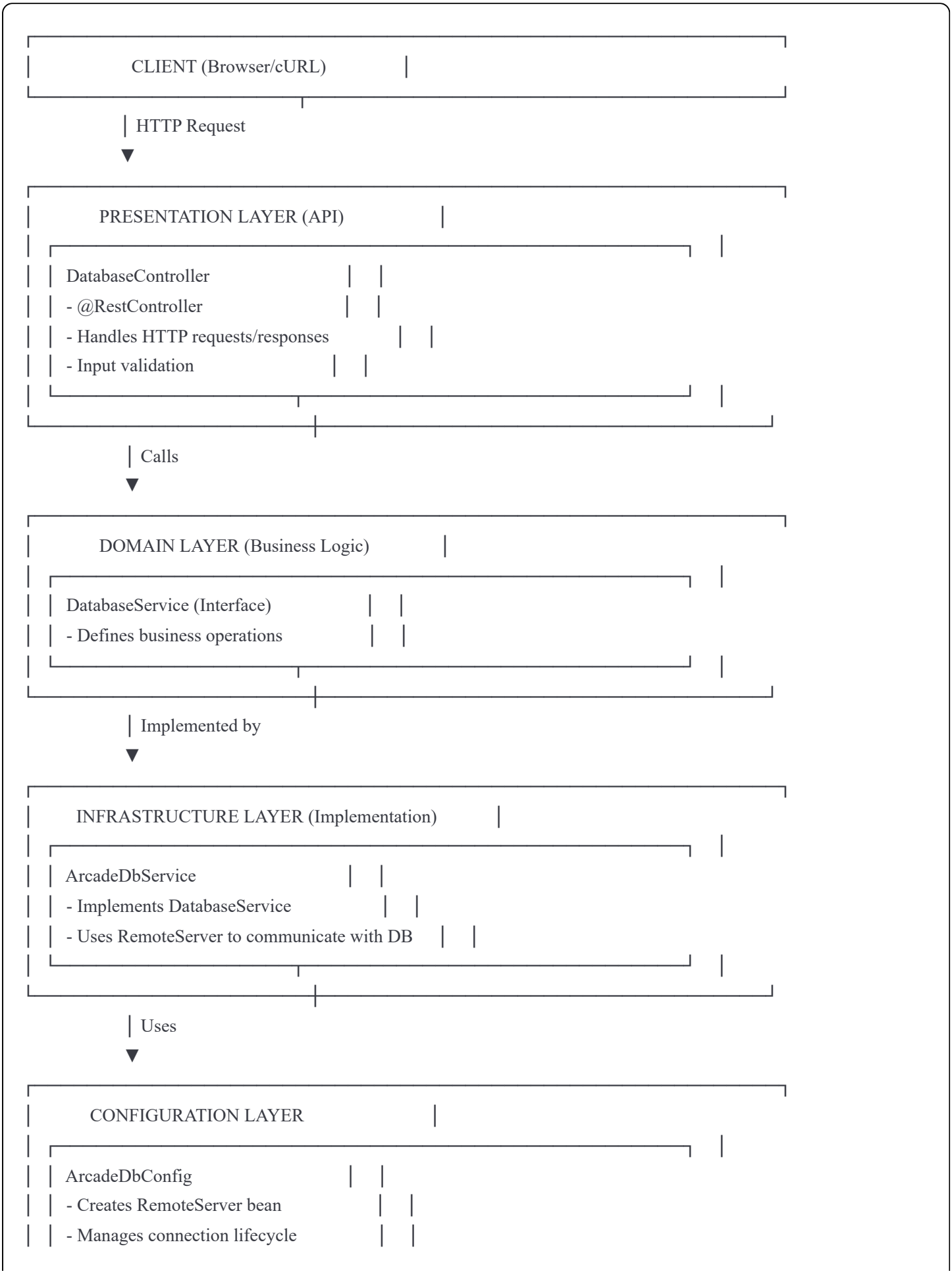
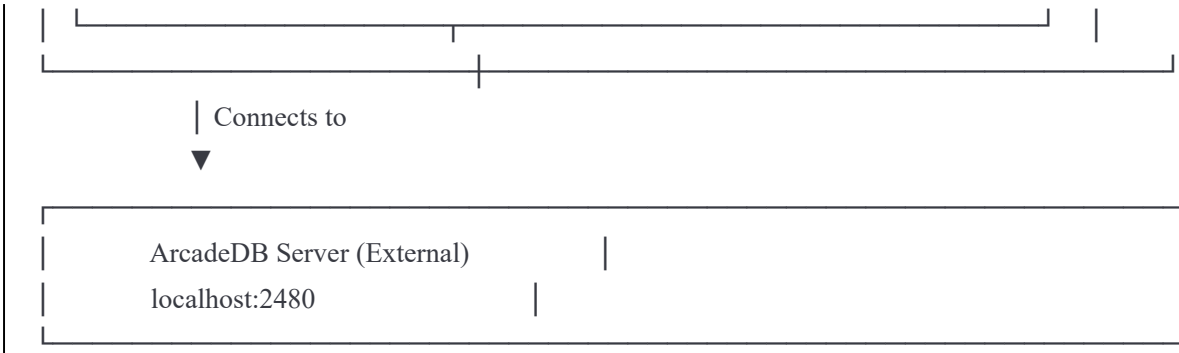
Spring Boot ArcadeDB Application - Architecture Deep Dive

