

How Object-Relational Mapping (ORM) influences software design and development in a web-based environment.

Abdulmalik Elghedafi

May 2025

1 Introduction

This report introduces the Conceptual foundations going over what OOP is, discussing the OOP paradigms, with examples, then provides an overview of web services vs REST. It discusses ORM and the significance in application development with the advantages and disadvantages. An analysis of how Spring integrates with OOP concepts is discussed, along with the design patterns and best practices in designing scalable and efficient web services using Spring boot and ORM. Common challenges and future perspectives is also discussed specifically issues such as performance overheads, lazy loading, and data consistency concerns. An exploration of emerging trends such as microservices is provided. As well as a recommendation in improving ORM strategies in enterprise applications

2 Conceptual foundations

2.1 What is OOP?

Object-oriented programming (OOP) is a software development paradigm that represents real-world entities using classes and objects. The main objective is to support core software qualities such as code reusability, reliability and ease of maintenance. In OOP, objects are the primary building blocks instantiated from class definitions. A class acts as a blueprint to specify the attributes and behaviours an object uses, constituting a core element of software development [1] [2] [3].

2.2 Describe Encapsulation

Encapsulation is a fundamental concept in object-oriented programming where variables and methods are grouped together in a single class. By declaring fields as private within the class and using public getter and setter methods,

this prevents data from being directly modified. Java provides three types of access modifiers which control visibility and different levels of security within a class [4] [3].

- **Public:** is accessible from outside the class and is accessed freely.
- **Private:** restricted access to the same class and protects internal data.
- **Protected:** is accessible from the same class and subclasses that are also defined in other packages.

A getter allows private data to be read safely. While a setter can modify the data by updating the field using additional logic [4] [3]. For example, a thermostat class would use a getter which provides read access to the private temperature field. Meanwhile, the setter provides logic i.e. ensuring the temperature value stays between 5 and 15 Celsius. This helps prevent negative or invalid values from being set.

2.3 Describe Inheritance

In OOP, Inheritance is a fundamental concept. A subclass can inherit features from the superclass such as methods and fields. This is done using the extend keyword to inherit properties from the superclass [5]. This represents the “is-a” relationship which allows a subclass to inherit all the non-primitive members of the superclass [3]. Java uses single inheritance by extending to one superclass. However, it does not support multiple inheritances between classes but can be done through interfaces [6].

An example of inheritance is the dog and cat class in listing 1 which introduce their own methods `fetch` and `climb`, which are not implemented in the superclass.

Animal.java

```
1
2 // Superclass
3 public class Animal {
4     public void speak() {
5         System.out.println("Animal is speaking");
6     }
7 }
8
9 // subclass
10 class Dog extends Animal {
11
12     public void fetch() {
13         System.out.println("Dog is fetching for sticks ");
14     }
15 }
16
17 // subclass
```

```

18     class Cat extends Animal {
19         public void climb() {
20             System.out.println("Cat is climbing");
21         }
22     }

```

Listing 1: Animal superclass with Dog and Cat subclasses

Main.java

```

1     public class Main {
2         public static void main(String[] args) {
3
4             Dog dog = new Dog();
5             Cat cat = new Cat();
6
7             dog.fetch();
8             cat.climb();
9         }
10    }

```

Listing 2: Main - method calling the subclass methods

By creating an instance of the dog and cat objects and calling their method in listing 2. A unique behaviour is printed in the output as shown in listing 3.

Main.java

```

1     Dog is fetching for sticks
2     Cat is climbing

```

Listing 3: Output for main

Advantages of inheritance is code reusability, where subclasses automatically have access to the superclass functionality, it also provides method overriding that is only possible through inheritance. Inheritance ensures code is maintainable, especially in the superclass without the requirement of updating in each subclass. However, the limitation of Inheritance is that it can lead to tight coupling, making it challenging if code is changed in the superclass, without modifying the subclass. It can also make readability of code difficult to understand if a deep hierarchy or multiple inheritance is implemented.

2.4 Describe Polymorphism

Polymorphism is another key fundamental in OOP where objects can demonstrate their own specific behaviour depending on the class type. It allows the same method from the super class to be overridden in the subclass and change behaviour based on the method called at runtime. A subclass can then add new functionality such as new methods or override existing methods to provide unique behaviour [7] [3].

There are two types of polymorphism, the first is compile time polymorphism known as static. In Java, this is known as having multiple methods with the same name but different number of parameters. The second is runtime polymorphism, this is where a subclass has its own unique implementation that is overridden from the same method defined in the superclass. Java decides which method to invoke during runtime and depends on the object type [7] [3].

An example of runtime polymorphism is shown in listing 4 where the subclasses override the method defined in animal superclass.

Animal.java

```
1      // Superclass
2      public class Animal {
3          public void speak() {
4              System.out.println("Animal sound");
5          }
6
7          public void move() {
8              System.out.println("Animal moving");
9          }
10     }
11
12     // subclass
13     class Dog extends Animal {
14         @Override
15         public void speak() {
16             System.out.println("Dog is barking ");
17         }
18     }
19
20     class Cat extends Animal {
21         @Override
22         public void speak() {
23             System.out.println("Cat is meowing");
24         }
25     }
```

Listing 4: Animal class - polymorphism runtime example

In Main.java the objects Dog() and Cat() determine which method of speak() is chosen during runtime listing 5.

Main.java

```
1
2      public class Main {
3          public static void main(String[] args) {
4              Animal animal = new Animal();
5              Animal dog = new Dog();
6              Animal cat = new Cat();
7          }
```

```

8         animal.speak();
9         dog.speak();
10        cat.speak();
11    }
12 }
13 }

```

Listing 5: Main class - showing runtime polymorphism

As shown in listing 6 is the result of runtime polymorphism.

Main.java

```

1
2     Animal sound
3     Dog is barking
4     Cat is meowing

```

Listing 6: Main class - console output

2.5 Describe Abstraction

Mention advantage and disadvantage of abstraction.

