TypeScript and an API client layer to isolate endpoint contracts.

Q12. What is the difference between fragments, themes, and widgets?

Fragments are reusable page-building components for content editors; themes provide global page styling and layout conventions; widgets/portlets implement interactive application behavior. Prefer fragments for presentation and editorial composition, not for complex business authorization or persistence. Themes should be as lightweight as possible to reduce upgrade friction. For a business application, a widget or client extension is generally more appropriate than embedding extensive application logic into a fragment.

4 Service Builder, Persistence, and Transactions

Q13. What does Liferay Service Builder generate?

Service Builder uses service.xml to generate model interfaces, persistence infrastructure, local-service contracts, and optionally remote-service scaffolding. Developers place business logic in implementation classes and run the build task to regenerate artifacts. Local services are internal Java APIs and are not automatically permission-checked; remote exposure requires explicit authorization design. Do not modify generated files manually because regeneration overwrites them.

Q14. Differentiate LocalService and remote Service in Liferay.

LocalService is intended for in-process calls and often participates in transaction boundaries. Remote Service methods may be exposed to external clients depending on configuration and framework, and must enforce permissions and validation. Never assume generated remote methods are safe simply because the caller is authenticated. Use resource permission checks and audit privileged operations. Consider headless REST endpoints as the preferred external contract when a modern API is required.

Q15. Explain entity finders, custom SQL, and indexing.

Generated finders handle common predicates defined in service.xml. Use custom SQL or DSL-Query for queries that are awkward to express with generated finders, subject to supported APIs on the target build. Add database indexes for selective filters and sort patterns, and inspect query plans before optimizing. Search indexes are appropriate for text relevance and faceted discovery, but are not a transactional source of truth. Keep the database and search index synchronized and plan reindexing for schema changes.

Q16. How do transactions work across Liferay and Spring Boot services?

A local service transaction can atomically commit changes to its own transactional database, but an HTTP call to a Spring Boot service is not part of that transaction. Avoid holding database transactions open while waiting on remote calls. For cross-service workflows, use an outbox pattern, idempotent consumers, compensating actions, or a saga. Document consistency guarantees explicitly: eventual consistency is acceptable for some notifications but not for all financial or entitlement operations.

5 Headless REST, API Design, and Integrations

Q17. How do you design a secure Liferay headless API?

Define resource-oriented endpoints, request and response schemas, validation, paging, filtering, sorting, and consistent error representations. Authenticate using a supported mechanism such as OAuth 2.0, then authorize each resource operation against business and portal permissions. Expose only necessary fields; avoid leaking internal entity identifiers when they reveal sensitive relationships. Include correlation IDs, rate limits where applicable, and contract tests. Use versioned APIs or compatibility rules to avoid breaking consumers.

Q18. REST versus GraphQL: when would you choose each?

REST is straightforward for resource operations, HTTP caching, monitoring, and conventional access-control policies. GraphQL can reduce over-fetching and combine related data in one request, but introduces query-cost control, resolver-level authorization, and caching complexity. Liferay supports headless capabilities whose exact endpoints depend on build and licensing. Choose based on client needs, operational maturity, and API governance rather than popularity.

Q19. How do you integrate Liferay with a Spring Boot backend?

Choose the system of record for each domain and avoid dual writes. Let Liferay handle identity, content, portal permissions, and experience composition when appropriate; let Spring Boot own independent domain workflows. Use OAuth 2.0/OIDC or another approved service-authentication model, enforce TLS, validate tokens, and implement timeouts, retries with jitter, circuit breakers, and idempotency. Propagate trace and correlation identifiers. Avoid forwarding portal session cookies as an ad hoc service-to-service security mechanism.

Q20. What makes an API idempotent and why does it matter?

An operation is idempotent if repeated equivalent requests have the same intended effect as one request. GET and PUT are defined as idempotent by HTTP semantics; POST is not automatically idempotent but can be made so using a client-generated idempotency key stored with the outcome. This matters when gateways retry after timeouts and the caller cannot know whether a write succeeded. Design a key scope, expiry, concurrency control, and response replay policy.

6 Java, Jakarta EE, Spring Boot, and Hibernate

Q21. Which Java features matter most for enterprise development?

