Important Errors in a Spring Boot Backend

Errors in a Spring Boot backend can generally be categorized into:

1. **Startup/Configuration Errors:** Prevent the application from even starting.
2. **Runtime Errors (Application Logic):** Occur during request processing within your business logic.
3. **Data Access Errors:** Related to database interactions.
4. **Security Errors:** Related to authentication and authorization.
5. **API/External Integration Errors:** When your backend interacts with other services.

1. Startup/Configuration Errors

1.1. Port Already In Use (`PortInUseException`)

- **Cause:** Another application (or a previous instance of your own application) is already running on the port Spring Boot tries to bind to (default 8080).
- **Detection:**
  - Error message in console/logs: `Address already in use: bind`, `PortInUseException`.
  - Application fails to start.
- **What to do:**
  - **Kill the process:** Find and terminate the process using the port (e.g., `lsof -i :8080` on Linux/macOS, `netstat -ano | findstr :8080` on Windows, then `taskkill /PID <PID> /F`).
  - **Change port:** Configure `server.port` in `application.properties` (e.g., `server.port=8081`).

1.2. Missing/Incorrect Dependency (`ClassNotFoundException`, `NoClassDefFoundError`)

- **Cause:** You're trying to use a class or feature for which the required JAR file is not on the classpath. Often happens when a `spring-boot-starter-*` is missing or a transitive dependency is excluded.
- **Detection:**
  - Error message in console/logs: `java.lang.ClassNotFoundException`, `java.lang.NoClassDefFoundError`. The stack trace will indicate which class is missing.
  - Application fails to start or crashes immediately after startup.
- **What to do:**
  - **Check `pom.xml`/`build.gradle`:** Ensure all necessary `spring-boot-starter` dependencies are included.
  - **Review `mvn dependency:tree` or `gradle dependencies`:** See the full dependency tree to identify missing transitive dependencies or unwanted exclusions.
  - **Rebuild:** Clean and rebuild your project.

1.3. Invalid Configuration (`InvalidConfigurationPropertyValueException`, `YAMLParseException`)

- **Cause:** Typos, incorrect syntax, or invalid values in `application.properties` or `application.yml`. Spring Boot might fail to bind configuration properties to beans.
- **Detection:**
  - Error messages during startup, often pointing directly to the property that couldn't be bound or parsed.
  - Spring Boot provides descriptive exceptions like `org.springframework.boot.context.properties.ConfigurationPropertiesBindingExc`

eption.

- **What to do:**
  - **Double-check syntax:** Ensure correct YAML indentation, property names, and valid values.
  - **Refer to documentation:** Consult Spring Boot reference documentation for correct property names.
  - **Use IDE auto-completion:** Modern IDEs with Spring Boot plugins often provide auto-completion for properties.

1.4. Bean Creation Failure (`UnsatisfiedDependencyException`, `NoUniqueBeanDefinitionException`, `NoSuchBeanDefinitionException`)

- **Cause:** Spring's IoC container cannot create a bean due to:
  - A required dependency not being found (`NoSuchBeanDefinitionException`).
  - Multiple beans of the same type found when only one is expected (`NoUniqueBeanDefinitionException`, resolve with `@Qualifier` or `@Primary`).
  - Circular dependencies between beans.
  - A class not being marked with a Spring stereotype annotation (`@Component`, `@Service`, `@Repository`, `@Controller`, `@Configuration`).
- **Detection:**
  - Error messages during application startup, indicating which bean failed to create and the reason.
  - Stack trace often points to the constructor or field where the dependency injection failed.
- **What to do:**
  - **Verify annotations:** Ensure all components are correctly annotated (`@Service`, `@Repository`, etc.) and are within a component scan path.
  - **Check constructor injection:** Ensure required beans are available for injection.
  - **Resolve circular dependencies:** Refactor code to break the cycle (e.g., by introducing an interface or splitting logic).
  - **Use `@Qualifier` or `@Primary`:** If you intentionally have multiple beans of the same type and need to specify which one to inject.

2. Runtime Errors (Application Logic)

2.1. Null Pointer Exception (NPE) (`java.lang.NullPointerException`)

- **Cause:** Attempting to use an object reference that currently points to `null`. This is a very common runtime error.
- **Detection:**
  - Stack trace clearly states `NullPointerException` and points to the exact line number.
  - HTTP 500 Internal Server Error response to the client.
- **What to do:**
  - **Debug:** Step through the code with an IDE debugger to find which object is `null`.
  - **Null checks:** Implement explicit `null` checks where appropriate.
  - **Use `Optional`:** Leverage `java.util.Optional` for methods that might return `null` to force consumers to handle the absence of a value.
  - **Defensive programming:** Ensure objects are initialized before use.
  - **Lombok's `@NonNull`:** Can help at compile time to identify potential null issues.

