

2420643_PJ_Rai

by Pj Rai

Submission date: 02-May-2025 11:19AM (UTC+0100)

Submission ID: 257583180

File name: 58810_Pj_Rai_2420643_PJ_Rai_664183_1737804618.docx (170.27K)

Word count: 6645

Character count: 42126

Introduction

Over the past few decades, the landscape of software development has experienced significant transformation, driven by the need for more organized, scalable, and maintainable solutions. One key development in software architecture is the rise of object-oriented programming (OOP), which has revolutionized how developers design and structure code [1-4]. Simultaneously, the evolution of web services - from SOAP (Simple Object Access Protocol) to REST (Representational State Transfer) - has simplified client-server communication across networks [5-10]. Furthermore, Object-Relational Mapping (ORM) technologies have emerged, seamlessly integrating with OOP languages such as Java and scalable frameworks like SpringBoot to improve database interaction and system performance [11-30].

This paper explores the evolution of these pivotal technologies - OOP, REST, and web services - and examines their impact on modern software development. It will also discuss the challenges posed by emerging technologies, such as the increasing demand for real-time data [31-40] and the growing need for scalable solutions in today's applications.

Object-Oriented Programming (OOP)

OOP which originated in the 1960s and gained widespread adoption in the 1990s, became popular for developing graphical user interfaces (GUIs) in operating systems like Windows 98 and Windows NT, using languages such as C++ [1,4]. Before OOP, software development relied on procedural programming, which followed a top-down approach where systems were structured into sequential functions. However, as software complexity increased, procedural approaches struggled with adaptability and maintainability. These challenges led to the rise of OOP, which introduced abstraction, encapsulation, and inheritance, allowing for more modular, scalable, and reusable code [2]. The adoption of OOP improved software design for client-server architectures and real-time systems, reducing development costs and enhancing flexibility [2]. As a result, OOP languages such as Java, C#, and Ruby gained popularity due to their reliability and ease of use in building complex, real-time applications [4].

Object-Oriented Programming (OOP) represents a shift from the execution-focused approach of procedural programming to a model where objects collaborate to share and manipulate data. Unlike procedural programming, which follows a step-by-step sequence of instructions, OOP structures code around objects that interact with each other, making programmes more modular and reusable [2, 4].

At the core of OOP are objects - self-contained entities that encapsulate both data and behaviour. Every object has a unique identity, a state defined by its attributes, and behaviour represented by the actions it can perform through methods. Objects do not function in isolation; they interact with one another through relationships, enabling more complex systems to be built from simple, independent components [2, 4].

Objects are created based on classes, which serve as blueprints defining shared characteristics and behaviours. A class groups objects with similar properties, making it easier to create and manage multiple instances without redundant code. Through methods, classes define how

their objects interact with others, promoting message passing as a way for objects to communicate. For example, a Car class might define an `accelerate()` method. A Driver class does not need to understand the internal workings of acceleration; it simply calls the method on a Car object. This approach ensures loose coupling, where components interact in a controlled manner without unnecessary dependencies [1].

Encapsulation further reinforces this controlled interaction by restricting direct access to an object's internal state. In OOP, a class binds data and behaviour together while preventing unintended modifications from external entities. This safeguards the integrity of objects and ensures that data can only be accessed through well-defined interfaces. Returning to the Car example, the `accelerate()` method belongs to the Car class and cannot be directly altered by the Driver class, preserving the internal logic of the object while allowing controlled interaction [1, 4].

Closely related to encapsulation is abstraction, which simplifies complex systems by exposing only the necessary details while hiding implementation specifics. When interacting with a Car object, the user only needs to know that calling `accelerate()` makes the car move, without needing to understand the mechanics behind it. While encapsulation focuses on restricting access to internal data, abstraction emphasises presenting a simplified interface, ensuring that objects are used efficiently without unnecessary complexity [4].

Inheritance allows a child class to inherit methods from a parent class and use those methods to define its own behaviour through polymorphism. Polymorphism enables objects or methods to operate in multiple forms, adapting to different requirements. There are two primary types of polymorphism: compile-time polymorphism and runtime polymorphism. Compile-time polymorphism is achieved through method overloading, where multiple methods with the same name exist but have different parameters [2,4]. For example, the Car class could have two versions of the `accelerate()` method: one with `accelerate(int speed)` and another with `accelerate(int time, boolean speedMode)`. The compiler determines which version of the method is called based on the number and type of arguments provided.

On the other hand, runtime polymorphism occurs when a method in a parent class is overridden by a subclass to provide its own implementation. This is resolved during runtime. For instance, the Car class could have a `makeSound()` method, and subclasses like `ElectricCar` and `RaceCar` would override this method with their own specific implementations: `ElectricCar.makeSound()` might output "go swoosh", while `RaceCar.makeSound()` might output "broom". This ability to define the same method with different implementations in various subclasses leads to code reusability and promotes cleaner, more flexible code [3].

