Critical Analysis of the Relationship Between OOP and Web Services: The Influence of ORM on Software Design in Web-Based Environments

Justin Hendy

June 2025

Abstract

This essay critically analyses the relationship between object-oriented programming (OOP) and web services, with a particular focus on the influence of object-relational mapping (ORM) on software design in web-based environments. It explores the conceptual foundations of OOP and web services, examines the role of ORM in simplifying database interactions, and evaluates how frameworks like Spring Boot enhance maintainability, modularity, and reusability. The essay also discusses the challenges and future perspectives of ORM-influenced software design, providing recommendations for improving ORM strategies in large-scale enterprise applications.

Contents

1 Introduction **2**

2 Conceptual Foundations **2**
2.1 Principles of Object-Oriented Programming (OOP) 2
2.2 Overview of Web Services 6

3 Object-Relational Mapping (ORM) in Web Services **7**
3.1 Definition and Significance of ORM 7
3.2 Advantages and Limitations of ORM 9

4 Spring Boot and OOP **9**
4.1 Spring Boot's Integration with OOP 9
4.2 Design Patterns in Spring Boot 11
4.3 Best Practices for Scalable Web Services 12

5 Challenges and Future Perspectives **14**
5.1 Common Challenges in OOP-based Web Services with ORM 14
5.2 Emerging Trends and Implications 15
5.3 Recommendations for Improving ORM Strategies 15

6 Conclusion **16**

1 Introduction

The key concepts and techniques associated with object-oriented programming (OOP) and web services have transformed the development of modern software solutions. OOP, with its core principles of encapsulation, inheritance, polymorphism, and abstraction, is a programming paradigm or set of ideas that, when implemented, allow modern software solutions to embrace modularity, reusability, and maintainability within their design. At the same time, web services provide standardised methods for communication over a network that simplify the exchange, interpretation, and processing of data between applications written in different languages and running on different platforms. The technique of object-relational mapping (ORM) facilitates integration between OOP and relational databases, providing a structure that simplifies database interactions by allowing developers to leverage OOP principles to create objects and set their properties thus enabling developers to focus on business logic rather than low-level database operations bridging the gap between the worlds of object-oriented programming and relational databases.

This essay will critically analyse the relationships between OOP and web services, with a particular focus on the influence of ORM on software design in web-based environments. It will provide a comprehensive analysis of these relationships, highlighting the benefits and challenges of integrating OOP, web services, and ORM in modern software design through four key topics: conceptual foundations, ORM in web services, Spring Boot and OOP and finally, challenges and future perspectives. To cover the conceptual foundations of OOP and web services, this essay will define the key principles of OOP and their role in software design and provide an overview of web services, especially RESTful architectures, and its relevance to modern software engineering. To explore the role of ORM in web services, this essay will define ORM and discuss its significance in application development, explain how ORM tools such as Hibernate simplify database interactions in an OOP paradigm and discuss the advantages and limitations of ORM in large-scale web applications. Further discussion, analysis and evaluation will examine how frameworks like Spring Boot integrate with OOP principles to enhance maintainability, modularity, and reusability; evaluate design patterns used in Spring Boot and their alignment with OOP principles and discuss best practices in designing scalable and efficient web services using Spring Boot and ORM. Finally, this essay will explore the challenges and future perspectives of ORM-influenced software design and web-based development by identifying common challenges faced when implementing OOP-based web services with ORM, such as performance overheads, lazy loading issues, and data consistency concerns and then explore emerging trends in web service architectures and their implications on OOP and ORM. Lastly, this essay will provide recommendations for improving ORM strategies in large-scale enterprise applications. Appropriately sourced and relevant literature will support key points of discussion, and snippets of code will illustrate pertinent techniques associated with specific topics.

2 Conceptual Foundations

2.1 Principles of Object-Oriented Programming (OOP)

This section defines the key concepts and principles of object-oriented programming (OOP), encapsulation, inheritance, polymorphism, and abstraction. This section will also discuss the benefits of OOP in software design, such as modularity, maintainability, and reusability.

Before explaining the key principles of OOP, a clear definition of the terms maintainability, reusability, and modularity is required to assess their role in software design.

Baggen et al. list six properties for defining the quality of a codebase's maintainability: volume, redundancy, unit size, complexity, unit interface size, and coupling [1]. Volume refers to the size of the application, where the larger the codebase, the harder it is to maintain. Redundancy occurs when

duplicated code exists, resulting in unnecessary additional maintenance. Unit size refers to the size of individual low-level pieces of functionality, where large units with an unclear focus increase the maintenance burden. A complex codebase is hard to understand and, therefore, difficult to maintain. Unit interface size refers to the number of parameters per unit, with more parameters indicating a lack of encapsulation. Finally, tightly coupled components are challenging to change and, therefore, harder to maintain, as any change to a unit will impact the other. Lerthathairat and Prompoon succinctly define maintainability as "time required to fix a fault" within a piece of software [2].

Frakes and Anguswamy highlight three frequently employed reuseability design principles: well-defined interface, documentation, clarity, and understandability [3]. A well-defined interface, measured by its simplicity, will ensure that a component can be "reused and interconnected with other components". Clarity and understandability relate to the ability to comprehend the purpose and structure of a component, thereby providing developers with confidence in how and where to use it. Frakes and Anguswamy also note that a component's size negatively impacts its reusability. Reusability improves code quality and productivity [4].

Modularity separates functionality and data into succinct units, clearly defined in their purpose and compatibility [5]. There are strategies for implementing modularity, such as the MVC model or Model-View-Controller, which sets out a pattern for separating concerns. Domain logic resides within the model, rendering for interaction sits with the view, and the controller covers the ability to interact with the system [5]. Shaw also highlights three clear reasons why modularity is important: intellectual control, segmentation of work, and evolution. Intellectual control enables the development of an understanding of each module without requiring understanding or knowledge of any other module within the system. Evolution ensures that updates or changes to individual modules have no impact on the broader system. Segmentation of work through modularisation provides an opportunity to assign responsibility based on the knowledge and skillset of the developer.