Abstraction is a core concept in object oriented programming where a superclass or interface can define abstract methods without including any logic in the method body. A subclass is then required to override the method and write its own logic. This promotes hiding implementation details and exposing only the essential behaviours. Abstract classes and Interfaces are the two ways of implementing data abstraction in Java. Horstmann states that a class that uses more than one abstract method, is required to be stated as abstract. It is used to define common behavior for all subclasses that implement them and can utilise multiple types of interfaces [8]. However abstract classes cannot be instantiated, and therefore use subclasses to instantiate an object. Another limitation of abstract classes, is that it can be extended to only one abstract class [8].

According to Horstmann, interfaces have no implementation details and are purely used to indicate methods within the interface body. All methods are automatically public, without defining the keyword in the code, instead it is written as such: `void shape()` instead of `public abstract void shape()` [8].

To use an interface after the methods are defined, a class must use the keyword **implements** and the interface name to create a contract with the class, which tells the class what methods it must use, but not how to implement them. interfaces can be used in the superclass, subclass. An major advantage from abstract classes, is that in Java, interfaces support multiple interfaces [8].

Joshua Bloch argues that interfaces should be preferred over abstract classes. As abstract classes can only extend to one superclass within the subclass. For example, if an abstract `Animal` class has a `abstract void moves()` method, a subclass `Zebra`, cannot extend to another class i.e `class Zebra extends Animal, Zoo`. That said, using interfaces is the most appropriate method in defining a type and are favoured as they provide multiple inheritance to classes in Java [9].

An example shown in listing 7 shows an interface which defines the methods of what `carVehicle` the superclass must implement. The `carVehicle` class in listing 8 which is the superclass, but can be a subclass too, will implement its own logic and is hidden away from the other classes. This results in the output shown in listing 9.

Car.java

```
1
2 public interface car {
3     void startEngine();
4     void accelerate();
5     void brake();
6 }
```

Listing 7: Interface defining methods

Main.java

```
1
2 public class carVehicle implements car {
3     @Override
4     public void startEngine() {
5         System.out.println("Starting Engine");
6     }
7
8     @Override
9     public void accelerate() {
10        System.out.println("Car is accelerating");
11    }
12
13
14    @Override
15    public void brake() {
16        System.out.println("Car is breaking");
17    }
18
19 }
```

Listing 8: carVehicle implementing the interface

Main.java

```
1
2 Starting Engine
```

```
3 Car is accelerating
4 Car is breaking
```

Listing 9: Main output of implementing the methods for interface

2.6 What are web services?

Web services can be defined as a standalone software application that contains all the necessary details about its operations and how it works. They allow systems to help solve issues, handle operations and accomplish tasks in response to a user or application. The architecture often includes many web service systems which all offer their own speciality of a service but work together by interacting over public and private networks (across the internet and web), collectively behaving like one logical system [10]. By implementing web services from third parties, it can heavily reduce time in creating custom code from scratch to handle different parts of the website.

According to Papazoglou (2008) web services can range from small to medium/large sized tasks, for example withdrawing and depositing capital from a bank account, controlling software using automation to checking if a user has a suitable credit score and this is combined on a more complex level, where information is pulled from different web services into one system i.e. an insurance brokering system [10].

To illustrate the concept of web services, consider an order management system where various web services work together when a customer places an order. Figure 1 shows a suppliers system of web services which operate the purchase order management system. To handle the order, web services are invoked which help complete the transaction. When a customer places an order, web services are initiated concurrently to check the customer has a good credit score. A stocking service will check if the item is available, a billing service will calculate the shipping and generate an invoice to the customer and a shipping service will select and schedule a delivery [10].

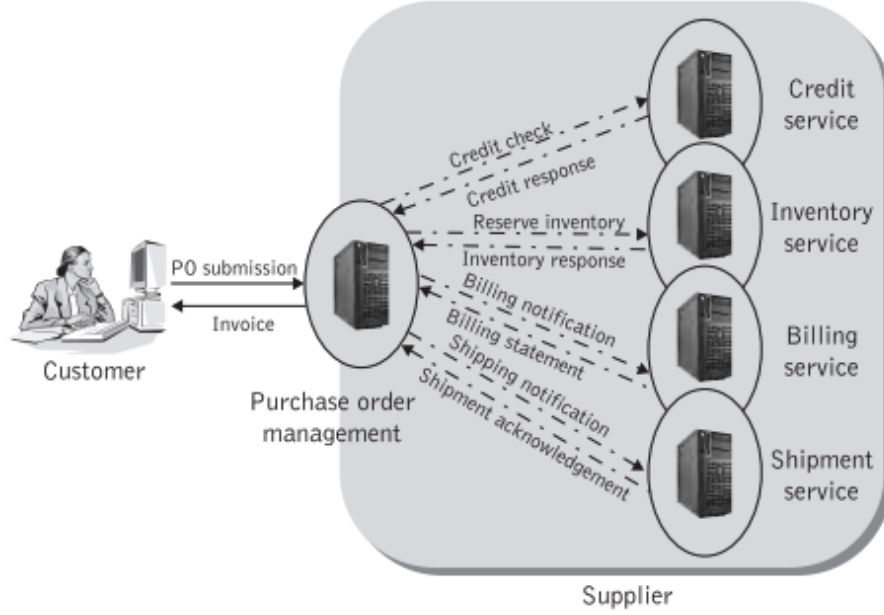


Figure 1: An order management process of how web services interact under the hood [10]

Hypertext transport protocol (HTTP) is a client protocol, which is the core foundation to communicate between a web browser (client) and web service (server) as shown in Figure 2. It is designed for accessing resources including HTML. It works by the web browser sending a GET request to the web service such as a web page made up of i.e. (index.html, style.css and script.js) and the web service sending a response back with the loaded document content, but if invalid, then status code and error is presented.