Advantages and Disadvantages

The major advantages of Object-Oriented Programming (OOP) stem from its unique features, such as reusability, extensibility, and flexibility, compared to other programming paradigms. Reusability ensures that code can be reused, significantly reducing development effort and enhancing reliability. Writing less code also accelerates software development, leading to increased efficiency. Extensibility, achieved through inheritance, allows new classes to inherit existing attributes while still enabling customisation. Subclasses can modify their methods to suit their specific functionality and context. However, modern best practices often favour composition over inheritance to avoid the complexities of deep class hierarchies. This

approach provides flexibility, as subclasses (or composed objects) can modify or extend functionality without altering the original codebase. Additionally, modularity improves software development productivity by enabling a clear separation of concerns. If a part of the system needs to be changed or updated, modifications can be made without affecting the rest of the system, leading to greater maintainability. Ultimately, these advantages contribute to the production of higher-quality software at a lower cost and within a shorter development time [1, 3, 4].

Despite its benefits, OOP is not without drawbacks. If poorly designed, tightly coupled code can emerge, making it difficult to maintain and modify. Instead of improving scalability, this can lead to increased complexity and decreased flexibility. Additionally, OOP programmes tend to have a larger codebase than procedural or functional programming styles. The added abstraction layers and object management mechanisms may introduce performance overhead, making OOP solutions slower in certain scenarios—especially in applications where execution speed is critical. Furthermore, OOP is not the best fit for all types of problems. Procedural programming may be more efficient for simple, linear tasks, while functional or logic programming paradigms are often preferred for mathematical computations and data transformations. Choosing the right paradigm depends on the specific needs of the project [1, 2].

Web services

A web service is a client-server application that acts as a conduit for various applications to communicate with each other over the network via the internet. It uses Extensible Markup Language (XML) messaging, which ensures that all transmissions are sent in a standard format. This can be done over transport protocols such as TCP/IP, HTTP, HTTPS, and other network protocols. Web services allow for interoperability across different applications, providing services throughout the internet. Fundamentally, a web service works by receiving a client's request and delivering a response back to the client application (see Fig. 1) [6,9].

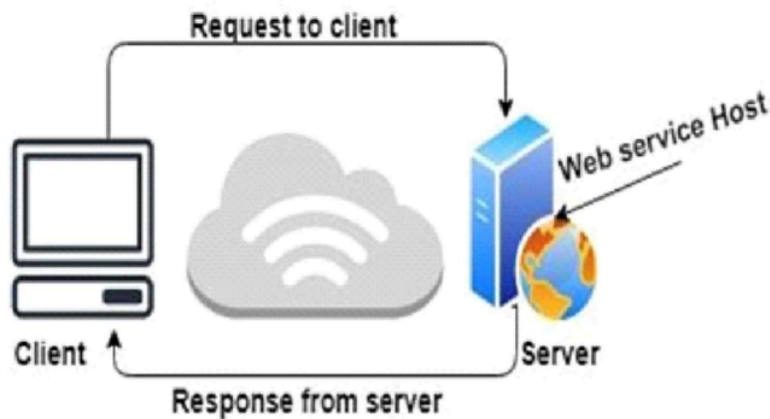


Figure 1: Virtual representation of web service

The three main components of web services:

1. SOAP – Simple Object Access Protocol
2. WSDL – Web Services Description Language
3. UDDI – Universal Description, discovery, and Integration

SOAP serves as an XML-based protocol that defines the rules for structuring messages and facilitates communication between web services. It ensures consistency and reliability in exchanging information over a network.

WSDL acts as a contract between the client and the server, describing how the web service functions. It defines the operations, data types, and message formats that the client and server will use for communication. The client reads the WSDL file to understand the available services and how to interact with the web service. In this sense, WSDL also acts as an interface, outlining the input/output data types, the operations available, and the message format required for interaction. UDDI acts as a registry where web services can be published and discovered. A service provider registers its web service in a UDDI registry, providing essential metadata, such as the WSDL location, business name, and other service-related details. A client then searches the UDDI registry to find the relevant web service and retrieves the WSDL file to interact with the service [6,9].

There are two main types of web services: SOAP and REST – Representational state transfer.

In the late 1990s, SOAP (Simple Object Access Protocol) was introduced by Microsoft as a standardised protocol for web services. It was designed to be flexible, supporting various transport protocols such as HTTP, SMTP (Simple Mail Transfer Protocol), JMS (Java Message Service), and TCP/IP). SOAP was built with high security and reliability in mind, offering built-in support for web standards like WS-Security, transactions, and ACID compliance. Due to these features, many industries, such as banking, finance, and telecommunications, adopted SOAP for their middleware applications, allowing them to send the same SOAP signals in a uniform format [7,9].

However, SOAP's reliance on XML messages led to some challenges. These messages were often large and slow to process, resulting in scalability issues. This inefficiency became particularly problematic for web-scale applications, which require fast performance and minimal latency. By the 2000s, a few milliseconds of delay could cost millions in lost revenue if customers switched to competitors [6].

This challenge drove companies like Amazon, Google, and Facebook to shift towards REST (Representational State Transfer) for their APIs [5]. REST offered an alternative architectural style for building web services. Unlike SOAP, which could use multiple transport protocols, REST relied solely on HTTP. Its simplicity, combined with its lightweight nature, made it easier to integrate with web applications, significantly improving deployability and scalability.

