Runtime repository generation in Spring Boot: Advancing REST API automation with ByteBuddy

N. WASZKOWIAK

norbert.waszkowiak@wat.edu.pl

Military University of Technology, Faculty of Cybernetics
Kaliskiego Str. 2, 00-908 Warsaw, Poland

Spring Boot simplifies REST API development but still requires repetitive code for controllers, services, and repositories. This article presents a proof-of-concept for the @RestEntity annotation, which uses ByteBuddy to dynamically generate repositories at runtime. By utilizing Java reflection and bytecode manipulation, our approach automatically creates repository interfaces that Spring Boot seamlessly integrates into its context, significantly reducing boilerplate code while maintaining framework compatibility.

Keywords: code generation, spring boot, reflection.

DOI: 10.5604/01.3001.0055.5707

1. Introduction

Modern web application development relies heavily on REST APIs, with Spring Boot serving as a popular framework for implementing such interfaces. While Spring Boot provides robust mechanisms for creating REST endpoints, the traditional approach often necessitates repetitive implementation of controllers, services, and repositories for each domain entity, resulting in significant boilerplate code and increased maintenance overhead.

Existing tools such as Project Lombok and Spring Data REST offer partial automation but do not fully eliminate the need for manual repository definition. This paper introduces a novel approach to automating repository generation and registration in Spring Boot, aiming to further reduce boilerplate and streamline API development.

2. Research Gap and Literature Review

The automation of code generation has become a significant area of research aimed at improving developer productivity and reducing software maintenance costs. [1] While modern frameworks like Spring Boot simplify the development of REST APIs, the conventional three-tier architecture still requires the manual creation of boilerplate code for each entity, including repositories, services, and controllers. This repetitive process is not only time-consuming but also prone to human error.

Existing solutions offer partial remedies to this issue. For instance, Spring Data JPA automates the implementation of repository interfaces, providing standard CRUD (Create, Read, Update, Delete) methods without requiring developers to write concrete implementation classes. [2] Similarly, libraries like Project Lombok reduce boilerplate for domain models by generating getters, setters, and constructors via annotations. Spring Data REST takes this a step further by exposing existing repository interfaces directly as REST endpoints, thereby eliminating the need for a separate controller layer. However, a critical limitation persists: developers must still manually define a Java interface file for each repository.

Recent advancements have explored runtime code generation as a more comprehensive solution. Libraries such as ByteBuddy provide a powerful API for creating and modifying Java classes dynamically during application execution. The concept of generating entities and repositories at runtime has been explored, but its practical implementation within the Spring ecosystem presents significant challenges. A primary obstacle is the integration of these dynamically generated repository beans into Spring's Application Context. The framework's dependency injection mechanism relies on discovering beans at startup, but runtime-generated classes are not available during the initial component scan. As a result, developers

have struggled to inject and use these dynamic repositories, often finding that attempts to autowire them result in null references.

While the broader field of code generation is rapidly advancing, with machine learning models showing promise in tasks like code-to-code translation and description-to-code synthesis [1], these approaches often introduce concerns regarding code quality and security [3]. They do not specifically address the architectural challenge of integrating dynamically created data access layers within a dependency-injection-based framework like Spring Boot.

This paper addresses a distinct and well--defined research gap: the lack of a fully automated, annotation-driven mechanism for generating Spring Data repository interfaces and ensuring their seamless registration and injection within the Spring Boot application context at runtime. Our work moves beyond the manual file creation required by Spring Data REST and addresses the integration challenges that have hindered previous runtime generation efforts, thereby providing a complete end-to-end solution for boilerplate reduction in REST API development.

3. Spring Boot REST API architecture

The traditional architecture of REST APIs in Spring Boot is a cornerstone of modern web application development. It provides a structured and modular approach to building scalable, maintainable, and efficient APIs. [4], [5] This section examines the conventional structure of REST APIs in Spring Boot, focusing on its core components: the domain model, data access layer, service layer, and controllers. Using a case study of an Employee entity, we explore how these components interact to deliver a robust API while highlighting the challenges posed by boilerplate code.

