

Object-Oriented Programming in Modern Web Service Design:

Principles, Frameworks, and Future Trends

Bradley Watkins

May 2025

Abstract

This essay explores the foundational role of Object-Oriented Programming (OOP) principles in modern web service development, focusing on their application in frameworks like Spring Boot. Encapsulation, cohesion, loose coupling, and modularity are examined as key indicators of reliability, maintainability, and scalability. The discussion extends to architectural styles, comparing SOAP and REST, and highlights the dominance of RESTful architectures due to their simplicity and efficiency [1, 2]. Object-Relational Mapping (ORM) strategies are evaluated, as well as their integration with the repository pattern to simplify database interactions [3, 4]. Emerging paradigms such as GraphQL and cloud-first approaches are explored, offering insights into the future of web service design. By integrating OOP principles, leveraging frameworks, and adapting to evolving technologies, developers can build robust systems that meet the demands of modern software engineering.

Contents

1	Introduction	3
2	Object-Oriented Programming	3
2.1	Encapsulation and Information Hiding	3
2.2	Cohesion and Modularity	4
2.3	Coupling	4
2.4	Inheritance and Polymorphism	5
2.5	Quality Metrics and Design Principles	6
3	Web Services	6
3.1	SOAP	7
3.2	REST	7
3.3	REST vs SOAP	8
3.4	Web Service Architectures	9
3.5	Quality Metrics and Design Principles	10
3.6	Security Challenges	11
4	Spring Boot Design Principles and Quality Metrics	11
4.1	Dependency Injection	12
4.2	Object-oriented Principles	12
4.3	Object Relational Mapping and the Repository Pattern	13
4.4	Advantages and Limitations of Object Relational Mapping	14
4.5	Best Practices in Spring Boot Applications	14
5	Challenges and Future Perspectives	15
6	Conclusion	16

1 Introduction

Web services have become a cornerstone of modern software engineering, enabling seamless machine-to-machine communication across the internet. Their ubiquitous nature can be attributed to their interoperability and the vast ecosystem of tools available for almost all modern programming languages.

Object-Oriented Programming (OOP) principles such as encapsulation, inheritance, polymorphism, and abstraction form the foundation for designing reliable, maintainable, and high-quality software. These principles are critical in developing web services that are modular and reusable [5, 6, 7].

This essay explores the core design principles and quality metrics in web service development, examining how frameworks like Spring Boot leverage OOP principles to create scalable and efficient systems. It also compares architectural styles like SOAP and REST, delves into the advantages and limitations of Object-Relational Mapping (ORM), and discusses emerging paradigms such as GraphQL [1, 2, 8]. The challenges of transitioning architectures and adopting cloud-first approaches are also considered, providing a holistic perspective on the future of web service development [9, 10].

2 Object-Oriented Programming

Object-Oriented Programming (OOP) is a paradigm that provides the foundational principles for designing reliable and high-quality software. The principles of encapsulation, inheritance, polymorphism, and abstraction form the core of this paradigm and enable the creation of modular, reusable, and maintainable software. These principles have a direct impact on the functionality, reliability, and overall quality of software systems, as outlined below.

2.1 Encapsulation and Information Hiding

Encapsulation refers to the bundling of data and data manipulation methods into a single unit. A class typically serves as this single unit, abstracting internal details from the end user. Information hiding is the abstraction of implementation details from external users, where a level of trust is established that the class will conform to its public interface. By designing with encapsulation and information hiding in mind, the integrity of internal data is protected and interactions with that data are tightly controlled. This also allows for changes to the internal implementation details without modifications to any dependent classes [5, 6].

Encapsulation enhances reliability and maintainability by ensuring that no unexpected modifications of internal state take place. Instead, the encapsulated unit is able to provide methods as a means of operating

on the data, and read-only views of the data can be provided. Encapsulated units should typically avoid using global or static state and should not produce side effects. Following these design principles allows classes to be tested as independent units, where any dependencies can be stubbed and realistic runtime scenarios can be observed [6, 11].