When introduced, OOP built upon and extended the first programming paradigm, known as procedural programming, employed by languages such as C, FORTRAN, and COBOL [6] [7]. In these languages, functions and procedures enabled developers to write code that performed specific tasks, but they had limited control over how data was structured or managed between these functions. For a time, the priority placed on procedural organisation was of the utmost importance, but gradually, the focus of software design shifted towards the organisation of data [6]. Blansit neatly places these differences in the context of grammar, where procedural programming has a verb focus, while object-oriented programming has a noun focus [7]. Object-oriented programming (OOP) places objects at the centre of software design, where an object has defined attributes and behaviours. The OOP paradigm provides a system for defining and modelling real-world entities and the interactions between them [8] [9].To illustrate this point, consider a series of real-world objects, such as a person, a jet, an animal, or a car, where their attributes and behaviours are a blueprint. A person's attributes could include their name, age, and hair colour, as well as behaviours such as walking, talking, and sleeping. A jet will have a wingspan, top speed, and weight and can fly, land and take off. The name given to these generalised objects within programming parlance is 'class'. A class is a template or a mould that a developer can use to define the attributes and behaviours of a general object and then instantiate multiple instances of an object from this template or mould [10]. The main principles that underpin the ideas of OOP include encapsulation, inheritance, polymorphism and abstraction, all of which help provide the key benefits of the OOP paradigm, specifically modularity, maintainability and reusability.

Encapsulation is the principle of creating objects or classes that contain attributes (data) and behaviours (methods) that operate on data related to that class; a developer uses encapsulation to hide or restrict access to a class's internal state or data [11] [10] [12]. A well-encapsulated class can, therefore, implement rules or business logic that determine how or when a class's data is updated, ensuring appropriate use of exposed elements [11]. For example, a class representing a bank account might encapsulate

attributes such as balance and account number and provide methods for depositing and withdrawing funds. Both methods can implement rules that determine how money can be deposited or withdrawn, such as not allowing withdrawals that exceed the sum of money currently available to the account holder. Direct access to the balance data type would open up the possibility of data corruption, as no control is in place to prevent incorrect usage. One of the key benefits of the OOP paradigm is modularity. The independence and interchangeable nature of components determine a system's modularity [5]. Well-encapsulated objects are key to achieving this state of modularity, as the interaction between each object is allowed but only under specific conditions, and each module or object can fulfil its purpose without requiring any further knowledge of the other objects within the system.

When creating a class or object, it is essential to recognise which attributes and behaviours may be required by multiple instances of that class. Within OOP, classes can inherit attributes and behaviours and then add further attributes and behaviours specific to that instance [13]. For example, a Dog class will inherit attributes such as size and weight, as well as behaviours for eating and sleeping, from the Animal class, however a dog may also have a behaviour that is only specific to a dog, like fetching a ball. In this example, the Animal class serves as the parent or base class, and the Dog class is a subclass that extends the Animal class. Relationships between classes formed by inheritance are called "is-a" relationships [10]. Developers create a hierarchy of classes, which in turn reduces redundancy and promotes code reuse, with reusability being a key benefit of OOP.

The principle of polymorphism adds flexibility and reusability to OOP by providing a structure for objects and methods to take on multiple forms [10]. In OOP, a base class defines a method, and polymorphism allows this same method to be overridden in subclasses, enabling different behaviours for the same method name depending on the object type. For example, a method called 'makeSound()' could be defined in an Animal class, and subclasses like Dog and Cat could override this method to produce different sounds (barking and meowing, respectively). Through polymorphism, developers write code that can work with objects of different classes while treating them uniformly and consistently. Polymorphism reduces the need for duplicate code and, therefore, also promotes code reuse [14].

Abstraction is a similar principle to encapsulation in that it deals with hiding aspects of a class or object. Abstraction focuses on hiding the internal logic and complexity of a class to simplify interactions with that object [13]. Abstract classes and interfaces provide a mechanism within OOP to achieve abstraction. Returning to the animal example, the term animal refers to an abstract idea, in that it is a template for an object, not a concrete example of an object, such as a cat or a dog; therefore, a developer can create an abstract class that declares certain behaviours where derived subclasses of animal must also contain the same behaviours and provide the implementation details. Abstraction may also sound similar to the principles of inheritance and polymorphism, and while there are relationships between them, they are distinct. Abstraction focuses on defining what a class does, inheritance sets out what a class is and what attributes and behaviours it inherits from its base class and polymorphism defines how the output of the same behaviour differs depending on the class carrying out that behaviour. By employing abstraction to simplify complex implementation details, developers can design software with improved maintainability, as they no longer need to understand the low-level details of individual classes or modules. Here is a code example that illustrates the principles of OOP, including encapsulation, inheritance, polymorphism, and abstraction using the animal example previously discussed.

```java
// Abstract class Animal
public abstract class Animal {

    // Encapsulated attributes
    private String name;
    private int age;

    public Animal(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    // Abstract method for sound
    public abstract String makeSound();
}

// Dog class inherits from Animal
public class Dog extends Animal {

    // Additional encapsulated attribute specific to Dog
    private String breed;

    public Dog(String name) {
        super(name);
    }

    // Overriding the makeSound method
    @Override
    public String makeSound() {
        return "Woof!";
    }

    // Additional method specific to Dog
    public void fetchBall() {
        System.out.println(getName() + " is fetching the ball!");
    }
}

// Cat class inherits from Animal
public class Cat extends Animal {

    // Additional encapsulated attribute specific to Cat
    private int lengthOfWhiskers;

    public Cat(String name) {
        super(name);
    }

    // Overriding the makeSound method
    @Override
    public String makeSound() {
        return "Meow!";
    }
}
```

```java

// Main class to demonstrate OOP principles
public class Main {
    public static void main(String[] args) {

        // Creating instances of Dog and Cat
        Animal dog = new Dog("Buckley");
        Animal cat = new Cat("Smokey");

        // Demonstrating polymorphism
        System.out.println(dog.getName() + " says: " + dog.makeSound());
        System.out.println(cat.getName() + " says: " + cat.makeSound());

        // Demonstrating additional Dog-specific behaviour
        ((Dog) dog).fetchBall();
    }
}
```

2.2 Overview of Web Services

This section outlines the definition of the web services paradigm and its relevance to modern software engineering with a particular focus on RESTful architectures.

