1. RESTful Architecture

Definition:

REST (Representational State Transfer) ही एक architectural style आहे जी APIs तयार करण्यासाठी वापरली जाते. यात communication HTTP methods वापरून होतं.

Main HTTP Methods:

GET → Data fetch करायला.

POST → New data create करायला.

PUT → Existing data update करायला.

DELETE → Data remove करायला.

Real Example:

Amazon app मध्ये → GET (view product), POST (add to cart), PUT (update address), DELETE (remove product).

2. Introduction to Spring REST

Definition:

Spring REST हा Spring Framework चा भाग आहे जो आपल्याला RESTful web services तयार करायला मदत करतो.

Features:

Annotations वापरून REST APIs build करता येतात.

कमी configuration लागतं.

JSON/XML responses सहज handle होतात.

Real Example: /users endpoint बनवलं तर ते थेट user data return करेल.

3. Benefits of Using Spring Boot for REST

1. Auto-configuration → कमी XML configuration लागतो.

2. Embedded Tomcat server → बाहेरून server install करण्याची गरज नाही.

3. Production-ready features → Actuator, health checks.

4. Less boilerplate code → कमी कोड लिहून API तयार होतं.

Real Example:

Spring Boot मध्ये फक्त काही lines मध्ये REST API तयार होतं, पण जुना Spring MVC वापरला तर XML config जास्त लागतं.

4. Setting up a Spring Boot Project

Steps:

1. start.spring.io ला जा.

2. Project Type → Maven / Gradle निवड.

3. Dependencies → Spring Web, Spring Boot DevTools निवड.

4. Project download करून IntelliJ/Eclipse मध्ये open कर.

5. Main class मधील SpringApplication.run() run कर → embedded Tomcat सुरू होईल.

5. What's New in Spring Boot 3

Java 17 minimum requirement.

AOT (Ahead of Time compilation) → app fast चालतो.

Jakarta EE 9 → javax.* packages आता jakarta.*.

Native image support → कमी memory usage, जलद performance.

📘 Controller in Spring REST

Definition

Controller हा एक Java class आहे जो HTTP requests handle करतो आणि response देतो.

👉 म्हणजे client ने GET, POST मारलं की ते controller methods कडे जातं.

Important Annotations

@RestController → class ला REST controller declare करतो.

@RequestMapping("/users") → base path define करतो.

@GetMapping → GET request handle करतो.

@PostMapping → POST request handle करतो.

@PutMapping → PUT request handle करतो.

@DeleteMapping → DELETE request handle करतो.

@PathVariable → URL मधून value घ्यायला.

@RequestParam → Query parameters घ्यायला.

@RequestBody → JSON → Java Object convert करायला.

Example Code

```
@RestController
@RequestMapping("/users")
public class UserController {
  // GET - fetch users
  @GetMapping("/all")
  public String getUsers() {
    return "All Users List";
  }
  // POST - add user
  @PostMapping("/add")
  public String addUser(@RequestBody String user) {
    return "User Added: " + user;
  }
  // PUT - update user
  @PutMapping("/update/{id}")
  public String updateUser(@PathVariable int id, @RequestBody String user) {
```

```
    return "User Updated with id " + id + ": " + user;
  }
  // DELETE - remove user
  @DeleteMapping("/delete/{id}")
  public String deleteUser(@PathVariable int id) {
    return "User Deleted with id " + id;
  }
}
```

📘 Interview Ready Short Explanation

👉 “Spring REST madhe controller HTTP requests handle karto. आपण annotations वापरतो जसे की @RestController, @GetMapping, @PostMapping इ. GET data fetch sathi, POST create sathi, PUT update sathi, DELETE remove sathi वापरतो. Spring Boot 3 मुळे configuration कमी लागतं आणि APIs पटकन तयार होतात.

🌟 Topics & Sub-Topics Notes

🔹 1. Building a Simple REST Controller

a) Creating a basic REST Controller

Definition: REST Controller हा एक Java class असतो ज्यात आपण annotations वापरून APIs तयार करतो.