2.2. Index Out of Bounds (`IndexOutOfBoundsException`, `ArrayIndexOutOfBoundsException`, `StringIndexOutOfBoundsException`)

- **Cause:** Trying to access an element in a collection (array, list, string) using an index that is outside its valid range.
- **Detection:**
  - Stack trace showing the specific `IndexOutOfBoundsException`.
  - HTTP 500 Internal Server Error response.
- **What to do:**
  - **Validate input:** Ensure indices are within bounds before accessing elements.
  - **Iterate safely:** Use enhanced for-loops or iterators.
  - **Check collection sizes:** Before attempting to access by index.

2.3. Illegal Argument (`IllegalArgumentException`)

- **Cause:** A method receives an argument that is not valid for its expected operation (e.g., a negative number for a quantity that must be positive).
- **Detection:**
  - Stack trace indicating `IllegalArgumentException`.
  - Often thrown by your own validation logic within service methods.
- **What to do:**
  - **Implement input validation:** At the controller layer (using JSR 303/380 annotations) and service layer.
  - **Throw custom exceptions:** For specific business rules violations, then map these to appropriate HTTP status codes (e.g., 400 Bad Request) using `@ControllerAdvice`.

3. Data Access Errors

3.1. Database Connection Failure (`CannotCreateStatementException`, `CommunicationsException`)

- **Cause:** Application cannot establish or maintain a connection to the database. Reasons include: incorrect database URL, credentials, database server is down, firewall blocking connection, exhausted connection pool.
- **Detection:**
  - Spring Boot startup logs will show connection errors.
  - During runtime, any database operation will throw related exceptions.
  - Specific driver exceptions (e.g., `com.mysql.cj.jdbc.exceptions.CommunicationsException`, `org.postgresql.util.PSQLException`).
- **What to do:**
  - **Verify `application.properties`:** Double-check `spring.datasource.url`, `username`, `password`.
  - **Check database server status:** Ensure the database is running and accessible from the application server.
  - **Firewall rules:** Verify network connectivity and firewall settings.
  - **Connection pool monitoring:** If using HikariCP, monitor its metrics for pool exhaustion.

3.2. SQL Grammar Error (`BadSqlGrammarException`, specific JDBC driver exceptions)

- **Cause:** Invalid SQL/JPQL syntax in `@Query` annotations or native queries. Mismatch between entity mapping and actual database schema.
- **Detection:**
  - `org.springframework.jdbc.BadSqlGrammarException` in logs.
  - Underlying JDBC driver exceptions (e.g., `PSQLException: ERROR: syntax error at or near "FROM"`) will provide more specific details from the database.
- **What to do:**
  - **Review SQL/JPQL:** Carefully check the query for typos or incorrect syntax.
  - **Compare entity with schema:** Ensure column names and types match between your JPA entities and the actual database table.
  - **Use database client:** Test the SQL query directly in a database client to ensure it's valid.

3.3. Data Integrity Violation (`DataIntegrityViolationException`)

- **Cause:** Attempting to perform a database operation that violates a database constraint (e.g., unique constraint, foreign key constraint, not-null constraint).
  - **Example:** Trying to insert a user with an email that already exists in a unique email column.
- **Detection:**
  - `org.springframework.dao.DataIntegrityViolationException` in logs.
  - Underlying JDBC driver exceptions provide details about the violated constraint.
- **What to do:**
  - **Implement validation:** At the application layer to prevent invalid data from reaching the database (e.g., check for existing email before attempting insert).
  - **Handle exceptions:** Catch `DataIntegrityViolationException` (or more specific Spring Data exceptions) and map them to appropriate HTTP status codes (e.g., 409 Conflict) and user-friendly messages.

4. Security Errors (Especially with JWT)

4.1. Invalid/Expired JWT (`SignatureException`, `ExpiredJwtException`, `MalformedJwtException`)

- **Cause:** Client sends a JWT that is invalid (e.g., tampered signature, malformed structure) or expired.
- **Detection:**
  - Exceptions thrown by the JWT library (JJWT) inside your `JwtAuthenticationFilter` or `JwtService`.
  - Your `JwtAuthenticationEntryPoint` will catch `AuthenticationException` and return a `401 Unauthorized` response with a specific message.
- **What to do:**
  - **Client-side:** The client should detect the 401 and either re-authenticate (login again) or use a refresh token to get a new access token.
  - **Server-side:** Ensure proper logging of these events, but no specific action is typically needed other than returning the 401.