Web services are a specific type of application programming interface (API) with standardised methods for communication over a network, typically the internet, that allow different software programs to complete various data-related transactions, where application-to-application communication is deemed interoperable; it occurs regardless of the platform or language [15]. The application-to-application relationship utilises the client-server architecture, where the client requests a particular service, and the server responds to the request. Web services will broadly fall under one of two types [16]: SOAP (Simple Object Access Protocol) and REST (Representational State Transfer). SOAP uses XML (Extensible Markup Language) for message formatting and typically uses HTTP (Hypertext Transfer Protocol) or SMTP (Simple Mail Transfer Protocol) for message transmission; REST provides a high-level framework with no specific implementation that uses standard HTTP methods (GET, POST, PUT, DELETE) to carry out operations on resources identified by URLs [17]. The remainder of this section will focus on RESTful architectures and their relevance to modern software engineering.

Pautasso et al. summarise the four technological principles that underpin the RESTful architectural style: resource identification through URI, uniform interface, self-descriptive messages and stateful interactions through hyperlinks [18] . Within this context, a resource refers to "the intended conceptual target of a hypertext reference" [19] or, in other words, the actual entity identified as existing at a given URI such as a document, image, video, web page, etc... A web service developed within this architecture style provides access to a required resource using a URI, such as a URL (Uniform Resource Locator). REST web services provide a uniform interface that facilitates interaction with the identified resource via the standard set of HTTP verbs - GET, PUT, POST and DELETE and subsequently provide full CRUD (Create, Read, Update and Delete) capabilities to any client utilising said service. Self-descriptive messages ensure that each message contains sufficient information, allowing the intended recipient to process and understand it, including content-type (data file type), resource representation (the format or layout of the data), metadata, and hypermedia (links to related resources or actions). This self-sufficiency ensures that each interaction with the web service is stateless, meaning knowledge of any prior interactions between the client and the server is unnecessary, and each request made by the client is processed individually by the server [20]. Due to the stateless nature of each interaction, the final technological principle of stateful interactions through hyperlinks enables a client to carry out

additional actions or view further resources via the aforementioned embedded hypermedia; Fielding referred to this as hypermedia as the engine of application state (HATEOAS) [19]. The four constraints outlined here provide a lightweight infrastructure for quickly developing a working web service, where, in some cases, client-side software is not even necessary to test the working web service, and a simple web browser will suffice [18]. Due to this simplicity and use of existing web protocols (HTTP, XML, URI, etc.) REST has become the most prevalent method for delivering web services on the Web [17].

To summarise, REST-based web services are relevant to modern software engineering for several reasons. Firstly, due to the close alignment and leveraging of the protocols and "principles that support the World Wide Web" [17], RESTful services are generally considered relatively simple [18]. Secondly, the stateless nature of RESTful interactions reduces the complexity of server-side development, resulting in more maintainable and scalable applications. Thirdly, and this applies to web services in general, due to the language and platform agnosticism, clients written in any language and on any platform can freely interact and consume a REST service, making them highly interoperable; where, due to the diversity of the modern software topography, interoperability has become a vital component in creating valuable and successful applications [20]. Finally, the HATEAOS principle provides clients with a mechanism to navigate through the application state without requiring prior knowledge of the system. However, Neumann et al. highlight that the implementation of this feature in modern web services is scarce [17].

3 Object-Relational Mapping (ORM) in Web Services

3.1 Definition and Significance of ORM

This section defines the key ORM concepts and their purposes, discussing their significance in modern software development. It will examine ORM tools, such as Hibernate, and their ability to simplify database interactions within the OOP paradigm.

Lorenz et al. highlight the standard practice of employing relational database systems (RDBS) to store or persist objects defined within an object-oriented environment [21]; however, the underlying concepts or principles of OOP and relational databases are not naturally compatible. As a result, object-relational mapping (ORM) techniques and tools emerged in the 1990s as a response to these challenges posed by the mismatched paradigms [22]. Take the OOP concept of inheritance as an example; this has no equivalence within the RDBS paradigm and its flat structure [22], or the use of objects as data models within OOP compared to the use of tables within an RDBS. The term often used to describe this issue of incompatibility is object-relational impedance mismatch [22]. The ability to map between the two and solve the impedance mismatch is a key purpose of the ORM programming technique and associated frameworks [23].

Through the application of ORM techniques, a developer can minimise, or even eliminate, the need to write complex SQL queries for database interactions, potentially resulting in increased code readability and maintainability [23]. ORM provides developers with the tools to create an application using object-oriented programming (OOP) and map key components to a relational database, such as objects and their properties to rows and columns within a database table. ORM carries significant importance within modern software development, primarily through its ability to simplify database interactions, improve code maintainability, and enhance productivity [21]. ORM frameworks, such as Hibernate, Doctrine, or SQLAlchemy, implement ORM techniques by generally providing a set of APIs and utilities that aim to automate the majority of the mapping process [24]. ORM frameworks also reduce the existence or need for boilerplate code, provide facilities for the inclusion of parameterised queries [23], and often include features for managing transactions, handling relationships between objects, and caching, further enhancing productivity and maintainability in large-scale applications. Ultimately, through the

successful implementation of ORM, a developer can significantly reduce development time and remove complex or lengthy SQL queries, which in turn lowers the risk of errors whilst improving the readability of the code [23].