3.1. Domain Model and Data Access Layer

In Spring Boot applications, the domain model represents the core business entities of the application.

These entities are typically implemented as Java classes annotated with Java Persistence API (JPA) annotations to map them to database tables. For example, an *Employee* entity might be defined as follows (Code 1).

Code 1. Entity "Employee"

```
@Entity
public class Employee {
    @Id
    @GeneratedValue(
strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String position;
   //Getters, setters, constructors
}
```

Key annotations such *as @Entity*, *@Id*, and *@GeneratedValue* enable automatic mapping between Java objects and database records. This abstraction reduces the need for manual SQL queries while maintaining a clear representation of the data model.

The data access layer in Spring Boot leverages Spring Data JPA to simplify interactions with the database. By extending predefined interfaces like *JpaRepository* (Code 2), developers can create repositories that provide built-in methods for common operations such as saving, updating, deleting, and retrieving entities.

Code 2. JPA repository "EmployeeRepository"

```
public interface EmployeeRepository
extends JpaRepository<Employee, Long> {}
```

This repository interface automatically supports CRUD operations without requiring additional implementation. Moreover, custom queries can be defined using method names or the *@Query* annotation (Code 3).

Code 3. Use *@Query* in "findByName" method

```
@Query("SELECT e FROM
       Employee e WHERE e.name = ?1")
List<Employee> findByName(String name);
```

The repository pattern abstracts database operations, promoting cleaner code and easier testing.

3.2. Service Layer and Controllers

The service layer acts as an intermediary between the data access layer and controllers. It encapsulates business logic and ensures that controllers remain focused on handling HTTP requests. A typical service class might look like this (Code 4),

Code 4. Example service class "EmployeeService"

```
@Service
public class EmployeeService {
    private final EmployeeRepository
repository;

    public EmployeeService(
EmployeeRepository repository)
    {this.repository = repository;}

    public Employee
createEmployee(Employee employee) {
        return repository.save(employee);
    }
}
```

By centralizing business logic in services, developers achieve better separation of concerns and improved reusability.

Controllers handle HTTP requests and map them to appropriate service methods. In Spring Boot, controllers are annotated with @RestController and use request mapping annotations such as *@RequestMapping*, *@PostMapping*, and *@GetMapping* (Code 5).

Code 5. Example controller class "EmployeeController"

```
@RestController
@RequestMapping("/api/employees")
public class EmployeeController {
    private final EmployeeService service;

    public
EmployeeController(EmployeeService
service) {
        this.service = service;
    }

    @PostMapping
    public ResponseEntity<Employee>
create(@RequestBody Employee employee) {
        return new
ResponseEntity<>(service.createEmployee(em
ployee), HttpStatus.CREATED);
    }
}
```

This approach ensures that each controller method corresponds to a specific HTTP endpoint and operation (e.g., POST for creating resources). The use of annotations simplifies routing while maintaining clarity.

3.3. Challenges of Traditional Architecture

While the traditional architecture of REST APIs in Spring Boot is effective, it often results in significant boilerplate code. For every new entity, developers must create:

- An entity class with getters, setters, and constructors;
- A repository interface for data access;
- A service class for business logic;
- A controller class for handling HTTP requests.

This repetitive process can lead to increased development time and potential errors. Tools like Project Lombok partially address this issue by generating boilerplate code such as getters and setters through annotations like *@Data* (Code 6).

Code 6. Example entity class "Employee" with *@Data* annotation

```
@Data
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy          =
GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String position;
}
```

However, Lombok does not eliminate the need for creating repositories, services, or controllers.

