
Object Oriented Programming And Web Services

Tudor Haba
2420645

Table of Contents

<i>Introduction</i>	3
<i>Object Oriented Principles</i>	3
Limitations	4
<i>RESTful Architecture</i>	4
<i>Object-relational mapping</i>	5
Advantages and Limitations.....	6
<i>Spring Boot and OOP</i>	7
Spring Boot and ORM.....	9
<i>Challenges and Future Perspectives</i>	10
Challenges	10
Emerging Trends	10
Large Scale Applications	11
<i>Conclusion</i>	12
<i>References</i>	13

Introduction

In this report I evaluate object-oriented programming (OOP) techniques such as inheritance, polymorphism, encapsulation and abstraction. I provide some of the benefits these bring to object-oriented programming and some of the disadvantages that could arise. I then explain what web services are and what REST is and how it links with web services. I explain what object relational mapping (ORM) is and give a detailed overview into some of the advantages it could bring when used to interact with relational databases in an object-oriented web application. I go into detail into some of the drawbacks, such as complex queries, that ORM frameworks still struggle with in the present. I explain how Spring Boot works and how it adheres to the principles outlined earlier in OOP to enhance the maintainability, modularity and reusability of an application. I evaluate the designed patterns found in Spring Boot and how they are similar to the design pattern from object-oriented applications and discuss the best practices in designing a web service with Spring Boot and an ORM framework. Toward the end of the report, I explore some of the current challenges of implementing OOP based web services with ORM frameworks and look at some of the new emerging trends in that area.

Object Oriented Principles

Object Oriented Programming (OOP) is a concept where a program is split into classes where each class has a unique scope. OOP's main principles are inheritance, encapsulation, polymorphism and abstraction [1].

Inheritance in OOP is the concept where a child class can inherit all the methods and attributes of a superclass [1]. All of the public methods and attributes of the superclass can now be accessed by creating an object of the child class. Along with inheriting all the methods from a superclass the child class can also create their own methods and attributes. This will allow a child class to specialize itself. The child class can also override the methods it inherits from the parent class furthering the idea of specialization. Inheritance allows for code reusability [2].

Another main principle of OOP is polymorphism. It is the idea of using the same methods with different implementation for different classes [1]. Polymorphism is achieved through inheritance and overriding the methods of a super class. This allows us to store instances of child classes into a superclass' object, and by accessing the same method the outcome would be different for every subclass object stored within the superclass object thus allowing for flexibility [3].

Encapsulation is another important principle OOP is built on. Encapsulation is hiding specific parts of an application behind private and protected methods and allowing the

access to that data through the use of public methods. This idea hides sensitive data to the users and also hides the implementation of some methods. By hiding the implementation to the user we can increase the maintainability of the code, changing the implementation without affecting the way others implement it [2].

Another principle of OOP is abstraction. Similar to encapsulation abstraction is used to hide data and complexity from the users [2]. In Java abstraction can be achieved by using abstract classes and interfaces to define the methods that other classes that inherit from them must implement. An abstract class in java is meant to act as the blueprint from the child that inherits its methods. On the other hand, an interface only defines what methods the child classes that inherit from it should always implement [2]. Both achieve abstraction by leaving the specific implementation of the methods to the classes that inherit from them.

Limitations

Although inheritance is a useful tool some problems could arise in some languages. Some languages allow multiple inheritance, which means a class could inherit from more than one class. This adds complexity to our application such as name collision, method combination and repeated inheritance. Repeated inheritance, for example, is when a class ends up inheriting from the same superclass more than once causing issues. This makes it difficult to implement and some languages, such as Java, completely prohibit the use of multiple inheritance to avoid the complexity issue that it brings to applications [4].

Object-oriented also comes with the problem of creating many classes. Since OOP encourages the use of abstraction and inheritance to make the code more modular and reusable it also has the problem of writing a lot of classes. Sometimes classes are only used once and for a single purpose [5]. This can cause the codebase to grow very large and makes it difficult to manage. Another drawback that is coming from this is that with a large codebase it makes it difficult to understand the code when first getting into a project.