ORM frameworks facilitate the implementation of Object-Relational Mapping (ORM) in applications. As previously mentioned, they provide a set of APIs and utilities that allow developers to interact with relational databases using object-oriented programming (OOP) principles. Some popular ORM frameworks include Hibernate (Java), Entity Framework (.NET), and SQLAlchemy (Python). Hibernate implements the Java/Jakarta Persistence API (JPA), a specification for object-relational mapping in Java. Hibernate has several key features, including automatic SQL generation (covered previously), caching, lazy loading, and transaction management [25]. Hibernate will cache frequently accessed data in memory, providing several benefits, including improved performance (reduced number of trips to retrieve data) and reduced database load (fewer database queries). Lazy loading is another feature that can improve performance. When a relationship exists between entities (objects), lazy loading ensures only the object currently being accessed is retrieved, with related objects "held back" until they are required [26]. In the code snippet below, a relationship exists between the Course entity and the CourseName entity, with the relationship fetch type set to lazy. When the course is loaded, the course name will only load when it is required.

```java

@Entity
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int courseId;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "degreeType")
    private DegreeType degreeType;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "courseName")
    private CourseName courseName;

    // Other attributes, getters, and setters
}

@Entity
public class CourseName {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int courseNameId;

    private String courseName;

    @OneToMany(fetch = FetchType.LAZY, targetEntity = Course.class, mappedBy = "
    courseName")
    private List<Course> courseNames;

    // Other attributes, getters, and setters
}
```

Hibernate ensures database transactions follow the ACID (Atomicity, Consistency, Isolation, Durability) rules through a transaction interface containing a set of key methods, such as begin(), commit(), rollback(), etc [27].

3.2 Advantages and Limitations of ORM

ORM has several advantages when applied to large-scale web applications. Firstly, ORM frameworks like Hibernate automatically map Java objects and database tables, reducing the amount of boilerplate code and allowing developers to allocate their time and resources to writing business logic [24]. Secondly, ORM frameworks are not tied to any single RDBS, allowing for greater flexibility, which is particularly beneficial for large-scale applications that may require multi-database support. Thirdly, by bridging the "semantic gap" [21] between OOP and RDBS, ORM frameworks can manage complex data structures and transactions, ensuring data integrity across large, distributed systems —a crucial advantage, considering the potential complexity of data models in large-scale web applications.

For example, a large university will enrol thousands of students, employ a large staff base, offer a wide range of courses and pathways, carry out and store research data, maintain complex financial records, and so on. Addressing the key university concern of student and course data, ORM can simplify data management by mapping course, student, course name, module, and course type data to objects and providing tools for defining relationships between them. Developers can perform CRUD operations on courses, students and modules without needing to write complex SQL queries. The ORM framework automatically generates the essential SQL statements and also provides intuitive systems for creating additional, complex queries, ensuring maintainability and reusability as business requirements change.

ORM also has several limitations when applied to large-scale web applications. Firstly, when dealing with large datasets or complex queries, ORM can encounter performance issues, such as the N+1 query problem or inefficient SQL queries [28]. N+1 query problems occur when ORM executes several SQL queries to extract relevant data when a single query would suffice. This problem is particularly prevalent when using lazy loading, especially with a large dataset and/or a loop that retrieves the associated entities. The second limitation stems from the potentially steep learning curve associated with using an ORM framework.. The additional layer of abstraction can present issues when trying to understand the underlying database interactions, and learning any ORM-related specialised syntax will also add to the development time. Finally, while database independence is an advantage of ORM, ORM frameworks do not always utilise the full range of features associated with a particular database; therefore, the importance of utilising these features may outweigh the benefits supplied by the ORM framework.

4 Spring Boot and OOP

4.1 Spring Boot's Integration with OOP

This section will analyse how Spring Boot integrates with OOP concepts of encapsulation, polymorphism, abstraction and inheritance to enhance maintainability, modularity, and reusability; provide an evaluation of Spring Boot design patterns and their alignment with OOP principles; and finally discuss best practices in designing a scalable and efficient web service using Spring Boot and ORM.

The Spring framework, at a basic level, provides developers with a suite of tools for producing enterprise applications within the Java programming language. Spring Boot simplifies the process of developing Spring web applications by minimising the amount of configuration required [29] [30]. Li and Gui further highlight that Spring Boot's strength lies in its ability to simplify processes that are ubiquitous in "real-world development" and, as a result, increase the rate at which developers produce

Java web applications [30].

Spring Boot leverages OOP concepts to enhance modularity, reusability, and maintainability within its applications. The code example provided below is a typical Spring Boot project that includes packages representing the different layers of an application, controllers, services, repositories, and models.

```java
package com.example.demo.controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import com.example.demo.service.UserService;

@RestController
public class UserController {

    private final UserService userService;

    // Constructor injection for UserService
    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping("/users")
    public List<User> getAllUsers() {
        return userService.getAllUsers();
    }
}


package com.example.demo.service;
import org.springframework.stereotype.Service;
import com.example.demo.repository.UserRepository;
import com.example.demo.model.User;

@Service
public class UserService {

    private final UserRepository userRepository;

    // Constructor injection for UserRepository
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User createUser(String name) {
        if (name == null || name.isEmpty()) {
            throw new IllegalArgumentException("Name cannot be null or empty");
        }
        User user = new User();
        user.setName(name);
        return userRepository.save(user);
    }
}


package com.example.demo.repository;
import org.springframework.data.jpa.repository.JpaRepository;
import com.example.demo.model.User;

```

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
}

package com.example.demo.model;
import jakarta.persistence.*;

@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    // Getters and setters
}
```

The 'UserController' handles HTTP web requests concerning the 'User' class. The 'UserService' contains business logic for interacting with the 'User' class, and the 'UserRepository' interacts with the persistence layer, such as a relational database. The UserService in this example provides logic that connects the creation of a User to the UserRepository, ensuring business rules are adhered to; in this case, the name cannot be empty. The UserRepository provides several data access and management commands 'out of the box', but Spring Boot also provides an intuitive system for writing more complex commands; for example, something like the following code would query the database ignoring the case of the entered letters, and return all users with a full or partial match:

```
List <User> fetchAllUsersByNameContainingIgnoreCase(String name);
```

Given the range of potential queries and depth of data extraction, the codebase's complexity is relatively low.

The structure is modular in design, with a clear separation of concerns between each package, resulting in a highly maintainable codebase. The use of encapsulation within every class ensures that public methods or interfaces are the only means of accessibility for the hidden data and implementation details. The use of interfaces for the repository layer also enables reusability through a technique called dependency injection, where the service layer's dependency on the repository layer necessitates the injection of the repository layer into the service layer. The repository layer also employs abstraction via the extension of the JpaRepository class. The JpaRepository class contains the CrudRepository and the PagingAndSortingRepository, providing the API for all CRUD operations and paging and sorting operations [31].

Spring Boot utilises annotations, such as @RestController, @Service, and @Repository, which enhance code readability while providing a clear definition of the roles of each module within the application. However, they also serve another purpose. Annotations like @RestController or @Service are known as beans; a bean instructs Spring Boot to create an object and make it available for use throughout the application via dependency injection. The use of beans within Spring Boot promotes modularity and reusability, whilst also ensuring loose coupling between modules.

4.2 Design Patterns in Spring Boot

The following section will evaluate specific design patterns employed by Spring Boot. Two key design patterns used in Spring Boot are Dependency Injection and the Repository Pattern.

Dependency Injection (DI) enables developers to deliver functionality from one class to another by injecting the required class rather than declaring it within the target class [32]. Sun et al. highlight

several ways in which DI aligns with OOP principles. DI improves maintainability by reducing developer effort in sharing functionality; this also leads to loose coupling between units and a reduction in class complexity. DI allows developers to swap out dependencies or mock them for testing purposes easily [33] . Within Spring Boot, the most common type of DI is constructor injection (as highlighted in previous code example) [32].

The code provided in section 4.1 utilised the repository design pattern. In this pattern, the repository serves as an abstraction layer for interacting with the underlying database, enabling developers to define repository interfaces that encapsulate CRUD operations whilst decoupling the business logic [34]. The article "Designing the infrastructure persistence layer" highlights the maintainability and decoupling of the data access layer (CRUD operations) from the domain model (business logic) as key benefits associated with the repository design pattern [35].

4.3 Best Practices for Scalable Web Services

Identifying key factors that contribute to the scalability and efficiency of a web service is essential for discussing best practices within the same field. Bondi highlights several issues to avoid or consider when designing software architecture that impact performance and scalability, such as a class that interacts with too many components, searching multiple tables to find the required data, and fixed data structure sizes [36]. Roe and Gonik outline a limited selection of principles that define scalable architecture; these include divide and conquer, asynchrony, encapsulation, concurrency, and parsimony [37]. Divide and conquer, encapsulation and parsimony are of particular interest concerning Spring Boot and ORM. Divide and conquer is similar to modularity, where, specifically, a "system should be partitioned into relatively small subsystems, each carrying out some well-focused function." Encapsulation in this context is no different to what has been discussed already. Parsimony requires lightweight and efficient implementation and code where "each line of code and each piece of state information has a cost", and object models should only contain data that is necessary [37]. Roe and Gonik also emphasise the importance of a "service-based layered architecture", including a breakdown of the application stack with its related layers and their respective responsibilities. The identified layers are presentation, application, services, domain and persistence. Finally, Jogalekar and Woodside define scalability within distributed systems, such as web-based applications, e-commerce, and distance learning, as "...not just the ability to operate, but to operate efficiently and with adequate quality of service..." [38], meaning a measure of a systems scalability is its ability to continue to providing a quality and efficient service as its load increases. With this information, implementing a layered architecture and using a Data Transfer Object (DTO) would be two practices well-suited to designing scalable and efficient web services.

A layered architecture separates concerns into distinct layers, thereby applying the divide-and-conquer principle, as outlined by Roe and Gonik. The Spring Boot layered architecture comprises a controller layer, a service layer, and a repository layer. The practice of employing a layered architecture also aligns partially with the aforementioned service-layered architecture, as referenced by Roe and Gonik. A layered architecture promotes modularity, allowing for easier testing and maintenance, independent scaling opportunities, and future development.

Using a DTO allows for better control over the data sent to and from clients. In the context of scalability, only the data required is sent to the client, thereby improving performance. This level of control will also improve security by hiding sensitive information, such as the user's password in the example above. Here is an example of a layered architecture in Spring Boot that also includes a DTO:

```java
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String username;
    private String password;
    private String email;
    // Getters and setters
}

public class UserDTO {

    private Long id;
    private String username;
    private String email;
    // Getters and setters
}

@Service
public class UserDTOService {

    @Autowired
    private UserRepository userRepository;

    private convertToDTO(User user) {
        UserDTO userDTO = new UserDTO();
        userDTO.setId(user.getId());
        userDTO.setUsername(user.getUsername());
        userDTO.setEmail(user.getEmail());
        return userDTO;
    }

    public List<UserDTO> getAllUserDTO() {
        List<User> users = userRepository.findAll();
        return users.stream().map(this::convertToDTO).collect(Collectors.toList());
    }

    public UserDTO getUserDTOByID(int id) {
        User user = userRepository.findById(id).orElseThrow(() -> new
   RuntimeException("User not found"));
        return convertToDTO(user);
    }
}

@RestController
public class UserDetailsController {

    @Autowired
    private UserDTOService userDTOService;

    @GetMapping("/userdetails")
    public List<UserDTO> getAllUsers() {
        return userDTOService.getAllUserDTO();
    }

```

```
    @GetMapping("/userdetailss/{id}")
    public UserDTO getUserById(@PathVariable int id) {
        return userDTOService.getUserDTOByID(id);
    }
}

