

Citi BRCC Project – 1-Hour Interview Q&A; (Abhishek Tiwari)

Prepared for: Abhishek Tiwari | Role: Software Engineer (Backend)

Agenda (1 hour)

- Project Overview & Context (5–7 min)
- Architecture & Design (8–10 min)
- APIs & Data Layer (8–10 min)
- PPT Automation (Apache POI) (7–8 min)
- Performance & Reliability (6–8 min)
- Security & Compliance (5–7 min)
- Testing & DevOps (5–6 min)
- Behavioral & Ownership (5–6 min)

1) Project Overview & Context

Q: What is BRCC and what problem does it solve?

A: BRCC (Business Risk & Compliance Controls) centralizes compliance data across business units and automates generation of audit-ready reports (including PPT decks), reducing manual effort and enabling timely compliance reviews.

Q: What was your role on the team?

A: Backend engineer building Spring Boot REST APIs, data transformation modules, and dynamic PowerPoint generation utilities; I also did performance tuning, secure integration, and production support.

Q: Who were the key users and what did they need?

A: Risk & compliance analysts and leadership. They needed accurate, standardized, and quickly generated reports with drill-down summaries and export-ready PPTs.

Q: What were your top deliverables?

A: (1) Optimized REST APIs for compliance data, (2) automated PPT generation with Apache POI, (3) performance improvements—pagination, projections, caching, and streaming responses.

Q: What measurable impact did you deliver?

A: Reduced manual PPT effort from hours to ~1–2 minutes per report; cut API latency by ~30–60% on heavy endpoints through query tuning and pagination.

—

2) Architecture & Design

Q: Describe the high-level architecture.

A: Layered Spring Boot app: Controller → Service → Repository. JDBC/SQL for data access, POI utilities for PPT, security with token/role checks, logging via SLF4J/Logback, CI/CD via Jenkins, Git/Bitbucket.

Q: How did you structure packages?

A: controller, service, repository, config, dto, mapper, util/report (POI), security for filters/interceptors.

Q: Why Spring Boot over alternatives?

A: Rapid REST development, autoconfiguration, actuator/metrics, profile-based config, and strong ecosystem.

Q: How did you handle configuration across environments?

A: Spring Profiles (application-dev/qa/prod.yml), secrets via environment variables or vault and feature flags via config.

Q: How did you model core entities/data?

A: DTOs for API contracts, projections for query efficiency, and mappers to decouple DB schema from API payloads.

Q: How did you ensure POI logic didn't pollute business logic?

A: Encapsulated PPT in ReportService + PptxBuilder; services produce structured data; POI strictly handles rendering.

—

3) APIs & Data Layer

Q: Example of a key API you built?

A: GET /reports/{period}?unit=...&category=... → aggregates controls, computes KPIs, and returns JSON or initiates PPT export.

Q: How did you design DTOs?

A: Minimal, purpose-specific DTOs; no overfetching; used projections for SQL to match DTO fields directly where possible.

Q: How did you manage large responses?

A: Pagination (page, size), filters, and server-side sorting; exported reports via streaming to avoid loading entire file in memory.

Q: Any complex query optimization you did?

A: Replaced N+1 reads with joins + projections; added indexes on filter columns; limited SELECT columns; batched fetches for reference data.

Q: How did you handle errors in APIs?

A: @ControllerAdvice with structured error codes/messages, correlation IDs in logs, and granular 4xx/5xx mapping.

Q: Did you use JPA or JDBC?

A: JDBC with named params / template and sometimes JPA for simple cases; used projections to avoid heavy entity hydration.

Q: How did you handle timezone and number formatting?

A: Standardized to UTC at DB and service layer; consistent formatting in DTOs; POI formatting aligned to report specification.

—

4) PowerPoint Automation (Apache POI)

Q: How did you generate PPTs programmatically?

A: Built reusable utilities using Apache POI (XSLF); used templates with placeholders and filled slides with titles, charts, and tables.

Q: How did you populate charts and tables?

A: Predefined chart shapes in template; replaced data series via POI APIs; constructed tables row-by-row with conditional formatting.

Q: How did you ensure brand-consistent layouts?

A: Template-driven approach with master slides, fixed placeholders, and style constants (fonts, colors, sizes).

Q: How did you stream PPT downloads?

A: Generated to a ByteArrayOutputStream or temp file and streamed as application/vnd.openxmlformats-officedocument.presentationml.presentation with appropriate headers.

Q: How did you avoid memory spikes with POI?

A: Reused objects, closed resources immediately, split very large decks into sections, and streamed output; avoided holding massive in-memory collections.

Q: How did you handle special characters or long text in slides?

A: Sanitized content, wrapped text, truncated with ellipsis where necessary, and used table column width management.