Understand generics, collections, concurrency, streams, Optional, exceptions, records where supported, and modern JVM memory behavior. Know the distinction between equals and hashCode, thread safety, immutable objects, and the cost of blocking operations. For production diagnosis, be comfortable with thread dumps, heap dumps, GC logs, and profiling. Align the JDK with the supported Liferay DXP update and Spring Boot version rather than assuming the newest JDK is supported.

Q22. Explain dependency injection and bean lifecycles in Spring Boot.

Spring manages beans in an application context, resolves dependencies through constructors or other injection points, and applies configuration and post-processors. Constructor injection makes required dependencies explicit and facilitates unit testing. Be aware that singleton-scoped beans are shared and must not hold per-request mutable state. @Transactional typically works through proxies; self-invocation can bypass proxy advice. Use @ConfigurationProperties for validated typed configuration and separate secrets from source code.

Q23. What is the N+1 query problem in JPA/Hibernate?

N+1 occurs when one query loads a list of parent entities and additional queries load associated records for each parent. It can cause severe latency under real data volumes. Detect it with SQL logging, metrics, and representative integration tests. Resolve with explicit fetch joins, entity graphs, DTO projections, or batched fetching as appropriate. Avoid switching all relationships to EAGER because that can cause over-fetching and cartesian-product issues. Always measure the resulting query plan.

Q24. Explain optimistic and pessimistic locking.

Optimistic locking uses a version column and checks that the record has not changed when committing an update; conflicting writers receive an exception and must retry or report a conflict. Pessimistic locking obtains database locks earlier to serialize conflicting work. Optimistic locking suits mostly non-conflicting updates; pessimistic locking can be appropriate for highly contended resources but increases deadlock and waiting risks. Keep transactions short and define retry behavior deliberately.

Q25. How would you design exception handling in Spring REST APIs?

Separate expected domain errors (validation, not found, conflict, forbidden) from unexpected infrastructure failures. Use @RestControllerAdvice to map exceptions to stable error codes and appropriate HTTP statuses. Return safe messages, correlation IDs, and field-level validation errors without exposing stack traces or SQL. Log exceptions once at the right boundary with structured context. Make client retryability explicit: a validation error should not be retried, while some transient failures may be.

7 React, JavaScript, TypeScript, HTML5, and CSS

Q26. Explain React rendering, hooks, and state management.

React re-renders components when props or state change. useState handles local state, useEffect synchronizes with external systems, and useMemo/useCallback can reduce specific expensive recalculations when profiling supports their use. Effects need correct dependency lists and cleanup to avoid stale requests and event listeners. Keep server data distinct from UI state; use a query/-cache layer for remote data when complexity warrants it. Prefer accessible semantic HTML over div-heavy custom controls.

Q27. How do you optimize a large React portal application?

Measure first using browser performance tools and React profiling. Reduce unnecessary renders through component boundaries, stable keys, memoization where effective, and virtualization for long lists. Code-split large routes, optimize bundles, compress assets, and cache safe API responses. For network-heavy pages, paginate and parallelize independent requests without overwhelming the backend. Avoid premature micro-optimizations that obscure code. Validate on lower-powered devices and slower networks.

Q28. Explain JavaScript event loop, promises, and async/await.

JavaScript runs application code on a main execution thread in typical browser contexts, while asynchronous operations schedule continuations through task and microtask queues. Promise

callbacks run as microtasks after the current stack clears, before the next task. async/await improves readability but does not make CPU-intensive work nonblocking. Use AbortController to cancel stale fetches, Promise.all for independent operations, and explicit error handling for rejected promises. A long synchronous computation still freezes the UI.

Q29. What are important accessibility and responsive design practices?

Use semantic headings, labels, buttons, landmarks, keyboard-operable controls, visible focus, sufficient contrast, and meaningful error messages. Test screen-reader flows and keyboard navigation, not only automated audits. For responsive layouts use flexible grids, CSS clamp and media queries, and content-driven breakpoints. Avoid hiding essential workflows on mobile. Tailwind is a utility CSS framework; ensure its styles do not conflict with portal or theme CSS, and scope custom components appropriately.

8 Authentication, Authorization, and Enterprise Security

Q30. Differentiate OAuth 2.0, OpenID Connect, JWT, and SAML.

OAuth 2.0 is an authorization framework for delegated access; OpenID Connect adds an identity layer and ID tokens. JWT is a token format, not an authentication protocol. SAML is an XML-based federation standard common in enterprise SSO. For SPA sign-in, use an approved OIDC Authorization Code flow with PKCE. Validate issuer, audience, signature, expiry, and relevant claims on APIs. Do not treat the presence of a JWT as proof that a request is authorized for a specific resource.