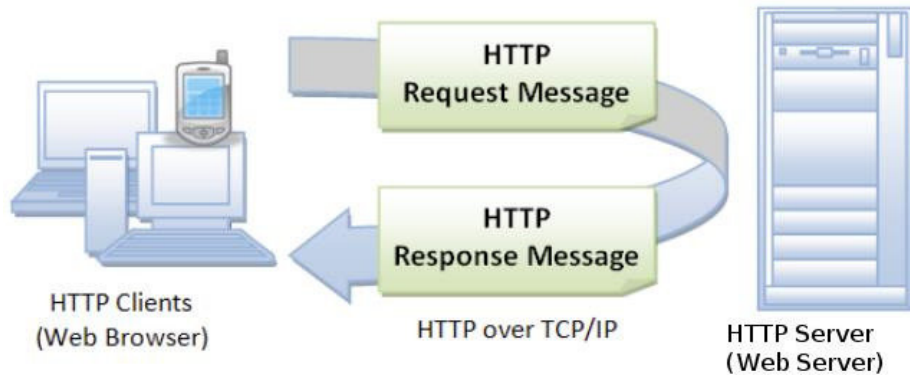


Figure 2: A HTTP request from the client to web server via TCP/IP. [11]

2.7 XML

Extensible Markup Language (XML) is a mark-up language used to transfer data from one location to another i.e. a website to a database and is popular among web services. XML is designed to move, hold and transfer text. It is written to be human readable and easily interpreted by computers. The text used within XML is defined using open and closing tags. XML lets hardware and software to communicate, irrespective of their compatibilities of how data is transported, making it advantageous to use, meanwhile ensuring its flexibility, as detailed data cannot be lost [12].

XML is commonly used for mapping data and one of the popular databases used is relational database (RDB), which is frequently implemented across various organisations, that said, the organisations depend on a well-defined mapping framework to store converted data in RBS using XML [12].

To allow a database to store XML, there are two ways that it can be implemented. Firstly, structured mapping needs a file called DTD which defines and verifies how the XML is set [12] [13].

2.8 JSON

JavaScript Object Notation (JSON) is a lightweight text-based file, that has been developed to provide a human readable syntax and the ability to transfer data between computers. It is made to be straightforward for computers to interpret and generate. It can be used with any language i.e. Java, Python. At its core, JSON is defined by two principal constructs [14] [15].

- A set of key/value pairs that represent an object but can be known differently depending on the language. It uses curly braces and divided by colons [14]

[15]. An example would be “Name”: “Abdulmalik”, “Module”: “Web Services”.
- A structured list in order, commonly known as an array with a group of elements. It uses brackets to form the start and end, separated by commas [14] [15]. An example would be [“A”, “B”, “C”, “D”, “E”]

2.9 Types of web services

SOAP:

Simple Object Access Protocol (SOAP) is a communication protocol utilised to transmit data through the network using XML and HTTP. SOAP operates across various platforms i.e. Windows, MacOS, Linux and programming languages without issues. SOAP is used to let a consumer (client) to send a request to the provider (server), where the provider, will send a response back to the consumer to action the request [16] [17].

There are three types of SOAP nodes as shown in Figure 3.

- Sender: creates and delivers messages [16]. - Receiver: receives and manages the SOAP message and might construct an outcome, alert or error [16]. - Intermediary: Acts as the middleman of the sender and receiver [16].



Figure 3: A diagram of the nodes for SOAP. [16]

SOAP is represented by four layers used from the server to the client:

- **Envelope:** Presents the start and end of a message [17].
- **Header:** Provides data such as authorisation / verification [17].
- **Body:** Contains the request/response data being delivered to application/website [17].
- **Fault:** Delivers error handling data and status i.e. a server/client fault incorrect version etc [17].

2.10 REST

During the year 2000, Roy Fielding had introduced Representational State Transfer (REST) in his dissertation named as Architectural Styles and the Design of Network-based Software Architectures". REST is used for building web services and is a software architectural paradigm that uses HTTP as a protocol to exchange data [17] [18].

REST is defined by six core principles:

- **Stateless:** It does not remember the previous response or request, therefore when a client initiates a new request, it should contain all the required information. This makes the system expand, grow and handle more requests [17].
- **Cache:** Saves repeated client request responses on the client side, to save time and prevent the web server from responding to the same request [17].
- **Uniform Interface:** It uses a standard way of identifying resources to access and update [19].
- **Client Server:** Isolates the client and server, which ensures the system is versatile and ability to grow [17].
- **Layered System:** Uses hierarchical layers, where each layer has its own specific responsibility between the client and server and only communicates with the previous or next layer to request/receive data [17].
- **Code on demand:** The server can send custom code such as JavaScript to run on the client dynamically. Thus improves the system flexibility [17].

RESTful is the preferred style within the software engineering industry especially mobile and web applications due to its factors of being lightweight, promotes accessibility, can be expanded and is compliant with many systems, enabling it to work well with modern technologies such as the Semantic Web [19] [20].

To allow a client and web server to communicate using resource methods, HTTP is used, as shown in figure 4. A **GET** request is requesting data from a website. The **POST** request is to submit data to the server for example a contact form. The **PUT** method is for updating data for example updating an address in a users record. Lastly the **DELETE** is for deleting for example, deleting a user from a database [17].

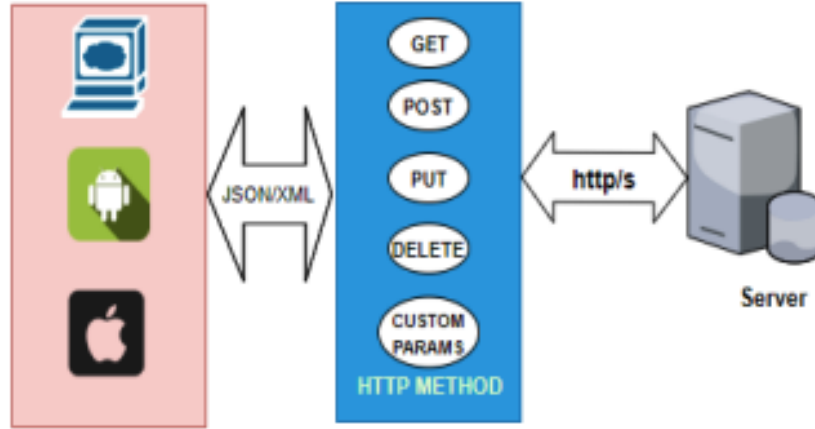


Figure 4: A diagram of the web/mobile app communicating with HTTP methods through the server. [17]

3 Object-relational mapping in Web Services

Object-Relational Mapping (ORM) is a widely recognised method in the software engineering industry [21]. ORM enables developers to retrieve and communicate with the database using classes and objects within Object Oriented (OO) instead of using pure SQL statements [22]. Before ORM was introduced, software developers would create SQL statements manually, which would develop many bugs, become tedious to maintain and be cumbersome. Therefore, ORM was introduced, which maximises developers efficiency in writing code and automatically translates database syntax which thereby supports portability by hiding the internal details of the Database Management System (DBMS) [21]. However, a limitation of ORM is that it can cause extra processing by translating OO code into SQL statements and keeping track of how objects and classes are linked to database tables, which can slow down the application further [23].