In his PhD dissertation, Roy Fielding, the inventor of REST, proposed that making web services stateless and allowing the client and server to evolve independently could achieve flexibility and scalability by simplifying server components. Unlike SOAP web services, which rely on complex, XML-based messages that can be heavyweight and often stateful (i.e., tracking interactions over multiple requests), REST APIs are designed to ensure that the

server does not maintain any session or context about the client between requests. This statelessness makes REST inherently scalable, as any server can handle any request without the need to manage client state or session information. These principles, along with cacheability, became core characteristics of REST. In RESTful systems, responses from the server can be explicitly marked as either cacheable or non-cacheable, improving performance.

Additionally, resources in REST are identified by URIs (Uniform Resource Identifiers), and interactions are carried out using standard HTTP methods. These methods include:

- GET – retrieves data from the server
- POST – sends data to the server to create a new resource
- PUT – updates an existing resource by replacing it with new data
- DELETE – removes a resource from the server [5-10].

Ultimately, web services and RESTful methods are critical in modern engineering due to their lightweight, scalable, and flexible nature. They serve as the foundation for enabling seamless communication and data exchange. In today's fast-paced environment, users expect a consistent and responsive experience across devices, regardless of geographic location. By leveraging web services, mobile applications can deliver up-to-date information, support real-time interactions, and meet the high-performance demands of users [7,9].

Object Relational Mapping

The relational database model gained popularity after its first commercial release by Oracle in the late 1970s. At the time, Relational Database Management Systems (RDBMS) were already widely used on mainframes. However, by the late 1980s, there was a significant technological shift from mainframes to the client-server model—a structure that remains popular today. While relational mainframes managed large volumes of business data, OOP languages, particularly Java, were simultaneously rising in popularity [14].

Relational databases and OOP are inherently different paradigms, which created a communication challenge between the two. For instance, an object in an OOP system needed to be mapped to columns in a relational database table. This disparity gave rise to a significant *impedance mismatch* problem [14].

To address this issue, the concept of Object-Relational Mapping (ORM) was introduced in 1994 by the Smalltalk community. The key idea behind ORM is to bridge the gap between OOP and relational databases by providing an interface that maps objects to rows in a relational table. In ORM, objects are temporarily instantiated at runtime, whereas the data in a relational database is persistent. This abstraction simplifies development and enhances code maintainability by automatically converting data structured in a relational format into forms more naturally used in OOP [14].

ORM soon became standard practice, particularly as Java - the most popular OOP language - grew in popularity and due to the open-source nature of many ORM tools [14]. The Jakarta Persistence API (JPA) is the standard ORM specification for Java, with Hibernate being its most widely used implementation [13].

One of the key reasons for Hibernate's widespread adoption is its implementation of the Data Mapper architectural pattern. In this model, Hibernate acts as an intermediary between the Java objects in an application and the underlying relational database tables. Rather than having objects manage their own persistence logic, Hibernate handles the mapping and data access layer. This separation allows domain objects to remain independent of database concerns, promoting cleaner code, greater flexibility, and improved maintainability [15].

Hibernate's popularity is further reinforced by its rich and versatile feature set. It provides automatic mapping of Java objects to database tables using annotations or XML configuration, significantly reducing the need for repetitive boilerplate code. Hibernate also automatically generates SQL for common operations such as Create, Read, Update, and Delete (CRUD), allowing developers to interact with the database without writing explicit SQL. Its database independence enables seamless migration between different relational database systems—such as MySQL, PostgreSQL, or Oracle—with minimal code changes.

Performance is another area where Hibernate excels. It incorporates an efficient caching mechanism, supporting both first-level (session-based) and second-level (shared across sessions) caching, which minimizes database queries and boosts application responsiveness. Furthermore, Hibernate supports lazy loading, meaning related data is only fetched when it is actually needed. This feature is particularly useful when working with large or complex datasets, as it helps optimise memory usage and application performance [12].

Beyond its technical strengths, Hibernate—and ORM tools in general—offer substantial development benefits. By abstracting away the complexities of low-level database interaction, Hibernate allows developers to focus more on solving domain-specific problems. This abstraction speeds up development and results in cleaner, more maintainable codebases. Automated ORM also helps reduce the volume of code required, making applications easier to understand, test, and refactor.

Although hand-coded persistence layers in low-level languages may offer better performance in specific use cases, Hibernate compensates with optimisation strategies such as application-tier caching, which can deliver high efficiency without excessive manual tuning. This allows developers to concentrate on optimizing true performance bottlenecks, rather than wasting resources on premature optimisations. Additionally, Hibernate's vendor independence mitigates the risk of database lock-in. It supports multiple DBMSs and enables flexible deployment scenarios - such as using a lightweight local database for development while relying on a more robust solution in production [15].

However, ORM tools like Hibernate are not without limitations. They introduce an additional layer of abstraction, which can add performance overhead - especially in high-throughput or resource-constrained environments. Each programming language has its own ORM solution, creating a learning curve when switching languages or frameworks. Moreover, although ORM promotes database independence, it can inadvertently create tool-level lock-in, where switching away from the ORM itself becomes challenging. Debugging can also be more difficult, as ORM-generated SQL is often abstracted away from the developer. For small-scale projects that only require basic CRUD functionality, ORM may add unnecessary complexity and overhead [16,17]. Like any abstraction, Hibernate introduces trade-offs - but when used appropriately, it enables developers to work more efficiently and build robust, adaptable systems with less effort.