RESTful Architecture

Web services are a set of protocols that are used together to create web applications [6]. They can be anything from accessing data within a database to more complex operations [7]. Once web services go live, they can be used by other applications or by other web services. The key advantage to web services is that they are self-contained and are loosely coupled, where changing one service does not affect the others [7].

REST is a web paradigm that focuses on the transfer of data over HTTP and how that data is altered by HTTP responses. REST is stateless meaning the client and server do not need to know what state the other is in [6]. Another benefit of a RESTful architecture is that the client and the server can be implemented separately resulting in them being loosely coupled. Changes to either the client or the server do not affect the other. Clients are concerned about sending request to access and modify the data and the server sends response to answer the requests [8].

In REST each operation matches the HTTP CRUD operations, create, read, update and delete [6]. The 5 most important Rest operations are GET, POST, PUT, PATCH and DELETE. The GET operation retrieves a record from the database, the POST operation will create a new record, the DELETE operation will delete an item from the database and PUT and PATCH both update a record [6]. However, there is a difference between the PUT and PATCH, the PUT operation overrides all of the record and requires data to be passed for every field in the record. On the other hand, the PATCH operation only overrides the fields provided and leaves the other fields untouched [8].

Object-relational mapping

Object relational mapping (ORM) systems are frameworks designed to solve the object relational impendence mismatch (ORIM) problem. ORIM refers to the mismatch happening when designing an object oriented application and integrating it with a relational database [9] [10]. The problem with ORIM is that some key concepts from OOP like inheritance and many to many are missing from relational databases. Vice versa, key concepts from relational databases are missing from OOP, such as primary key and foreign keys without which a database could not function [10]. There are many ways to overcome ORIM, one would be writing raw SQL queries to directly access the database in the OOP application. However, with that comes the problem of tightly coupling the application with the database model and having a hard time to switch between different databases if needed in the future [9].

ORM systems are frameworks, such as Hibernate, that are able to easily interact with a relation database from our OOP application. ORM frameworks are a layer that sit between the OOP application and the relational database and converts the data using the OOP's application language [11][12]. Hibernate, lets the developer use the created entities, Java classes, that will be mapped to a table within the relational database. This is where we define the structure of the table in our database, by creating new attributes under the entity we are adding new columns to the table [11].

Advantages and Limitations

An advantage of ORM frameworks is that it abstracts the raw SQL and it allows the developers to use the OOP language to access the data within the database. This reduces the time it would take a developer to write SQL code and it makes for a clearer and more object oriented focused approach to implementing the queries [12]. This also reduces the amount of code a developer writes as complex SQL queries can get long and hard to follow.

Using a ORM makes the application much more flexible as a result of not having to write pure SQL. This is due to the fact that it does not matter what relational database you are using, using a framework will make it easy to switch between databases [9].

Along with not having to write SQL there is also the benefit of the improved security a ORM provides. ORM frameworks take in account some security flaws that could be introduced while writing pure SQL and protect against it [13]. One of the security flaws it protects against is injection attacks, where a person attempts to gain access or cause damage to a system by using user input to enter malicious set of strings [14].

Another benefit of using ORMs is that it hides away the business logic and while still giving you access to the data stored in the database [12].

One of the limitation coming from object relational mapping frameworks is the addition of anti-patterns by the ORM generated queries, inheritance being the biggest issues [9]. Since inheritance allows the child classes to inherit everything from a super class then the issue comes when the super class is given additional permission levels. Every child of the super parent would inherit the same permission level which might allow the introduction of unwanted access to the database thus reducing the cohesion of the application [9]. A lower cohesion in OOP would mean that the class does more than it needs to do. This means that it will reduce the reusability of the class and in OOP the focus is to make the classes are reusable as possible [15].