4. Reflection and its role in ByteBuddy and Spring Boot

Reflection is a powerful programming concept that allows a program to inspect and modify its own structure and behavior at runtime. It plays a critical role in modern frameworks such as Spring Boot, enabling dynamic code generation, dependency injection, and runtime adaptability. This section explores the concept of reflection, its types and levels, and its application in tools like ByteBuddy. Furthermore, we discuss the differences between runtime code generation and compile-time approaches, focusing on their implications for performance, flexibility, and debugging

4.1. Reflection in Java: Concept and Levels

Reflection in Java refers to the ability of a program to examine or modify its structure during execution. [6] This includes inspecting classes, methods, fields, annotations, and even dynamically invoking methods or creating objects. The Java Reflection API (*java.lang.reflect*) provides the tools necessary for these operations.

Reflection is widely used in frameworks like Spring Boot for tasks such as:

- Dependency Injection: Dynamically injecting beans into the application context;
- Annotation Processing: Reading custom annotations to configure application behavior;
- Dynamic Proxy Creation: Generating proxy classes for method interception.

Reflection can be categorized into four distinct levels based on its scope and impact (Table 1).

Tab. 1. Reflection levels

Level	Description	Example
Introspection	Passive inspection of code structure (e.g., listing class methods or fields).	Debugging tools, ORM frameworks (e.g., Hibernate).
Structural	Modification of code structure (e.g., adding/removing fields at runtime).	Dynamic proxy generation, AOP frameworks.
Computational	Alteration of program execution flow (e.g., intercepting method calls).	Security policies, runtime optimizations.
Linguistic	Modification of programming language constructs at runtime.	Domain-specific language extensions.

These levels differ in their impact on runtime behavior. While introspection is read-only and safe for performance, structural and computational reflection introduce greater flexibility but come with potential overheads.

4.2. ByteBuddy: Dynamic Code Generation Using Reflection

ByteBuddy is a Java library designed for runtime code generation and manipulation [7]. It simplifies the process of creating new classes or modifying existing ones by abstracting away the complexities of bytecode manipulation. ByteBuddy relies heavily on reflection to analyze class structures and dynamically generate new types.

Key features of ByteBuddy include:

- **Dynamic Class Creation:** Generating new classes or interfaces at runtime;
- **Method Interception:** Modifying method behavior using proxies;
- **Annotation Injection:** Adding annotations to classes or methods dynamically.

ByteBuddy operates primarily at the structural level of reflection by allowing developers to define new classes or interfaces programmatically.

In Spring Boot applications, ByteBuddy is often used for advanced tasks such as:

1. **Dynamic Proxy Creation:** Spring uses ByteBuddy to generate proxies for beans annotated with *@Transactional* or *@Async*;
2. **Repository Generation:** tools like Spring Data REST leverage ByteBuddy to create dynamic repositories based on entity definitions;
3. **Aspect-Oriented Programming (AOP):** ByteBuddy enables method interception for cross-cutting concerns like logging or security.

For example, in the context of *@RestEntity*, ByteBuddy can scan annotated classes during runtime and generate corresponding repository interfaces dynamically (Code 7).

This approach reduces boilerplate code while maintaining flexibility.

Code 7. Creating new repository interfaces using ByteBuddy

```
DynamicType.Unloaded<?> unloaded = new
                                  ByteBuddy()
    .makeInterface(TypeDescription.Generic
       .Builder.parameterizedType(
         JpaRepository.class,
         entityClass,
         Long.class
     ).build())
    .name("pl.edu.wat.demo.repository." +
entityClass.getSimpleName() + "Repository")
    .annotateType(AnnotationDescription
          .Builder.ofType(
       RepositoryRestResource.class)
       .build())
    .make();
```

4.3. Runtime vs Compile-Time Code Generation

Runtime code generation refers to creating or modifying classes during application execution. Reflection plays a central role in this process by enabling dynamic analysis and manipulation of class structures.

Advantages:

- **Flexibility:** Classes can adapt to changing requirements during runtime;
- **Reduced Boilerplate:** Frameworks like Spring Data REST eliminate the need for manually written repositories.

