**Runners in Spring Boot**

* **Runners** are Java classes that behave like Spring Beans in Spring Boot and implement one of the Runner interfaces, typically:
  + CommandLineRunner or
  + ApplicationRunner.
* Each Runner class must override the run() method, which contains logic that should execute **once** during application startup.
* The logic inside the run() method executes **after the Spring application context is initialized**, meaning:
  + All beans are instantiated,
  + Dependencies are injected,
  + The application is almost fully up.
* The run() method of each Runner is automatically executed by the **IoC container** exactly **once**, as part of the startup process triggered by SpringApplication.run().

**Example:**

@Component

public class MyRunner implements CommandLineRunner {

@Override

public void run(String... args) throws Exception {

System.out.println("This runs after Spring Boot startup.");

}

}

### ✅ CommandLineRunner

* **Method Signature**:

void run(String... args) throws Exception;

* **Parameter Type**: A String[] of raw command-line arguments.
* **Usage**:

@Component

public class MyCommandLineRunner implements CommandLineRunner {

@Override

public void run(String... args) {

for (String arg : args) {

System.out.println("CommandLineRunner arg: " + arg);

}

}

}

* **Example Input**:

--name=John --age=30 customArg

* **args[]** will be:

[--name=John, --age=30, customArg]

Eg: RunnersCommandLineRunners

### ✅ ApplicationRunner

* **Method Signature**:

void run(ApplicationArguments args) throws Exception;

 **Parameter type**: ApplicationArguments args

 **What it is**: A helper object that separates and parses the command-line arguments into **option arguments** (with --) and **non-option arguments**.

@Component

public class MyApplicationRunner implements ApplicationRunner {

@Override

public void run(ApplicationArguments args) {

System.out.println("Option Names: " + args.getOptionNames());

System.out.println("Non-option args: " + args.getNonOptionArgs());

System.out.println("Value of 'name': " + args.getOptionValues("name"));

}

}

Example Input:

--name=John --age=30 customArg

args.getOptionNames() → ["name", "age"]

args.getOptionValues("name") → ["John"]

args.getNonOptionArgs() → ["customArg"]

Eg: RunnersApplicationRunner

Spring Jdbc:

### ✅ ****Plain JDBC**** (Traditional Approach)

**Responsibilities of Developer (Manual Work):**

1. **Load JDBC Driver**

Class.forName("oracle.jdbc.driver.OracleDriver");

1. **Establish Connection**

Connection con = DriverManager.getConnection(...);

1. **Create Statement Object**

PreparedStatement ps = con.prepareStatement("...");

1. **Send and Execute Query**

ResultSet rs = ps.executeQuery();

1. **Process Results (Application Logic)**

while(rs.next()) {

// logic to process result

}

1. **Exception Handling**

try-catch blocks around every DB interaction

1. **Transaction Management**

con.setAutoCommit(false);

con.commit();

con.rollback();

1. **Close JDBC Objects**

rs.close(); ps.close(); con.close();

✅ **Only step 5 (processing results)** is the actual business logic.  
❌ All other steps are repetitive boilerplate.

✅ **Spring JDBC (SpringDAO with JdbcTemplate)**

### ✅ Focus on Business Logic

* **What It Means:**  
  You only write logic that actually solves your business problem. All technical, repetitive parts are handled by Spring.
* **Example:**

public Employee getEmployeeById(int id) {

String sql = "SELECT \* FROM employee WHERE id = ?";

return jdbcTemplate.queryForObject(sql, new Object[]{id}, new EmployeeRowMapper());

}

* **Why It Helps:**  
  You focus on what you want to do (get employee by ID) instead of writing many lines just to set up and run the query.

### ✅ No Need to Manage Low-Level Details

#### 1. **Connection Creation and Closing** — AUTOMATIC

@Autowired

private JdbcTemplate jdbcTemplate;

Spring uses the injected DataSource to create and manage connections automatically. No need to call DriverManager.getConnection() or conn.close() manually.

