

lestyn Green Web Services OOP essay

by lestyn Green

Submission date: 20-May-2025 06:42PM (UTC+0100)

Submission ID: 259954437

File name: 58847_lestyn_Green_lestyn_Green_Web_Services_OOP_essay_664183_1157549250.docx (53.79K)

Word count: 5691

Character count: 34164

Web Services, Object Oriented Programming Report
Iestyn Green
2420522

Table of Contents

Introduction 3

Conceptual Foundations 4

Object-Relational Mapping (ORM) in Web Services 6

Spring Boot and OOP 8

Challenges and Future Perspectives..... 11

Conclusion..... 14

References..... 15

Introduction

Object-oriented programming (OOP) is one of the most fundamental programming models used today. OOP structures software to be designed around objects containing data and methods. The key principles of OOP are Encapsulation, inheritance, Polymorphism and Abstraction, each of these will be studied in more depth within this report. By properly utilising OOP, software can be made more modular, reusable, and easier to maintain. This report will delve into both OOP and introduce Web Services as a concept and how they can be paired and bridged using Object-Relational Mapping (ORM).

Web services facilitate communication and data exchange across different applications or systems over a network. Acting as a standardised way of communication, allowing applications that are written in different languages to run on different platforms and interact seamlessly. Today, web services are important and used for various applications, such as cloud computing, APIs, and connecting different systems.

ORM is a programming technique that allows developers to interact with a database using object-oriented languages rather than raw SQL queries. It creates an abstraction layer between OOP code and a relational database, allowing developers to work with data as objects. This is especially useful in web services where backend logic uses OOP and needs to interact with a database.

The primary aim of this report is to examine the relationship between OOP and Web services, while shedding light on the role of ORM in modern web-based software development. Touching on how specific OOP principles influence web design and how ORMs, such as Hibernate, help with handling data within systems. Lastly, this report aims to highlight how to integrate these techniques in the spring boot framework and delve into the current challenges and emerging trends.

The objectives for this report are to:

1. Define and explain object-oriented programming, focusing on how it impacts software design.
2. Explore Web Services, providing an overview of what their role is in modern software engineering
3. Explain and evaluate the impact on application development that Object-Relational Mapping has
4. Analyse the integration of ORM tools within the context of OOP and Web Services
5. Assess how Spring Boot supports OOP principles by frameworks, patterns and best practices
6. Identify the main challenges faced when using ORM in large-scale web applications
7. Identify future trends and suggest improvements for ORM

Conceptual Foundations

Object Oriented Programming

There are a few fundamental principles that are essential to object-oriented programming's design: Encapsulation, Inheritance, Polymorphism, and Abstraction. In this report segment, each of these principles will be defined and their benefits will be highlighted.

Firstly, Encapsulation will be covered. Encapsulation is a principle in OOP that bundles data with the procedures that are used to operate on it, restricting external access to this data. Access to an object's internal data is tightly controlled, ensuring that this data cannot be accessed directly outside the object, reinforcing the principle of information hiding. This controlled access method partitions memory into separate blocks, each accessible only by the object's procedures. This isolation helps to stop unintended interfaces between different objects and supports modularity in the process, making programs easier to read, resulting in less errors in development and maintenance. [1]

Inheritance is another principle of OOP. Inheritance allows for new object definitions to be built based upon existing ones. This means that when a new object class is being defined, only the properties that differ from the parent class must be defined explicitly; the other properties get automatically inherited. Inheritance works to make slight changes or improvements to existing code, without needing to retype all of the existing code, reducing code duplication. It is very effective since it allows developers to modify and extend existing code without altering the original. [2]

Polymorphism is a fundamental feature of OOP, used to enhance how easy code is to read and how it can be reused in different scenarios. It allows a single interface or method name to be used with different underlying forms, enabling methods and objects to behave in different ways, adapting depending on the context in which they are used. Polymorphism supports using multiple forms for functions, variables, and objects by overloading, overriding, and virtual functions. By using polymorphism, developers can write adaptable, scalable and easy-to-maintain code. It supports efficient code organisation, reuse and system design by reducing repetition and improving structure. [3]