Q31. How should roles and permissions be modeled in a multi-site portal?

Use least privilege and resource-level checks. Define global administration separately from site and organization responsibilities. For domain entities, authorize operations against both role and ownership or tenancy context. Keep a permission matrix for create, view, update, delete, publish, approve, and export actions. Test negative cases and cross-site access. A Liferay site alone does not guarantee tenant-grade isolation if data services or search filters fail to enforce boundaries.

Q32. How do you protect APIs against common OWASP risks?

Enforce server-side authorization for every object and function, validate input, parameterize SQL, limit payload sizes, and constrain file uploads. Use TLS, secure cookie attributes, CSRF protection where cookie-based authentication applies, and appropriate CORS restrictions. Keep secrets in managed storage and rotate credentials. Apply dependency scanning, secure logging, audit trails, and abuse protections such as throttling. Threat-model sensitive workflows rather than relying only on static scanners.

Q33. How do you manage secrets and environment configuration?

Separate nonsecret configuration from credentials. Use environment injection or a secrets manager with least-privilege access, rotation, and audit logs. Avoid committing tokens to Git or placing secrets in browser bundles, portal pages, or verbose logs. For production, prefer workload identities or short-lived credentials where supported. Define configuration validation and safe startup failure behavior. Ensure backup, incident-response, and credential-revocation procedures are tested.

9 Performance, Scalability, Search, and Caching

Q34. How do you investigate a slow Liferay portal page?

Start with end-to-end timing: browser waterfall, proxy, servlet processing, portlet render time, database queries, search, and downstream API calls. Use APM traces and correlation IDs to identify the critical path. Check thread pools, JDBC pool saturation, cache misses, expensive

JSP/fragment rendering, and slow asset delivery. Reproduce with representative users and data. Apply the smallest measurable fix and compare p50, p95, and p99 latency before and after.

Q35. Explain cache layers and cache invalidation in enterprise portals.

Caching can occur in browsers/CDNs, reverse proxies, application caches, distributed caches, ORM second-level caches, and search indexes. Choose TTL and invalidation rules based on data volatility and authorization scope. Never share a user-specific cached response across principals. For distributed systems, invalidation events can arrive late, so design for temporary staleness. Monitor hit ratio, memory pressure, stampedes, and eviction. Use cache keys that include tenant and locale where relevant.

Q36. What does horizontal scaling require for Liferay?

A cluster needs consistent application versions, shared or coordinated content storage, appropriate session strategy, compatible search infrastructure, database capacity, and cluster-aware caching or messaging. A load balancer should perform health checks and support draining during deployments. Verify document library storage, background jobs, scheduled tasks, and index consistency. Sticky sessions can mitigate some issues but are not a substitute for understanding state management. Validate supported topology for the exact DXP release.

Q37. How do you plan database and Elasticsearch/OpenSearch capacity?

Estimate concurrent users, request rates, write amplification, document counts, index growth, query patterns, and retention. Size the database for transactional load and the search engine for indexing and retrieval; do not treat search as the authoritative datastore. Test reindex time and recovery procedures. Monitor query latency, shard health, heap, disk utilization, and slow logs. Supported search engine versions are tied to the Liferay release, so verify the compatibility matrix before upgrading.

10 CI/CD, Containers, Upgrades, and Operations

Q38. Design a CI/CD pipeline for Liferay modules and Spring Boot services.

A typical pipeline checks out code, runs static analysis and dependency scanning, compiles, executes unit and integration tests, packages immutable artifacts, and publishes signed or traceable builds. Deploy to a staging environment matching production, run smoke tests, and promote with approval gates as required. Keep database migrations versioned and backward compatible. Use environment-specific configuration outside artifacts. Add rollback criteria, observability checks, and release notes to the pipeline.

Q39. How do you approach a Liferay 7.4 upgrade?

Inventory current GA/update, JDK, database, search engine, custom modules, themes, client extensions, and integrations. Read the target release notes and supported compatibility matrix. Back up the database, document library, and configuration; rehearse the upgrade in an isolated clone. Compile custom modules, inspect API changes, execute database upgrade processes, reindex if required, and test key business journeys and permissions. Define a rollback strategy that restores a consistent snapshot rather than merely redeploying an old WAR.

Q40. How do you implement zero- or low-downtime deployment?

