PRO3 Exam Guide

Eduard Fekete

Character Count: 20437
Word Count: 2974

January 9, 2026

PRO3 Exam Questions, January 2026

1 Exam Notes and Tips From Joseph

One slide with bullet points (this is what I'll present) and maybe one picture to explain

For the other topics 1-6, open IntelliJ and show stuff there. Use your course Assignment a lot.

Only use assignment but you can refactor it.

IDE doesn't matter but it's better to use IntelliJ.

Ask for feedback but it will not be given broadly because it is a part of the exam.

Don't make any client, just use postman and bloomrpc.

Even for 1-6 prepare presentation with 2 slides, one bullet points and one maybe system architecture (or a couple slides but then switch to IntelliJ)

15 min talking

webclient and restclient from springboot, they also use http and grpc I guess

multiduplexing, duplex communication (HTTP/2)

deployment diagram for sep3

distributed system:

- pass messages
- several machines
- acting as a single system
- in a network

Joseph expects a happy ending

2 Distributed systems

Q: Define distributed systems. Give an overview of different distributed system architectures.

Full Answer:

2.0.1 15-Minute Speech Script

Intro (0-2 min): Start by defining a distributed system. I like the definition by George Coulouris: a system where hardware or software components located at networked computers communicate and coordinate their actions only by passing messages. The key is that these components appear to the user as a single, unified system. Mention why we use them: to share resources, improve performance, and ensure availability.

Core Characteristics (2-5 min): Explain the "Five Pillars" (or at least 3-4):

1. **Heterogeneity**: The system works across different OSs, languages, and hardware. (Mention how Java helps here with JVM).
2. **Scalability**: Can we add more users/resources? Horizontal vs. Vertical scaling (KIWI - Kill It With Iron).
3. **Fault Tolerance**: If one node fails, the system stays up. Redundancy is key.
4. **Transparency**: The user doesn't need to know where the data is (Location transparency) or that it's replicated (Replication transparency).

Architectural Models (5-12 min): This is the meat of the presentation.

1. **Client-Server**: The most common. Servers provide resources, clients request them. Great for centralized control but has a single point of failure.
2. **N-Tier (usually 3-tier)**: Presentation, Logic, and Data layers. Improves maintainability.
3. **Peer-to-Peer (P2P)**: No central authority. Every node is both client and server. Extreme scalability and fault tolerance (e.g., BitTorrent) but complex coordination.
4. **Microservices**: Breaking a monolith into small, independent services. Each has its own DB, communicates via REST or gRPC. Very popular now (mention project context if applicable).
5. **SOA (Service Oriented Architecture)**: Like microservices but usually more enterprise-focused with an "Enterprise Service Bus" (ESB).

Summary (12-15 min): Recap that there is no "one size fits all". The choice depends on the CAP theorem (Consistency, Availability, Partition Tolerance) and the specific needs of the project. Mention that in our assignments, we usually used a 3-tier approach with Spring Boot.

Slides Suggestion:

- Slide 1: Definition + 4 Icons for Pillars (Heterogeneity, Scalability, Fault Tolerance, Transparency).
- Slide 2: Diagram comparing Client-Server vs. P2P vs. Microservices.

3 Web Services client-server architecture (REST)

Q: Explain the ideas behind REST. Show some implementation details and how it works.

Full Answer:

3.0.1 15-Minute Speech Script

Intro (0-3 min): REST stands for REpresentational State Transfer. It's not a protocol (like HTTP), but an architectural style for building web services. It was defined by Roy Fielding in 2000. It relies heavily on HTTP as the underlying transport protocol.

Core Principles/Constraints (3-8 min):

1. **Client-Server**: Separation of concerns. The client handles UI/UX, the server handles data/logic.
2. **Stateless**: Each request from client to server must contain all the information necessary to understand the request. The server doesn't store session state between requests. This makes it very scalable.
3. **Cacheable**: Responses must define themselves as cacheable or not to improve performance.
4. **Uniform Interface**: This is the most important. We use URIs for identifying resources (e.g., `/users/1`). We use HTTP verbs (GET, POST, PUT, DELETE) to express actions.
5. **Layered System**: A client cannot tell whether it is connected directly to the end server or to an intermediary (like a load balancer or proxy).

Implementation Details (8-13 min): Talk about how we do this in Spring Boot:

- **@RestController**: Marks a class as a handler for web requests.
- **Request Mappings**: `@GetMapping`, `@PostMapping`.
- **DTOs (Data Transfer Objects)**: We don't expose our database entities directly; we use DTOs to define the interface with the client.
- **JSON**: The standard data format used for representation.
- **Postman**: Mention using Postman to test these endpoints.