Abstraction is the final OOP paradigm to be highlighted. It defines classes and objects in terms of their interfaces and expected behaviours rather than how those behaviours are executed internally, like Encapsulation. This design principle supports a top-down development process, starting with general definitions of shared attributes and behaviours and refining them into more specific, implementable classes. Abstraction helps developers to organise code by grouping components based on what they are and what they do. By structuring systems this way, the system offers a cleaner picture of how different system parts relate. As a result, Abstraction encourages reusability, since a general interface can be applied in different contexts without rewriting logic. It also promotes flexibility, as abstract super classes allow specific implementations to be modified or replaced with minimal impact on the whole system. Abstraction also enhances maintainability by isolating

changes to a specific subclass and leveraging polymorphism, allowing the system to handle different object types dynamically through a standard interface. [4] [5]

Web Services and RESTful

The next fundamental topics to be covered in this report are web services and RESTful architectures. Web services consist of a software module that enables interoperability between different systems that may use different platforms, languages, or technologies. Two primary approaches for building web services are SOAP (Simple Object Access Protocol) and REST (Representational State Transfer). SOAP uses XML for data transmission and relies on multiple protocols like HTTP and SMTP. SOAP also adds headers and body to messages, resulting in more data but leading to potential performance issues. SOAP is best suited for enterprise-level systems needing high security, as standards like WSDL, WS-Security, and WS-Addressing support it. It is also chosen due to its reliability, asynchronous communication and integration into systems used for banking or payment systems. REST, on the other hand, is a newer, more lightweight alternative to SOAP. REST uses HTTP and standard methods like GET, POST, PUT, and DELETE. These methods are used to perform operations in a way that aligns with the CRUD model (Create, Read, Update, Delete). So, GET is used to retrieve or read a resource from a web server, POST creates a new resource on a server, PUT is used to update or change the state of a resource, and DELETE is used to remove or delete a resource [6]. It also supports data formats like XML and JSON. It has been shown to be simpler to implement and faster in execution, making it ideal for scalable, performance-sensitive, and mobile-accessible applications. When comparing them both, REST tends to be a lighter alternative to SOAP. REST benefits from its implementation complexity, execution speed, used memory resources and overall performance being better than SOAP. Also, REST is better for simple integrations, especially if the project requires a simple point-to-point integration or large-scale availability from mobile devices, making REST the better choice in that context. So, to summarise, REST provides a flexible, efficient and scalable angle to building web services. While SOAP provides a good option for high-security, high-complexity environments. REST is often the preferred choice for most modern applications due to its simplicity, performance, and compatibility with the web. [7]

Object-Relational Mapping (ORM) in Web Services

Object-Relational Mapping (ORM) is a design approach which connects OOP models with relational database systems, enabling developers to abstract and manage data persistence by keeping it separate from business logic within their application. Since their adoption in the early 2000s, ORMs have significantly accelerated the software development process for many programmers. By using ORM frameworks, developers are relieved from manually writing database code, streamlining the development process, minimising repetitive coding, particularly for CRUD operations, allowing developers to focus more on application-specific logic. However, using ORM also brings disadvantages, such as having a huge learning curve, performance issues, and object-relational impedance mismatch. To expand on the learning curve issue, ORMs have shown to have complex configurations, despite abstractions. ORMs like Hibernate or Entity Framework (EF) require a deep understanding of annotations or configurations and an understanding of proper tuning to avoid greater issues [8]. The performance issues stem from ORMs introducing substantial performance overhead in database operations, especially if not used optimally. Even after performance tuning, a gap often remains when compared to hand-crafted SQL. [9] Lastly, multiple studies have found a fundamental mismatch between OOP and relational databases. Objects possess identities, states, and behaviours, while relational models are stateless and data-centric. Creating challenges in mapping class hierarchies, object identity, and relationships and managing semantics related to inheritance and Encapsulation. [10] [11]