Hibernate is a popular framework used within ORM that helps facilitate the development of software applications through tight cohesion and loose coupling [23]. These work well together as it can make code easier to change as components do not rely on other libraries, making it reliable and reusable [24]. The Hibernate framework is widely used among Java developers, considered for its extensive functionality and active user base. Hibernate helps to streamline the process of linking objects within Java and relational database tables, in turn, this offers integration with a range of relational database technologies [22] and eliminates the need for boilerplate code [21].

An advantage of using Hibernate is that it can automatically generate the schema and update the database structure without manual intervention, as well as provides features such as lazy loading and caching [22]. An additional benefit is that developers do not require to write SQL or use JDBC explicitly, CRUD actions are managed within methods, objects and classes. Hiding the internal details of the database, ensures that ORM can work with applications that use different SQL dialects [25]. Nevertheless, a few drawbacks of Hibernate is that it requires significant effort to set up correctly and optimise for performance. Becoming proficient with the framework also demands a considerable time investment. Many developers have described the learning process as particularly strenuous [22].

4 Spring boot and OOP

4.1 Inversion of Control

In Spring, the user is providing access to the application to manage and constructing of objects through the framework Spring boot. By using Inversion of Control (IoC) it allows the software application to do all the work. The advantages are that it enhances the modularity of the software and separates the application from how it is implemented, therefore making it easier to test, by isolating parts of the program [26] [27].

4.2 Dependency Injection

Dependency Injection (DI) is an approach that is used along side IoC where objects can declare dependencies such as arguments through the constructor, factory methods arguments or properties initialised to the object once it has been constructed or returned from the factory method. It is then injected by the container to the dependency, when it is produced the bean. This is known as inverse, essentially where Spring boot, will control the creation and managing of dependencies to the bean. [27] [26].

4.3 How Spring boot supports maintainability, reusability and modularity Dependency Injection

Dependency Injection (DI) enhances maintainability within software applications by decreasing the work required from a software engineer to manually write tightly coupled components. In turn, a developer can use the Spring framework to externally inject a dependency using the XML configuration file or an object within a class. When coupled dependencies begin to decline, then the difficulty of a class begins to diminish. This helps ensure classes are less dependent of each other, therefore DI provides a straightforward way by grouping all the dependencies in one file, known as the injector. By having one file where

all the dependencies are stored, it allows errors and mistakes to be recognised and resolved imminently [28].

When a service is required to be changed, it can be edited within the injector, with no impact to the client. This allows the client to remain untouched, and the code becomes more maintainable and modular [28].

Furthermore DI, supports reusability, in high scale projects i.e. enterprise level where a dependency is used in multiple locations, by allowing a class that uses constructor parameters to be generated without requiring knowledge of the parameter as long as the code behaviour remains consistent across the application structure [28].

Empirical evidence demonstrates that smaller projects see a rise of tightly coupled code because of overhead resulting from DI injection. Open-source projects tend to rise in higher usage due to DI. Therefore, further investigation is required to differentiate the difference of projects that use DI and ones that use pass parameters without using DI [28].

4.4 Springboot three layer architecture

Applications written in Spring, typically use a layered architecture as shown in figure 5 that split into three layers, the controller layer (presentation), service layer (business), repository layer (data access), data source (persistence).

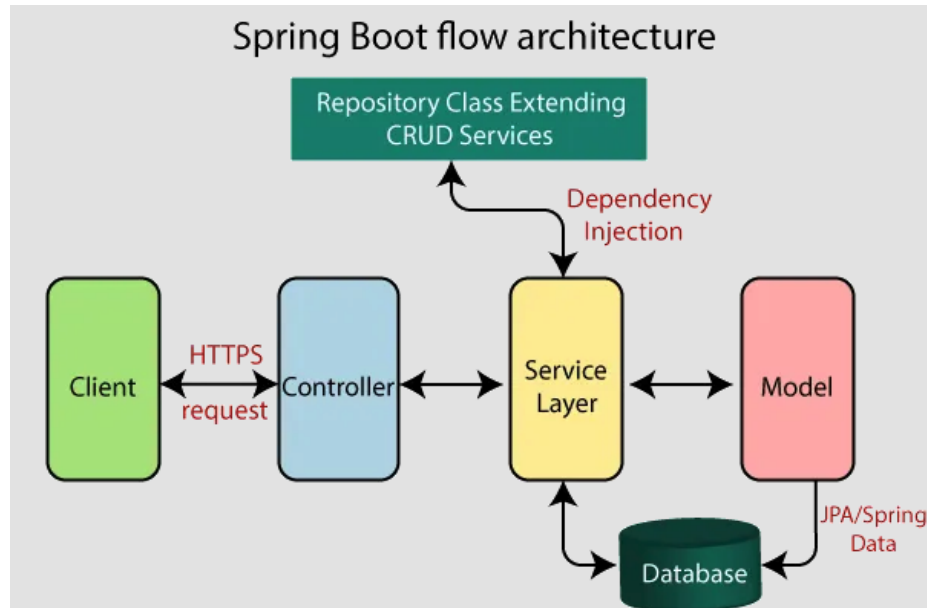


Figure 5: The Spring boot architecture layers [29] [30]

The controller is responsible for processing incoming HTTP, typically RESTful from the client. It handles a request from the client, processes the request by redirecting it to the service layer and once the service layer has completed the request it will format into a JSON format and send a response back to the client [31] [32] [33]

The service layer contains the core business logic of an application and is the layer intermediate to the Controller and Data Access Object (DAO). In Spring boot, annotations are used to specify the layer, using @Service annotation. This helps support a modular design by ensuring business logic is isolated from the web layer, therefore can help minimise the layers becoming tightly coupled. The service layer will then collect and combine data from many sources and combine it into one result, before handing it over to the DAO to handle database transactions [31] [32] [33] [34].