Spring boot

Spring is an open-source web application framework written in Java, created by Rod Johnson. As software development evolved from monolithic architectures to more distributed systems, the need for reusable components to accelerate development became increasingly apparent [18, 19]. In response, Spring emerged as an open-source framework that unified various Java libraries for web development, encompassing areas such as database access, security, and web services.

The core features of the Spring Framework—such as Inversion of Control (IoC), Dependency Injection (DI), widely adopted annotations, and foundational implementations like Spring MVC - serve as the backbone for many frameworks within its ecosystem. Building on this robust foundation, Spring Boot was released in April 2014 as an extension of the Spring Framework. It leverages these core features while further simplifying application configuration through production-ready defaults, auto-configuration, and embedded servers like Tomcat, Jetty, and Undertow. This integration enables developers to build stand-alone, production-grade Spring applications with minimal setup, significantly accelerating development and reducing boilerplate code [19].

In modern software development, applications are typically composed of components that perform specific functions within the system. These components often include business logic and persistent objects. Business logic dictates how an application behaves and processes its core functions, while persistent objects represent crucial data that must be stored and managed across the application's lifecycle. Efficient management of these components is essential to maintain data consistency and ensure accessibility when needed.

A common challenge arises when these components need to interact, particularly in managing their dependencies. This is where Spring Boot's implementation of IoC plays a key role. Inversion of Control is a design paradigm where the framework, not the application code, takes responsibility for creating and managing objects. Upon running a Spring Boot application, the IoC container automatically initializes components, manages their lifecycles, and connects them in a logical order.

The IoC container also manages the lifecycle of "beans"—objects instantiated, assembled, and managed by the container. A specific implementation of IoC is Dependency Injection (DI), which allows the framework to inject dependencies into classes at runtime. By doing so, classes remain focused on their specific functionality, without needing to handle the creation or configuration of their dependencies. This separation of concerns supports better encapsulation and enhances maintainability.

IoC and DI are closely aligned with key object-oriented programming (OOP) principles, such as encapsulation, abstraction, and modularity. By externalizing dependency management, these principles foster loose coupling, where classes depend on abstractions (e.g., interfaces) instead of concrete implementations. This decoupling ensures that changes to dependencies can be accommodated without requiring modifications to the dependent classes, increasing the flexibility and extensibility of the system. Furthermore, it enhances testability, as dependencies can be substituted with mock or stub implementations during testing. For instance, a class that depends on a Car interface can be tested with various implementations (e.g., ElectricCar, HybridCar, DieselCar) without altering the class itself, making the codebase more adaptable and maintainable [18 - 20].

All these IoC features in Spring Boot contribute to the adoption of the Model-View-Controller (MVC) design pattern. The View is responsible for displaying the interface to the user, while the Model serves as the data transfer object, enabling data exchange between the browser, controller, and application. The Controller handles user requests from the View, processes them, and updates the Model layer, which in turn updates the View. This structure promotes loose coupling, flexibility, and a focus on design, enabling developers to create clean, maintainable applications.

Another complementary programming paradigm within Spring is Aspect-Oriented Programming (AOP). While OOP excels in organising behaviour into classes, it struggles with cross-cutting concerns - features like logging, security, or performance monitoring that span multiple modules. Implementing these concerns in OOP can lead to code duplication and clutter. AOP solves this problem by modularising such concerns into separate units called "aspects," which are applied declaratively across the codebase. Spring utilizes IoC and DI to inject these aspects transparently, allowing for cleaner, more maintainable code without disrupting core business logic [20].

The Repository Pattern is another design pattern commonly used in modern frameworks like Spring Boot. This pattern abstracts data access by providing a clean interface for interacting with data, decoupling the domain logic from the underlying persistence mechanism. Repositories often extend interfaces like `JpaRepository` from Spring Data JPA, which automatically generates standard CRUD operations and query methods. This reduces boilerplate code and simplifies database interactions. While the Data Access Object (DAO) pattern focuses on low-level persistence operations, the Repository Pattern builds on that by emphasising domain-centric access and business logic encapsulation. By promoting abstraction, loose coupling, and single responsibility, the Repository Pattern aligns well with OOP principles, making the codebase more maintainable, testable, and scalable.

Spring Boot and ORM frameworks are widely used to build scalable and efficient web services. Several best practices can be followed to ensure performance, reliability, and maintainability of such systems. One key aspect is the efficient use of databases through ORM. While frameworks like Hibernate and JPA simplify data access, they can also generate inefficient queries if not properly configured. Techniques such as batch processing and lazy loading can significantly improve performance by loading data only when needed. Furthermore, the use of database indexing can speed up common queries and joins, especially when working with large datasets. Connection pooling mechanisms like HikariCP help reduce the overhead of establishing database connections by reusing existing ones, improving responsiveness under load [24]. Caching is another critical practice for enhancing performance and scalability. It reduces the need for repeated database access, particularly when the same data is requested frequently. Spring Boot supports in-memory caching solutions such as Ehcache, Caffeine, and Guava Cache, which can significantly reduce the load on the database server. For distributed systems, implementing external caching solutions like Redis allows data to be cached across multiple nodes, supporting scalability and high availability [24]. In terms of architectural design, a stateless approach is recommended. RESTful web services using HTTP methods (CRUD operations) paired with stateless security mechanisms like JWT (Java Web Token) enable secure, lightweight communication between services while avoiding the complexity and overhead of maintaining server-side session states [22]. When designing microservices, scalability is further achieved by developing loosely coupled, independently deployable services. However, this introduces challenges such as increased network overhead from inter-service communication. Again,

strategic caching can be used to minimize redundant calls and reduce latency [25]. Building scalable and efficient web services with Spring Boot and ORM requires a combination of smart database practices, strategic caching, stateless design principles, and thoughtful microservice architecture - all aimed at maximizing performance, maintainability, and scalability in modern web systems [21-25].