#### 2. **Statement Creation and Execution** — AUTOMATIC

String sql = "SELECT \* FROM employee WHERE id = ?";

jdbcTemplate.queryForObject(sql, new Object[]{id}, new EmployeeRowMapper());

Spring creates a PreparedStatement, sets the parameters, and executes it behind the scenes.

#### 3. **Result Processing** — Simplified with RowMapper

java

CopyEdit

public class EmployeeRowMapper implements RowMapper<Employee> {

public Employee mapRow(ResultSet rs, int rowNum) throws SQLException {

return new Employee(rs.getInt("id"), rs.getString("name"));

}

}

Instead of looping through ResultSet manually, you just define how one row should be converted into an object.

#### 4. **Transaction Management** — HANDLED BY SPRING

@Transactional

public void saveEmployee(Employee emp) {

String sql = "INSERT INTO employee(id, name) VALUES(?, ?)";

jdbcTemplate.update(sql, emp.getId(), emp.getName());

}

Use @Transactional to let Spring manage commit/rollback. No need for conn.setAutoCommit(false) or conn.commit().

### ✅ Built-In Support for Exception Translation

#### Without Spring:

try {

// JDBC logic

} catch (SQLException e) {

e.printStackTrace(); // or handle various SQL error codes manually

}

#### With Spring:

try {

jdbcTemplate.queryForObject("SELECT \* FROM employee WHERE id = ?",

new Object[]{id}, new EmployeeRowMapper());

} catch (DataAccessException e) {

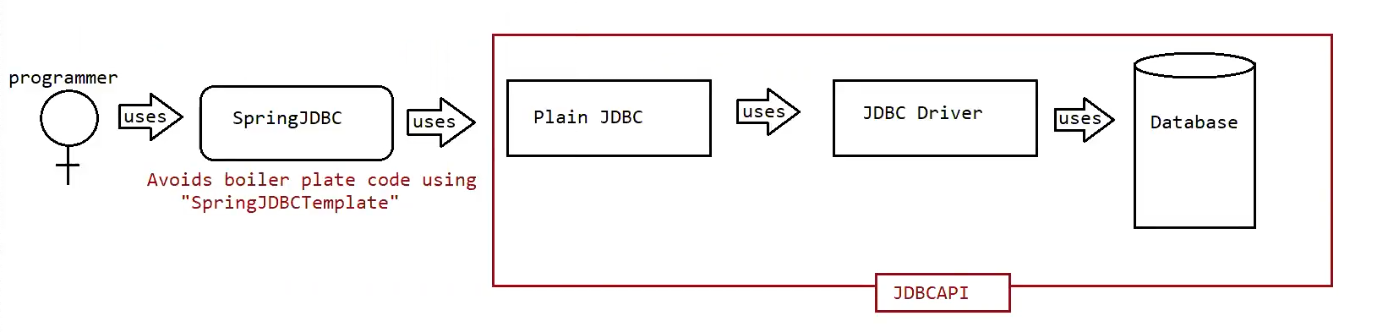
System.out.println("Database error occurred: " + e.getMessage());

}

Spring catches low-level exceptions and wraps them in DataAccessException, giving you a consistent, cleaner way to handle DB errors.

### 📌 Summary

| **✅ Feature** | **🔧 Spring Code Example** |
| --- | --- |
| Inject JdbcTemplate | @Autowired private JdbcTemplate jdbcTemplate; |
| Execute SQL | jdbcTemplate.update(...); or queryForObject(...) |
| Process ResultSet | RowMapper<T> implementation |
| Handle Transactions | @Transactional annotation |
| Handle Exceptions | catch (DataAccessException e) |



### 🔍 ****Diagram Explanation (In Simple Words)****

#### 🧑‍💻 Programmer

* The developer **uses Spring JDBC** to interact with the database.
* No need to manually write boilerplate JDBC code.