4.2. Unauthorized Access (Authentication Failure - 401)

- **Cause:** An unauthenticated user (or a user with an invalid/missing token) tries to access a protected resource.
- **Detection:**

    - Spring Security's `ExceptionTranslationFilter` catches an `AuthenticationException` and delegates to your `AuthenticationEntryPoint` (`JwtAuthenticationEntryPoint`).
    - Client receives HTTP `401 Unauthorized`.
- **What to do:**
    - **Client-side:** Prompt the user to log in or refresh their token.
    - **Server-side:** Ensure your `JwtAuthenticationEntryPoint` returns a clear JSON error response.

4.3. Forbidden Access (Authorization Failure - 403)

- **Cause:** An authenticated user tries to access a resource or perform an action for which they do not have the necessary permissions/roles (e.g., a `USER` trying to access an `ADMIN`-only endpoint).
- **Detection:**
    - Spring Security's `ExceptionTranslationFilter` catches an `AccessDeniedException` and delegates to your `AccessDeniedHandler` (`CustomAccessDeniedHandler`).
    - Client receives HTTP `403 Forbidden`.
    - Look for logs from `FilterSecurityInterceptor` or method security (`@PreAuthorize`).
- **What to do:**
    - **Client-side:** Inform the user they lack permissions.
    - **Server-side:** Ensure your `AccessDeniedHandler` returns a clear JSON error response. Review `@PreAuthorize` annotations and user roles.

5. API/External Integration Errors

5.1. External Service Unreachable/Timeout (`ConnectException`, `SocketTimeoutException`)

- **Cause:** Your backend tries to connect to an external API (e.g., payment gateway, third-party service) but cannot reach it or it takes too long to respond.
- **Detection:**
    - `java.net.ConnectException` (connection refused).
    - `java.net.SocketTimeoutException` (read or connection timeout).
    - Custom exceptions if you're wrapping external API calls.
- **What to do:**
    - **Implement timeouts:** Configure appropriate connection and read timeouts for your HTTP client (e.g., RestTemplate, WebClient).
    - **Retry mechanisms:** Implement retries for transient failures.
    - **Circuit Breakers (e.g., Resilience4j, Spring Cloud Circuit Breaker):** Prevent cascading failures by opening the circuit when an external service is unhealthy.
    - **Graceful degradation:** Provide fallback behavior if the external service is unavailable.
    - **Monitoring:** Set up alerts for external service outages.

General Error Handling Best Practices & Detection

1. **Centralized Error Handling (`@ControllerAdvice` & `@ExceptionHandler`):**
    - **Purpose:** Catch exceptions thrown by controllers, services, or repositories and transform them into consistent, user-friendly JSON error responses.
    - **Benefits:** Avoids scattering `try-catch` blocks everywhere, provides a single place to define your error response format.

- **Detection:** When an unhandled exception occurs, Spring's default error handler will return a generic error. Your custom handler should log the exception and return a structured JSON.

2. **Detailed Logging:**
    - **Log Levels:** Use `INFO` for normal operations, `DEBUG` for detailed execution paths, `WARN` for potential issues, and `ERROR` for critical failures and exceptions.
    - **Structured Logging:** Consider tools like Logback with JSON appenders or Logstash/Splunk for easy parsing and analysis in production.
    - **Correlation IDs:** Add a unique ID to each request log to trace its journey through your services.
    - **What to log:** Exception type, message, stack trace, relevant request parameters, and user info (avoid sensitive data).
3. **Use IDE Debugger:**
    - Set breakpoints, step through code, inspect variable values at runtime. This is invaluable for pinpointing the exact line where an error originates.
4. **Unit & Integration Tests:**
    - Write tests that cover expected error scenarios (e.g., invalid input, non-existent ID, database constraint violation). This helps catch errors early during development.
5. **Monitoring and Alerting:**
    - Use tools like Spring Boot Actuator with Prometheus/Grafana or commercial APM (Application Performance Monitoring) tools (e.g., New Relic, Datadog) to track application health, metrics, and errors in production. Set up alerts for high error rates.
6. **API Client Tools (Postman, Insomnia, Curl):**
    - Test your API endpoints manually. Observe the HTTP status codes and response bodies for errors.

