🔹 Project Overview

BRCC (Business Risk & Compliance Controls) is an internal Citi platform designed to streamline **regulatory risk reporting**, automate **control assessments**, and generate **audit-ready documentation**. The system centralizes data from multiple Citi business units and produces standardized reports required for **risk governance**, **compliance checks**, and **internal/external audits**.

You worked on the backend modules responsible for **data processing**, **automated reporting**, and **dynamic PowerPoint generation**, which are critical for presenting compliance outcomes to leadership and audit teams.

🔹 Your Key Responsibilities & Contributions

1️⃣ Developed & Optimized REST APIs

- Designed, implemented, and optimized **RESTful services** using **Spring Boot** to support BRCC's core workflows.
- Built APIs for:
    - Fetching compliance data
    - Generating consolidated control summaries
    - Pushing processed data to front-end dashboards
- Improved API response time by applying:
    - Efficient SQL queries
    - Caching strategies
    - Pagination for large datasets

2️⃣ Automated Dynamic PowerPoint (PPT) Generation

One of your major contributions.

- Developed backend utilities that automatically generate **customized PowerPoint reports** based on:
    - Risk ratings
    - Compliance metrics
    - Control maturity indicators

- Implemented:
    - Dynamic slide creation
    - Chart/graph population
    - Table generation
    - File download endpoints
- Reduced manual PPT creation time for compliance teams from **hours to minutes**.

3️⃣ Data Transformation & Processing Pipelines

- Implemented logic to normalize, aggregate, and structure raw compliance data coming from various sources.
- Applied mapping and transformation rules to convert data into **audit-ready formats**.
- Ensured data accuracy by adding:
    - Validation checks
    - Schema constraints
    - Error-handling layers

4️⃣ Integration with Citi Internal Systems

- Integrated backend services with multiple internal Citi modules (with appropriate security controls).
- Worked with:
    - Token-based authentication
    - Secure APIs
    - Access-controlled endpoints
- Ensured full **compliance with Citi's security guidelines**.

5️⃣ Improved Code Quality & Performance

- Refactored legacy modules to follow clean coding practices.

- Added logging using **SLF4J + Logback** for full audit traceability.
- Implemented:
    - Multithreaded data processing
    - Efficient in-memory operations
- Helped reduce server load during peak report-generation cycles.

6️⃣ Collaboration & Agile Delivery

- Collaborated closely with:
    - Business analysts
    - Onsite leads
    - QA teams
- Participated in:
    - Sprint planning
    - Code reviews
    - Daily standups
- Ensured on-time delivery of features aligned with Citi's audit schedule.

🔹 Tools & Technologies Used

- **Java 8+**, **Spring Boot**, **REST APIs**
- **Maven**, **Git**, **Jenkins**, **Bitbucket**
- **SQL**, **JDBC**, **Stored Procedures**
- **Apache POI** (for dynamic PPT creation)
- **JSON**, **XML**
- **Agile/Scrum methodology**

🔹 Impact of Your Work

- Enabled compliance teams to generate audit presentations **10x faster**.
- Reduced manual reporting effort significantly.

- Improved system reliability and ensured data correctness.
- Helped Citi meet **regulatory deadlines** more efficiently.
- Delivered clean, optimized code that strengthened backend stability.

Actual contributions (Spring Boot backend, API design, data processing, and dynamic PPT automation).

🔹 **30–45 sec Elevator Pitch (Simple)**

"At LTIMindtree, I worked on Citi's BRCC (Business Risk & Compliance Controls) platform, which generates audit-ready risk and compliance reports used by leadership. I built and optimized Spring Boot REST APIs to fetch and transform compliance data from multiple sources, and I automated dynamic PowerPoint report generation using Apache POI—reducing manual effort from hours to minutes. I also improved performance with pagination, efficient SQL, and caching,

and ensured secure, token-based access to endpoints. Overall, I helped the team deliver faster, more reliable reporting aligned with audit deadlines."

🔹 **2–3 min Technical Deep Dive**

1) Problem & Context

- BRCC centralizes **risk & compliance data** across business units.
- Users needed **standardized, audit-ready reports** (especially PPT decks) with control scores, exceptions, and trends.

2) Your Scope

- **Backend APIs** in **Spring Boot**: data retrieval, aggregation, and export.
- **Dynamic PPT generation** (Apache POI) → charts/tables/slides.
- **Data validation & transformation**: converting raw domain data into compliant structures.