2.2 Cohesion and Modularity

Cohesion is the degree to which data and methods within a class are related and work together to achieve a single function. Within a highly cohesive class, there is an expectation that internal methods will make liberal usage of local variables. Each class should address a single responsibility or be divided into logical sub-classes to accommodate multiple responsibilities. This allows classes to align with key object-oriented principles such as the single responsibility principle and separation of concerns [12, 13].

Beyond their internal implementation and usage of local variables, a desirable trait for cohesive modules is that they map to a single semantic idea or concept. This allows for self-descriptive classes that can be reasoned and understood more easily. Reasoning classes in this way can also help to reduce overlap in functionality within the system and find logical groupings of semantic modules [14, 15].

High cohesion is a key attribute in high-quality software. Cohesive modules are typically easier to understand, maintain, reuse, and test. Due to its importance, cohesion metrics are a highly researched area, and many automated tools exist for quantifying cohesion within software modules [16, 17, 18]. It is not so simple to quantify semantic cohesion within a software system, which can influence how measurements should be interpreted; however, there is research being done in this area [14].

2.3 Coupling

Coupling is the degree to which classes, or units, within the software system depend on one another. Loosely coupled units with fewer dependencies benefit by minimising the impact that changes within the system can produce. This also allows for more modular and reusable components within the system, as with fewer dependencies, the component can be useful in a greater number of contexts. While some level of coupling is inevitable for a software system to be useful, excessive coupling can negatively impact software quality [19, 17].

Many object-oriented principles support the development of loosely coupled systems. Dependency inversion and inversion of control are two such concepts that are deeply ingrained in many modern frameworks. Dependency inversion states that software modules should rely on abstractions or interfaces and not on low-level concrete implementations. Inversion of control states that control flow should be handled externally. These

principles typically take shape via dependency injection, where the creation and supply of dependencies are abstracted and occur at runtime. This allows for faster development and ease of testing through stubs and mocks [20, 21].

Another principle that supports loose coupling is the Law of Demeter [22]. The Law of Demeter says that modules should depend only on direct dependencies and nothing else. This principle prohibits the extraction and usage of classes that are not either stored as local variables or have been passed into the method in use [20]. This ensures maximum modularity and minimises side effects and unexpected behaviour within the system.

Alongside cohesion, the level of coupling within a software system is an important metric. Many measurement techniques exist which work to help developers make logical and well-educated decisions within their systems [17, 23].

2.4 Inheritance and Polymorphism

Inheritance and polymorphism are the true foundational principles that object-oriented programming derives from. Inheritance is a means of grouping related functionality into a hierarchical system of parent-child relationships. The parent class, or superclass, typically provides a less specific default behaviour which can then be extended and overridden for more specific implementation in child, or derived, classes. Polymorphism allows derived classes to be used as if they are of the same type as their parent class. This means that dependent classes can rely on the more abstract parent class while still allowing dynamic and specialised behaviour from the derived classes [5, 24].

Parent classes in inheritance hierarchies need not be concrete implementations. If no default behaviour can be provided, then an abstract class can be used that places the responsibility for implementation on the deriving classes. Abstract classes can not be directly instantiated [5].

In the case that there is no logical semantic grouping, but unrelated classes must still share method signatures, then interfaces can be used. An interface provides method definitions with no internal state or implementation. All instantiable implementors of the interface must provide specific functionality for these methods, similar to when deriving abstract methods [24].

Both abstract data types and interfaces can be used as polymorphic types in place of concrete data types. This allows for the grouping and management of these types, with the only stipulation being that you interface only via the method definitions provided by the polymorphic API. Method definitions that exist in derived or implementing classes will not be available for use in this case. In the case that no specific API is required when managing or collecting other classes, then generics or parameterised types can be provided for further

abstraction and reusability [5].

2.5 Quality Metrics and Design Principles

High-quality object-oriented software is typically characterised by reliability, maintainability, and usability [25]. As mentioned, measurable quality metrics such as cohesion and coupling levels are highly desirable and researched [16, 17, 26, 23, 21, 18]. Many other areas of object-oriented design are measurable and help support the production of high-quality software, such as depth of inheritance [13].