Overall Architecture





🔌 Connection Establishment Flow

Step 1: Application Startup

```
Application.main()
    |
    ├─> Spring Boot initializes
    |
    ├─> Scans for @Configuration classes
    |
    └─> Finds ArcadeDbConfig
```

Step 2: Configuration Loading

java

```
// application.yml is loaded
arcadedb:
  host: localhost
  port: 2480
  username: root
  password: arcadedb
```

What happens:

1. Spring Boot reads `application.yml`
2. `@ConfigurationProperties` annotation on `ArcadeDbProperties` tells Spring to map these values
3. Spring creates `ArcadeDbProperties` bean with values from YAML

java

```java
@ConfigurationProperties(prefix = "arcadedb")
public class ArcadeDbProperties {
    private String host = "localhost";  // ← Gets "localhost" from YAML
    private int port = 2480;            // ← Gets 2480 from YAML
    private String username = "root";   // ← Gets "root" from YAML
    private String password = "arcadedb"; // ← Gets "arcadedb" from YAML
    // ... getters/setters
}
```

Step 3: Bean Creation (ArcadeDbConfig)

java

```java
@Configuration
@EnableConfigurationProperties(ArcadeDbProperties.class)
public class ArcadeDbConfig {

    @Bean(destroyMethod = "close")
    public RemoteServer remoteServer(ArcadeDbProperties properties) {
        // This method is called by Spring during startup

        // 1. Create connection to ArcadeDB
        RemoteServer server = new RemoteServer(
            properties.getHost(),     // "localhost"
            properties.getPort(),     // 2480
            properties.getUsername(), // "root"
            properties.getPassword()  // "arcadedb"
        );

        // 2. Test the connection
        server.databases(); // If this fails, app won't start

        // 3. Return the connected server (Spring manages it)
        return server;
    }
}
```

What `@Bean` does:

- Tells Spring: "This method creates an object I want you to manage"
- Spring calls this method once during startup
- The returned `RemoteServer` is stored in Spring's container
- Spring injects this bean wherever it's needed

What `destroyMethod = "close"` does:

- When application shuts down, Spring calls `server.close()`
- Ensures clean connection closure

Step 4: Behind the Scenes - RemoteServer Connection

java

```
// Inside ArcadeDB's RemoteServer class (simplified)
public RemoteServer(String host, int port, String username, String password) {

    // 1. Create HTTP client
    this.httpClient = new HttpClient();

    // 2. Build base URL
    this.baseUrl = "http://" + host + ":" + port;
    // Result: "http://localhost:2480"

    // 3. Store credentials for authentication
    this.username = username;
    this.password = password;

    // 4. Create connection pool for efficiency
    this.connectionPool = new ConnectionPool();

    // Note: Connection is lazy - doesn't actually connect until first use
}
```