While Object-Oriented Programming (OOP) with ORM frameworks offers many advantages, such as simplifying development and providing an abstraction layer over database interactions, it comes with significant performance and scalability challenges. One major limitation is that ORM frameworks can obscure how high-level code is translated into database queries, making it difficult for developers to understand and optimize the queries being executed. This lack of transparency can hinder efficient database interaction and obscure potential performance bottlenecks.

Additionally, ORM frameworks like Hibernate often struggle to generate efficient query execution plans. They typically prioritise ease of use over optimisation, leading to suboptimal queries. For instance, the ORM may generate unnecessary queries, especially in cases involving lazy loading. Lazy loading can result in the N+1 query problem, where an extra query is executed for each related entity, unless the data has been cached or preloaded. This can lead to a large number of queries, significantly degrading performance. The situation worsens when bugs are introduced, such as with improper session management or entity state mismatches, which can cause incorrect interactions with the underlying database, resulting in crashes or improper data storage and retrieval.

Although ORM frameworks often support caching to mitigate some of these issues, the cache itself must be carefully managed to avoid consistency problems and resource overuse. Additionally, while ORM frameworks provide a powerful abstraction for developers, they can become a performance bottleneck in high-traffic applications or when dealing with complex queries and large datasets.

To address these limitations, developers can use a hybrid approach, leveraging ORM for basic operations and resorting to custom SQL queries for more performance-critical sections of the application. This allows for fine-tuned control over query generation while still benefiting from the ORM's productivity advantages [27-31].

Emerging trends in web services

One of the key trends in modern web applications is the increasing reliance on cloud services to enable horizontal scaling. This shift has necessitated more complex network communication and emphasized the importance of web APIs built on cloud-native principles. Cloud-native architectures prioritise scalability, modularity, and resilience, typically achieved through the use of microservices and containers [36]. However, traditional database access methods such as ORM can be challenging to use effectively in containerized environments. While Object-Oriented Programming (OOP) and ORM perform well in monolithic or modular architectures, their tight coupling between application and database can limit scalability, portability, and maintainability [32].

ORM continues to serve an essential role when working with relational databases, especially where transactional consistency and strong referential integrity are critical. However, the massive growth of data on the internet has led to the rise of microservices, which often favour document-based and NoSQL data models. These models are better suited for handling large volumes of data with high flexibility and performance requirements. As a result, Object-Document Mapping (ODM) has emerged as a viable alternative to ORM for working with NoSQL databases such as MongoDB. NoSQL databases offer schema flexibility and horizontal scalability, making them well-aligned with cloud-native architectures. ODM facilitates this by providing a natural and efficient way to map application data to JSON-like document structures, minimising translation overhead.

The advantages of ODM include improved performance at scale, schema flexibility, and more intuitive data mapping in document-oriented applications. However, NoSQL databases typically support limited join operations, making complex queries less efficient. This limitation makes them less suitable in scenarios where strong relational integrity and transactional consistency are required - conditions under which relational databases and ORM remain the better choice [40].

Another significant shift in web service architecture has been the adoption of GraphQL. Initially developed by Facebook to address the limitations of REST, GraphQL gained widespread popularity by the late 2010s. RESTful web services are built on top of HTTP and leverage its native caching mechanisms to ensure high availability. In contrast, GraphQL is transport-agnostic and can operate over any client-server protocol. While REST uses standard HTTP methods to represent actions (e.g., GET, POST, PUT, DELETE), GraphQL introduces two core interaction types: queries and mutations. Queries are used to retrieve data, while mutations handle data modifications.

GraphQL allows developers to define resources hierarchically through types that are interconnected via graph relationships. At the core of GraphQL's architecture is a strongly typed schema that defines the data models and their relationships. This schema acts as a contract between the client and server, ensuring predictable and transparent communication. One of GraphQL's key advantages is its ability to return only the data explicitly requested by the client, effectively eliminating the issues of over-fetching and under-fetching that often arise with REST APIs.

Furthermore, GraphQL consolidates data retrieval into a single endpoint, simplifying communication and reducing the complexity of managing multiple REST endpoints. Its syntax, modelled after JSON, is intuitive and closely aligns with modern frontend data

structures. In addition, GraphQL supports real-time data through subscriptions, allowing clients to receive live updates in response to specific events. This capability is particularly valuable for applications that require high interactivity, such as chat platforms, collaborative editing tools, and live dashboards. By enabling seamless integration of live data streams, GraphQL enhances the responsiveness and user experience of modern web applications [31-34].