While ORM tools suggest a simplified database interaction, the process is somewhat complex and partially manual, acting as a dynamic link between objects and data structures without being fully automated and often requiring thoughtful configuration. Developers must make informed decisions when mapping object structures to relational tables, balancing database design concerns such as normalisation and key relationships with OOP principles like Encapsulation and low coupling. ORM can help tackle the fundamental differences between OOP and relational databases, which operate on differing principles of data manipulation. ORM frameworks often implement well-established design patterns that guide how data is stored, retrieved, and maintained, creating a separation of concern, improving maintainability and making more precise alignment between analysis models and implementation. Using ORM, developers can avoid writing SQL directly, instead interacting with a layer that automatically translates requests into database queries and returns results as objects, allowing developers to focus on how the application processes data rather than how that data is stored or retrieved. Successful use of ORM requires understanding its limitations and the trade-offs involved, especially when adapting abstract domain models to the practical constraints of relational databases. [12] [13]

The number of frameworks and SQL dialects available can make selecting the right one for a new application challenging. The decision typically involves evaluating factors such as code complexity, required dependencies and performance. Two popular frameworks are Hibernate and Java Persistence API (JPA). Both these

frameworks offer comparable performance, while Spring Data JPA introduces additional overhead; it also simplifies development by reducing boilerplate code and streamlining data access and modifications by providing a higher level of Abstraction over JPA. This Abstraction can benefit developers, particularly when rapid development is needed. JPA itself is a specification that standardises ORM in Java. It allows developers to use annotations to define how Java objects are mapped to database tables, specifying primary keys, table relationships, and field constraints. Managing CRUD operations uses the EntityManager interface, which handles persistence logic without requiring custom SQL. This standardised approach helps decouple the persistence layer from the business logic, making cleaner, more maintainable code. [14] [15]

Hibernate plays a significant role in abstracting database interactions. It handles the generation of SQL automatically and offers a clean abstraction layer, reducing the need for manual database code. The framework supports ACID-compliant transactions. ACID stands for Atomicity, Consistency, Isolation, and Durability, four critical properties that ensure data operations remain consistent and reliable, even during situations with high-concurrency or unexpected failures. Alongside automatic SQL generation and ACID compliance, Hibernate also provides robust transaction management, making it an ideal solution for applications that require high availability and scalability. Through its ORM features, Hibernate also aids in simplifying complex data relationships and mapping Java objects to relational database tables. These capabilities allow developers to manage complex data models without diving into SQL intricacies. Hibernate's integration with existing Java frameworks ensures seamless and efficient interaction with enterprise-level applications. [16]

Spring Boot and OOP

This report segment will delve into what Spring Boot is, how Spring Boot uses OOP principles to encourage modular code, the different design patterns in Spring Boot, and the best practices for scalable and efficient design. Spring is a comprehensive Java framework that combines various tools and libraries for web development. It provides many features such as database integration, security, cloud deployment, and support for web services. The Spring framework was created in 2003 by Rod Johnson to simplify enterprise Java development. It emerged as a response to the complexity of J2EE, offering a more lightweight alternative through various concepts like Plain Old Java Objects (POJOs), Inversion of Control (IoC), dependency injection and standardised annotations. Spring Boot, released in 2014, takes these principles further by reducing boilerplate code and handling configuration automatically, making it easier to create standalone and enterprise-scale applications. Spring Boot promotes a clean, modular design using OOP principles such as Encapsulation, Abstraction and separation of concerns. It uses IoC to manage how parts of an application depend on each other. Rather than manually wiring components together, Spring automatically detects and connects them at runtime based on configuration and annotations. This allows developers to break applications into independent services or classes, making the code easier to test, maintain and reuse. These modular parts are often called "beans" and are organised and managed by Spring within an application context. [17] [18]