3) Architecture (High Level)

- **Client/UI** → **Spring Boot APIs** → **Service Layer** → **Repository Layer (SQL)**.
- **Data Flow**: DB/JDBC → Service transformations → POI utility → secure file download.
- **Security**: Token-based (e.g., OAuth/JWT/Session token as per Citi standards), role checks at controller/service level.
- **Operational**: Logging via SLF4J/Logback; config via Spring Profiles; CI/CD through Jenkins & Bitbucket.

4) Key Technical Decisions

- **Pagination** and **projection queries** to handle large datasets (avoid N+1 and heavy joins).
- **Caching** (where safe) for reference data (e.g., control catalogs).
- **Streaming file responses** for PPT downloads to reduce memory spikes.
- **POI Templates**: Used pre-defined slide templates and filled placeholders to ensure consistent branding & layout.

5) PPT Automation (Example)

- For each report request:

- Query: controls + risk ratings + exceptions.
- Transform: calculate KPIs (e.g., % compliant, # open exceptions).
- Generate PPT:
  - Title + summary slide
  - Trend charts (e.g., category-wise compliance)
  - Detailed tables with conditional formatting (e.g., red for high risk)
- Return: stream application/vnd.openxmlformats-officedocument.presentationml.presentation.

6) Performance & Reliability

- **Reduced PPT generation time** from **~10–15 min manual** to **~1–2 min automated**.
- **API response time** improvements using:
  - **SELECT with projections**, indexed filters
  - **Batch fetch** where applicable
  - **DTO mapping** to avoid entity bloat
- **Resilient error handling**: user-friendly status codes and traceable logs.

🔹 **Common Interview Q&A (With Crisp Answers)**

Q1. How did you ensure the system is performant with large datasets?

- Used **pagination** on list APIs (page/size), **indexed queries**, and **projections** to fetch only needed columns. Cached frequently used reference data. For PPT, generated sections incrementally and streamed the response.

Q2. How did you design the PPT generation to be maintainable?

- Abstracted a **PPTService** with methods like addTitleSlide, addChartSlide, addTableSlide. Used **templates** with placeholders to keep layout consistent. Kept **business logic** in service layer; the **POI layer** is purely presentational.

Q3. What were the main challenges?

- **Memory management** during PPT creation (POI can be heavy) → solved using streaming, closing resources, and splitting very large exports.

- **Data correctness** → added validations and reconcilers; wrote unit tests for transformation rules.
- **Security** → enforced token validation and role checks; ensured no PII leaks in logs.

Q4. How did you handle errors in report generation?

- Standardized error responses (HTTP 4xx/5xx with codes). Logged **correlation IDs** for each request. In PPT flow, partial failures resulted in **graceful fallback** (e.g., skip a broken section and inject an error slide).

Q5. How did you test it?

- **Unit tests** for services and mappers (JUnit/Mockito).
- **Integration tests** for API + DB using test containers/in-memory DB.
- **Golden-file** tests for PPT: validate slide counts, titles, table data, and file format integrity.

Q6. How did you ensure data security?

- Token-based auth, role-based authorization, parameterized SQL, no sensitive data in logs, and validated all inputs. Followed client's secure coding checklist.

Q7. What metrics did you improve?

- **PPT generation time** → minutes to ~1–2 min.
- **API latency** → improved by ~30–60% on heavy endpoints.
- **Manual effort** → reduced significantly due to automation.

🔹 **Short Stories (STAR format)**

1) Performance bottleneck

- **S**: Large compliance datasets made report APIs slow.
- **T**: Optimize without affecting accuracy.
- **A**: Added pagination, projections, tuned indexes, and cached reference data.
- **R**: Reduced latency by ~40% on peak endpoints.

2) Memory spikes in PPT

- **S**: Long reports caused memory issues.
- **T**: Make PPT generation stable.

- **A**: Switched to streaming response; closed POI resources promptly; split very large decks.
- **R**: Stable generation with lower memory footprint, no OOM in load tests.

🔹 **One-Liners for Specific Tech**

- **Why Spring Boot?** Rapid REST development, actuator for health, easy config via profiles.
- **Why Apache POI?** Fine-grained control over PPTX (charts, tables), supports enterprise templating.
- **DB efficiency?** Projection queries, indexed columns, pagination, minimal joins where possible.

If you want, I can **rehearse with you** in mock Q&A style—ask me anything the interviewer might, and I'll craft sharp, 20–30 sec answers tailored to your profile.