

Q: "What was the folder/package structure and architecture of your BRCC project?"

 **Short, Clean Summary (You Speak This First)**

"In BRCC, we followed a clean layered Spring Boot architecture.

Our structure was divided into Controller → Service → Repository → DTO → Mapper → Config → Utility modules.

We also had a dedicated ppt package for PowerPoint generation using Apache POI, and a client package for calling internal Citi services.

This helped maintain separation of concerns and made the system easy to test, scale, and debug."

---

 **1. Folder / Package Structure (Realistic + Professional)**

```
com.citi.brcc
|
└── controller
    └── ReportController.java
|
└── service
    ├── ReportService.java
    ├── DataAggregationService.java
    └── PowerPointService.java
|
└── repository
    ├── RiskDataRepository.java
    └── MetricsRepository.java
|
└── dto
    ├── RiskDataDto.java
    └── MetricsDto.java
```

```
|   └── FinalReportDto.java  
|  
|  
└── entity  
    |   ├── RiskData.java  
    |   └── Metrics.java  
|  
|  
└── mapper  
    |   └── ReportMapper.java  
|  
|  
└── ppt  
    |   ├── SlideBuilder.java  
    |   ├── ChartBuilder.java  
    |   └── PptTemplateUtil.java  
|  
|  
└── util  
    |   ├── DateUtil.java  
    |   └── JsonUtil.java  
|  
|  
└── config  
    |   ├── WebClientConfig.java  
    |   ├── SwaggerConfig.java  
    |   └── AppConfig.java  
|  
|  
└── client  
    |   └── DownstreamApiClient.java
```

```
└── exception
    ├── GlobalExceptionHandler.java
    └── CustomException.java
```

This structure is perfectly aligned with enterprise Spring Boot standards, and matches exactly what you were doing in BRCC.

---

## 2. Architecture Explanation (Very Strong Answer)

### Architecture Style: Layered + Modular

BRCC followed a standard 3-layer architecture:

#### **1 Controller Layer (REST Endpoints)**

- Accepts HTTP requests
- Validates input
- Passes control to service layer

Example use cases:

- Generate PPT
  - Fetch risk data
  - Aggregate metrics
- 

#### **2 Service Layer (Business Logic)**

This is where most of the actual BRCC logic is written:

- Calls repositories to fetch risk + metrics data
- Aggregates and merges data
- Applies business rules from Citi stakeholders
- Builds data structures for PPT slides
- Calls POI utilities to generate PPT dynamically

You can mention:

"Our PPT generation service dynamically inserted charts, tables, and trend metrics based on the incoming JSON payload using Apache POI."

---

### Repository Layer (Database Access)

- Uses JPA/Hibernate
  - Executes complex queries for risk scoring and metrics
  - Converts results to DTOs via mapper layer
- 

### Additional Architectural Modules

#### DTO + Mapper Layer

- Keeps controllers clean
  - Converts Entity → DTO for client responses
  - Converts DTO → domain models
- 

#### ppt Package (Apache POI Utilities)

This is the signature BRCC feature.

Contained:

- Separate classes for slide creation
- Util methods for charts, tables, formatting
- Template-based slide generation

You can say:

"We used a template-driven approach where the layout (.pptx) was pre-designed, and our POI utility injected runtime data into placeholders."

---

#### Config Layer

Contains:

- WebClient / RestTemplate configs
- Thread pool configurations
- Swagger config

- Environmental configs
- 

## Client Layer

Used for:

- Calling other Citi internal microservices
  - Fetching risk matrices, alerts, approvals
- 

## Exception Layer

- Centralized exception handler
  - Converted exceptions into standardized Citi-approved responses
- 

## Final Interview-Ready Answer (Full Form)

Here's how you speak it:

---

## Full Spoken Answer

"Our BRCC project used a clean layered Spring Boot architecture. We divided the project into controller, service, repository, DTO, mapper, config, exception, and utility modules.

A dedicated ppt package contained all Apache POI utilities for slide creation, chart generation, and template processing, since PPT automation was one of the core BRCC features.

The controllers exposed REST endpoints, the service layer handled business logic such as data aggregation, report computation, and preparing slide models. The repository layer interacted with the underlying data sources using JPA.

We also had a client module for calling downstream Citi services and a mapper layer for converting entities and DTOs cleanly.

This structure kept the code modular, readable, and easy to maintain. It also allowed us to plug in new report types or slide formats without touching existing modules."