Although GraphQL provides significant flexibility and efficiency in data fetching, it also introduces several challenges that should be considered. One of the primary drawbacks is the difficulty in implementing effective caching. Unlike REST, GraphQL does not include a built-in caching mechanism, and the highly dynamic and unpredictable nature of queries makes it difficult to apply standard caching strategies. Additionally, GraphQL can add complexity to the architecture, particularly for teams new to its concepts, resulting in a steep learning curve. Another notable disadvantage is the risk of performance issues, especially the N+1 query problem. This occurs when resolvers fetch related data individually for each item in a list, leading to multiple inefficient database calls - particularly problematic in deeply nested queries. These drawbacks underscore the importance of thoughtful design and tooling when adopting GraphQL in production systems [33,34].

Improving ORM strategies in large-scale enterprise applications

Modern application development requires developers to use ORMs strategically to ensure scalability and performance. Two of the most important practices in this context are indexing and caching. ORMs often allow developers to define indexes directly in their models, which can significantly improve query efficiency. Different indexing strategies are beneficial depending on the system: for example, OLTP (Online Transaction Processing) systems benefit from high write performance, while OLAP (Online Analytical Processing) systems prioritize fast read performance. More advanced indexing techniques, such as adaptive or hybrid indexing, can further optimize specific query patterns, though these are typically implemented at the database level [37].

When it comes to write performance, ORMs provide a range of tools to handle high-frequency write operations efficiently. They offer bulk insert and batch update features to reduce the overhead of individual write operations, which is crucial for transactional systems. Many ORMs also support transaction management, ensuring that write operations are consistent and efficient even under heavy loads. For example, Django ORM allows bulk inserts, which can be particularly useful in applications with frequent data changes, like e-commerce sites. On the other hand, in read-heavy systems (like OLAP), ORMs can sometimes introduce inefficiencies due to the way they generate SQL queries. Developers often need to optimise ORM-generated queries or complement them with caching strategies like read-through caching, where data is first checked in the cache before querying the database. This ensures that repeated data requests don't overload the database. Some ORMs, such as Prisma, also allow for efficient query generation and support caching mechanisms that help speed up repeated queries [37, 38].

Caching is another powerful tool that complements ORM usage and can be implemented at various levels of an application. With caching, developers can store frequently accessed data in a cache (like Redis or Memcached), reducing the number of times the database needs to be queried. This significantly improves performance, especially for data that doesn't change often but is accessed repeatedly.

A common caching strategy is read-through caching, where the application first checks the cache before querying the database. If the data isn't in the cache (a cache miss), it's fetched from the database via the ORM and then stored in the cache for future use. This approach ensures that the database is not repeatedly queried for the same data, which can be expensive in terms of time and resources. Many ORMs integrate easily with caching systems, allowing developers to implement this pattern effectively. For instance, Django ORM can be combined with Redis to cache ORM query results, reducing the number of database queries.

While caching can dramatically improve read performance, it also introduces complexity in ensuring data consistency. Cache invalidation - when data in the cache becomes outdated or inconsistent with the database - needs careful handling. However, when implemented properly, caching can drastically reduce the load on the database and improve overall application performance [37, 38].

To conclude, the evolution of software architecture reflects a continuous pursuit of simplicity, scalability, and responsiveness. Object-Oriented Programming provided the foundation for building structured and maintainable systems, with Java leading the way. RESTful web services, built on HTTP, addressed the need for lightweight, stateless communication, replacing the complexity of earlier SOAP-based solutions. As enterprise systems grew, ORMs like Hibernate emerged to seamlessly connect object-oriented applications with relational databases, enabling cleaner code and efficient data handling. Together with frameworks like Spring Boot, they formed the backbone of scalable and modular enterprise software. However, the modern web demanded more: real-time updates, flexible data access, and optimized performance for increasingly dynamic user experiences. GraphQL rose to meet these needs, offering precision, efficiency, and built-in support for live data. In the end, there is no one-size-fits-all solution. REST remains ideal for transactional and resource-based systems, while GraphQL thrives in data-rich, interactive environments. The key lies in choosing the right tool for the problem - and understanding how each fit into the evolving landscape of software development.

References

- [1]
N. Singh, S. S. Chouhan, and K. Verma, "Object Oriented Programming: Concepts, Limitations and Application Trends," *2021 5th International Conference on Information Systems and Computer Networks (ISCON)*, Oct. 2021, doi: <https://doi.org/10.1109/iscon52037.2021.9702463>.
- [2]
A. S. C and Elugwu F, "Object Oriented Programming Approach: A Panacea for Effective Software Development," *Zenodo (CERN European Organization for Nuclear Research)*, Sep. 2022, doi: <https://doi.org/10.5281/zenodo.7321809>.
- [3]
H. Ali, M. Hamza, and T. Rashid, "Exploring Polymorphism: Flexibility and Code Reusability in Object- Oriented Programming," 2023. Available: https://passer.garmian.edu.krd/article_189667_b5fb20044fc0571614a42f29beeced39.pdf
- [4]
"A Research on Object Oriented Programming and Its Concepts," *International Journal of Advanced Trends in Computer Science and Engineering*, vol. 10, no. 2, pp. 746–749, Apr. 2021, doi: <https://doi.org/10.30534/ijatcse/2021/401022021>.
- [5]
J. Yousif, "The 'RESTful' Services: Are they 'RESTful' Yet? : - A Follow-up Study," *DIVA*, 2022. <https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1672919&dswid=6146>
- [6]
Abhijit Banubakode and Priya Chore, "Growth of Individualizing Web Services using APIs: REST and SOAP," *SAMRIDDHI : A Journal of Physical Sciences, Engineering and Technology*, vol. 14, no. Spl-2 issu, pp. 284–290, Jun. 2022, doi: <https://doi.org/10.18090/samriddhi.v14spli02.15>.
- [7]
Sharma Sakshi, "Assessment of Web Services based on SOAP and REST Principles using Different Metrics for Mobile Environment and Multimedia Conference," *Philpapers.org*, 2023. <https://philpapers.org/rec/SAKAOW> (accessed Apr. 03, 2025).
- [8]
A. Jatain, S. Bhaskar Bajaj, S. Chaudhary, and P. Batra, "Rest Web Services: An Elementary Learning," *Research Journal of Engineering and Technology*, pp. 75–78, Sep. 2021, doi: <https://doi.org/10.52711/2321-581x.2021.00012>.
- [9]
A. Soni and V. Ranga, "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, doi: <https://doi.org/10.35940/ijitee.i1107.0789s19>.
- [10]