Maintaining high-quality code means making changes easily and without disrupting other parts of the system. By breaking code into smaller, more modular units, such as services and components, it makes it easier to test and modify. For testing, unit testing is often best applied, focusing on small parts of the application with full knowledge of how they work. Each method is tested with various inputs, improving code coverage. Spring Boot's use of POJOs and services supports this by making individual components easy to isolate and test. However, some argue that simple POJOs do not always need to be tested because they tend to be straightforward and low-risk. [19] [18]

Inversion of Control (IoC) refers to a general principle where the flow of control in an application is reversed compared to traditional programming. Rather than components actively seeking the resources they need or controlling the flow themselves, control is handed over to a framework or external system. IoC is a broad concept and can be applied to many areas such as event handling, plugin systems and test frameworks; it is not tied to any one pattern, unlike Dependency Injection (DI). DI is a specific form of IoC where a component's dependencies (the services or objects it needs to function) are provided by an external source, usually a library or a framework. Instead of a component creating or locating its dependencies, these are "injected" from the outside, either through constructors, setter methods, or interfaces. This approach promotes loose coupling, making systems more modular, easier to test and easier to maintain. DI helps eliminate repetitive wiring code and allows for greater flexibility when changing implementations. For example, rather than a service class manually creating its database connection or logger, a DI supplies those objects, allowing the service to focus only on its main responsibilities. [20] [18]

Another design pattern present in Spring Boot is the repository pattern. The repository pattern is an abstraction that sits between the business logic and the data

access logic of an application. It moves data operations into a single interface, making code more maintainable, testable, and less error-prone. This layer hides the complexities of data access, offering a simplified interface for the rest of the application to interact with. It serves as a bridge that separates the two, allowing the business logic to focus on the application's functionality without worrying about how data is fetched or saved. The pattern also reduces code duplication by streamlining everyday data operations and allows developers to work with higher-level objects instead of dealing directly with the database. The repository layer typically handles operations like collecting, saving or updating data and transforming these into value objects, a design approach commonly seen in backend systems. [21] [18]

Pairing with the repository pattern is the Service pattern, which acts as a middle layer that captures business logic and orchestrates interactions between repositories and controllers. In service-oriented architectures (SOAs), services are defined as reusable, coherent sets of operations that interact through well-defined interfaces. These services are orchestrated into processes that define the behaviour and workflow of distributed systems across layers. In CSR (Controller-Service-Repository) architectures, commonly implemented in RESTful Spring Boot applications, each controller typically utilises one service, a decision to ensure clean separation of responsibilities and avoid tight coupling between components. Its implementation allows for the isolation and modularisation of business rules, improving code maintainability and supporting changes and reuse. Utilising design patterns focuses on the functional properties and preserves the quality characteristics of a system, which is important in a large-scale, distributed application. [22] [18]

At the top of the stack, the controller layer defines REST API endpoints and handles the receipt and dispatch of HTTP requests. It also controls input validation, path parameter parsing, and response structuring tasks. In layered architectures such as the CSR model, controllers are also tasked with translating domain-specific objects into Data Transfer Objects (DTOs) when the internal domain model does not align with the external API specification. DTOs map and expose only the required data, thereby preserving Encapsulation and improving clarity. These patterns closely align with core OOP principles. For example, Abstraction is achieved by these patterns through the separation of concerns. The repository pattern abstracts the data access logic from the rest of the application, whereas the service pattern abstracts the business logic from the data and presentation layers. These patterns also demonstrate Encapsulation as they package related functionality into specific layers. The repository encapsulates all database interactions, hiding the underlying implementation details. Similarly, the service layer encapsulates business logic and workflows, allowing for changes to be made internally without affecting other parts of the application. [23] [18]