### 🧱 ****Spring JDBC****

* Acts as a **helper layer** on top of plain JDBC.
* It provides a class called JdbcTemplate that:
  + Automatically handles connection creation and closing
  + Automatically prepares and executes SQL statements
  + Handles exceptions
  + Simplifies result mapping using RowMapper

🟢 **Main Advantage:**

**Avoids boilerplate code** using "Spring JdbcTemplate".

### 🔗 ****Plain JDBC****

* Still used internally by Spring JDBC.
* But **programmer doesn’t directly deal with it**.

### 🔄 ****JDBC Driver****

* Translates Java calls into **database-specific calls**.
* Required to connect and communicate with the actual database.

### 🗄️ ****Database****

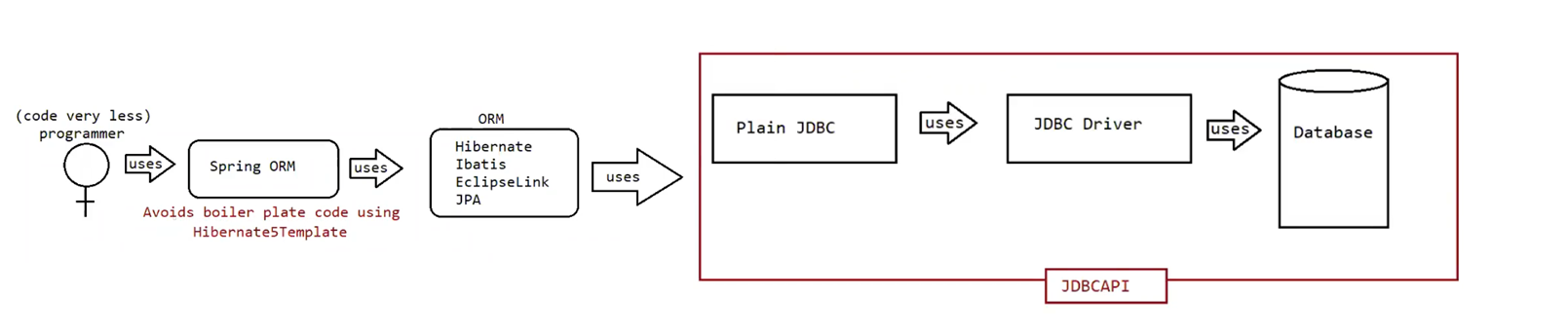
* The final destination for all data operations (CRUD).

### 🔁 ****JDBC API Boundary (Red Box)****

* Shows that all these components (Plain JDBC, Driver, Database) are **part of standard JDBC architecture**.
* Spring JDBC just sits on top of it and simplifies it for the developer.

### ✅ Key Takeaway:

**Spring JDBC is not a replacement for JDBC — it's an abstraction over it that makes development faster and cleaner.**



### 🧠 ****Diagram Explanation (Simple Terms)****

#### 👩‍💻 (Code very less) Programmer

* The developer writes **very little code**.
* Uses **Spring ORM**, which internally uses **HibernateTemplate**, JpaTemplate, etc., to make database operations even easier.