Spring Boot मध्ये → फक्त @RestController लावलं की तो class REST APIs साठी ready होतो.

Example:

```
@RestController
public class HelloController {
  @GetMapping("/hello")
  public String sayHello() {
    return "Hello, REST API!";
  }
```

}

b) Defining Request Mappings

Definition: @RequestMapping किंवा short forms (@GetMapping, @PostMapping) वापरून आपण URL path define करतो.

म्हणजे client ने कोणता path hit केल्यावर कोणती method चालेल हे ठरतं.

Example:

@RequestMapping("/users")

public class UserController { }

c) Handling HTTP Methods (GET, POST, PUT, DELETE)

GET → Fetch data

POST → Create new data

PUT → Update existing data

DELETE → Remove data

Example:

@GetMapping("/all")

public List<User> getAllUsers() { ... }

@PostMapping("/add")

public String addUser(@RequestBody User user) { ... }

@PutMapping("/update/{id}")

public String updateUser(@PathVariable int id, @RequestBody User user) { ... }

@DeleteMapping("/delete/{id}")

public String deleteUser(@PathVariable int id) { ... }

d) Returning JSON responses

By default Spring Boot REST APIs return data in JSON format (because of Jackson library).

आपल्याला फक्त object return करायचं → ते auto JSON मध्ये convert होतं.

Example:

```
@GetMapping("/user")
public User getUser() {
   return new User(101, "Anurag", "Aurangabad");
}
```

👉 Response:

```
{
 "id": 101,
 "name": "Anurag",
 "city": "Aurangabad"
}
```

🔹 2. Request and Response Handling

a) Handling Path Variables and Query Parameters

Path Variable → URL मधून value घ्यायला.

```
@GetMapping("/users/{id}")
public String getUserById(@PathVariable int id) {
   return "User with id " + id;
}
```

Query Parameter → URL मधील ? नंतर parameter घ्यायला.

```
@GetMapping("/search")
public String searchUser(@RequestParam String name) {
   return "Searching user with name " + name;
}
```

👉 URL: /search?name=Anurag

b) Request Body and Form Data Processing

RequestBody: JSON data → Java Object मध्ये convert होतं.

```
@PostMapping("/add")
public String addUser(@RequestBody User user) {
   return "Added: " + user.getName();
}
```

Form Data: HTML form submit केल्यावर parameters handle करायला → @RequestParam.

c) Customizing Response Status and Headers

Spring मध्ये आपण ResponseEntity वापरून status code आणि headers control करू शकतो.

Example:

```
@GetMapping("/custom")
public ResponseEntity<String> customResponse() {
   return ResponseEntity
        .status(201) // Created
        .header("MyHeader", "TestValue")
        .body("Custom Response Created!");
}
```

👉 Output → status: 201, header + body.

d) Exception Handling in REST Controllers

Error आल्यावर custom response द्यायला @ExceptionHandler किंवा @ControllerAdvice वापरतो.

Example:

```
@RestControllerAdvice
public class GlobalExceptionHandler {
```

```
  @ExceptionHandler(Exception.class)
  public ResponseEntity<String> handleException(Exception e) {
    return ResponseEntity.status(500).body("Error: " + e.getMessage());
  }
}
```

✅ Interview Ready Short Lines

Basic REST Controller: “@RestController वापरून आपण REST APIs तयार करतो.”

Request Mapping: “@GetMapping, @PostMapping use करून आपण path आणि method define करतो.”

HTTP Methods: “GET fetch, POST create, PUT update, DELETE remove data.”

JSON Response: “Spring Boot मध्ये by default object auto JSON मध्ये convert होतं.”

PathVariable vs RequestParam: “PathVariable URL मधून value घेतो, RequestParam query parameter मधून value घेतो.”

RequestBody: “Client ने पाठवलेला JSON Java object मध्ये convert होतो.”

ResponseEntity: “Response customize करायला वापरतो – status code, headers, body.”