Disadvantages:

- **Performance Overhead:** Reflection introduces latency due to runtime type checks;
- **Debugging Complexity:** Dynamically generated classes can be harder to trace during debugging.

Compile-time code generation involves creating classes during the build process using annotation processors or other tools (e.g., Lombok). This approach avoids runtime overhead by generating static code that is included in the compiled application.

Advantages:

- **Performance Efficiency:** No runtime processing is required;
- **Early Error Detection:** Compile-time errors are easier to catch than runtime issues.

Disadvantages:

- **Limited Flexibility:** Classes cannot adapt dynamically after compilation;
- **Additional Build Complexity:** Requires integration with build tools like Maven or Gradle.

Spring Boot combines both approaches:

1. **Runtime Generation:**
   - Dynamic proxies for repositories (JpaRepository);
   - AOP aspects using reflection-based proxies.
2. **Compile-Time Generation:**
   - Annotation processors for configuration classes (@Configuration);
   - Lombok-generated boilerplate code (@Data, @Builder).

The choice between runtime and compile--time generation depends on the application's requirements for performance versus flexibility.

5. Custom annotation @RestEntity

Our approach implements a custom annotation-driven mechanism for generating Spring Data REST repositories using ByteBuddy. The *@RestEntity* annotation triggers entity scanning during application startup, where ByteBuddy dynamically creates repository interfaces annotated with *@RepositoryRestResource*. These generated repositories integrate seamlessly with Spring Boot's classpath scanning mechanism, enabling automatic registration as Spring beans. Consequently, full CRUD endpoints become available without manual implementation of controllers, services, or repository interfaces. The following code implements this mechanism. The code that implements this looks as follows (Code 8–11).

Code 8. "@RestEntity" implementation. Class: "RestEntityScanner"

```
//ReflectionEntityScanner.java

//import…

@AllArgsConstructor
@Slf4j
public class RestEntityScanner {

    private final TypePool typePool;
    private final String basePackage;
    private final String sourceRootPath;

    public Set<TypeDescription> scanForAnnotatedEntities() {
        Set<TypeDescription> annotatedEntities = new HashSet<>();
        String packagePath = basePackage.replace('.', '/');
        Path basePath = Paths.get(sourceRootPath, packagePath);

        try (Stream<Path> pathStream = Files.walk(basePath)) {
            Set<String> javaClasses = pathStream
                    .filter(Files::isRegularFile)
                    .filter(path -> path.toString().endsWith(".java"))
                    .map(path -> {
                        // Convert file path to class name
                        String relativePath = basePath.relativize(path).toString();
                        String className = relativePath.replace(".java",
"").replace('/', '.');
                        return basePackage + (className.isEmpty() ? "" : "." +
className);
                    })
                    .collect(Collectors.toSet());

            // Check each class for @RestEntity annotation
            for (String className : javaClasses) {
                try {
                    TypeDescription typeDesc = typePool.describe(className).resolve();
                    if
(typeDesc.getDeclaredAnnotations().isAnnotationPresent(typePool.describe(RestEntity.clas
s.getName()).resolve())) {
                        log.debug("Found @RestEntity annotated class: {}", className);
                        annotatedEntities.add(typeDesc);
                    }
                } catch (Exception e) {
                    log.warn("Error resolving class {}: {}", className, e.getMessage());
                }
            }
        } catch (IOException e) {
            log.error("Error scanning for annotated entities: {}", e.getMessage());
        }

        return annotatedEntities;
    }
}
```

Code 9. "@RestEntity" implementation. Annotation „RestEntity"

```
//RestEntity.java

//import…

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface RestEntity {
}
```