### 🧱 ****Spring ORM****

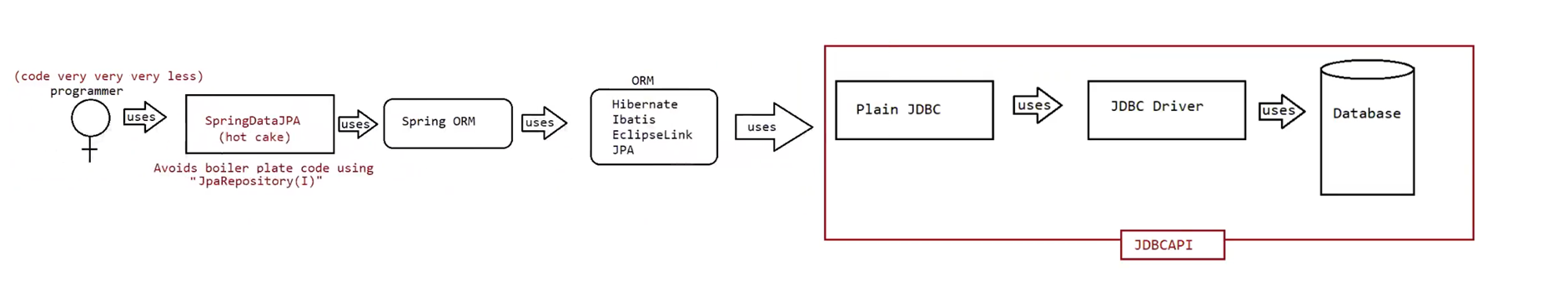
* A module provided by Spring that **integrates with ORM tools** like Hibernate, JPA, iBatis, etc.
* It helps in:
  + Managing **Hibernate sessions**
  + Simplifying **transaction management**
  + Avoiding Hibernate's own boilerplate code

🟢 **Main Advantage:**

Avoids boilerplate code using tools like HibernateTemplate, JpaTemplate, etc.

### ⚙️ ****ORM Framework (Hibernate / JPA / iBatis / EclipseLink)****

* These frameworks **map Java objects to database tables**, and handle the conversion of data automatically.



### 🔍 ****Diagram Explanation (Simplified)****

#### 👩‍💻 Programmer (Code very very very less)

* The developer hardly writes any code — **Spring Data JPA handles almost everything**.
* Just define an interface that extends JpaRepository, and Spring auto-generates the implementation.

### 🧱 ****Spring Data JPA (Hot Cake!)****

* It is the **most powerful and modern** approach to database interaction in Spring projects.
* Built on top of **Spring ORM + JPA/Hibernate**.
* Main feature: Avoids even the need to write SQL queries or DAO classes.

📌 **No need to write**:

* SQL queries
* JDBC code
* RowMappers
* Transactions

**🧠 Spring ORM**

* Internally used by Spring Data JPA for ORM support.
* Still integrates with tools like Hibernate and JPA.

**⚙️ ORM Tools (Hibernate / JPA / EclipseLink / iBatis)**

* Still responsible for mapping Java classes to database tables.
* Handle advanced features like:
  + Lazy loading
  + Caching
  + Entity relationships (OneToMany, ManyToOne)

**🔗 Plain JDBC → JDBC Driver → Database**

* As always, everything eventually ends up in **JDBC API** calls to talk to the actual database.

**✅ Key Takeaways:**

| **Layer** | **What It Does** |
| --- | --- |
| Spring Data JPA | Auto-generates DAO implementations using JpaRepository |
| Spring ORM | Connects to ORM tools like Hibernate |
| ORM (JPA/Hibernate) | Handles Java-to-DB mapping and SQL generation |
| JDBC API | Executes the final SQL on the database |

**💎 Why It's Called "Hot Cake"**

* **Very popular in industry today**
* Reduces **95% of boilerplate code**
* Ideal for **CRUD-based enterprise apps**
* Automatically provides methods like:
  + findById(), findAll(), delete(), save()
  + Plus custom query methods using method names like findByName() etc.

## ✅ **Spring JDBC Advantages (Simple Explanation)**

### 1. ****Supports Both Positional and Named Parameters****

* You can write SQL queries using:
  + **Positional parameters**: use ? placeholders
  + **Named parameters**: use names like :empId

📌 Example:

// Positional

jdbcTemplate.query("SELECT \* FROM emp WHERE id = ?", new Object[]{101});

// Named

String sql = "SELECT \* FROM emp WHERE id = :empId";

namedJdbcTemplate.queryForObject(sql, Map.of("empId", 101), new EmpMapper());

✅ **Named parameters make queries more readable and easier to manage**.

**2. ✅ If you want Spring to process the ResultSet automatically**

(use built-in formats)

➡️ **Use ready-made methods** like:

* query()
* queryForList()
* queryForMap()
* queryForObject()

📌 These methods handle everything: running the query, reading data, converting to list/map/value — all done for you.