Pros/Cons (13-15 min):

- **Pros**: Simple, highly scalable, widely supported, works over standard HTTP (firewall friendly).
- **Cons**: Can be "chatty" (many requests needed for one page), "over-fetching" (getting more data than needed), lacks a strict contract (though OpenAPI/Swagger helps).

Slides Suggestion:

- Slide 1: The 5 Constraints of REST (statelessness as the highlight).
- Slide 2: A table of Resource URIs vs. HTTP Methods (e.g., GET `/books` vs. POST `/books`).
- Slide 3: Screenshot of a simple Spring `RestController`.

4 gRPC architecture

Q: Explain the ideas behind gRPC. Show some implementation details and describe how it works.

Full Answer:

4.0.1 15-Minute Speech Script

Intro (0-3 min): gRPC is a modern, high-performance RPC framework developed by Google. It's often used for internal microservices communication. It's different from REST because it focuses on *actions* (procedures) rather than *resources*.

Key Technologies (3-8 min):

1. **HTTP/2**: Unlike REST (which usually uses HTTP/1.1), gRPC uses HTTP/2. This allows for:
   - **Binary framing**: Much faster than text/JSON.
   - **Multiplexing**: Many requests over a single connection.
   - **Streaming**: Supports client-side, server-side, and bi-directional streaming.
2. **Protocol Buffers (Protobuf)**: A language-neutral, platform-neutral mechanism for serializing structured data.
   - You define your messages and services in a `.proto` file.
   - This acts as a **strict contract** between client and server.

How it works / Implementation (8-13 min): Identify the workflow:

1. Define the service in `.proto`.
2. Generate code (using `protoc` or a build tool like Maven/Gradle plugin). This gives you the Base class for the server and the Stub for the client.
3. Server: Implement the generated interface and start the gRPC server.
4. Client: Use the "Stub" to call methods on the server as if they were local. Mention **BloomRPC** or **Postman** for testing (since they support gRPC now).

Comparison with REST (13-15 min):

- **gRPC**: Faster (binary), strict contract (Protobuf), supports streaming, better for microservices.
- **REST**: Easier to use for public APIs, browser-friendly, JSON is human-readable, better documentation (Swagger).

Slides Suggestion:

- Slide 1: Protobuf example (a message and a service definition).
- Slide 2: Diagram of the gRPC workflow (Proto -> Code Gen -> Client/Server).
- Slide 3: Comparison table (gRPC vs. REST).

5 Indirect Communication

Q: Explain reasons for indirect communication. Give examples of indirect communication. Define time and space uncoupling.

Full Answer:

5.0.1 15-Minute Speech Script

Intro (0-3 min): In direct communication (like REST or gRPC), the sender and receiver must both be online and know each other's addresses. Indirect communication breaks this dependency. The key concept here is "Uncoupling".

Uncoupling (3-8 min):

1. **Space Uncoupling**: The sender does not need to know who the receivers are (IP addresses, count, etc.). They just send to a middleman.
2. **Time Uncoupling**: The sender and receiver do not need to be online at the same time. The message is stored until the receiver retrieves it. This leads to better **Scalability** and **Reliability**. Even if a service is down, messages are queued up.

Examples of Indirect Communication (8-13 min):

1. **Message Queues (e.g., RabbitMQ, ActiveMQ)**: Point-to-point where one message is consumed by one receiver. Good for load balancing.
2. **Publish-Subscribe (Pub/Sub)**: One message is broadcast to many subscribers (e.g., Kafka themes, RabbitMQ Exchanges).
3. **Group Communication**: Messages sent to a group of processes (multicast).
4. **Shared Memory (Tuple Spaces)**: Not very common in enterprise now, but conceptually interesting.

Implementation (13-15 min): Mention using something like **RabbitMQ**. You have a **Producer**, an **Exchange**, a **Queue**, and a **Consumer**. The Producer "fires and forgets" the message to the Exchange. This is perfect for long-running tasks (e.g., generating a PDF) so the user doesn't wait for a response.

Slides Suggestion:

- Slide 1: Diagram showing Direct vs. Indirect Communication.
- Slide 2: Explaining Space and Time Uncoupling with a "Post Office" analogy.
- Slide 3: Simple RabbitMQ workflow (Producer -> Exchange -> Queue -> Consumer).

6 Middleware

Q: Explain the use of Middleware in distributed systems. Explain the ideas behind Inversion of Control and dependency injection. Describe some implementation details and how it works.

Full Answer:

6.0.1 15-Minute Speech Script

Intro (0-3 min): Middleware is "software that glue parts together". It sits between the User Applications and the Operating System/Network. Its main purpose is to mask the complexity of distribution (Transparency) and provide common services like security, message passing, and transaction management.