The data access object (DAO) handles database storage, retrieval of data and communications with the database. In Spring boot, the annotation @Repository is a component that is used, which demonstrates the usage of database transactions. The DAO layer can hide internal data and provide a controlled way of engaging with data, provides simple functions that can help with writing queries and setting up the database. Hibernate and JPA within Spring boot are known to be the frameworks used to communicate with the database [32].

The data source typically handles the configuration of the database and defines the settings to be applied. As ORM can handle many types of databases, this can range from relational such as PostgreSQL to non-relational such as MongoDB. Once the request has been complete, it will then respond and return the data back to each service layer that was called and send the response back to the client [35].

The key benefit that the architecture offers is that development time can be enhanced by using Spring boot to develop large and complex applications, by reducing the boilerplate code required to implement a service, using Spring syntax which can maximise performance and reduce error-prone mistakes within the code [31].

The layered architecture aligns with encapsulation by requesting data from the following layer and cannot request data from the controller to persistence layer as well as it cannot call a service from the previous service. In Spring boot, encapsulation it hides the internal data of each layer and isolates the service to have a single responsibility, Incorporating Spring and industry level architecture within applications, it can minimise module dependencies while promoting high cohesion with low coupling [31].

4.5 Repository Pattern

Martin Fowler states that the repository pattern acts as a bridge layer between the business logic and data access layer which is responsible for the management of data storage. It provides a virtual collection of domain entities. The client can then request search queries by sending it to the repository to process matching results. This allows a developer to add and delete objects, just like a standard list of objects. Under the hood, the database carries out automatic execution of the database operations [36].

The repository pattern aligns well with encapsulation, as it hides the internal workings of the data access layer. For example, database management and SQL query creation are hidden behind an interface that clients interact with. Access is controlled through well-defined methods. In Spring, `JpaRepository` provides controlled access to predefined methods such as `getReferenceById()`, `deleteById()`, and `save()` [37]. These methods enhance code readability by allowing Spring to generate the implementation automatically, reducing errors as no boilerplate code is required. Furthermore, the `@Repository` annotation in Spring promotes data hiding by concealing implementation details. This annotation centralises data access management in a single location, helping developers quickly identify and resolve issues when errors occur [38].

4.6 Encapsulation integrating with Springboot

Encapsulation is a core concept in object-oriented programming, of hiding the internal data of a class / method and restricting access directly to the field. It is best to declare fields within a class to private, which can prevent it from being reassigned or changed from outside the class. Therefore, controlled access using additional logic, can be provided using public getter and setter methods to read data safely and update the field [4] [3]

In Spring, the framework Springboot builds on top of Encapsulation, through using Dependency Injection to inject a bean to create an object using the component `@Bean`, where the location of a dependency and instantiating the object is handled by the Inversion of Control (IoC) container [39].

Using `@Beans` in Springboot can help abstract the internal workings and help the developer to focus on creating business logic by using '`@Beans`' to create objects in fewer lines, rather than creating boilerplate code. Developers that make use of these components can improve readability as components are universal in the Spring documentation, maintainable, as the components are injected using components, therefore does not rely on other dependencies and code to work that is written manually and is modular, as each component is designed to handle a single responsibility which help facilitates unit testing code and testing isolated code. This adheres to the SOLID principle well, in particular the Single-Responsibility Principle (SRP). As Robert C. Martin states that each class should include one responsibility, and if a class includes more than one

responsibility, it can therefore lead to coupled code and a weak design which are unanticipated when the code / design is changed. The advantages of this, is that this can create robust applications, that are modular and easier to test, as each class follows SRP [40].

4.7 Abstraction

Abstraction is a fundamental principle in object-oriented programming, where a parent class or interface can assert abstract methods without requiring to implement them in the code. A child class must then override the abstract method and provide its own logic [8]. In Spring boot, there are two ways abstraction is implemented, Spring Java Persistence API (JPA) and Crud Repository.

Spring Data JPA, is a highly robust framework, that is designed to streamline interactions with the database through the abstraction layer. Craig Walls highlights an example in listing 10, in the context of Spring boot of how a splitter repository can be implemented using an interface. The splitter repository can now access the JPA repository and crudRepository that have 18 methods which can update, find, delete Spitter objects without requiring the developer to write the implementation. Spring will then automatically constructs the application during runtime, therefore allowing Spring to do the heavy lifting [41].

```
1 public interface SpitterRepository extends JpaRepository<
   Spitter, Long> {
2 }
```

Listing 10: JPA Example [41]

This allows developers to reduce writing boilerplate code and focus on using interfaces to implement business logic, therefore reducing the lines of code written, enhances readability of code, reducing error prone mistakes and providing a simple way of testing and debugging functions [41].

4.8 Inheritance

Inheritance within Spring boot, can help launch java web applications to a server by using a WAR (Web Application Archive) file, that is configured to run on servlet containers such as WebSphere and Tomcat. This will allow the server to understand how to execute the application [42]. SpringBootServletInitializer is a custom implementation from Spring boot, that extends from the Spring interface WebApplicationInitializer which helps to configure ServletContext rather than the established web.xml method [42] [43].