Code 10. "@RestEntity" implementation. Class „ReflectionSetter". Part 1

```
//ReflectionSetter.java part 1

//import…

@AllArgsConstructor
@Slf4j
public class ReflectionSetter {

    @Getter
    private final TypePool typePool;
    private final String repositoryPackage;
    private final ByteBuddy byteBuddy;
    private final String basePackage;
    private final String sourceRootPath;
    private final File outputDir;

    public static void main(String[] args) {
        new ReflectionSetter(
                TypePool.Default.ofSystemLoader(),
                "pl.edu.wat.demo.repository",
                "pl.edu.wat.demo",
                "src/main/java",
                new File("build/classes/java/main")
        ).generateRepositories();
    }

    @Builder
    public ReflectionSetter(
            TypePool typePool,
            String repositoryPackage,
            String basePackage,
            String sourceRootPath,
            File outputDir) {

        // Required parameters
        this.typePool = Optional.ofNullable(typePool).orElseThrow(() ->
                new IllegalArgumentException("TypePool cannot be null"));
        this.outputDir = Optional.ofNullable(outputDir).orElseThrow(() ->
                new IllegalArgumentException("OutputDir cannot be null"));

        // Parameters with defaults
        this.basePackage = Optional.ofNullable(basePackage).orElse("pl.edu.wat.demo");
        this.repositoryPackage =
Optional.ofNullable(repositoryPackage).orElse(this.basePackage + ".repository");
        this.sourceRootPath =
Optional.ofNullable(sourceRootPath).orElse("src/main/java");

        // Create ByteBuddy instance
        this.byteBuddy = new ByteBuddy();
    }

    public void generateRepositories() {
        RestEntityScanner scanner = new RestEntityScanner(typePool, basePackage,
sourceRootPath);
        Set<TypeDescription> annotatedEntities = scanner.scanForAnnotatedEntities();

        log.info("Found {} entities annotated with @RestEntity",
annotatedEntities.size());

        for (TypeDescription entityClass : annotatedEntities) {
            generateRepositoryForEntity(entityClass);
        }
    }
```

Code 11. "@RestEntity" implementation. Class „ReflectionSetter". Part 2

```
//ReflectionSetter.java part 2

    private void generateRepositoryForEntity(TypeDescription entityClass) {
        String entityName = entityClass.getSimpleName();
        String repositoryName = entityName + "Repository";

        // Get JpaRepository type description
        TypeDescription repositoryTypeDescription =
typePool.describe("org.springframework.data.jpa.repository.JpaRepository").resolve();
        TypeDescription idTypeDescription =
typePool.describe("java.lang.Long").resolve();

        // Create repository interface with generic types
        DynamicType.Unloaded<?> unloaded = byteBuddy.makeInterface(
                        TypeDescription.Generic.Builder
                                .parameterizedType(
                                        repositoryTypeDescription,
                                        entityClass,
                                        idTypeDescription)
                                .build())
                .name(repositoryPackage + "." + repositoryName)

.annotateType(AnnotationDescription.Builder.ofType(typePool.describe("org.springframewor
k.data.rest.core.annotation.RepositoryRestResource").resolve()).build())
                .make();

        // Save the generated interface to the output directory
        try {
            unloaded.saveIn(outputDir);
            log.info("Generated repository interface: {}", repositoryName);
        } catch (IOException e) {
            log.error("Failed to save repository interface {}: {}", repositoryName,
e.getMessage());
        }
    }
}
```

5.1. Annotation Definition and Entity Scanning

The *@RestEntity* (Code 9) annotation is defined with runtime retention and targets types, serving as a marker for entities requiring a REST repository interface. A custom scanner, *RestEntityScanner* (Code 8), traverses the project's source directory, converts file paths to class names, and uses a *TypePool* to load and inspect each class's annotations; only classes annotated with *@RestEntity* are collected for repository generation. This approach confines the developer's responsibility to adding a single annotation on an entity, delegating discovery to a reusable, annotation-driven scanner.