Types of Middleware (3-7 min):

- **Communication Middleware**: gRPC, REST, RabbitMQ.
- **Application Middleware**: Spring Boot, .NET Core.
- **Database Middleware**: Hibernate/JPA (mapping objects to relational data).

Inversion of Control (IoC) & Dependency Injection (DI) (7-12 min): This is a critical part of modern middleware (like Spring).

- **Inversion of Control**: Instead of the developer controlling the flow of the program (like `main()` creating every object), the *Framework* controls it.
- **Dependency Injection**: A specific form of IoC where the framework "injects" the required dependencies into a class.
- **Why?**: It makes code loosely coupled, easier to test (using mocks), and more maintainable.

Implementation in Spring Boot (12-15 min):

- **@Component / @Service / @Repository**: These tell Spring "hey, manage this object for me".
- **ApplicationContext**: This is the "Container" where all the beans live.
- **@Autowired / Constructor Injection**: This is how we ask Spring for a dependency. Explain that the "Bean" is just a Java object that is instantiated, assembled, and managed by the Spring IoC container.

Slides Suggestion:

- Slide 1: The "Layers" diagram (Application | Middleware | OS/Network).
- Slide 2: IoC/DI Concept (e.g., Comparing `new MyService()` vs. `@Autowired`).
- Slide 3: Diagram of the Spring Application Context / Bean Lifecycle.

7 Contract-based Communication

Q: Compare contract and convention-based communication. Explain the reasons for using contract-based communication. Show examples of both forms of communication.

Full Answer:

7.0.1 15-Minute Speech Script

Intro (0-3 min): When two services talk, they need to agree on a format. We can do this through **Contracts** or **Conventions**.

Convention-based (3-7 min):

- **Example**: REST with JSON.
- **Idea**: We "agree" that if I send a GET to `/users/1`, I'll get back a JSON object with `id` and `name`.
- **Problems**: If the server changes `name` to `fullName`, the client breaks. There is no automatic validation (in the code) until runtime. It's "loose".
- **Pros**: Flexible, easy to get started, human-readable.

Contract-based (7-12 min):

- **Example**: gRPC with Protobuf, SOAP with WSDL.
- **Idea**: We write a formal file (the `.proto` or `.wsdl`) that defines exactly what data types are allowed.
- **Code Generation**: We use this file to generate code. If the server doesn't follow the contract, it won't even compile (or the bridge will fail immediately).
- **Pros**: Type safety, better documentation (the file *is* the documentation), smaller payload size (binary), earlier error detection.

Why use Contracts? (12-15 min): In large-scale distributed systems (Microservices), contracts are essential. Teams can work independently because they have a "Shared Truth" (the contract file). You can use **OpenAPI/Swagger** to add a "pseudo-contract" to a REST API, providing some of the benefits of contract-based communication to an otherwise convention-based style.

Slides Suggestion:

- Slide 1: JSON vs. Protobuf comparison (Side-by-side).
- Slide 2: Workflow diagram (Contract File -> Code Gen -> Both sides use same types).
- Slide 3: Table comparing "Convention" vs "Contract" (Safety, Speed, Flexibility).

8 Encryption

Q: Compare symmetric and asymmetric encryption. Give examples of attacks against encryption.

Full Answer:

8.0.1 15-Minute Speech Script

Intro (0-3 min): Encryption is about providing **Confidentiality** (the C in the CIA triad). We transform "plaintext" into "ciphertext" so only authorized parties can read it.

Symmetric Encryption (3-7 min):

- **Definition**: Uses the **same key** for both encryption and decryption.
- **Example**: AES (Advanced Encryption Standard).
- **Pros**: Very fast, good for encrypting large amounts of data.
- **Cons**: "Key Distribution Problem" – how do you share the key securely? If the key is stolen, everything is compromised.

Asymmetric Encryption (7-12 min):

- **Definition**: Uses a **key pair**: a **Public Key** (can be shared with anyone) and a **Private Key** (must be kept secret).
- **Example**: RSA.
- **Mechanics**: Data encrypted with the Public Key can only be decrypted with the corresponding Private Key.
- **Pros**: Solves the key distribution problem.
- **Cons**: Much slower than symmetric.
- **Hybrid approach**: In the real world (like HTTPS), we use Asymmetric to exchange a Symmetric key, and then use that Symmetric key for the actual data transfer (Best of both worlds).

Attacks (12-15 min):

- **Man-in-the-Middle (MITM)**: Attacker sits between client and server, impersonating both to intercept traffic. This is why we need certificates!
- **Replay Attack**: Attacker captures a valid encrypted message (e.g., "Pay $100") and sends it again later. We solve this with nonces or timestamps.
- **Brute Force**: Trying every possible key.
- **Known-Plaintext Attack**: If the attacker has both the plaintext and the ciphertext, they can try to find the key.