To utilise SpringBootServletInitializer, the developer is required to extend through a subclass and overriding the method configure() to define the configuration class for Spring [42]. An example of Inheritance is provided in 11

Within the code below, the method ‘configure()’ is used, which allows a developer to override the implementation within a subclass, known as the template method. This empowers the application to handle the booting up of the application and allow the developer override the subclass with a different configuration. This ensures the main code of the application is kept unchanged, while leaving subclasses to be flexible and reusable [42] [44].

```
1 package readinglist;
2 import org.springframework.boot.builder.SpringApplicationBuilder;
3 import org.springframework.boot.context.web.
    SpringBootServletInitializer;
4
5 public class ReadingListServletInitializer extends
    SpringBootServletInitializer {
6     @Override
7     protected SpringApplicationBuilder configure(
        SpringApplicationBuilder builder) {
8         return builder.sources(Application.class);
9     }
10 }
```

Listing 11: SpringBootServletInitializer example [42]

5 Best practices for designing a scalable and efficient web service using ORM and Spring boot

Best practices in Spring boot can range from connection pooling, caching, lazy loading, transaction management and thread safety. This section will introduce and discuss why they matter.

5.1 Connection pooling

Connection pooling in Spring boot is a method where it reuses data connections as opposed to creating a new connection every time a HTTP request is sent from the client. By using reusing connections that are in idle state, connection pooling helps reduce overhead costs of opening a new database connection, which can be expensive. Benefits can include quicker responses times, as setting a new database connection can take much longer than reusing one that isn't being used [45] [46] [47]

A widely adopted technique in delivering performance improvements is using connection pooling. This is a method, that is utilised in reusing database connections that are in idle state, which can be quicker as they are already available. This reduces the need of opening a new database connection, which can be slightly slower, increase overhead costs as database resources are expensive and result in unused connections especially in large enterprises that make requests often [45] [46] [47]. A best practice in mitigating the effects is to prioritise by selecting an ORM tool such as Hibernate in Java, this ensures that connections

are reused and let a developer set custom settings for when the system is in high demand. The advantage of using a tool such as HikariCP in connection pooling is that it is light weight and is the default configuration and is set up when using the spring-boot-starter-jdbc module. A developer can utilise it immediately as no startup configuration is required and completely customisable by setting the pool size, connection management and performance tuning [48].

5.2 Caching

Caching is a standard practice that is implemented in real world applications. It has many benefits from storing frequently accessed data on the client, improving user experience through faster response times, thus reducing database load and costs on sending the same frequently requested data.

A developer should think of when to apply caching, specifically where it benefits the user in accessing data / a page more frequently to reduce database requests. There are multiple methods in Spring boot to use caching. The most prominent being Spring annotations, specifically the use of @Cacheable, it allows for data to be cached, if it is different, then it uses the server to request for new data. While this is a great option, Redis is known to be more robust offering, specifically in highly used web applications, a use case of enterprise applications, is that it offers in-memory caching, offering advanced reading speeds and can restart and backup data. Redis also allows for data to be accessed much faster than locally cached data and can be manipulated [48] [49] [50].

In ORM systems, Hibernate can help maintain data consistency, by removing cached data that are no longer inline with what is stored on the database. This helps minimise the risks of sending outdated information [51].

5.3 Lazy loading and Eager loading

Lazy loading and batch operations are two ways that offer ORM optimisation in building efficient and scalable web services. Lazy loading is a design pattern used in ORM (Hibernate) where beans and dependencies are not created and injected until explicitly required. A developer should use the @Lazy annotation in a configuration or bean class to mark it as Lazy loaded [52].

This is helpful, as it can improve the applications startup runtime, by reducing the loaded data in memory. Lazy loading is useful in development that use massive or sophisticated datasets that are optional. [53] [54].

On the other hand, eagerly, lazy loading can be useful for developers as it fetches all the data at once. Specifically, if all the information will be used, it can save trips to the database requesting for information, ultimately maximising runtime performance [53] [54]. Developers should use the annotation when

writing entity relationships for each field to specifically define which fields can be fetched when required.

6 Challenges and Future Perspectives

6.1 Common challenges in OOP

In Hibernate, proxies are used as placeholders with no data. It is only when a request is sent to the database to fill the placeholder with data requested. The issue this presents is that when the database session closes and the proxy tries to access data from the database, it is faced with the lazy loading error known as being in free state and results in a `LazyInitializationException` error message. Therefore it is crucial that the database session stays open when the proxy tries to request data [55].

6.2 Performance overhead

While ORM can save time and reduce boilerplate code in querying SQL queries, it can also cause performance issues. A notable example is known as the N+1 query problem. An issue where data fetched, that is not optimised for performance can cause hundreds to thousands of requests to the database just to load one page and cause slower performance. An example the author states, is that a figure of 1082 out of 1705 round trips were to queried to the database. Secondly, while ORM provides an easier way of working with querying databases, when utilising objects as opposed to writing raw SQL queries, it can lead to objects requiring to be converted to SQL queries, which can degrade application performance. The author noted that a study taken place in 2010 by Akamai, had stated about 57% of the application users will not stay on the website if it takes longer than three seconds to finish loading [56].

6.3 Data consistency concerns

Data consistency in relation to ORM can introduce many problems. One of them being the “Update All” whereby if one field in a database is changed, it triggers ORM to update every field within the column in the database regardless of which field was changed. This can cause inconsistencies of data and accidentally overwriting changes made by other transactions [57].

A general downside of caching, is that the term dirty read can cause issues where served data is outdated due to the database and caching not being in sync together [48] [49].

6.4 Exploring emerging trends

6.5 Microservices

A Microservice is a design paradigm where services are broken down into smaller services that are independent of each other. A few benefits is that it allows for each service to be loosely coupled, easier to maintain, test, as well as scalable and offers delivering agile development. A suggestion in creating a modular system, would be to separate the data model into related microservices that can capture and focus on one area. By keeping the logic in relevance of the service, it provides an easier way of testing, as it will be independent focusing on one area of the software. That said, the implications on ORM can result in having multiple databases for each microservice, which facilitates in services being difficult to manage due to the relationships of each service. A thoughtful design and plan is crucial in migrating from a monolithic system to microservices [58].

6.6 Recommendations on improving ORM in large scale enterprise applications

6.7 Lazy loading exception error

A recommendation to preventing the lazy loading error, would be to use the filter label class as shown in listing 12 which is a configuration to be used in the Spring boot application. This keeps the session open until the webpage has fully loaded and prevents it from closing early causing the lazy loading exception error [55].