Exception Handling: “@RestControllerAdvice वापरून global error handling करू शकतो.”

🌟 RESTful Resource Representation with DTOs

1. Introduction to Data Transfer Objects (DTOs)

Definition: DTO म्हणजे एक plain Java object (POJO) जे फक्त data transfer करण्यासाठी वापरलं जातं, business logic ठेवत नाही.

Why? → Entities direct expose करू नयेत कारण त्यात sensitive data असू शकतं. DTO वापरून आपण फक्त आवश्यक fields client ला देतो.

Example:

```
// Entity
class UserEntity {
  int id;
  String name;
```

```
  String password; // हे expose करायचं नाही
}
// DTO
class UserDTO {
  int id;
  String name;
}
```

👉 म्हणजे Entity मध्ये सगळा data असतो, पण DTO मध्ये फक्त client ला लागणारा data जातो.

2. Mapping Entities to DTOs

Manual Mapping: setters/getters वापरून copy करणं.

Automatic Mapping: ModelMapper / MapStruct सारख्या libraries वापरणं.

Example:

```
UserDTO dto = new UserDTO();
dto.setId(entity.getId());
dto.setName(entity.getName());
```

3. Customizing JSON Serialization and Deserialization

Serialization: Java object → JSON

Deserialization: JSON → Java object

Annotations: @JsonIgnore, @JsonProperty, @JsonInclude वापरून customize करता येतं.

Example:

```
class UserDTO {
  private int id;
  @JsonProperty("full_name")
  private String name;
  @JsonIgnore
  private String password;
```

}

👉 आता JSON मध्ये name हे full_name म्हणून दिसेल आणि password field दिसणार नाही.

4. Managing Versioning and Backward Compatibility

कधी कधी API बदलतो (v1 → v2).

जुन्या clients ना error न देता नवीन version handle करायला लागतो.

आपण URL मध्ये किंवा request header मध्ये version ठेवतो.

Example:

/api/v1/users → जुना version

/api/v2/users → नवीन version (ज्यात DTO structure बदलू शकतो)

🌟 RESTful CRUD Operations

1. Implementing Create, Read, Update, Delete

Create (POST) → नवीन resource तयार करायला.

Read (GET) → existing resource fetch करायला.

Update (PUT/PATCH) → resource बदलायला.

Delete (DELETE) → resource काढून टाकायला.

Example (UserController):

@PostMapping("/users")

public String createUser(@RequestBody UserDTO user) { return "User Created"; }

@GetMapping("/users/{id}")

public UserDTO getUser(@PathVariable int id) { return new UserDTO(id, "Anurag"); }

@PutMapping("/users/{id}")

public String updateUser(@PathVariable int id, @RequestBody UserDTO user) { return "User Updated"; }

@DeleteMapping("/users/{id}")

public String deleteUser(@PathVariable int id) { return "User Deleted"; }

2. Utilizing HTTP methods for CRUD

POST → Create

GET → Read

PUT/PATCH → Update

DELETE → Delete

👉 Interview मध्ये सोपा line:

"CRUD operations directly map होतात HTTP methods वर."

3. Validating Input Data with Annotations

Spring Boot मध्ये आपण javax.validation annotations वापरतो.

Controller मध्ये @Valid वापरून validation enable करतो.

Example:

```
class UserDTO {
  @NotNull
  private String name;
  @Email
  private String email;
  @Size(min = 6)
  private String password;
}

@PostMapping("/users")
public String addUser(@Valid @RequestBody UserDTO user) {
   return "Valid User Added!";
}
```

---

4. Optimistic Locking for Concurrent Updates

जेव्हा multiple users एकाच resource ला update करत असतात तेव्हा data consistency टिकवायला Optimistic Locking वापरतो.

यासाठी @Version field वापरला जातो.

Example (Entity):

```
@Entity
class UserEntity {
  @Id
  private int id;

  private String name;

  @Version
  private int version; // used for optimistic locking
}
```

👉 म्हणजे दोन users एकाच वेळी update करत असतील तर version mismatch झाल्यास error येतो आणि data corrupt होत नाही.