A. Ehsan, M. A. M. E. Abuhaliqa, C. Catal, and D. Mishra, "RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions," *Applied Sciences*, vol. 12, no. 9, p. 4369, Apr. 2022, doi: <https://doi.org/10.3390/app12094369>.

[11]
D. Colley, C. Stanier, and M. Asaduzzaman, "The Impact of Object-Relational Mapping Frameworks on Relational Query Performance," *2018 International Conference on Computing, Electronics & Communications Engineering (iCCECE)*, Aug. 2018, doi: <https://doi.org/10.1109/iccecome.2018.8659222>.

[12]
C. Tudose and C. Odubășteanu, "Object-relational Mapping Using JPA, Hibernate and Spring Data JPA," *2021 23rd International Conference on Control Systems and Computer Science (CSCS)*, Bucharest, Romania, 2021, pp. 424-431, doi: 10.1109/CSCS52396.2021.00076.

[13]
Alexandre Bonvoisin, C. Quinton, and Romain Rouvoy, "Understanding the Performance-Energy Tradeoffs of Object-Relational Mapping Frameworks," vol. 7, pp. 626–636, Mar. 2024, doi: <https://doi.org/10.1109/saner60148.2024.00069>.

[14]
K. Dumas, "ORMs and Database Design," *Digital Commons@University of Nebraska - Lincoln*, 2024. <https://digitalcommons.unl.edu/honorstheses/734/> (accessed Apr. 04, 2025).

[15]
Catalin Tudose, *Java Persistence with Spring Data and Hibernate*. Simon and Schuster, 2023.
<https://hoclaptrinhdanang.com/downloads/pdf/spring/Java%20Persistence%20with%20Hibernate.pdf>

[16]
V. Sivakumar, T. Balachander, J.Ramu "Object Relational Mapping Framework Performance Impact," 2021, *Turkish Journal of Computer and Mathematics Education*, vol. 12, no. 7, pp. 2516–2519, Available: <https://www.proquest.com/docview/2623612701?pq-origsite=gscholar&fromopenview=true&sourcetype=Scholarly%20Journals>

[17]
Kuldeep Hule and Rekha Ranawat, "Analysis of Different ORM Tools for Data Access Object Tier Generation: A Brief Study," *International journal of membrane science and technology*, vol. 10, no. 1, pp. 1277–1291, Oct. 2023, doi: <https://doi.org/10.15379/ijmst.v10i1.2842>.

[18]
I. Madurapperuma, Shafana, and M. Sabani, "State-of-Art Frameworks for Front-end and Back-end Web Development." Available: <http://ir.lib.seu.ac.lk/bitstream/123456789/6339/3/62-67.pdf>

[19]
X. Luan, "IMPLEMENTATION AND ANALYSIS OF SOFTWARE DEVELOPMENT IN SPRING BOOT," 2021. Available: <https://scholarworks.calstate.edu/downloads/zg64ts132>

[20]

S. Sharma, *Modern API Development with Spring and Spring Boot*. Packt Publishing Ltd, 2021.

[21]

Y. Jani, "Spring Boot for Microservices: Patterns, Challenges, and Best Practices," *European Journal of Advances in Engineering and Technology*, vol. 2020, no. 7, pp. 73–78, Accessed: Apr. 06, 2025. [Online]. Available: <https://ejaet.com/PDF/7-7/EJAET-7-7-73-78.pdf>

[22]

V. Desai, "Microservices: Architecture and Technologies," *International Journal for Research in Applied Science and Engineering Technology*, vol. 8, no. 10, pp. 679–686, Oct. 2020, doi: <https://doi.org/10.22214/ijraset.2020.31979>.

[23]

B. M. Abdelhafiz, "Distributed Database Using Sharding Database Architecture," *IEEE Xplore*, Dec. 01, 2020. <https://ieeexplore.ieee.org/abstract/document/9411547/>.