When actual connection happens:

java

```java
// When you call server.databases()
public List<String> databases() {
    // 1. Creates HTTP request
    HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create(baseUrl + "/api/v1/databases"))
        .header("Authorization", "Basic " + encodeCredentials())
        .GET()
        .build();

    // 2. Sends request to ArcadeDB server
    HttpResponse<String> response = httpClient.send(request);

    // 3. Parses JSON response
    return parseJsonToList(response.body());
}
```

Step 5: Dependency Injection

```java
@Service
public class ArcadeDbService implements DatabaseService {

    private final RemoteServer remoteServer;

    // Constructor injection - Spring automatically injects RemoteServer
    public ArcadeDbService(RemoteServer remoteServer) {
        this.remoteServer = remoteServer;  // ← Spring provides this
    }
}
```

How Spring does this:

1. Spring has `RemoteServer` bean (from Step 3)
2. Spring sees `ArcadeDbService` needs `RemoteServer` in constructor
3. Spring passes the bean to the constructor automatically

Step 6: Controller Gets Service

```java
```

```
@RestController
public class DatabaseController {

    private final DatabaseService databaseService;

    // Spring injects ArcadeDbService (implements DatabaseService)
    public DatabaseController(DatabaseService databaseService) {
        this.databaseService = databaseService;
    }
}
```

🔄 Request Flow Example

Let's trace a complete request: `GET /api/v1/databases`

Step-by-Step Execution:

```
1. Client sends HTTP request
   ↓
   curl http://localhost:8080/api/v1/databases

2. Spring's DispatcherServlet receives request
   ↓
   Matches URL pattern to DatabaseController.getAllDatabases()

3. DatabaseController.getAllDatabases() is called
   ↓
   @GetMapping
   public ResponseEntity<DatabaseResponse> getAllDatabases() {
       List<DatabaseInfo> databases = databaseService.getAllDatabases();
       // ...
   }

4. databaseService.getAllDatabases() is called
   ↓
   public List<DatabaseInfo> getAllDatabases() {
       List<String> databaseNames = remoteServer.databases();
       // ...
   }

5. remoteServer.databases() makes HTTP call to ArcadeDB
   ↓
   HTTP GET → http://localhost:2480/api/v1/databases
   Authorization: Basic cm9vdDphcmNhZGVkYg==
```

6. ArcadeDB Server processes request
↓
- Validates credentials
- Queries internal database catalog
- Returns list of databases as JSON

7. RemoteServer receives response
↓
["mydb", "testdb", "productdb"]

8. ArcadeDbService transforms to domain objects
↓
List<DatabaseInfo> with DatabaseInfo("mydb"), etc.

9. DatabaseController transforms to DTOs
↓
DatabaseResponse with DatabaseDto objects

10. Spring serializes to JSON
↓
```
{
  "databases": [{"name": "mydb"}, {"name": "testdb"}],
  "count": 2,
  "timestamp": "2025-11-05T10:30:00"
}
```