Another problem introduced by ORMs is the eager fetching problem. This problem, also called the excessive data problem is one that's caused by the fetching of additional columnar data that is not needed by the query [9]. After the application fetches the extra data, it then discards it. ORMs can sometimes hide this behaviour from the user by using pre-fetching techniques, fetching the data before it is needed. This also introduces its own set of problems. Pre-fetching data has the drawback of increasing the resources a system will use thus slowing the execution time of the application and causing more traffic between the database and the application [9].

Handling difficult SQL queries proves to be a challenged object relational mapping frameworks are facing [13]. It was found that some ORM frameworks do not support some SQL queries such as cross joins. Instead, they would try and replicate a cross join by doing multiple simpler operations. However, this increases the time it would take for a

query to execute significantly. It took a ORM generated query up to 300x times longer to perform a cross-join operation than a non-ORM generated query. Fig. 1 shows that as we increase the complexity of a query the longer the execution time of that query gets when generated by an ORM [9].

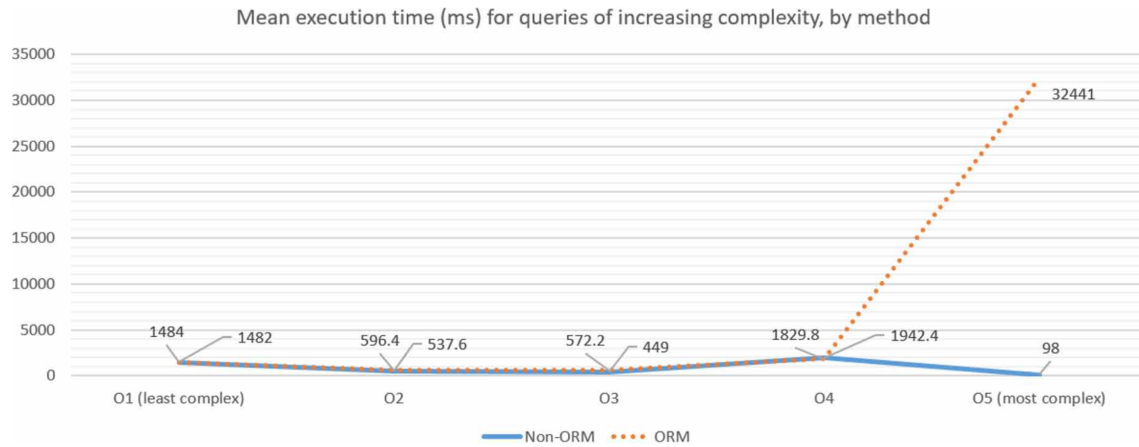


Fig. 1 Execution time of queries ORM generated and non-ORM [9]

Spring Boot and OOP

Spring Boot is built on top of the Java Spring framework. Spring boot is built with the intention to simplify the creation of independent, production level applications that are easily ran. It reduces the set-up time the developers need, and it reduces boiler plate code that would otherwise be repeated. Spring boot simplifies the set up process otherwise required with a spring application through the use of auto configuration and managing the available dependencies [16] [17].

Spring boot encourages developers to integrate object-oriented programming concepts such as encapsulation by separating the application into different layers. Each layer will come with the benefit of hiding and restricting access to the data and with this comes the enhancement of maintainability. The layers spring boot encourages developers to use are broken down into controllers, services and repositories. This makes sure that each layer is responsible for a single action and promotes the loose coupling seen from OOP.[17] The controller layers is responsible for receiving HTTP requests, mapping them to a certain endpoint in your application. After mapping the request to one of the endpoint it then processes the request and then returns a response after calling the corresponding server logic [16]. The controller is responsible for logic that might include verification, requesting data from the database. The service layer is responsible for holding all the business logic of the application, it has access to the repositories and other external services and all the requests passed from the controller are going through the service layer. Finally, the repositories are what handle the data, they are where the CRUD

operation that have direct access to the database are stored [16]. From this we can see that Spring boot promotes encapsulation by breaking down the layers and making changes to one layer will not affect the other thus improving the maintainability of the application as a whole.