Other practices worth highlighting include pagination, DTO usage, and transaction management. These practices complement the layered CSR architecture well by addressing performance, data integrity, and decoupling concerns, which are all key factors in building more modular and testable software. Pagination is an API design technique that returns extensive data sets into smaller, more manageable chunks rather than delivering all data elements at once. As a result, this helps both the client and the infrastructure handle the data more efficiently. The primary reasons

developers implement pagination include improving performance, reducing resource usage and handling large datasets more gracefully for clients. Pagination can be implemented by changing how data is returned from an API operation. Instead of sending all the data, the API includes control parameters in the request and metadata in the response that specify how much data is sent and what subset. However, a few factors must be considered when implementing pagination to ensure it remains effective. One example is order consistency; it is crucial to maintain a consistent order of the data across multiple pages. If the order of the data changes between requests, then clients may see inconsistent, duplicated data when navigating through the pages. Another risk is with stateful APIs and failover, so if the pagination relies on session or stateful data, failover scenarios can introduce issues where a server instance fails, and the client must re-request data. Proper handling is needed, such as ensuring session continuity or retry mechanisms. One thing that may be required before pagination can be added is DTOs. The response message structure often needs to be adjusted to include pagination metadata, which is usually done by introducing a DTO which encapsulates the original data and additional metadata required for pagination. It is important because the data elements should remain organised, while the pagination details must be communicated effectively. [24]

DTO usage goes beyond facilitating data transfer between layers in an application. It is important in improving the API's maintainability, scalability, and performance. It achieves this by separating the data transmission from business logic, ensuring that changes in one area do not disrupt the entire system. DTOs act as a way of communicating between the controller, service and data access levels, among others. They also improve security by only sending essential data, reducing the risk of leakage, and ensuring data consistency by following the predefined standards, making APIs more interoperable and reliable. [25]

Another practice worth highlighting is transaction management. When implemented, this ensures that reliable and consistent operations are handled on data in an isolated way, especially when multiple steps need to succeed or fail as one unit. In modern applications, it supports scalability and efficiency by keeping operations modular and separating business logic from data integrity concerns. In Java, Spring Boot and JPA, developers can define declarative transactions, which means that the framework automatically manages the start, commit, or rollback of transactions, allowing developers to focus on other logic while the system ensures ACID properties. This modular approach can help build applications that are easier to maintain, extend, and scale, especially when using microservices or working with multiple databases. [26] [18]

Challenges and Future Perspectives

ORM frameworks are often used to simplify the development of database-backed web applications by enabling developers to interact with databases more objectively. However, as applications scale and handle more data, they could run into performance challenges that appear from inefficient ORM interactions. This section will highlight these common issues relating to ORM performance focusing on lazy loading, memory usage, transaction boundaries, and data consistency.

Lazy loading is a mechanism where related objects are loaded only when accessed, which can be more efficient for reducing the retrieval of unneeded data. However, it can also lead to performance issues if not managed correctly. Specifically, N+1 queries occur when a query is issued for each individual related object, which results in multiple database queries that could otherwise be combined [27]. ORM frameworks also often translate database queries into memory-related objects. Data handled inefficiently can lead to excessive memory usage, particularly when many associated records are all loaded simultaneously, causing memory pressure and potentially freezing the application.

In a database application, managing transaction boundaries is essential. Transaction boundaries are the logical beginning and end points of a transaction, and they define the scope within which a set of operations is treated as a single unit of work. These boundaries ensure the ACID properties, especially atomicity and consistency, are followed. All data changes are applied or discarded entirely at these boundaries, depending on whether the transaction commits or rolls back. A DBMS uses transaction boundaries to decide when to apply discarded changes, maintaining transaction integrity. [28] [29]

ORMs do introduce tuning difficulties, particularly for performance-critical applications. These difficulties often stem from an object-relational impedance mismatch (ORIM) between object-oriented applications and relational databases. ORM frameworks attempt to bridge the mismatch by generating SQL queries from object-oriented code. However, in doing so, they often produce inefficient queries that can significantly degrade performance and scalability. ORM-generated queries can be inefficient because they rely on general-purpose heuristics rather than context-specific optimisations. This leads to poor query plans, poor index usage, and the inability to reuse execution plans effectively. ORM tools also obscure low-level SQL control, making it hard for developers to fine-tune performance-critical paths without circumventing the ORM, which in the end defies the point of having it. Lastly, ORMs struggle with compatibility regarding essential RDBMS features, such as join optimisation, execution plan reuse and indexing strategies. [30]