Slides Suggestion:

- Slide 1: Symmetric vs. Asymmetric diagram (One key vs. Two keys).
- Slide 2: The "Hybrid" approach (Asymmetric for Handshake -> Symmetric for Session).
- Slide 3: Diagram of a MITM attack.

9 Digital signatures and certificates

Q: Explain what digital signatures are and how they are used with Certification Authorities. Describe how TLS/HTTPS works.

Full Answer:

9.0.1 15-Minute Speech Script

Intro (0-3 min): Digital signatures and certificates are the bedrock of trust on the internet. While encryption provides confidentiality, digital signatures provide **Authenticity** (knowing who sent it), **Integrity** (knowing it wasn't changed), and **Non-repudiation** (the sender can't deny sending it).

Digital Signatures (3-7 min):

- **How they work**: They use Asymmetric encryption but "in reverse".
- The sender hashes the message and encrypts that hash with their **Private Key**. This is the "Signature".
- The receiver decrypts the signature with the sender's **Public Key** to get the hash, then hashes the message themselves. If the hashes match, the signature is valid.
- **Why?**: Only the owner of the private key could have created that signature.

Certification Authorities (CAs) (7-11 min):

- **The Problem**: How do I know that "Alice's Public Key" actually belongs to Alice and not an attacker?
- **The Solution**: Digital Certificates issued by a CA.
- A certificate (X.509) binds a public key to an identity.
- The CA signs this certificate with *its* private key.
- Our browsers come with the public keys of "Root CAs" pre-installed. This creates a **Chain of Trust**.

How TLS/HTTPS Works (11-15 min): The "Handshake" process:

1. **Client Hello**: Client sends supported cipher suites and a random number.
2. **Server Hello**: Server sends its certificate and another random number.
3. **Verification**: Client checks the certificate with a Root CA.
4. **Key Exchange**: Client generates a "Pre-master secret", encrypts it with the server's public key (from the certificate), and sends it to the server.
5. **Session Keys**: Both compute a "Symmetric Session Key" from the random numbers and the pre-master secret.
6. **Encrypted Communication**: All subsequent traffic uses the symmetric key (for speed).

Slides Suggestion:

- Slide 1: Diagram of Digital Signature generation and verification.
- Slide 2: The "Chain of Trust" diagram (User -> Intermediate CA -> Root CA).
- Slide 3: Step-by-step TLS Handshaking flow.

10 Integrity and cryptographic hash functions

Q: Explain cryptographic hash functions and give examples of how they can be used to support data integrity. Give examples of attacks against cryptographic hash functions.

Full Answer:

10.0.1 15-Minute Speech Script

Intro (0-3 min): Integrity is about ensuring that data has not been altered during transit or storage. Cryptographic hash functions are the primary tool for this. A hash is like a "digital fingerprint".

What is a Hash Function? (3-8 min): It takes an input of any length and produces a fixed-length string of characters (the digest). Key properties:

1. **Deterministic**: Same input always equals same output.
2. **Fast**: Efficient to compute.
3. **Pre-image resistant**: Given a hash, it's impossible (computationally) to find the original input.
4. **Second pre-image resistant**: Given any input, it's impossible to find another input that produces the same hash.
5. **Collision resistant**: Impossible to find *any* two different inputs that produce the same hash.

Use Cases for Integrity (8-12 min):

1. **File Checksums**: Download a file and verify its hash (e.g., MD5 or SHA-256) to ensure it wasn't corrupted or tampered with.
2. **Password Hashing**: We never store passwords in plaintext. We store the hash (plus a **Salt** to prevent rainbow table attacks).
3. **Digital Signatures**: As mentioned before, we sign the hash of a message, not the whole message.
4. **Blockchain**: Each block contains the hash of the previous block, creating an immutable chain.

Attacks (12-15 min):

1. **Brute Force / Dictionary Attack**: Trying many inputs until one matches the hash (mostly for passwords).
2. **Rainbow Tables**: Using huge pre-computed tables of hashes to "reverse" a hash instantly.
3. **Birthday Attack**: A mathematical attack that exploits the "Birthday Paradox" to find collisions faster than pure brute force.
4. **Length Extension Attack**: (Optional mention) Some older hashes (like MD5 or SHA-1) are vulnerable to adding data to the end of a hashed message without knowing the secret.

Slides Suggestion:

- Slide 1: The "Black Box" of a Hash Function (Input -> [Hash Func] -> Fixed-length Digest).
- Slide 2: Table of common hash functions (MD5 - broken, SHA-1 - weak, SHA-256 - secure, SHA-3).
- Slide 3: Explaining a "Collision" (Two different inputs -> same output).