11. Response sent to client
↓
HTTP 200 OK with JSON body

🔐 Connection Details

Authentication Flow

java

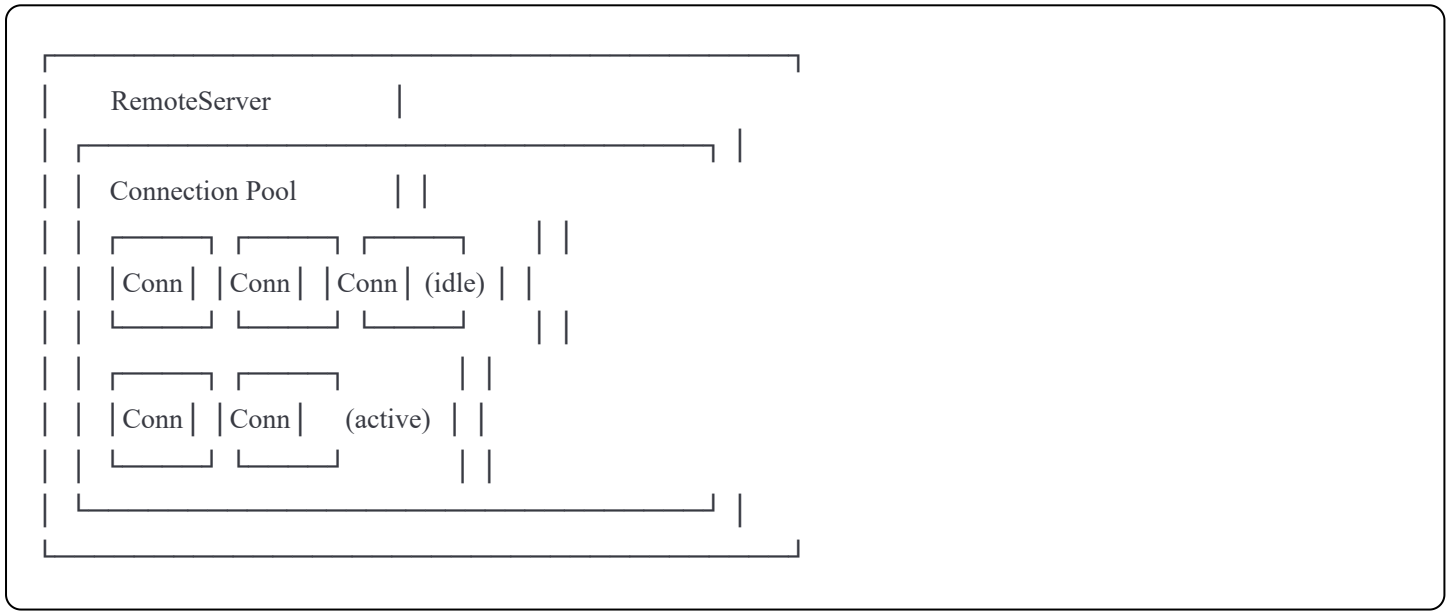
```
// When RemoteServer makes a request:

1. Encode credentials
   username:password → root:arcadedb
   Base64 encode → cm9vdDphcmNhZGVkYg==

2. Add to HTTP header
   Authorization: Basic cm9vdDphcmNhZGVkYg==

3. ArcadeDB validates
   - Decodes Base64
   - Checks username/password against user database
   - Returns 401 if invalid, 200 if valid
```

Connection Pooling (if implemented)



💡 Key Concepts Explained

1. Dependency Injection (DI)

Without DI (Bad):

```java
public class DatabaseController {
    // Tightly coupled - hard to test
    private DatabaseService service = new ArcadeDbService();
}
```

With DI (Good):

```java
public class DatabaseController {
    private final DatabaseService service;

    // Spring provides the implementation
    public DatabaseController(DatabaseService service) {
        this.service = service;
    }
}
```

Benefits:

- Easy to test (can inject mock)
- Easy to swap implementations
- Spring manages lifecycle

2. Bean Lifecycle

```
Application Starts
    ↓
1. Spring Container Created
    ↓
2. Configuration Classes Processed (@Configuration)
    ↓
3. Beans Created (@Bean methods called)
    ↓
4. Dependencies Injected (Constructor injection)
    ↓
5. Post-Processing (@PostConstruct methods)
    ↓
Application Running (Beans in memory)
    ↓
Application Shuts Down
    ↓
6. Pre-Destroy (@PreDestroy, destroyMethod)
    ↓
7. Beans Destroyed
    ↓
Spring Container Closed
```