Many trends have emerged, either in response to the challenges faced with ORM's or as a form of advancement. One of these trends is microservices. This is an architectural approach where an extensive software application is broken down into smaller, loosely coupled services, each responsible for a specific function. These services can then be developed, deployed and maintained independently. This trend has gained significant traction due to its potential to offer flexibility and scalability, especially in cloud-native environments. Microservices-based architecture often uses a database-per-service pattern, where each microservice manages its own

database, creating challenges for managing data across the different services. ORM tools, which facilitate the interaction between the application and relational databases, were initially designed for monolithic applications with a centralised data model, so when applied to microservices, these ORM tools can have difficulties due to the decentralisation of databases and the need for services to interact across different databases. [31]

Cloud-native development is gaining traction as an emerging trend, mainly due to its flexibility, scalability and reduced maintenance efforts provided by cloud environments. As organisations increasingly design systems specifically for the cloud, they can then better align the resource demands with the supply, reducing the cost and operational efficiencies. However, despite these benefits, cloud migration remains a challenging and complex process, requiring adjustments to ORM models to support different database technologies. Cloud providers offer various services to host databases, and each service may have its specificities, meaning that traditional ORM tools may need to be extended or customised to handle cloud-specific databases, potentially leading to performance issues if not properly adapted. [32]

The emerging trend of GraphQL APIs represents a recent shift in how applications interact with data, promoting flexible, client-driven queries. It is meant to query data more precisely than traditional REST APIs. Unlike REST, which delivers fixed data from predefined endpoints, GraphQL allows dynamic, client-defined queries that streamline data access and improve performance. This shift has gained popularity because developers seek more control and visibility in how applications communicate with data sources. However, this flexibility introduces challenges for ORMs and OOP models, which are typically structured around static entity relationships. The dynamic nature of GraphQL can have inefficiencies like excessive database calls or complex join operations when using conventional ORMs. To adapt, developers would need to reconsider their plans for ORM to better align with GraphQL's needs and maintain scalable, better-performing systems, such as introducing parallel processing. [33]

As highlighted previously, implementing DTOs could also have a positive effect, allowing data to be shaped and transferred more efficiently and controlled, reducing the need to load entire entities with unnecessary associations. Another strategy worth implementing would be batch processing, which refers to handling multiple database operations, such as inserts and updates, in grouped executions rather than one-by-one. This technique significantly reduces the overhead caused by repeated database communication. This improves performance and reduces memory usage, particularly during large-volume data operations. [34]

While this essay has explored several limitations and performance concerns related to ORMs, it is important to remember that ORMs still offer valuable abstraction benefits when used appropriately. One practical approach is to use ORM selectively, combining it with raw SQL queries in performance-critical areas. This hybrid model allows developers to retain the convenience and maintainability of ORM for standard operations, while using hand-optimised SQL for complex or large data operations. Another strategy for improvement is query optimisation. This is essential, especially in large-scale applications; one key recommendation is using lazy fetching, similar to lazy loading, instead of eager loading to reduce unnecessary data retrieval. Cache

management and navigation strategies are heavily encouraged as they are critical to performance. Implementing first- and second-level caches can reduce redundant database access by storing often used data. [35]

Conclusion

Object-oriented programming principles such as Encapsulation, Inheritance, Polymorphism, and Abstraction significantly influence the development of modular, maintainable and reusable software. These principles help organise code in a way that is flexible and simplifies maintenance. When applied to web services, these principles become very important. Polymorphism and Abstraction enable flexible and reusable service designs, allowing different objects or methods to be used interchangeably. Encapsulation protects data while providing a straightforward interface for interacting with web services, ensuring data security and reducing complexity. Inheritance reduces repeated code, allowing web services to be built on existing structures, making it easier to maintain and extend in the future.