@RepositoryRestResource(collectionResourceRel = "user", path = "user")
public interface UserRepository extends JpaRepository<User, Long> {
    List<User> findByUsernameContainingIgnoreCase(String username);
}
```

In this example, as in the previous code example in section 4.1, the User entity sets the data model, except this time with fields for username, password and email. The UserDTO data model sets the specific user data to transfer between layers and expose to the client; in this case, it is just the ID, username, and email. The UserDTOService contains business logic methods, consisting of two public and one private method. To persist any data handled within this service, an instance of the UserRepository is injected into this class as a field injection. The private convertToDTO method converts a User object to a UserDTO object. The two public methods return either all users as UserDTOs or a single user as a UserDTO. Both methods retrieve data from the database via the UserRepository and convert any User objects received into UserDTOs through calls to the private convertToDTO method. The first public method returns a list of all users converted to UserDTOs, and the second takes an ID parameter passed by the client, returning either the requested user as a UserDTO or a message indicating that no user with that ID exists. The UserRepository provides communication with the database via standard JpaRepository queries, as well as an additional query for a partial, case-insensitive username search. Finally, the UserDetailsController provides the RESTful interface for HTTP requests, where the @GetMapping annotation defines the HTTP GET method and the path for each request, as well as any parameters passed within the URL, as is the case with the getByUserId method.

5 Challenges and Future Perspectives

5.1 Common Challenges in OOP-based Web Services with ORM

In OOP-based web services that utilise ORM, several common challenges can arise, particularly concerning performance overheads, data consistency and transaction management.

Performance overheads occur due to the computational cost of generating SQL queries from object-oriented defined entities and relationships [39]. Colley et al. identify a range of specific performance issues related to ORM's dynamically created database queries, such as eager fetch, large execution plans that reduce the effectiveness of the plan cache, and the non-employment of window functions [28]. When employed, eager fetch loads all data related to an entity, regardless of whether all the data is required. For example, returning all social media posts made by a user when only the most recent is required is not an effective use of computational resources. When generating the results of an SQL query, the database creates an execution plan that seeks to optimise the query through the ordering of each operation, use of indexes, joins, and filters, and table scans [40]; the plan cache stores the execution plans to provide efficiency with repeated queries. The dynamic nature of ORM query generation and the numerous options available to developers via the repository layer can result in overly complex queries, leading to the creation of large execution plans that consume a significant portion of the plan cache. Minor variations to queries will also result in unique execution plans even if the majority of the resources employed by the database are similar. Without the use of window functions, any query that requires some form of ranking or cumulative summing will become unnecessarily complex [41].

Data inconsistencies arise due to the impedance mismatch that occurs when the data required by the application and the data requested by the ORM do not match [42], resulting in performance issues.

Database transactions are a collection of queries that either commit changes to the database upon successful completion of all queries or rollback to the previous state if any query fails to complete. The execution of overly complex queries can cause unexpected concurrency issues and prevent transactions from committing [43].

5.2 Emerging Trends and Implications

There are several emerging trends in web services architecture, such as microservices, cloud-native development and GraphQL, that will have implications for OOP and ORM practices.

Microservices are small, independent applications that communicate via interfaces or set events [44]. In a microservices architecture, each service may store its data in a decentralised database, which has implications for managing entities and their relationships across a collection of microservices.

With the increasing development of cloud-native software that utilises cloud computing for hosting and storage, new approaches to data management have emerged, such as the data lakehouse architecture (DLH) [45]. DLH handles both structured and unstructured data, offering flexibility in terms of storage formats and scalability. Where ORM frameworks map data to objects and types, it is clear that modern cloud data storage of the DLH type does not fit this object model due to the aforementioned variety of data types and storage systems.

GraphQL is a query language considered as an alternative to REST-based interfaces, providing users with the ability to create complex bespoke requests for the exact data they require in a single query as opposed to multiple queries with REST [46]. Marchuk et al. tested the performance of multiple database fetching methods, including direct database access, ORM, REST APIs and GraphQL and found that GraphQL required additional resources to match the performance of ORM or REST, however, GraphQL provided greater flexibility [47]. Praschal et al. developed a full-stack application with a GraphQL API interface that utilises ORM principles [48], highlighting the ability to leverage the benefits of both technologies in modern web solutions.

5.3 Recommendations for Improving ORM Strategies

Choosing the best ORM framework requires considering several key factors, including platform, performance, maintainability, and specific project requirements. Klimiuk et al. tested the performance of several ORM tools from the NuGet .NET package manager [49]. They specifically tested the speed at which each tool performed the standard database operations of SELECT, INSERT and UPDATE, with results showing that Dapper ORM had the highest performance. Alvarez-Eraso and Arango-Isaza highlight the tension between the maintainability benefits of an "off the shelf" ORM, such as Hibernate, and the performance of managing complex queries [50], suggesting that in non-performance-critical environments, the maintainability benefits provided by Hibernate make it a good choice but where higher performance is required these benefits can be lost. Połeć et al. found, when comparing the performance of different ORM frameworks available in Java, Apache OpenJPA exhibited the best performance but due to its complex configuration and documentation would limit its adoption; however, Hibernate excelled in these areas [51].