The clear separation of the controller, services and repositories layers that spring boot implements can show how spring boot promotes modularity within the applications it is a part of. Modularity can be seen by the fact that these layers are all separated by individual responsibilities and each layers interacts with another layer, Fig 2 [16].

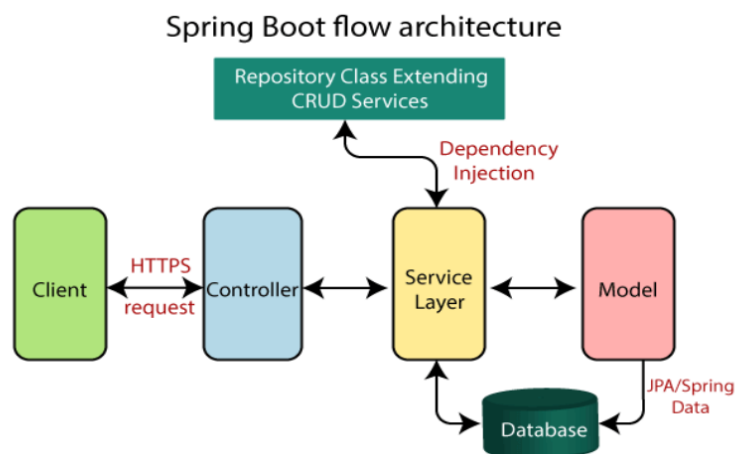


Fig 2. Spring Boot flow architecture [16]

A dependency injection (DI) in Java, or any other OOP language, is a design pattern where a class receives their dependencies from an external source rather than creating them from within itself [18]. A dependency means that in order for the class to work as expected it depends on some other bit of code coming from an external source. Therefore, a DI would insert that dependency directly into the class through the use of methods or constructors. In Spring boot a DI is a core concept of the framework. The framework uses annotations to inject dependencies into object which require them. Developers should only ask a class to insert the dependencies it needs, this is it took promote the loose coupling principle and to keep the code easy to maintain and testable. DI in Spring boot promotes the use of encapsulation and modularity by following the Law of Demeter. This is where each class should only talk to classes it relates to, and each class should have limited knowledge of the other classes it interacts with [18].

One of the design patterns used in Spring Boot is the repository pattern. The repository pattern allows us to define read and write operations in a structured way [19]. This pattern uses OOP interfaces to define what methods are available to access the data within a database, but it leaves the implementation of the methods undefined. The service layer that sits between the repository and the controller is responsible for implementing the methods the interface defines. The developer then decides in the service layer how each method is implemented and implements the business logic behind them. This way the

developer hides the business logic from the controller promoting abstraction and keeping the code more maintainable overall. The controller does not have access to the business logic and only calls available operations implemented by the service layer [19]. In Spring Boot this pattern allows developers to abstract the operations accessing the database. However, the framework allows for some automatic implementation of basic operations such as create, read, update and delete by the use of special tags [20].

A design pattern that Spring Boot makes use of is the singleton pattern. The singleton pattern is designed to ensure there is only a single active instance of the class created for the duration of the application. It also ensure that there is global access to that instance. This pattern encapsulates the instantiation process and promotes flexibility of the application along with scalability. This pattern is used in Spring Boot by how the framework manages beans [21]. A bean in the Spring framework is a Java object that is created and managed by the framework. When a developer creates a bean, by default the bean will be a singleton. This essentially means that every time the developer uses the bean, the same instance will be used thus aligning with how the singleton pattern is described in an OOP application [22].

Spring Boot and ORM

It is easy to build scalable and efficient web services using Spring Boot and ORMs because of the features they provide and because Spring Boot can handle large amounts of data, such as big data [23]. Spring Boot, for example, reduces the cluttering in a project, the auto configuration it provides help by not having to configure as many files. Also, the easy managing of our dependencies we discussed previously plays a big role into making Spring Boot more scalable. With a single file that manages all the dependencies we can transfer and run the application on any machine. The Spring security add a protective layer that is more effective than the ones previously used in Node JS. A more secure framework helps when we are using a large amount of data as we want to protect the data as much as possible. With the application getting larger, more possible bugs that might make the data more vulnerable could be introduced, so the extra security Spring provides is beneficial [23]. ORM frameworks will make a web service easy to scale as they make it easier for the developer to map the object in an application to a relational database. As an application becomes more complex and grows in size, using an ORM framework to manage the relationships between objects, and write SQL queries for developers, makes it more efficient [9].