✅ Interview Ready Short Lines

DTO: “DTO म्हणजे data transfer object – entities direct expose न करता फक्त आवश्यक fields client ला देण्यासाठी वापरतो.”

Entity → DTO Mapping: “Manual mapping karu शकतो किंवा ModelMapper वापरू शकतो.”

Serialization/Deserialization: “Jackson annotations वापरून JSON customize करू शकतो.”

Versioning: “API बदलल्यावर backward compatibility ठेवायला versioning वापरतो → /v1, /v2.”

CRUD: “Create (POST), Read (GET), Update (PUT), Delete (DELETE).”

Validation: “@NotNull, @Email सारख्या annotations वापरून input validate करतो.”

Optimistic Locking: “Multiple updates साठी @Version field वापरून conflicts handle करतो.

Anurag, तुला हवंय का मी हे सगळं एक mini-project example म्हणून तयार करून दाखवू (Entity + DTO + CRUD Controller + Validation + Error Handling)? म्हणजे तुला interview मध्ये दाखवायला perfect demo मिळेल.

1. RESTful HATEOAS (Hypermedia as the Engine of Application State)

👉 Definition:

HATEOAS म्हणजे REST API madhe resources sobat links provide karne, ज्यामुळे client ला पुढचं action kसं घ्यायचं हे API मधूनच समजतं.

म्हणजेच, API la self-discovery cha power milto.

👉 Key Points:

HATEOAS = Resource sobat hypermedia links.

Client ला फक्त response मधून पुढे कुठल्या URL ला call करायचं हे समजतं.

यामुळे documentation वर कमी depend karava lagto कारण API स्वतः guidance deto.

👉 Example (Spring Boot):

```
@GetMapping("/students/{id}")
public EntityModel<Student> getStudent(@PathVariable int id) {
   Student student = new Student(id, "Anurag", "CSE");
   return EntityModel.of(student,
WebMvcLinkBuilder.linkTo(WebMvcLinkBuilder.methodOn(this.getClass()).getAllStudents()).withRel("all-students"),

WebMvcLinkBuilder.linkTo(WebMvcLinkBuilder.methodOn(this.getClass()).getStudent(id)).withSelfRel()
   );
```

}

Response (JSON with Links):

{

"id": 101,

"name": "Anurag",

"branch": "CSE",

"_links": {

"self": { "href": "http://localhost:8080/students/101" },

"all-students": { "href": "http://localhost:8080/students" }

}

}

✅ Real-life Example:

Amazon REST API madhun order details fetch keletar response madhe “track-order”, “cancel-order”, “download-invoice” असे links येतात.

2. Content Negotiation & Media Types

👉 Definition:

Content Negotiation म्हणजे client cha request header pramane response format decide karane.

म्हणजेच ekach API JSON/XML donhi format madhe output deu shakto.

👉 Key Points:

Accept header vaparla jato (उदा. Accept: application/json किंवा Accept: application/xml).

Spring Boot automatically JSON & XML support deto (Jackson & JAXB vaprun).

Custom media types pan define karu shakto (उदा. application/vnd.company.app-v1+json).

👉 Example (Spring Boot):

```
@RestController
@RequestMapping("/products")
public class ProductController {
   @GetMapping(produces = {"application/json", "application/xml"})
   public Product getProduct() {
      return new Product(101, "Laptop", 60000);
   }
}
```

👉 Example Requests:

JSON Request:

```
GET /products
Accept: application/json
```

Response:

```
{
 "id": 101,
 "name": "Laptop",
 "price": 60000
}
```

XML Request:

```
GET /products
Accept: application/xml
```

Response:

```
<Product>
   <id>101</id>
   <name>Laptop</name>
   <price>60000</price>
```

</Product>

✅ Short Interview Points (तुला सांगायला easy होतील):

HATEOAS: Resource sobat links देतो → client ला पुढचं action समजतं.