[24]

J. George, "Optimizing Database Queries for AngularJS and Java-Based Web Applications," Sep. 07, 2024. https://www.researchgate.net/publication/389635985_Optimizing_Database_Queries_for_AngularJS_and_Java-Based_Web_Applications

[25]

N. H. Ahmad, "Architectural Patterns and Challenges in Spring Boot for Microservices: Evaluating Automation Strategies for Scaling, Monitoring, and Deployment in Complex Software Ecosystems," *International Journal of Information Technologies and Artificial Intelligence*, vol. 8, no. 8, pp. 1–9, 2024, Accessed: Apr. 07, 2025. [Online]. Available: <https://norislab.com/index.php/IJITAI/article/view/88>

[26]

D. V. Kornienko, S. V. Mishina, and M. O. Melnikov, "The Single Page Application architecture when developing secure Web services," *Journal of Physics: Conference Series*, vol. 2091, no. 1, p. 012065, Nov. 2021, doi: <https://doi.org/10.1088/1742-6596/2091/1/012065>.

[27]

Kuldeep Hule and Rekha Ranawat, "Analysis of Different ORM Tools for Data Access Object Tier Generation: A Brief Study," *International journal of membrane science and technology*, vol. 10, no. 1, pp. 1277–1291, Oct. 2023, doi: <https://doi.org/10.15379/ijmst.v10i1.2842>.

[28]

U. Yang, C. Yan, P. Subramaniam, S. Lu, and A. Cheung, "How not to structure your database-backed web applications: A study of performance bugs in the wild," in *Proc. 2018 IEEE/ACM 40th Int. Conf. Software Engineering (ICSE)*, 2018, pp. 800–810.

[29]

W. Liu, S. Mondal, and T.-H. P. Chen, "An Empirical Study on the Characteristics of Database Access Bugs in Java Applications," *ACM Transactions on Software Engineering and Methodology*, Jun. 2024, doi: <https://doi.org/10.1145/3672449>.

- [30]
T. Sotiropoulos, S. Chaliasos, V. Atlidakis, D. Mitropoulos, and D. Spinellis, "Data-Oriented Differential Testing of Object-Relational Mapping Systems," *IEEE Xplore*, May 01, 2021.
<https://ieeexplore.ieee.org/abstract/document/9401963>
- [31]
M. Gorodnichev, M. Moseva, K. Poly, K. Dzhabrailov, and R. Gematudinov, "Exploring object-relational mapping (ORM) systems and how to effectively program a data access model," *PalArch's J. Archaeol. Egypt / Egyptology*, vol. 17, no. 3, pp. 615-627, Nov. 2020. [Online]. Available: <https://doi.org/10.48080/jae.v17i3.141>.
- [32]
B. H. V. Wahono, Asfihani, and U. L. Yuhana, "Comparative Analysis of Database Access Method Performance on Containerized Application," *2024 IEEE International Conference on Artificial Intelligence and Mechatronics Systems (AIMS)*, pp. 1–6, Feb. 2024, doi: <https://doi.org/10.1109/aims61812.2024.10513113>.
- [33]
G. Brito and M. T. Valente, "REST vs GraphQL: A Controlled Experiment," 2020 IEEE International Conference on Software Architecture (ICSA), Salvador, Brazil, 2020, pp. 81-91, doi: 10.1109/ICSA47634.2020.00016. keywords: {Web services;Facebook;Database languages;Uniform resource locators;Servers;Task analysis;Standards;GraphQL;REST;Web APIs},
- [34]
S. L. Vadlamani, B. Emdon, J. Arts, and O. Baysal, "Can GraphQL Replace REST? A Study of Their Efficiency and Viability," *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, Jun. 2021, doi: <https://doi.org/10.1109/ser-ip52554.2021.00009>.
- [35]
Aleksējs Sergejevs and Sergejs Kodors, "COMPOSITION AND INHERITANCE MODEL COMPARISON," *HUMAN ENVIRONMENT TECHNOLOGIES Proceedings of the Students International Scientific and Practical Conference*, no. 23, pp. 64–71, Apr. 2019, doi: <https://doi.org/10.17770/het2019.23.4394>.
- [36]
S. Deng *et al.*, "Cloud-Native Computing: A Survey From the Perspective of Services," *Proceedings of the IEEE*, vol. 112, no. 1, pp. 12–46, Jan. 2024, doi: <https://doi.org/10.1109/JPROC.2024.3353855>.
- [37]
Y. Jani, "Optimizing Database Performance for Large-Scale Enterprise Applications," *International Journal of Science and Research (IJSR)*, vol. 11, no. 10, pp. 1394–1396, Oct. 2022, doi: <https://doi.org/10.21275/sr24716121211>.
- [38]
S. Youssef, "Optimizing Database Performance for Large-Scale Enterprise Applications: A Comprehensive Study on Techniques, Challenges, and the Integration of SQL and NoSQL Databases in Modern Data Architectures," vol. 7, no. 1, pp. 81–92, Aug. 2023,

[39]

J. Griffin, "DTOs, Entities, and Value Objects," *Apress eBooks*, pp. 365–408, Oct. 2020, doi: https://doi.org/10.1007/978-1-4842-6023-4_12.

[40]

D. P. Mishra, K. K. Rout, and S. R. Salkuti, "Modern tools and current trends in web-development," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 24, no. 2, p. 978, Nov. 2021, doi: <https://doi.org/10.11591/ijeecs.v24.i2.pp978-985>.

FINAL GRADE

GENERAL COMMENTS

Avery good piece of work, with a nice use of sources.

Change your spell checker to UK English for future work.

86/100

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

PAGE 16