ORM frameworks are built on these OOP principles to connect object-oriented code and relational databases. Frameworks such as Hibernate or EF allow developers to interact with databases using object-oriented models, removing the difficulty of writing raw SQL queries. While ORMs improve code readability and maintainability, they also have their own sets of challenges, such as object-relational impedance mismatch, where the object-oriented nature of code conflicts with the relational database structure, which can lead to inefficiencies later down this line, particularly in large-scale applications where mapping objects to relational data becomes complex and slow.

Integrating ORMs with web services can simplify data access through object models, but can also cause performance issues when handling large datasets or complex relationships. For example, lazy loading can result in many minor database queries, which might degrade performance if not managed carefully. Additionally, managing transaction boundaries within ORM systems ensures that data integrity is maintained, but scaling ORM performance for high-volume applications can be difficult. The need for performance tuning and handling the mismatch between object-oriented and relational structures often complicates the integration of ORM with web services, especially in situations that are performance-sensitive.

To conclude, while OOP principles, web service and ORM frameworks work together to build efficient and scalable systems, careful management must be implemented to address the performance issues and difficulties that may appear as applications grow.

References

- [1] A. T. Cohen, "Data abstraction, data encapsulation and object-oriented programming," Department of Computer and Information Sciences University of Delaware, Newark, DE, 1984.
- [2] A. TAIVALSAARI, "On the Notion of Inheritance," Nokia Research Center, Helsinki, 1996.
- [3] H. M. Ali, M. Yashar Hamza and T. A. Rashid, "Exploring Polymorphism: Flexibility and Code Reusability in ObjectOriented Programming," International Conference on Recent Innovation in Engineering, Duhok, 2023.
- [4] O. L.V, "MAIN CONCEPTS OF OBJECT-ORIENTED PROGRAMMING," National Aviation University, Kyiv, 2017.
- [5] R. Klump, "Understanding Object-Oriented Programming Concepts," IEEE, Vancouver, 2001.
- [6] A. Rodriguez, "RESTful Web services: The basics," IBM, 2008.
- [7] J. Tihomirovs and J. Grabis, "Comparison of SOAP and REST Based Web Services Using Software Evaluation Metrics," Information Technology and Management Science, Riga, 2016.
- [8] J. Rojas, O. Fragoso, R. Santaolaya and J. Soto, "Generation of POJOs and DAOs Classes from Metadata Database," IEEE Latin America Transactions, 2021.
- [9] K. Drzazga, M. Bobel and M. Skubiewska-Paszkowska, "Comparative analysis of selected object-relational mapping systems for the .NET platform," Journal of Computer Sciences Institute, 16, 285–292, Lublin, 2020.
- [10] D. B. M. N. a. K. W. C. Ireland, "A Classification of Object-Relational Impedance Mismatch," First International Conference on Advances in Databases, Knowledge, and Data Applications, Gosier, 2009.
- [11] K. Hule, "Analysis of Different ORM Tools for Data Access Object Tier Generation: A Brief Study," Cosmos Scholars Publishing House, 2023.
- [12] 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, Brazil, 2017.
- [13] D.-P. POP and E. C. BUTOIU, "AN ANALISYS OF RDBMS, OOP AND NOSQL CONCEPTS," JOURNAL OF INFORMATION SYSTEMS & OPERATIONS MANAGEMENT, Bucharest, 2015.
- [14] A. M. Bonteanu and C. Tudose., "Performance Analysis and Improvement for CRUD Operations in Relational Databases from Java Programs Using JPA,

Hibernate, Spring Data JPA," Multidisciplinary Digital Publishing Institute, Basel, 2024.