Content Negotiation: Client cha Accept header pramane response JSON/XML format madhe yeto.

Real-life: Amazon order API (links for next actions), Gmail API (JSON/XML response).

1. Spring Boot Actuator for REST Monitoring

👉 Definition:

Spring Boot Actuator म्हणजे एक production-ready tool आहे जे application चं health, metrics, info, monitoring externally expose करतं.

म्हणजेच REST APIs चालू असताना ते कसे perform होत आहेत, error आहेत का, किती requests आले इ. माहिती मिळते

👉 Key Sub-points:

1. Integrating Spring Boot Actuator

Add dependency in pom.xml:

<dependency>

  <groupId>org.springframework.boot</groupId>

  <artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

By default काही endpoints मिळतात → /actuator/health, /actuator/info.

2. Monitoring & Managing REST services

/actuator/metrics → memory, CPU, request count

/actuator/health → app healthy आहे का?

/actuator/env → environment variables

3. Exposing Custom Metrics

आपण आपले custom counters define करू शकतो.

```
@RestController
public class StudentController {
  private Counter counter;

  public StudentController(MeterRegistry registry) {
    counter = Counter.builder("student.api.calls").register(registry);
  }

  @GetMapping("/students")
  public String getStudents() {
    counter.increment();
    return "All students fetched";
  }
}
```

4. Securing & Customizing Actuator Endpoints

application.properties मधे configure करू शकतो:

```
management.endpoints.web.exposure.include=health,info,metrics
management.endpoint.health.show-details=always
```

Actuator endpoints secure करायला Spring Security वापरतो.

✅ Real-life Example:

Netflix, Amazon सारख्या मोठ्या apps production मध्ये Actuator + Micrometer + Prometheus + Grafana वापरून APIs monitor करतात.

2. Testing RESTful APIs (Security + JWT + CORS)

(a) Securing RESTful Endpoints with Spring Security

REST APIs open ठेवली तर कोणीही access करू शकतो.

म्हणून authentication & authorization लावतो.

👉 Example config:

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable()
      .authorizeRequests()
      .antMatchers("/public/**").permitAll()
      .anyRequest().authenticated()
      .and()
      .httpBasic();
  }
}
```

इथे /public/** open आहे, बाकीच्या endpoints साठी login लागेल.

(b) Token-based Authentication (JWT)

👉 Definition:

JWT (JSON Web Token) म्हणजे token-based authentication mechanism.

म्हणजे user authenticate झाल्यावर server ek signed token generate karto, आणि पुढील requests मध्ये तो token वापरून user ओळखला जातो.

👉 Flow:

1. Client → username/password पाठवतो

2. Server → JWT generate करून client ला देतो

3. Client → पुढच्या सर्व requests मध्ये Authorization: Bearer <token> पाठवतो

4. Server → token verify करून data देतो

👉 Example Token (JWT):

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

(c) Handling Cross-Origin Resource Sharing (CORS)

👉 Problem:

जर frontend (React/Angular) आणि backend (Spring Boot) वेगळ्या servers वर चालत असतील तर Cross-Origin error येतो.

👉 Solution:

Spring Boot मध्ये CORS enable करतो.

```
@Configuration
public class CorsConfig {
  @Bean
  public WebMvcConfigurer corsConfigurer() {
    return new WebMvcConfigurer() {
      @Override
      public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**").allowedOrigins("http://localhost:3000");
      }
    };
  }
}
```

👉 आता React app (3000 port) Spring Boot (8080 port) कडून data घेऊ शकते.

✅ Interview साठी Short बोलायला Points:

Actuator: REST API monitoring tool → health, metrics, custom monitoring.

Security: Endpoints secure करण्यासाठी Spring Security वापरतो.

JWT: Token-based authentication, stateless, scalable.

CORS: Frontend & backend वेगवेगळ्या domains असतील तर data share करण्यासाठी enable करतो.

1. Testing RESTful APIs

👉 Definition:

Testing म्हणजे आपल्या REST APIs नीट काम करतायत का हे automatically verify करणं. यामुळे bugs कमी होतात आणि production ला जाण्याआधीच समस्या सापडतात.

Sub-points:

(a) Unit Testing REST Controllers (JUnit + Mockito)

Unit test म्हणजे फक्त individual method check करणे.

External calls (database, service) mock करतो.

👉 Example:

```
@WebMvcTest(StudentController.class)
public class StudentControllerTest {

  @Autowired
  private MockMvc mockMvc;

  @MockBean
  private StudentService studentService;

  @Test
  public void testGetStudent() throws Exception {
    Student s = new Student(1, "Anurag", "CSE");
    Mockito.when(studentService.getStudent(1)).thenReturn(s);
```

```
mockMvc.perform(get("/students/1"))
    .andExpect(status().isOk())
    .andExpect(jsonPath("$.name").value("Anurag"));
```

(b) Integration Testing REST Services

इथे पूर्ण flow (Controller → Service → Repository → DB) check करतो.

Database साठी in-memory DB (H2) वापरतो.

(c) Spring Test & MockMvc

MockMvc वापरून HTTP request simulate करून response check करतो.

फायदे: Server start न करता API test करता येते.

(d) Test Coverage & Best Practices

Coverage → किती code test झाला (%).

✅ Real-life Example:

Flipkart/Paytm सारख्या companies मध्ये JUnit + Mockito + Postman automated tests वापरले जातात जेणेकरून मोठ्या scale वर APIs fail होऊ नयेत.

2. Documenting RESTful APIs

👉 Definition:

API documentation म्हणजे developer-friendly guide ज्यात API endpoints, inputs, outputs, errors clearly दाखवलेले असतात.

म्हणजेच developers ला तुमची API कशी वापरायची हे पटकन समजतं.

Sub-points:

(a) Introduction to Tools

Swagger (OpenAPI Specification)

Springdoc OpenAPI (Spring Boot integration)

(b) Documenting REST APIs with Swagger/OpenAPI

👉 Add Maven dependency:

```
<dependency>
   <groupId>org.springdoc</groupId>
   <artifactId>springdoc-openapi-ui</artifactId>
   <version>1.7.0</version>
</dependency>
```

👉 Now run app → open browser:

http://localhost:8080/swagger-ui.html

(c) Example with Annotations

```
@RestController
@RequestMapping("/students")
@Tag(name = "Student API", description = "Operations on students")
public class StudentController {
   @Operation(summary = "Get Student by ID")
   @GetMapping("/{id}")
   public Student getStudent(@PathVariable int id) {
      return new Student(id, "Anurag", "CSE");
   }
}
```

👉 Swagger UI मध्ये हे neat form madhe दिसेल.

(d) Best Practices

प्रत्येक endpoint ला summary + description द्या

Input/Output examples द्या

Authentication details (JWT token usage) दाखवा

Versioning सांगा (/api/v1, /api/v2)

✅ Real-life Example:

Google Maps API किंवा Razorpay API → Swagger सारख्या docs वरूनच developers integration करतात.

✅ Interview साठी Short बोलायला Points:

Unit Testing: Individual methods test करतो (Mockito use).

Integration Testing: Full flow (Controller to DB).

MockMvc: REST API simulate करायला वापरतो.

Coverage: जास्त असेल तर code जास्त safe.

API Docs (Swagger): Developers ला API वापरणं सोपं होतं.

Best practice: Swagger annotations वापरा, inputs/outputs clearly दाखवा, versioning follow करा.

🚀 Stage 2 – Spring REST using Spring Boot 3 (Interview Notes)

1. Introduction to Spring REST & Spring Boot 3

REST (Representational State Transfer):Architecture style for web services.

Uses HTTP methods (GET, POST, PUT, DELETE).

Resource-based URLs (उदा. /students/1).

Spring REST: REST APIs तयार करायला support.

Spring Boot: Simplified setup, auto-configuration, embedded server.

Spring Boot 3 Features:

Java 17+, improved performance, native image support.

2. Building a Simple REST Controller

Controller: Request handle करणारा class (@RestController).

Request Mappings:

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping.

JSON Responses: Jackson library automatically Java → JSON convert करते.

👉 Example:

```
@RestController
@RequestMapping("/students")
public class StudentController {
  @GetMapping("/{id}")
  public Student getStudent(@PathVariable int id) {
    return new Student(id, "Anurag", "CSE");
  }
}
```

3. Request & Response Handling

Path Variables: @PathVariable (उदा. /students/{id}).

Query Params: @RequestParam (उदा. /students?id=10).

Request Body: @RequestBody → JSON → Java object.

Custom Response: ResponseEntity वापरून status code, headers बदलता येतात.

Exception Handling: @ExceptionHandler किंवा @ControllerAdvice.

4. RESTful Resource Representation with DTOs

DTO (Data Transfer Object): External world ला data देण्यासाठी safe object.

Mapping Entities → DTOs: ModelMapper/MapStruct वापरतो.

Custom Serialization/Deserialization: Jackson annotations (@JsonProperty).

Versioning: /api/v1/students vs /api/v2/students.

5. RESTful CRUD Operations

CRUD = Create, Read, Update, Delete.

HTTP mapping:

POST → Create

GET → Read

PUT → Update

DELETE → Delete

Validation: @NotNull, @Size, @Email.

Optimistic Locking: @Version (JPA) for concurrent updates.

6. RESTful HATEOAS

Definition: API responses सोबत links देणे → client ला पुढचं action समजतं.

Spring HATEOAS library: EntityModel, WebMvcLinkBuilder.

👉 Example JSON:

```
{
 "id": 101,
 "name": "Anurag",
 "_links": {
  "self": {"href": "http://localhost:8080/students/101"},
  "all-students": {"href": "http://localhost:8080/students"}
 }
}
```

7. Content Negotiation & Media Types

Definition: Response format client च्या Accept header वर depend करतो.

Supported → JSON, XML.

Custom media types पण बनवू शकतो.

👉 Example:

```
@GetMapping(produces = {"application/json", "application/xml"})
public Student getStudent() {
   return new Student(1, "Anurag", "CSE");
}
```

8. Spring Boot Actuator for REST Monitoring

Actuator: Production-ready monitoring tool.

Endpoints: /actuator/health, /actuator/metrics, /actuator/info.

Custom Metrics: Micrometer वापरून बनवता येतात.

Securing Actuator: Spring Security वापरतो.

9. Securing RESTful APIs

Spring Security: Authentication + Authorization.

Basic Auth / Form Login.

JWT (JSON Web Token):

Stateless auth → scalable.

Flow → Login → Token generate → पुढच्या requests मध्ये Authorization: Bearer <token>.

CORS: Different domains वरून access allow करायला.

10. Testing RESTful APIs

Unit Testing: Individual methods → JUnit + Mockito.

Integration Testing: Full flow (Controller → DB).

MockMvc: Server start न करता REST APIs test करतो.

Coverage: जास्त coverage म्हणजे जास्त safe code.

11. Documenting RESTful APIs

Tools: Swagger, Springdoc OpenAPI.

Swagger UI: http://localhost:8080/swagger-ui.html → interactive docs.

Annotations:

@Operation(summary = "Get Student by ID")

@Tag(name = "Student API")

Best Practices:

Input/Output examples द्या

Authentication माहिती दाखवा

Versioning करा

✅ Quick Interview Answers (1–2 lines each):

REST: Resource-based architecture using HTTP methods.

Controller: Request handle करणारा class → @RestController.

DTO: Data transfer object, secure API contract.

HATEOAS: Response मध्ये links देऊन next actions guide करणे.

Content Negotiation: JSON/XML response based on Accept header.

Actuator: API monitoring tool (health, metrics).

JWT: Stateless authentication using tokens.

CORS: Different domains वरून API access enable करणे.

Unit Testing: Individual methods test.

Swagger: API documentation and testing tool.