**3. ✅ If you want to process the ResultSet manually**

(because you need custom logic)

➡️ **Use callback interfaces** like:

* RowMapper
* ResultSetExtractor
* RowCallbackHandler

📌 These let **you control how each row is read and converted** — perfect for custom Java objects or special processing.

**🎯 Final Check:**

❓ Want auto-processing? → Use Spring’s built-in methods  
❓ Want full control? → Use RowMapper or other interfaces

## 4. ✅ 4. **Avoids Boilerplate Code (Code Reuse)**

### 🔰 What is Boilerplate Code?

Boilerplate code is the **repeated code** you have to write every time, even though it's not related to your business logic.

In **plain JDBC**, you always have to write:

Class.forName("oracle.jdbc.driver.OracleDriver"); // Load driver

Connection con = DriverManager.getConnection(...); // Get DB connection

PreparedStatement ps = con.prepareStatement(...); // Create statement

ResultSet rs = ps.executeQuery(); // Run query

while(rs.next()) { // Loop through results

// Read data manually

}

rs.close(); ps.close(); con.close(); // Close everything

Also:

* You must handle SQLException using try-catch blocks
* You must manage transactions manually if needed

📌 This makes your code **long, repetitive, and error-prone**

**✅ What You Write (Only the Essentials)**

Instead of 15–20 lines of setup, you write just:

java

CopyEdit

String sql = "SELECT \* FROM employee WHERE id = ?";

Employee emp = jdbcTemplate.queryForObject(sql, new Object[]{101}, new EmpMapper());

✔ No connection  
✔ No closing  
✔ No try-catch  
✔ No driver loading

**🔍 How Does Spring Do It Internally?**

* Spring uses a configured DataSource (via XML, Java Config, or Spring Boot application.properties)
* The JdbcTemplate class:
  + Gets the connection from the DataSource
  + Creates the statement
  + Executes the SQL
  + Closes all resources safely
  + Catches SQLException and converts it into a meaningful DataAccessException
* If you annotate with @Transactional, Spring also manages transactions automatically

## ✅ 5. **Built-In Exception Handling (DataAccessException Hierarchy)**

Spring JDBC provides a **powerful and simplified way to handle exceptions** using the DataAccessException class and its subtypes.

In **plain JDBC**, you need to handle low-level SQLException, which is:

* A **checked exception** (must be caught or declared)
* Not specific (you get an error code, not a clear exception type)
* Difficult to propagate and handle in a meaningful way

### ✅ In Spring JDBC:

All SQLExceptions are **automatically converted** into **unchecked exceptions** using a class called DataAccessException.

Spring has created a **common exception hierarchy** for all JDBC-related errors — this is known as the **DataAccessException Hierarchy**.

### 🔹 Key Features (With Explanation)

#### a. ✅ **Unchecked Exceptions**

You don’t need to write try-catch blocks unless you want to.

* DataAccessException is a **RuntimeException**, so Java doesn’t force you to catch it.
* Cleaner and shorter code.

📌 Example:

// No try-catch needed unless you want to handle it specially

Employee emp = jdbcTemplate.queryForObject(sql, new Object[]{id}, new EmpRowMapper());

#### b. ✅ **Handling is Optional**

You can choose when and where to handle exceptions.

* If you don’t catch it in the DAO layer, it will automatically bubble up to the service/controller.
* You can add a global exception handler if needed (e.g., in Spring MVC or Spring Boot).

#### c. ✅ **Supports Exception Propagation**

If an error occurs, Spring will **pass the exception automatically** to the calling method.

📌 Example:

// DAO method throws exception

public Employee getEmployee(int id) {

return jdbcTemplate.queryForObject(...); // exception goes to service

}

// Service method receives and may log or re-throw

No need to write:

try {

...

} catch (SQLException e) {

...