Q: What were the biggest pitfalls with POI?

A: Chart XML complexity, memory usage on large decks, and template corruption if edited improperly—solved with strict template control and utility wrappers.

—

5) Performance & Reliability

Q: What were your top performance wins?

A: Pagination + projections, index tuning, caching static reference data, and streaming large file responses.

Q: How did you profile/measure performance?

A: Actuator metrics, application logs with timing, SQL execution time logging, and load tests on key endpoints.

Q: How did you cache safely?

A: Cached read-only/reference data with TTL; avoided caching per-user sensitive data; invalidated on configuration changes.

Q: How did you handle bulk operations?

A: Batch queries and transformation pipelines; chunked processing where data volume was high.

Q: Any concurrency considerations?

A: Thread-safe POI usage (per-request instances), careful use of parallel streams for compute, and avoiding shared mutable state.

Q: How did you design for failure?

A: Timeouts/retries for integrative calls (if any), graceful fallbacks in PPT (error slide), transactional boundaries for data integrity.

—

6) Security & Compliance

Q: How was authentication/authorization handled?

A: Token-based security (client standard) with role checks at controller/service layer; secure endpoints and request validation.

Q: How did you protect against injection?

A: Parameterized SQL, input validation, sanitized content for PPT, and strict DTO mapping.

Q: How did you handle sensitive data in logs?

A: Redacted sensitive fields, correlation IDs instead of PII, structured logs via SLF4J/Logback.

Q: Any audit requirements?

A: Traceable logs for report generation and data access; included request IDs; stored metadata for regeneration when required.

Q: How did you ensure least privilege?

A: Role-based access controls at endpoints and service methods; minimal DB permissions for service accounts.

7) Testing & DevOps

Q: How did you test the APIs?

A: Unit tests for services and mappers (JUnit/Mockito), integration tests with in-memory DB/test containers, and contract tests for edge cases.

Q: How did you test PPT outputs?

A: Validated slide counts/titles, table contents, and presence of key shapes; checksum comparisons for templates; manual visual checks for charts.

Q: What was your CI/CD pipeline?

A: Jenkins for build/test/quality gates, Bitbucket for PR reviews and merges; artifacts deployed to client environments with approvals.

Q: How did you do feature toggling or safe releases?

A: Config-based toggles and backward-compatible DTO evolution; dark launches for risky features.

Q: Any production incident you handled?

A: Memory spike during peak report runs—resolved by streaming output, reducing in-memory aggregation, and adding back-pressure.

8) Behavioral & Ownership

Q: Biggest challenge and how you solved it?

A: Managing large PPT generation efficiently—abstracted POI, streaming response, and template hygiene; stabilized memory and speed.

Q: How did you work with non-technical stakeholders?

A: Converted requirements into data contracts and PPT templates; demoed increments; agreed SLAs and iterated based on feedback.

Q: How did you ensure quality and maintainability?

A: Layered design, DTO + mapper patterns, utility reuse for PPT, code reviews, and automated tests.

Q: Example of proactive improvement?

A: Introduced pagination and projections on heavy endpoints; reduced latency significantly and lowered DB load.

Q: What would you improve if given more time?

A: Add async job queue for very large report generations with user notifications, and template-driven configuration for new report types.

Deeper Technical Drill-Down (If Interviewer Probes)

Q: Show how you stream PPT in Spring.

A: Generate with POI into a ByteArrayOutputStream or temp file; return as ResponseEntity with proper content type and Content-Disposition for download.

Q: How to build chart slide via POI?

A: Use template slide with preset chart; access chart data via POI APIs, update categories and values, refresh chart cache.

Q: How to prevent OOM during export?

A: Avoid accumulating entire dataset; process sections; stream write; close slide/shape resources promptly; consider temp-file-backed streams for very large exports.

Q: How to ensure idempotency for report generation?

A: Include a request ID; cache/download link for the same request; avoid duplicate DB writes; log correlation.

Q: How to design an endpoint for long-running reports?

A: Submit job → return job ID → polling endpoint for status → download on completion; or server-sent events/notifications.

Q: DTO vs Entity—when and why?

A: DTOs are tailored for API contracts to avoid overfetching and leaking schema; entities are internal and should not be exposed.

Q: What metrics do you track?

A: API latency, throughput, error rates, report generation time, PPT size distribution, and DB query timings.

Q: How do you handle versioning of APIs?

A: URI (/v1, /v2) or content negotiation; maintain backward compatibility and deprecate old versions gradually.

Q: How to handle partial failures in PPT?

A: Insert an 'Error Summary' slide for sections that fail while keeping the rest complete; optionally return HTTP 206 with details.

—