Main.java

```
1 <filter-class>
2 org.springframework.orm.hibernate3.support.OpenSessi
3 onInViewFilter
4 </filter-class>
```

Listing 12: Filter class to prevent lazy loading exception error

References

- [1] Efan, Krismadinata, J. Jama, and R. Mulya, "A systematic literature review of teaching and learning on object-oriented programming course," *International Journal of Information and Education Technology*, vol. 13, no. 2, pp. 302–312, 2023.
- [2] H. Hourani, H. Wasmi, and T. Alrawashdeh, "A code complexity model of object oriented programming (oop)," in *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*. IEEE, 2019, pp. 560–564.

- [3] K. K. Zaw, N. Funabiki, E. E. Mon, and W.-C. Kao, "An informative test code approach for studying three object-oriented programming concepts by code writing problem in java programming learning assistant system," *2018 IEEE 7th Global Conference on Consumer Electronics (GCCE)*, pp. 629–633, oct 2018.
- [4] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," *SIGPLAN Not.*, vol. 21, no. 11, pp. 38 – 45, jun 1986. [Online]. Available: <https://doi.org/10.1145/960112.28702>
- [5] M. Amarif, "Applying the genetic algorithm concept for determining the inheritance relationship between objects in object oriented languages," in *2023 IEEE 3rd International Maghreb Meeting of the Conference on Sciences and Techniques of Automatic Control and Computer Engineering (MI-STA)*. IEEE, May 2023, pp. 276–280.
- [6] K. H. Wai, N. Funabiki, S. T. Aung, X. Lu, Y. Jing, H. H. Sandi Kyaw, and W.-C. Kao, "Code writing problems for basic object-oriented programming study in java programming learning assistant system," in *2023 IEEE 12th Global Conference on Consumer Electronics (GCCE)*. IEEE, Oct. 2023, pp. 5–6.
- [7] M. Hammer, R. Maschotta, and A. Zimmermann, "Extended polymorphism semantics for fuml models," in *2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, aug 2024, pp. 142–151.
- [8] C. S. Horstmann and C. S. Horstmann, *Object-oriented design & patterns*, 2nd ed. Hoboken, N.J.: John Wiley & Sons, 2006, includes index. - Previous ed.: 2004.
- [9] J. Bloch, *Effective Java*, 3rd ed. Boston: Addison-Wesley, 2018, titelzusätze auf dem Umschlag: "Updated for Java 9. Best practices for ... the Java platform".
- [10] M. P. Papazoglou, Ed., *Web services*, 1st ed., ser. Pearson education. Harlow: Pearson/Prentice Hall, 2010.
- [11] S. Suroto, "A review of defense against slow http attack," *JOIV: International Journal on Informatics Visualization*, Nov. 2017.
- [12] E. Song and S.-C. Haw, "Xml-reg: Transforming xml into relational using hybrid-based mapping approach," *IEEE Access*, vol. 8, pp. 177 623–177 639, 2020.
- [13] I. Corporation, "Document type definitions (dtds) - overview," Website. Online Documentation, Mar. 2021. [Online]. Available: <https://www.ibm.com/docs/en/dmrt/9.5.0?topic=dtds-document-type-definitions-overview>

- [14] A. A. Abd El-Aziz and A. Kannan, "Json encryption," in *2014 International Conference on Computer Communication and Informatics*. IEEE, Jan. 2014, pp. 1–6.
- [15] A. A. Frozza, R. d. S. Mello, and F. d. S. d. Costa, "An approach for schema extraction of json and extended json document collections," in *2018 IEEE International Conference on Information Reuse and Integration (IRI)*. IEEE, Jul. 2018, pp. 356–363.
- [16] F. Halili and E. Ramadani, "Web services: A comparison of soap and rest services," *Modern Applied Science*, vol. 12, no. 3, p. 175, Feb. 2018.
- [17] V. R. Anshu Soni, "Api features individualizing of web services: Rest and soap," *International Journal of Innovative Technology and Exploring Engineering*, vol. 8, no. 9S, pp. 664–671, Aug. 2019.
- [18] S.-P. Ma, Y.-J. Chen, Y. Syu, H.-J. Lin, and Y.-Y. Fanjiang, "Test-oriented restful service discovery with semantic interface compatibility," *IEEE Transactions on Services Computing*, vol. 14, no. 5, pp. 1571–1584, Sep. 2021.
- [19] L. A. d. A. Rodriguez and J. M. P. de Oliveira, "A semantic model to describe restful services," *IEEE Access*, vol. 13, pp. 72 402–72 426, 2025.
- [20] R. Ramanathan and T. Korte, "Software service architecture to access weather data using restful web services," in *Fifth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*. IEEE, Jul. 2014, pp. 1–8.
- [21] T. Sotiropoulos, S. Chaliasos, V. Atlidakis, D. Mitropoulos, and D. Spinellis, "Data-oriented differential testing of object-relational mapping systems," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, May 2021, pp. 1535–1547.
- [22] K. Durvas Jayaraman and E. Siddharth, "Harnessing the power of entity framework core for scalable database solutions," *Journal of Quantum Science and Technology (JQST)*, vol. 2, p. 21, 03 2025.
- [23] 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.
- [24] A. Elghedafi, "Detailing the importance to software development of high cohesion, loose coupling, data hiding and information hiding," Jan. 2025, unpublished, University of Wales Trinity Saint David.