- [15] N. Dhingra, "Review on ORM based JPA implementations," Proceedings of the 2016 APCBEES International Conference, Toronto, 2016.
- [16] A. K. Bayya, "A Comprehensive Study On Hibernate As A Data Persistence Solution For Financial Applications," Roman Science Publications and Distributions, London, 2024.
- [17] X. Luan, "Implementation and Analysis of Software Development in Spring Boot," California State Polytechnic University, Pomona, 2021.
- [18] R. Johnson, J. Hoeller, K. Donald, C. Sampaleanu, R. Harrop, T. Risberg, A. Arendsen, D. Davison, D. Kopylenko, M. Pollack, T. Templier, E. Vervaet, P. Tung, B. Hale, A. Colyer and J. Lewis, Spring Framework Reference Documentation, San Francisco: SpringSource, 2013.
- [19] O. Patlak, "Strategies to Ensure Software Quality in Existing Java Applications," University of Applied Sciences FH Campus Wien, Vienna, 2023.
- [20] D. R. PRASANNA, Dependency Injection, DESIGN PATTERNS USING SPRING AND GUICE, Greenwich: Manning Publications Co, 2009.
- [21] D. Damyanov, Z. Varbanov and S. Varbanova, An improved approach of using data storage services in ASP.NET Core, Varna: IEEE, 2022.
- [22] C. Pahl and R. Barrett, "Pattern-based Software Architecture for Service-oriented Software Systems," *e-Informatica Software Engineering Journal*, vol. 4, no. 1, pp. 31-45, 2010.
- [23] O. Özkan, Ö. Babur and M. van den Brand, "Refactoring with domain-driven design in an industrial context: An action research report," Springer, Dordrecht, 2023.
- [24] M. tocker and O. Zimmermann, "API Refactoring to Patterns Catalog, Template and Tools for Remote Interface Evolution," Association for Computing Machinery, Rapperswil, 2024.
- [25] S. Kumar, "Spring Boot Application using Three Layered Architecture," aypee University of Information Technology, Solan, H.P., Solan, 2023.
- [26] P. K. Joshi, "Concept to Production: Crafting Payment Solutions with Java and Spring Boot," *International Journal of Science and Research (IJSR)*, vol. International Journal of Science and Research (IJSR), p. 1590–1596, 2021.
- [27] J. Yang, P. Subramaniam, S. Lu, C. Yan and A. Cheung, "How not to structure your database-backed web applications: a study of performance bugs in the wild," in *ICSE '18 — 40th International Conference on Software Engineering*, Gothenburg, 2018.

- [28] H. Desamsetti, "Relational Database Management Systems in Business and Organization Strategies," *Global Disclosure of Economics and Business*, vol. 9, no. 2, p. 151–162, 2020.
- [29] E. E. Cobb, "The impact of object technology on commercial transaction processing," Springer-Verlag, San Jose, 1995.
- [30] 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, 2020.
- [31] F. Freitas, A. Ferreira and J. Cunha, "A Methodology for Refactoring ORM-Based Monolithic Web Applications into Microservices," Elsevier, Amsterdam, 2023.
- [32] M. Ellison, "Evaluating Cloud Migration Options for Relational Databases," University of York, York, 2017.
- [33] F. Jovanov, V. Zdraveski, M. Gusev and M. Kostoska, "Optimization and Parallelization of Object-Relational Mappers," in *2023 46th MIPRO ICT and Electronics Convention (MIPRO)*, Opatija, 2023.
- [34] H. Houssein, "Are ORMs the end of stored procedures?," Linnaeus University, Faculty of Technology, Department of Computer Science, Växjö, 2017.
- [35] S. Saddar, J. Baloch, M. Sami, N. Pirzada, V. Khaliq and A. A. Memon, "Evaluating Performance of Hibernate ORM based Applications using HQL Query Optimization," Oriental Scientific Publishing Company, Varanasi, 2018.

Iestyn Green Web Services OOP essay

GRADEMARK REPORT

FINAL GRADE

79/100

GENERAL COMMENTS

A good report with nice use of sources.

Some sections could be more terse without losing meaning - consider whether you can use fewer words to convey the same meaning for the upcoming major project.

Very good work overall.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