Use rolling or blue-green strategies when the application and database changes are compatible with mixed versions. Keep schema changes expand-and-contract: add new fields first, migrate readers and writers, then remove obsolete fields later. Drain connections, honor readiness probes, and monitor error rates and latency after release. Some Liferay platform upgrades involve database transformations that require a maintenance window; never promise zero downtime without validating the specific upgrade path.

Q41. What metrics and alerts should a production team monitor?

Track request throughput, error rate, p95/p99 latency, JVM heap and GC, CPU, thread pools, JDBC pool usage, database waits, search health, disk, queues, and external dependency failures. Monitor business indicators such as failed logins, publishing failures, and incomplete workflows. Build dashboards around service-level objectives rather than raw metric volume. Alerts should be actionable and include runbook links, ownership, and escalation rules.

11 Testing, Quality, and Troubleshooting

Q42. Explain the testing pyramid for a Liferay plus React plus Spring Boot solution.

Unit tests validate business rules and components quickly. Integration tests verify persistence, service wiring, security, and API contracts against realistic dependencies. End-to-end tests cover critical user journeys such as login, approvals, publishing, and role-sensitive actions. Avoid making every scenario a slow browser test. Use test fixtures and isolated environments, and run contract tests to detect breaking API changes. Include accessibility, performance, and security tests as cross-cutting quality gates.

Q43. How would you debug intermittent 500 errors after a deployment?

Correlate errors with deployment time, instance, request path, and trace ID. Inspect application and proxy logs, thread dumps, database pool state, and external dependencies. Compare environment configuration and artifact versions across nodes. Check OSGi components for unsatisfied references and look for missing schema migrations or class-loading issues. Roll back if customer impact exceeds the agreed threshold, then reproduce and add a regression test. Avoid restarting blindly because it destroys useful evidence.

Q44. How do you review code as a technical lead?

Review correctness, security, testability, transaction scope, performance risks, error handling, API compatibility, readability, and operational observability. Focus on concrete failure modes rather than stylistic preferences already covered by automated formatting. Encourage small pull requests, meaningful test evidence, and documented architectural decisions. For critical changes, request threat-model or load-test evidence. Use mentoring comments that explain why a change matters and offer an actionable alternative.

12 Enterprise Workflows, Agile, and Architecture Leadership

Q45. How do you model a multi-stage approval workflow?

Define states, transitions, actors, permissions, escalation, timeout behavior, audit history, and notification semantics. A workflow should handle rejection, resubmission, delegation, and concurrent edits. Use the platform workflow engine where it fits content and asset lifecycles; use a domain workflow service when processes span multiple systems and need independent orchestration. Store durable state and make side effects idempotent. Test all invalid transitions and authorization boundaries.

Q46. How do you gather requirements for a customer-facing enterprise portal?

Identify user personas, journeys, integrations, compliance obligations, accessibility needs, content ownership, nonfunctional requirements, and measurable success criteria. Separate business goals from proposed technical solutions. Map functional requirements to domain boundaries and acceptance tests. Define service-level objectives, availability expectations, peak load, data retention, and disaster recovery. Run discovery workshops and validate assumptions with prototypes before committing to an architecture.

Q47. What is your approach to architectural trade-offs and ADRs?

Write concise architecture decision records with context, options, decision, rationale, consequences, and revisit triggers. Compare choices using security, scalability, maintainability, cost, delivery time, and team capability. For example, choose between an embedded Liferay module and a separate microservice based on independent scaling and ownership rather than a blanket microservices preference. Record risks and mitigation plans, then validate the decision with measurable outcomes.

Q48. How do you lead an Agile team through a difficult release?

Make the release goal and exit criteria explicit. Identify critical-path dependencies, establish code freeze or change control appropriate to risk, and triage defects by customer impact. Keep stakeholders informed with evidence-based status and clear decisions. Run rehearsals, verify rollback, and assign incident roles. After release, hold a blameless retrospective focused on systemic improvements: test gaps, ambiguous ownership, missing telemetry, or unsafe deployment steps.

13 Architect-Level Scenario Questions

Q49. Scenario: Design a B2B wholesale portal on Liferay.