- [25] C. Tudose and C. Odubasteanu, "Object-relational mapping using jpa, hibernate and spring data jpa," in *2021 23rd International Conference on Control Systems and Computer Science (CSCS)*. IEEE, May 2021, pp. 424–431.
- [26] R. Manchana, "Building scalable java applications: An in-depth exploration of spring framework and its ecosystem," *International Journal of Science, Engineering and Technology*, 2016. [Online]. Available: https://www.ijset.in/wp-content/uploads/IJSET_V4_issue2_103.pdf
- [27] Spring, "Dependency injection," Online Documentation, 2005–2025. [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html>
- [28] P. Sun, D.-K. Kim, H. Ming, and L. Lu, "Measuring impact of dependency injection on software maintainability," *Computers*, vol. 11, no. 9, 2022. [Online]. Available: <https://www.mdpi.com/2073-431X/11/9/141>
- [29] U. Patil, "Spring boot architecture," Online website, Oct. 2024, medium website Spring boot Architecture. [Online]. Available: <https://medium.com/@udaypatil318/spring-boot-architecture-39935654ce5c>
- [30] M. Mythily, A. Samson Arun Raj, and I. Thanakumar Joseph, "An analysis of the significance of spring boot in the market," in *2022 International Conference on Inventive Computation Technologies (ICICT)*. IEEE, Jul. 2022, pp. 1277–1281.
- [31] Z. Li, L. Wang, W. Piao, and L. Zhang, "Design and implementation of production visual report subsystem based on echarts," in *2022 IEEE Asia-Pacific Conference on Image Processing, Electronics and Computers (IPEC)*. IEEE, Apr. 2022, pp. 696–701.
- [32] S. C. Hubli and D. R. C. Jaiswal, "Efficient backend development with spring boot: A comprehensive overview," *International Journal for Research in Applied Science and Engineering Technology*, vol. 11, no. 11, pp. 759–765, Nov. 2023.
- [33] K. Gos and W. Zabierowski, "The comparison of microservice and monolithic architecture," in *2020 IEEE XVth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*. IEEE, Apr. 2020.
- [34] F. Zhang, G. Sun, B. Zheng, and L. Dong, "Design and implementation of energy management system based on spring boot framework," *Information*, vol. 12, no. 11, 2021. [Online]. Available: <https://www.mdpi.com/2078-2489/12/11/457>
- [35] GeeksforGeeks, "Spring boot - architecture," Online, Mar. 2025. [Online]. Available: <https://www.geeksforgeeks.org/spring-boot-architecture/>

- [36] M. Fowler, *Patterns of enterprise application architecture*, 19th ed., ser. The Addison-Wesley Signature Series. Boston: Addison-Wesley, 2013.
- [37] P. Software, “Interface jparepository<t,id>,” Online Documentation, 2011-2025. [Online]. Available: <https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html>
- [38] Spring, “Working with spring data repositories,” Online Documentation. [Online]. Available: <https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html>
- [39] —, “Introduction to the spring ioc container and beans,” Online Documentation, 2025. [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/introduction.html>
- [40] R. C. Martin, *Agile software development, principles, patterns, and practices*, 1st ed. Harlow: Pearson, 2014.
- [41] C. Walls, *Spring in action*, 4th ed. Shelter Island, NY: Manning, 2015, die Nutzung des codierten Zugangs zu zusätzlichen Online-Ressourcen ist aus lizenzrechtlichen Gründen für Bibliotheksnutzer nicht möglich.
- [42] —, *Spring Boot in Action*. New York: Manning Publications Co. LLC, 2016, description based on publisher supplied metadata and other sources.
- [43] C. Beams, “Interface webapplicationinitializer,” Online Spring framework docs. [Online]. Available: <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/WebApplicationInitializer.html>
- [44] A. Shvets, “Template method,” Online, 2014 - 2025. [Online]. Available: <https://refactoring.guru/design-patterns/template-method>
- [45] L. Ju, A. Yadav, A. Khan, A. P. Sah, and D. Yadav, “Using asynchronous frameworks and database connection pools to enhance web application performance in high-concurrency environments,” in *2024 8th International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*. IEEE, Oct. 2024, pp. 742–747.
- [46] K. Gupta and M. Mathuria, “Improving performance of web application approaches using connection pooling,” in *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)*, vol. 2. IEEE, 2017, pp. 355–358.
- [47] D. V. K. P. Sohel S. Shaikh, “A comparative study of database connection pooling strategy,” *International Research Journal of Engineering and Technology (IRJET)*, vol. 4, May 2017. [Online]. Available: <https://www.irjet.net/archives/V4/i5/IRJET-V4I539.pdf>

- [48] K. S. P. Reddy and S. Upadhyayula, “Beginning spring boot 3: Build dynamic cloud-native java applications and microservices,” *Beginning Spring Boot 3: Build Dynamic Cloud-Native Java Applications and Microservices*, 2023.
- [49] Z. Wang, F. Tang, and Z. L. Yu, “Design and implementation of a health status reporting system based on spring boot,” in *2020 International Conference on Artificial Intelligence and Computer Engineering (ICAICE)*. IEEE, 2020, pp. 453–457.
- [50] Spring, “Caching,” *Caching*, n/a. [Online]. Available: <https://docs.spring.io/spring-boot/reference/io/caching.html>
- [51] B. Sun, M. Dong, W. Yang, and S. Bi, “The solution of server for the smart health monitor system for the pupil,” in *2017 IEEE 7th Annual International Conference on CYBER Technology in Automation, Control, and Intelligent Systems (CYBER)*. IEEE, 2017, pp. 1581–1585.
- [52] Spring, “Lazy-initialized beans,” Online Documentation. [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-lazy-init.html>
- [53] F. Majerik and M. Borkovcova, “Verify the effectiveness of the intelligent data layer on different hardware configurations,” in *2024 IEEE 17th International Scientific Conference on Informatics (Informatics)*. IEEE, 2024, pp. 481–486.
- [54] —, “Verification of idl architecture performance for larger datasets,” in *2024 36th Conference of Open Innovations Association (FRUCT)*. IEEE, 2024, pp. 475–480.
- [55] C. Shan and J. Yuntao, “The discussion on improvement of the j2ee framework based on ssh,” in *2011 International Conference on Future Computer Science and Education*. IEEE, 2011, pp. 518–521.
- [56] A. Cheung, S. Madden, and A. Solar-Lezama, “Sloth: Being lazy is a virtue (when issuing database queries),” *ACM Transactions on Database Systems (ToDS)*, vol. 41, no. 2, pp. 1–42, 2016.
- [57] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, “Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks,” *IEEE Transactions on Software Engineering*, vol. 42, no. 12, pp. 1148–1161, 2016.
- [58] Y. Romani, O. Tibermacine, and C. Tibermacine, “Towards migrating legacy software systems to microservice-based architectures: a data-centric process for microservice identification,” in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2022, pp. 15–19.