5.2. Dynamic Repository Generation with ByteBuddy

The core of the proof-of-concept is implemented in *ReflectionSetter* (Code 10, 11), which instantiates a *ByteBuddy* engine and iterates over the discovered entity types. For each entity, it constructs a parameterized interface extending *JpaRepository<TEntity, Long>*, names it after the entity (e.g.,*EmployeeRepository*), and adds the *@RepositoryRestResource* annotation. ByteBuddy then emits the interface's bytecode into *.class* files in the build output directory, effectively creating repository interfaces without writing a single line of repository code.

5.3. Automatic Detection and Integration in Spring Boot

Once generated, the repository interfaces reside under *build/classes/java/main* in the appropriate package, which Spring Boot includes in its classpath at startup. Spring Data REST's component scan detects interfaces extending *JpaRepository* and annotated with *@RepositoryRestResource*, dynamically creating controllers and mapping CRUD endpoints for each entity. As a result, full RESTful API operations become available at paths derived

from repository definitions-no explicit controller or service beans are required, and standard HATEOAS support (pagination, sorting, hypermedia links) is provided out of the box.

6. Conclusion

This research demonstrates that *@RestEntity* represents a meaningful evolution beyond Spring Data REST's *@RepositoryRestResource* annotation. While Spring Data REST eliminates controller implementation by exposing repositories as REST endpoints, developers must still manually create repository interfaces. Our *@RestEntity* approach takes automation one step further by dynamically generating these repositories at runtime using ByteBuddy, requiring only a single annotation on domain entities.

This proof-of-concept illustrates how reflection and bytecode manipulation can further streamline Spring Boot development, reducing boilerplate code compared to standard Spring Data REST implementations. As microservices and API-first architectures continue to dominate enterprise development, such annotation-driven automation techniques offer promising avenues for improving developer productivity while maintaining Spring Boot's flexibility and conventions.

7. Bibliography

[1] Siswo Utomo M., Utami E., Kusrini K., Setyanto A., "Machine Learning Innovations in Code Generation: A Systematic Literature Review of Methods, Challenges and Directions", *2024 International Conference on Information Technology and Computing* (*ICITCOM*), Yogyakarta, Indonesia, 2024, pp. 24–29, 10.1109/ICITCOM62788.2024.10762291.

[2] Spring Data JPA – Reference , https://docs.spring.io/spring-data/jpa/docs/3.1.7/reference/html/ (accessed: 13 May 2025).

[3] Ramírez L.C., Limón X., Sánchez--García Á.J., Pérez-Arriaga J.C., "State of the Art of the Security of Code Generated by LLMs: A Systematic Literature Review", *2024 12th International Conference in Software Engineering Research and Innovation* (*CONISOFT*), Puerto Escondido, Mexico, 2024, pp. 331–339, 10.1109/CONISOFT63288.2024.00050.

[4] Spring Framework Documentation, https://docs.spring.io/spring-framework/reference/index.html (Accessed: 13 May 2025).

[5] Walls C., *Spring Boot in Action*, Manning Publications, 2023.

[6] Java Reflection API Documentation, https:// docs.oracle.com (accessed: 13 May 2025).

[7] ByteBuddy Documentation, https:// bytebuddy.net (accessed: 13 May 2025).

Generowanie repozytorium w Runtime w Spring Boot: zaawansowana automatyzacja API REST z ByteBuddy

N. WASZKOWIAK

Spring Boot upraszcza tworzenie REST API, ale nadal wymaga powtarzalnego kodu dla kontrolerów, serwisów i repozytoriów. Artykuł przedstawia dowód koncepcji adnotacji *@RestEntity*, która używa ByteBuddy do dynamicznego generowania repozytoriów w czasie wykonania. Wykorzystując refleksję w Javie i manipulację kodem bajtowym, przedstawiono podejście, w którym automatycznie tworzone są interfejsy repozytoriów. Spring Boot płynnie integruje je ze swoim kontekstem, znacznie redukując kod przy jednoczesnym zachowaniu kompatybilności oprogramowania.

Słowa kluczowe: generowanie kodu, spring boot, refleksja.