6 Conclusion

In conclusion, this essay has examined the integration of OOP principles with web services and ORM, with particular focus on the Spring Boot framework. Key findings highlight the importance of OOP concepts such as encapsulation, inheritance, polymorphism and abstraction in enhancing maintainability, modularity and reusability in web development as well as demonstrating how Spring Boot's integration with OOP principles provides developers with the tools for creating scalable and maintainable applications by leverging design patterns such as dependency injection and the repository pattern and through the use of best practices of employing DTOs and a layered architecture.

The essay also discussed the advantages and limitations of ORM frameworks, highlighting their role in simplifying database interactions and improving code maintainability, while drawing attention to several issues related to performance overheads, data consistency, and transaction management. Implications surrounding emerging trends, such as the use of microservices, cloud-based development, and GraphQL, were also discussed and investigated.

Finally, the essay presented a selection of recommendations based on various studies for enhancing ORM strategies in large-scale applications, with a focus on strategies for selecting the most suitable ORM framework and optimising ORM performance.

In summary, the integration of OOP principles with web services and ORM frameworks provides a powerful environment for building scalable and maintainable applications. By employing OOP concepts, design patterns, and best practices, developers can create robust web services that meet the demands of modern software development. The ever-changing landscape of web services will ensure that tools and strategies continue to adapt to stay relevant in the world of modern software development.

References

[1] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, no. 2, pp. 287–307, May 2011.

[2] P. Lerthathairat and N. Prompoon, "An approach for source code classification to enhance maintainability," in *2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE)*. IEEE, May 2011, pp. 319–324.

[3] R. Anguswamy and W. B. Frakes, "A study of reusability, complexity, and reuse design principles," in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, ser. ESEM '12. ACM, Sep. 2012, pp. 161–164.

[4] W. Frakes and C. Terry, "Software reuse: metrics and models," *ACM Computing Surveys*, vol. 28, no. 2, pp. 415–435, Jun. 1996.

[5] M. Shaw, "Modularity for the modern world: summary of invited keynote," in *Proceedings of the tenth international conference on Aspect-oriented software development*, ser. AOSD'11. ACM, Mar. 2011, pp. 1–6.

[6] B. Stroustrup, "What is object-oriented programming?" *IEEE Software*, vol. 5, no. 3, pp. 10–20, May 1988.

[7] B. D. Blansit, "Object oriented programming: What is it talking about?" *Journal of Electronic Resources in Medical Libraries*, vol. 7, no. 1, pp. 90–97, Mar. 2010.

[8] S. Xinogalos, "Object-oriented design and programming: An investigation of novices' conceptions on objects and classes," *ACM Transactions on Computing Education*, vol. 15, no. 3, pp. 1–21, Jul. 2015.

[9] F. Fallucchi and M. Gozzi, "Puzzle pattern, a systematic approach to multiple behavioral inheritance implementation in object-oriented programming," *Applied Sciences*, vol. 14, no. 12, p. 5083, Jun. 2024.

[10] J. Keogh and M. Giannini, *OOP demystified*, ser. Demystified series, J. E. Keogh, Ed. New York [u.a.]: McGraw-Hill/Osborne, 2004, name des 1. Verf. auf dem Umschlag: Jim Keogh.

[11] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," *ACM SIGPLAN Notices*, vol. 21, no. 11, pp. 38–45, Jun. 1986.

[12] N. Schärli, A. P. Black, and S. Ducasse, "Object-oriented encapsulation for dynamically typed languages," in *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA04. ACM, Oct. 2004, pp. 130–149.

[13] D. Halbert and P. O'Brien, "Using types and inheritance in object-oriented programming," *IEEE Software*, vol. 4, no. 5, pp. 71–79, Sep. 1987.

[14] H. Ali, "Exploring polymorphism: Flexibility and code reusability in object-oriented programming," *Passer Journal of Basic and Applied Sciences*, vol. 6, no. Special Issue, Jan. 2024.

[15] K. Gottschalk, S. Graham, H. Kreger, and J. Snell, "Introduction to web services architecture," *IBM Systems Journal*, vol. 41, no. 2, pp. 170–177, 2002.

[16] X. Shi, "The semantics of web services: An examination in giscience applications," *ISPRS International Journal of Geo-Information*, vol. 2, no. 3, pp. 888–907, Sep. 2013.

[17] A. Neumann, N. Laranjeiro, and J. Bernardino, "An analysis of public rest web service apis," *IEEE Transactions on Services Computing*, vol. 14, no. 4, pp. 957–970, Jul. 2021.

[18] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. "big"' web services: making the right architectural decision," in *Proceedings of the 17th international conference on World Wide Web*, ser. WWW '08. ACM, Apr. 2008, pp. 805–814.

[19] R. T. Fielding and R. N. Taylor, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, Donald Bren School of Information & Computer Sciences, 2000, aAI9980887.

[20] S. Ahmad, S. Ali, N. Waqar, N. S. Naz, and M. Hassaan Mehmood, "Comparative evaluation of the maintainability of restful and soap-wsdl web services," in *2023 International Conference on Business Analytics for Technology and Security (ICBATS)*. IEEE, Mar. 2023, pp. 1–9.

[21] M. Lorenz, G. Hesse, and J.-P. Rudolph, "Object-relational mapping revised - a guideline review and consolidation," in *Proceedings of the 11th International Joint Conference on Software Technologies*. SCITEPRESS - Science and Technology Publications, 2016, pp. 157–168.

[22] A. Torres, R. Galante, M. S. Pimenta, and A. J. B. Martins, "Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design," *Information and Software Technology*, vol. 82, pp. 1–18, Feb. 2017.

[23] G. Vial, "Lessons in persisting object data using object-relational mapping," *IEEE Software*, vol. 36, no. 6, pp. 43–52, Nov. 2019.