}

#### d. ✅ **Same Exceptions Used in Spring ORM and Spring Data JPA**

The same DataAccessException base class is used in:

* Spring JDBC
* Spring ORM (HibernateTemplate, JpaTemplate)
* Spring Data JPA

📌 **You can write common error handling logic**, regardless of which technology is used.

#### e. ✅ **Uses Exception Rethrowing (Checked → Unchecked)**

Internally, Spring catches checked exceptions (like SQLException) and **throws corresponding unchecked ones**.

📌 Example:

catch(SQLException ex) {

throw new DataIntegrityViolationException(ex.getMessage(), ex);

}

This is called **Exception Translation / Rethrowing**.

## ✅ 6. **Simplifies Calling Stored Procedures**

### 🔰 In Plain JDBC:

Calling a stored procedure is usually **long and complex**.

You have to:

1. Get a connection manually
2. Create a CallableStatement
3. Register input/output parameters
4. Set values for each parameter
5. Execute the call
6. Get the result
7. Close everything
8. Handle all exceptions

📌 This makes the code big, messy, and hard to maintain.

### ✅ What Spring JDBC Does Better:

Spring gives us a class called **SimpleJdbcCall** — it handles **all the complexity** for you.

You don’t have to:

* Register parameters manually
* Handle output types
* Manage connections or resources

You only need to:

* Give the procedure name
* Pass the input parameters

## ✅ 7. **Supports Generics and Var-Args**

### 🔹 What are ****Var-Args****?

Var-args means:  
👉 You can pass **any number of values** to a method, without needing to create an array.

In Java, this looks like:

Object... args

### ✅ In Spring JDBC

Spring JDBC methods (like query(), queryForObject(), etc.) support **var-args** so that you can pass query parameters **directly**, without extra code.

### 📌 Example — Using Var-Args:

String sql = "SELECT name FROM employee WHERE id = ?";

String name = jdbcTemplate.queryForObject(sql, String.class, 101);

**🔹 Let's break down the parameters:**

| **Parameter** | **What it is** |
| --- | --- |
| sql | The SQL query string |
| String.class | The **return type** you're expecting (Generic) |
| 101 | A **query parameter value** (Var-arg) |

**✅ So what is actually var-args here?**

Only this part:

101

It is passed as part of **var-args** (Object... args) for the ? placeholder in the SQL.

**🔹 Explanation:**

* queryForObject(String sql, Class<T> requiredType, Object... args)
  + sql → SQL query
  + requiredType → Generic (like String.class, Integer.class)
  + Object... args → This is the **var-args part** (used to fill ? in SQL)

So:

| **What you passed** | **Role** |
| --- | --- |
| "SELECT..." | SQL query |
| String.class | Return type (Generic) |
| 101 | Var-arg (for SQL ?) |

✔ Here, 101 is passed directly as a parameter.  
✔ You don’t need to write this:

Object[] params = new Object[]{101};

✅ **Less code**, easier to read, and cleaner.

### 🔹 What are ****Generics****?

Generics allow you to tell Java:

“This method will return a specific type like String, Integer, or Employee.”

This makes your code:

* **Type-safe** (compiler checks the type)
* **No need to cast manually**

### 📌 Example — Using Generics:

String sql = "SELECT COUNT(\*) FROM employee";

int count = jdbcTemplate.queryForObject(sql, Integer.class);

✔ Spring knows you're expecting an Integer.  
✔ You get the result **already typed**, no casting needed.

Without generics, you might have to do:

Object result = ...;

int count = (Integer) result; // Manual casting

8. Can Auto-Generate INSERT Queries

Using SimpleJdbcInsert, you can insert data without writing SQL.

📌 Just provide the table name, column names, and values — Spring builds the SQL for you.

SimpleJdbcInsert insert = new SimpleJdbcInsert(jdbcTemplate)

.withTableName("employee")

.usingColumns("id", "name");

insert.execute(Map.of("id", 101, "name", "Pavan"));