Challenges and Future Perspectives

Challenges

Implementing OOP based web services with some ORM frameworks such as Hibernate could cause some performance overheads. This is when the performance of the application is impacted negatively because of how the ORM framework functions. The annotations developers can use such as @Transactional can also introduce these performance overheads if not used correctly, for example when used when there is no need for it. Using nested transaction within our services, where a transaction can call other methods from within itself, can lead to problems such as timeout and deadlock while trying to access the same resource. These overheads affect the overall performance of the whole application and can slow it down [24].

One of the big challenges object relational mapping introduces to OOP based web services is the problem of lazy loading. Lazy loading is a technique that has been introduced to try and solve the problem of slow loading times and improving performance caused by the initial loading of all the data on a web page. Lazy loading is the idea of not loading unnecessary data until the data is requested by the service. By using lazy loading annotation in our web service, we are telling it to only fetch the data when the service is being called. The problem this then introduces is that lazy loading sends individual queries to the database [9]. When batch data is needed and the developer chooses to use lazy loading it causes the $N + 1$ query problem. This problem is when we are querying the database for a list of items and for each of these items we are querying the database again for more detail. This sends too many queries to the database and depending on the size of the data requested we could end up affecting the performance of the application negatively [24].

Emerging Trends

Microservices are an emerging trend in recent years because of the many benefits it brings and the use of cloud. Microservices are an architectural pattern which break the application up into small independently deployable services, each designed to implement a specific business need. Each service is fully autonomous, meaning they are independent from each other [25] [26]. Communication between the services happens between interfaces therefore ensuring that changing one service will have no impact on the other services following object-oriented patterns. The implementation details are hidden between the services, and this helps to achieve loose coupling and high cohesion in the application. Microservices achieve functional decomposition, breaking down the service into smaller tasks, which is key for achieving a microservice architecture. This allows for high flexibility of systems and it makes scalability easier [25].

Along with the trend of microservices, a new trend emerged called cloud-native development, that helped with breaking down the application into microservices and packaging those microservices together into lightweight container for deployment. Once deployed the microservices can be easily orchestrated across multiple servers on the cloud. Cloud native development focuses on the following characteristics, containerization and orchestration [26]. Containerization is the idea of isolating all the applications code, the libraries it uses and any other dependencies that it needs to run into containers. The benefit of this approach is that each of the container can then simply run on any type of system. This promotes isolation of resources and reproducibility. One of the most popular tools for implementing containerization is Docker. The other characteristic making up cloud-native development, orchestration, refers to the automated management, coordination and arrangement of microservices to build and maintain scalable applications. Orchestration reduces the effort of managing the containers' lifecycle, scaling up and down and deployment of microservices on the cloud. A cloud native application is a horizontally scalable application, in which it is easy to add new servers to improve the performance of the application, made up of microservices [26].

According to Google's statistics another emerging trend in web service architecture is GraphQL with an average 14% increase yearly in the search for the term as of 2023 [27]. GraphQL is a query language and server-side runtime for executing the queries. GraphQL lets you define the exact data you want to return from your server as opposed to REST which will bring everything, having to then disregard the additional data it brought. In the past REST was considered standard for implementing APIs however GraphQL solves some of the drawbacks of REST. Compared to REST which makes multiple endpoints calls to retrieve data for more complex queries, GraphQL only does a single call making it much easier to work with and implement [28]. GraphQL is very useful for modern web applications and mobile applications for this reason of not making unnecessary calls and not bringing extra data with every query. It was invented by Facebook in 2012 and it got more popular since [28].