Important HTTP Status Codes

HTTP status codes are returned by the server in response to a client's request to indicate the outcome of the request. They are crucial for REST APIs to communicate the state of operations.

2xx: Success

- `200 OK`:
    - **Meaning:** The request has succeeded.
    - **When returned:** Generic success. Used for GET requests where data is successfully retrieved, PUT/PATCH for successful updates if a specific 204 or 201 isn't more appropriate.
- `201 Created`:
    - **Meaning:** The request has been fulfilled and resulted in a new resource being created.
    - **When returned:** Primarily for **POST** requests that successfully create a new resource on the server. The response should typically include the URI of the newly created resource in the `Location` header.
- `204 No Content`:

- **Meaning:** The server successfully processed the request, but is not returning any content.
- **When returned:** Often used for **DELETE** requests, or PUT/PATCH requests where the client doesn't need to know the updated state of the resource (e.g., just confirming success).

4xx: Client Errors

- `400 Bad Request`:
  - **Meaning:** The server cannot or will not process the request due to something that is perceived to be a client error.
  - **When returned:**
    - **Validation failures:** Malformed JSON, missing required parameters, invalid data formats (e.g., `MethodArgumentNotValidException` when `@Valid` fails).
    - Semantic errors in the request that prevent processing.
- `401 Unauthorized`:
  - **Meaning:** The request lacks valid authentication credentials for the target resource.
  - **When returned:**
    - When a client tries to access a protected resource without an authentication token (e.g., missing JWT).
    - When the provided authentication token is invalid or expired.
    - (Handled by `AuthenticationEntryPoint` in Spring Security for unauthenticated requests).
- `403 Forbidden`:
  - **Meaning:** The server understands the request but refuses to authorize it. Authentication was successful, but the authenticated user does not have the necessary permissions to access the resource.
  - **When returned:** When an authenticated user with a specific role (e.g., `USER`) tries to access an endpoint that requires a higher role (e.g., `ADMIN`) (`AccessDeniedException`).
    - (Handled by `AccessDeniedHandler` in Spring Security for authorized but forbidden requests).
- `404 Not Found`:
  - **Meaning:** The server cannot find the requested resource.
  - **When returned:**
    - When a client requests a non-existent URL (typo in endpoint path).
    - When a client requests a resource by ID that does not exist in the database (e.g., `/users/999` where user 999 doesn't exist).
- `405 Method Not Allowed`:
  - **Meaning:** The request method (e.g., GET, POST) is not supported for the resource identified by the URL.
  - **When returned:** When a client sends a POST request to an endpoint that only accepts GET requests (or vice-versa).
- `409 Conflict`:
  - **Meaning:** The request could not be completed due to a conflict with the current state of the target resource.
  - **When returned:**
    - When trying to create a resource that already exists (e.g., registering a user with an email that is already taken due to a unique constraint).
    - Optimistic locking failures (concurrent updates).

- `429 Too Many Requests`:
  - **Meaning:** The user has sent too many requests in a given amount of time ("rate limiting").
  - **When returned:** If you implement rate limiting on your API endpoints (e.g., for brute-force prevention on login).

5xx: Server Errors

- `500 Internal Server Error`:
  - **Meaning:** A generic error message, given when an unexpected condition was encountered and no more specific message is suitable.
  - **When returned:**
    - Uncaught exceptions in your backend code (e.g., `NullPointerException`, `IndexOutOfBoundsException`).
    - Any unhandled business logic error.
    - Database connection issues or other unhandled critical failures within the server.
    - (You should strive to map these to more specific 4xx codes where possible or return a detailed JSON error body for debugging).
- `502 Bad Gateway`:
  - **Meaning:** The server, while acting as a gateway or proxy, received an invalid response from an upstream server.
  - **When returned:** If your Spring Boot app is behind a proxy (like Nginx, Apache, API Gateway) and the proxy gets an invalid response from your app (or your app is down).
- `503 Service Unavailable`:
  - **Meaning:** The server is not ready to handle the request. Common causes are a server that is down for maintenance or is overloaded.
  - **When returned:** If your application instance is temporarily unavailable (e.g., during deployment, or if it's gracefully shutting down).
- `504 Gateway Timeout`:
  - **Meaning:** The server, while acting as a gateway or proxy, did not receive a timely response from an upstream server.
  - **When returned:** Similar to 502, but indicates a timeout specifically. If your backend calls another slow service and exceeds its timeout configured on the proxy or gateway.