3. Layered Architecture (SOLID)

```
API Layer (DatabaseController)
  ↓ Depends on interface
```

```
Domain Layer (DatabaseService interface)
  ↓ Implemented by
Infrastructure Layer (ArcadeDbService)
  ↓ Uses
External Systems (ArcadeDB)
```

Why this matters:

- Change database? Only modify Infrastructure layer
- Change API format? Only modify API layer
- Business logic stays isolated in Domain layer

4. Configuration Properties

```yaml
# application.yml
arcadedb:
  host: localhost
  port: 2480
```

↓ Automatically maps to ↓

```java
@ConfigurationProperties(prefix = "arcadedb")
public class ArcadeDbProperties {
    private String host;  // Gets "localhost"
    private int port;     // Gets 2480
}
```

Validation:

```java
@Min(value = 1, message = "Port must be greater than 0")
private int port = 2480;
```

If port is invalid, application fails to start with clear error.

🎯 SOLID Principles in Action

1. Single Responsibility Principle (SRP)

- `DatabaseController`: Only handles HTTP
- `ArcadeDbService`: Only handles business logic
- `ArcadeDbConfig`: Only handles configuration

2. Open/Closed Principle (OCP)

```
java

// Want to add caching? Extend without modifying existing code
@Service
public class CachedDatabaseService implements DatabaseService {
    private final DatabaseService delegate;
    private final Cache cache;

    // Decorator pattern - open for extension
}
```

3. Liskov Substitution Principle (LSP)

```
java

// Any DatabaseService implementation can be used
DatabaseService service = new ArcadeDbService(remoteServer);
// OR
DatabaseService service = new CachedDatabaseService(delegate);
// OR
DatabaseService service = new MockDatabaseService();
```

4. Interface Segregation Principle (ISP)

```
java

// Small, focused interface
public interface DatabaseService {
    List<DatabaseInfo> getAllDatabases();
    boolean databaseExists(String name);
    DatabaseInfo createDatabase(String name);
}
// Not: getAllDatabases(), createUser(), deleteRecord(), etc.
```

5. Dependency Inversion Principle (DIP)

```java
// Controller depends on abstraction (interface)
public class DatabaseController {
    private final DatabaseService service; // ← Interface, not concrete class
}
```

🧪 Testing Benefits

Because of this architecture, testing is easy:

```java
@Test
public void testGetAllDatabases() {
    // Mock the service
    DatabaseService mockService = mock(DatabaseService.class);
    when(mockService.getAllDatabases())
        .thenReturn(List.of(new DatabaseInfo("testdb")));

    // Inject mock into controller
    DatabaseController controller = new DatabaseController(mockService);

    // Test without touching real database
    ResponseEntity<DatabaseResponse> response = controller.getAllDatabases();
    assertEquals(200, response.getStatusCodeValue());
}
```

🚀 Performance Considerations

1. **Connection Reuse**: `RemoteServer` bean is singleton - one connection for all requests
2. **Thread Safety**: Spring controllers are thread-safe by default
3. **Connection Pooling**: Multiple threads can use the same `RemoteServer`
4. **Lazy Loading**: Connection only established when first used

---

🔍 Troubleshooting Connection Issues

Check 1: Is Spring creating the bean?

```bash
# Add to application.yml
logging:
  level:
    org.springframework.beans: DEBUG
```

Check 2: Can you reach ArcadeDB?

```bash
curl http://localhost:2480/api/v1/server
```

Check 3: Are credentials correct?

```bash
curl -u root:arcadedb http://localhost:2480/api/v1/databases
```

Check 4: Is RemoteServer created?

Add debug logging in `ArcadeDbConfig`:

```java
@Bean
public RemoteServer remoteServer(ArcadeDbProperties properties) {
    logger.info("Creating RemoteServer with host={}, port={}",
            properties.getHost(), properties.getPort());
    // ...
}
```

This architecture ensures:

- ✅ Clean separation of concerns
- ✅ Easy to test
- ✅ Easy to maintain
- ✅ Easy to extend
- ✅ Type-safe configuration

- ✅ Proper resource management