Large Scale Applications

As we discussed in the challenges faced when implementing web services with ORM frameworks, large applications suffer from performance overheads and lazy loading issues. A way to overcome these issues in a large application would be to integrate GraphQL to take care of some of the more complex queries, where a framework such as Hibernate would struggle, to overcome the performance overheads [24]. This way we would essentially use both an ORM framework and GraphQL to improve our queries.

Conclusion

In conclusion in this report, I've covered topics starting from object-oriented principles and what benefits OOP brings to programming, such as making code more reusable and less dependent on each other. I've also covered why OOP might introduce some complexities to our code base, having a larger code base, essentially making it harder to navigate through. I have looked at the REST architecture and what that means for web services. I've described what ORM frameworks are and how they make it easier for developer to write SQL queries by writing boiler plate code for them and making the application easier to integrate between different relational databases by making the code loosely coupled and how it links to OOP principles. I went into detail about how Spring Boot helps manage our dependencies and how it promotes core OOP principles like reusability, maintainability and modularity. I evaluate some of the designed patterns that Spring Boot implements and discussed how we could use Spring Boot and ORM together to build scalable and efficient web services. Finally, I wrote about challenges faced by ORMs such as performance issues it could cause because of complex queries. I looked at some of the current trends like microservices, cloud development and GraphQL and how some of these compare to REST. Key differences being that REST architecture makes use of multiple endpoints and GraphQL only making use of a single endpoint to bring only essential data back.

References

- [1] 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)*, Apr. 2019, pp. 560–564. doi: 10.1109/JEEIT.2019.8717448.
- [2] I. Lavy, Rashkovits, Rami, and R. and Kouris, 'Coping with abstraction in object orientation with a special focus on interface classes', *Comput. Sci. Educ.*, vol. 19, no. 3, pp. 155–177, Sep. 2009, doi: 10.1080/08993400903255218.
- [3] G. Smith and S. Helke, 'Refactoring Object-Oriented Specifications with Inheritance-Based Polymorphism', in *2011 Fifth International Conference on Theoretical Aspects of Software Engineering*, Aug. 2011, pp. 35–41. doi: 10.1109/TASE.2011.31.
- [4] G. B. Singh, 'Single versus multiple inheritance in object oriented programming', *ACM SIGPLAN OOPS Messenger*, vol. 6, no. 1, pp. 30–39, Jan. 1995, doi: 10.1145/209866.209871.
- [5] N. Ragonis and M. and Ben-Ari, 'A long-term investigation of the comprehension of OOP concepts by novices', *Comput. Sci. Educ.*, vol. 15, no. 3, pp. 203–221, Sep. 2005, doi: 10.1080/08993400500224310.
- [6] 'Definition of REST web services with JSON schema'. Accessed: Apr. 08, 2025. [Online]. Available: https://onlinelibrary.wiley.com/doi/epdf/10.1002/spe.2466?saml_referrer
- [7] H. Wang, J. Z. Huang, Y. Qu, and J. Xie, 'Web services: problems and future directions', *J. Web Semant.*, vol. 1, no. 3, pp. 309–320, Apr. 2004, doi: 10.1016/j.websem.2004.02.001.
- [8] A. Golmohammadi, M. Zhang, and A. Arcuri, 'Testing RESTful APIs: A Survey', *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 1, pp. 1–41, Jan. 2024, doi: 10.1145/3617175.
- [9] D. Colley, C. Stanier, and M. Asaduzzaman, 'Investigating the Effects of Object-Relational Impedance Mismatch on the Efficiency of Object-Relational Mapping Frameworks', *J. Database Manag.*, vol. 31, no. 4, pp. 1–23, Oct. 2020, doi: 10.4018/JDM.2020100101.
- [10] 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', *Inf. Softw. Technol.*, vol. 82, pp. 1–18, Feb. 2017, doi: 10.1016/j.infsof.2016.09.009.
- [11] E. J. O'Neil, 'Object/relational mapping 2008: hibernate and the entity data model (edm)', in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, Vancouver Canada: ACM, Jun. 2008, pp. 1351–1356. doi: 10.1145/1376616.1376773.
- [12] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, 'Detecting performance anti-patterns for applications developed using object-relational mapping', in *Proceedings of the 36th International Conference on Software Engineering*, Hyderabad India: ACM, May 2014, pp. 1001–1012. doi: 10.1145/2568225.2568259.
- [13] E. Sierra and U. L. Yuhana, 'Optimizing Software Development through Data Access Speed using Object-Relational Mapping (ORM) on Credit Risk Application', in *2023 3rd International Conference on Smart Cities, Automation & Intelligent*