[24] C. Xia, G. Yu, and M. Tang, "Efficient implement of orm (object/relational mapping) use in j2ee framework: Hibernate," in *2009 International Conference on Computational Intelligence and Software Engineering*. IEEE, Dec. 2009, pp. 1–3.

[25] M. Borys and B. Panczyk, "Improving data processing performance for webapplications using object-relational mapping," *Aktual'Ni Problemy Ekonomiky = Actual Problems in Economics*, no. 164, pp. 387–395, 2015. [Online]. Available: https://www.proquest.com/scholarly-journals/improving-data-processing-performance-web/docview/1661350747/se-2

[26] R.-M. BÂRA, C. A. BOIANGIU, and C. TUDOSE, "Analysing the performance impacts of lazy loading in webapplications," *Journal of Information Systems & Operations Management*, vol. 18.1, May 2024.

[27] S. Pandey, "Advanced java — transaction management (hibernate)," https://medium.com/@SachinPandeyOnline/advanced-java-transaction-management-hibernate-7e8262f26484, Feb. 2025. [Online]. Available: https://medium.com/@SachinPandeyOnline/advanced-java-transaction-management-hibernate-7e8262f26484

[28] D. Colley, C. Stanier, and M. Asaduzzaman, "Investigating the effects of object-relational impedance mismatch on the efficiency of object-relational mapping frameworks," *Journal of Database Management*, vol. 31, no. 4, pp. 1–23, Oct. 2020.

[29] D. D. Saxena, D. J. Shah, V. K. Singh, H. Yadav, and V. K. Singh, "Transforming placement workflow: Springboot approach to campus recruitment management," in *2025 International Conference on Next Generation Communication & Information Processing (INCIP).* IEEE, Jan. 2025, pp. 909–913.

[30] L. Li and L. Gui, "Design and development of an online education system based on springboot," in *2024 3rd International Conference on Artificial Intelligence and Computer Information Technology (AICIT).* IEEE, Sep. 2024, pp. 1–6.

[31] GeekForGeeks, "Spring boot jparepository with example," https://www.geeksforgeeks.org/spring-boot-jparepository-with-example/, Aug. 2024. [Online]. Available: https://www.geeksforgeeks.org/spring-boot-jparepository-with-example/

[32] T. D. Fernando, "Understanding dependency injection in spring boot," https://dev.to/tharindufdo/understanding-dependency-injection-in-spring-boot-2ll0, Jun. 2025. [Online]. Available: https://dev.to/tharindufdo/understanding-dependency-injection-in-spring-boot-2ll0

[33] P. Sun, D.-K. Kim, H. Ming, and L. Lu, "Measuring impact of dependency injection on software maintainability," *Computers*, vol. 11, no. 9, p. 141, Sep. 2022.

[34] Java-Design-Patterns, "Repository pattern in java: Simplifying data access with abstracted persistence," https://java-design-patterns.com/patterns/repository/, Jun. 2025.

[35] Microsoft, "Design the infrastructure persistence layer," https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design, Feb. 2023. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design

[36] A. B. Bondi, "The software architect as the guardian of system performance and scalability," in *2009 ICSE Workshop on Leadership and Management in Software Architecture.* IEEE, May 2009, pp. 28–31.

[37] C. Roe and S. Gonik, "Server-side design principles for scalable internet systems," *IEEE Software*, vol. 19, no. 2, pp. 34–41, 2002.

[38] P. Jogalekar and M. Woodside, "Evaluating the scalability of distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 6, pp. 589–603, Jun. 2000.

[39] N. R. D. Riyanto and S. Rochimah, "Optimization of object relational mapping: Query profiling integration based on frequency and timestamps implementation in lru," in *2025 International Conference on Advancement in Data Science, E-learning and Information System (ICADEIS).* IEEE, Feb. 2025, pp. 1–6.

[40] D. Bhatia, "Understanding sql query execution plans: Unveiling the path to database performance," https://divyanshbhatia.medium.com/understanding-sql-query-execution-plans-unveiling-the-path-to-database-performance-ad9aee04b9f7, Aug. 2024. [Online]. Available: https://divyanshbhatia.medium.com/understanding-sql-query-execution-plans-unveiling-the-path-to-database-performance-ad9aee04b9f7

[41] V. Leis, K. Kundhikanjana, A. Kemper, and T. Neumann, "Efficient processing of window functions in analytical sql queries," *Proceedings of the VLDB Endowment*, vol. 8, no. 10, pp. 1058–1069, Jun. 2015.

[42] T.-H. Chen, "Improving the quality of large-scale database-centric software systems by analyzing database access code," in *2015 31st IEEE International Conference on Data Engineering Workshops*. IEEE, Apr. 2015, pp. 245–249.

[43] A. Bonvoisin, C. Quinton, and R. Rouvoy, "Understanding the performance-energy tradeoffs of object-relational mapping frameworks," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Mar. 2024, pp. 626–636.

[44] R. Laigner, Y. Zhou, and M. A. V. Salles, "A distributed database system for event-based microservices," in *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*, ser. DEBS '21. ACM, Jun. 2021, pp. 25–30.

[45] A. AbouZaid, P. J. Barclay, C. Chrysoulas, and N. Pitropakis, "Building a modern data platform based on the data lakehouse architecture and cloud-native ecosystem," *Discover Applied Sciences*, vol. 7, no. 3, Feb. 2025.

[46] O. Hartig and J. Pérez, "Semantics and complexity of graphql," in *Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW '18*, ser. WWW '18. ACM Press, 2018, pp. 1155–1164.

[47] Y. Marchuk, I. Dyyak, and I. Makar, "Performance analysis of database access: Comparison of direct connection, orm, rest api and graphql approaches," in *2023 IEEE 13th International Conference on Electronics and Information Technologies (ELIT)*. IEEE, Sep. 2023, pp. 174–176.

[48] C. Praschl, S. Bauernfeind, C. Leitner, and G. A. Zwettler, "Domain-driven design as model contract in full-stack development," in *2023 3rd International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*. IEEE, Jul. 2023, pp. 1–6.

[49] M. Klimiuk, P. Karabowicz, and M. Plechawska-Wójcik, "Comparative analysis of database mapping frameworks available in nuget manager," *Journal of Computer Sciences Institute*, vol. 32, pp. 231–238, Sep. 2024.

[50] D. A. Alvarez-Eraso and F. Arango-Isaza, "Hibernate and spring - an analysis of maintainability against performance," *Revista Facultad de Ingeniería Universidad de Antioquia*, no. 80, Sep. 2016.

[51] M. Połeć, J. Pitera, and G. Kozieł, "Comparing the performance of the object-relational mapping program-ming frameworks available in java," *Journal of Computer Sciences Institute*, vol. 22, pp. 59–65, Mar. 2022.