Use Liferay for buyer and dealer experiences, identity, sites, content, and governance. Model accounts, price lists, catalogs, inventory, quotations, orders, and approvals in the commerce/-domain layer appropriate to the chosen edition and capabilities. Define ERP integration with asynchronous events for stock and order status and synchronous APIs only where necessary. Enforce account-specific price visibility, tax rules, and auditability. Add search, pagination, caching, and robust failure handling. Validate whether native Liferay Commerce is supported in the exact licensed/CE distribution rather than assuming parity.

Q50. Scenario: A React page is fast locally but slow in production.

Compare browser waterfalls, bundle sizes, cache headers, CDN behavior, and API latency. Determine whether time is spent in JavaScript parsing, hydration/rendering, backend calls, or authentication redirects. Inspect production source maps and profiling evidence. Check whether an API performs N+1 queries or a cross-region call. Apply targeted fixes: code splitting, caching, pagination, DB indexing, or reducing sequential calls. Re-test under representative network and concurrency conditions.

Q51. Scenario: Two portal tenants can see each other’s records. What do you do?

Treat this as a security incident. Contain exposure by disabling the affected endpoint or feature, preserve logs and evidence, and notify the incident owner according to policy. Trace tenant context through authentication, service methods, persistence queries, search filters, and caches. Fix authorization at the server-side data-access boundary, not only in the UI. Add cross-tenant negative tests, review other endpoints for the same defect, and follow the organization’s disclosure and remediation process.

Q52. Scenario: A downstream service times out during an approval workflow.

Define whether the approval state is committed before the remote action. Prefer a durable event or outbox so approval and event publication are coordinated within one local transaction. A worker retries transient failures with bounded exponential backoff and an idempotency key. Expose a pending or failed integration status rather than silently losing the action. Alert on dead-letter events and provide a safe manual replay path. Avoid keeping a database transaction open across the remote call.

Q53. Scenario: You must modernize a monolithic Liferay solution.

Begin with an application inventory, dependency map, traffic baseline, and pain-point analysis. Extract independently owned capabilities incrementally behind stable APIs, preserving business behavior and permission checks. Use a strangler pattern for selected domains while keeping content and identity integration coherent. Establish contract tests, shared observability, and deployment automation. Measure the modernization by release lead time, incident rate, latency, and cost, not by number of microservices created.

14 Behavioral Interview and Final Revision

Q54. How should you introduce yourself for this senior role?

Structure a 90-second answer around enterprise portal architecture, Java/Liferay delivery, integration experience, leadership, and measurable business outcomes. Mention React and Spring Boot as complementary technologies, not disconnected keywords. Explain your role in architecture decisions, production ownership, and mentoring. Close with why the job fits your strengths. Replace general claims with real project metrics you can defend; do not invent team size, revenue impact, or performance improvements.

Q55. How do you answer a question about a major production failure?

Use STAR: situation, task, action, result. Describe customer impact, how you narrowed the fault domain, mitigation or rollback, root cause, and the preventive changes. Distinguish what you personally did from team efforts. Include measurable recovery time or incident reduction only if verified. Strong senior answers demonstrate calm prioritization, communication, and learning rather than blaming a developer or vendor.

Q56. How should you handle a question you cannot answer?

Clarify the version and constraints, state what you know, and distinguish it from what you would verify. Explain your investigation approach: official documentation, compatibility matrix, a reproducible proof of concept, and logs or metrics. Avoid inventing method names or platform capabilities. Interviewers often value systematic reasoning and honest technical boundaries more than a memorized but inaccurate answer.

30-minute rapid revision checklist

1. Explain OSGi bundle resolution versus DS component satisfaction.
2. Explain MVC action/render/resource command responsibilities and permission checks.
3. Differentiate Service Builder local service, remote service, and headless REST.
4. Design a secure React–Liferay–Spring Boot integration using OAuth/OIDC.
5. Describe database transactions, N+1, locking, and cross-service consistency.
6. Demonstrate p95 latency investigation, JVM diagnosis, caching, and search capacity.
7. Describe a tested upgrade plan, CI/CD gates, rollback, and observability.
8. Prepare three STAR stories: delivery success, production incident, technical leadership.

Version-sensitive topics to verify

- Exact DXP 7.4 update or Liferay Portal CE 7.4 GA version and Java compatibility.
- Availability of Objects, Commerce, client extensions, headless APIs, and enterprise-only capabilities.
- Supported database and Elasticsearch/OpenSearch versions and upgrade procedure.
- Deployment topology, license, identity provider, CI/CD platform, and React version.

This handbook is a conceptual preparation resource, not a substitute for release-specific vendor documentation.