Computing Systems (ICON-SONICS), Dec. 2023, pp. 201–206. doi: 10.1109/ICON-SONICS59898.2023.10435255.

- [14] D. Ray and J. Ligatti, 'Defining code-injection attacks', *ACM SIGPLAN Not.*, vol. 47, no. 1, pp. 179–190, Jan. 2012, doi: 10.1145/2103621.2103678.
- [15] V. Saxena and S. Kumar, 'Impact of Coupling and Cohesion in Object-Oriented Technology', *J. Softw. Eng. Appl.*, vol. 05, no. 09, pp. 671–676, 2012, doi: 10.4236/jsea.2012.59079.
- [16] object Object, 'A Review of Spring Boot Framework Based on REST API', Accessed: Apr. 11, 2025. [Online]. Available: <https://core.ac.uk/reader/586551330>
- [17] 'Spring Boot', Spring Boot. Accessed: Apr. 11, 2025. [Online]. Available: <https://spring.io/projects/spring-boot>
- [18] C. Muldoon, 'Engineering testable and maintainable software with Spring Boot and React'.
- [19] H. Ouyang and T. Palaoag, 'A Java EE Data Persistence Layer Model Based on Multi Persistence Technology', in *2022 International Conference on Management Engineering, Software Engineering and Service Sciences (ICMSS)*, Jan. 2022, pp. 9–14. doi: 10.1109/ICMSS55574.2022.00009.
- [20] 'DAO vs Repository Patterns | Baeldung'. Accessed: Apr. 12, 2025. [Online]. Available: <https://www.baeldung.com/java-dao-vs-repository>
- [21] J. Asaad and E. Avksentieva, 'A Review of Approaches to Detecting Software Design Patterns', in *2024 35th Conference of Open Innovations Association (FRUCT)*, Apr. 2024, pp. 142–148. doi: 10.23919/FRUCT61870.2024.10516345.
- [22] 'Bean Overview :: Spring Framework'. Accessed: Apr. 13, 2025. [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/definition.html>
- [23] 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)*, Jul. 2022, pp. 1277–1281. doi: 10.1109/ICICT54344.2022.9850910.
- [24] B. Alshemaimri, R. Elmasri, T. Alsahfi, and M. Almotairi, 'A survey of problematic database code fragments in software systems', *Eng. Rep.*, vol. 3, no. 10, p. e12441, 2021, doi: 10.1002/eng2.12441.
- [25] C. Pahl and P. Jamshidi, 'Microservices: A Systematic Mapping Study', in *Proceedings of the 6th International Conference on Cloud Computing and Services Science*, Rome, Italy: SCITEPRESS - Science and Technology Publications, 2016, pp. 137–146. doi: 10.5220/0005785501370146.
- [26] S. Deng et al., 'Cloud-Native Computing: A Survey From the Perspective of Services', *Proc. IEEE*, vol. 112, no. 1, pp. 12–46, Jan. 2024, doi: 10.1109/JPROC.2024.3353855.
- [27] A. Quiña-Mera, P. Fernandez, J. M. García, and A. Ruiz-Cortés, 'GraphQL: A Systematic Mapping Study', *ACM Comput. Surv.*, vol. 55, no. 10, pp. 1–35, Oct. 2023, doi: 10.1145/3561818.
- [28] A. Lawi, B. L. E. Panggabean, and T. Yoshida, 'Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System', *Computers*, vol. 10, no. 11, Art. no. 11, Nov. 2021, doi: 10.3390/computers10110138.