These measures typically take into account coarse-grained design principles that are abstract and don't provide much guidance at the design stage. Examples of coarse-grained principles include high cohesion, low coupling, low complexity and encapsulation. These are all aspirational qualities of good software and should be sought after [16, 15, 27], but it is not always clear how to achieve these qualities.

By considering fine-grained design principles at the design stage, engineers get a more concrete view of how to move forward and model their software for success. Fine-grained principles include single responsibility, separation of concerns, information hiding, the open-closed principle, and don't repeat yourself (DRY). Taking a more holistic view of software development allows developers to take a step back and consider how to improve their solution, which is not always possible with a single quality metric [13].

The production of high-quality software is essential for success in the long term. By producing modular software that is reusable, reliable, and maintainable, you not only reduce the burden on engineers but also on business. High-quality software is cheaper and is able to achieve greater agility, allowing businesses to respond to changing requirements and capitalise on an ever-changing market [12, 23, 27].

3 Web Services

Web services are a cornerstone of modern software engineering, enabling machine-to-machine communication over the internet. They follow a variety of architectural styles, such as SOAP and REST, with RESTful architectures being among the most widely adopted due to their simplicity, scalability, and interoperability. This section explores the principles of web services, the role of RESTful architectures, and their relevance to contemporary software development.

3.1 SOAP

Simple Object Access Protocol (SOAP) is a messaging protocol that allows communication between machines over the internet. SOAP is comprised of a provider, a requestor, and a registry. In this case the provider is the service, and the requestor is the client making the request. When making a request, the client will first query the registry for available endpoints, making use of the description provided to produce a request [1]. The provider would maintain the state of its exchange with a requestor, influencing behaviour and next actions.

SOAP requests are composed of an optional header and a mandatory body that are contained in the message envelope. The header may contain information such as the date of the message, authentication data and other useful metadata. The body bears the message payload in XML format [28].

There are many useful features provided out of the box when using SOAP. These include built-in fault tolerance via message retries, security enhancements, and extensive metadata that allows message tracing [29, 30]. One of SOAP's issues, however, is the verbosity and deep technical knowledge required to make use of it. Payloads tend to be large and difficult to parse, and managing the stateful nature of SOAP can be incredibly complex [31].

SOAP was an industry standard for many years but has seen a decrease in usage in recent years, making way for the simplicity of RESTful architectures.

3.2 REST

Representation State Transfer (REST) is an architectural paradigm that leverages HTTP to provide a scalable and performant framework for distributed systems. Restful architectures rely on Uniform Resource Identifiers (URIs) for resource detection and HTTP verbs to describe actions such as GET, POST, PUT, and DELETE [2]. One of the fundamental principles of REST is statelessness, which ensures that each request is independent and contains all necessary information for the completion of the request. Caching is another key feature, improving performance by reducing redundant server calls [29].

A key philosophy in designing RESTful architectures is that available operations are resource-centric, allowing users to retrieve or modify persistent data. The mapping of HTTP verbs to logical URIs is determined by the resource being operated on. This allows URI overloading, where the correct endpoint is determined based on the HTTP verb provided. This is an important principle for resource addressability; however, many services still provide operation-specific information within the URI, leading to a remote procedure call (RPC)-like service structure [2].

RESTful architectures are designed to be simple yet robust. They tend to be relatively small, offering between

2 and 50 operations, although outliers like Facebook and eBay provide hundreds of operations in their services. Though Restful architectures are the industry standard, few practical implementations fully comply with REST standards. Features such as Hypermedia as the Engine of Application State (HATEOAS), providing authorised and valid next actions to clients, are extremely rare in industry. When HTTP is in use, it is not unusual to see services ignoring headers and opting to provide the return type and other such details in the request URI. These are clear violations of resource addressability, but the industry has self-organised and determined which standards are important and which can be forgotten. Where REST standards are known to have an impact on understandability and usage, there is a much higher level of conformance [2, 32].

Perhaps in part because of its simplicity, REST lacks a built-in security model, relying instead on HTTPS and TLS for secure communication [30]. When designing Restful architectures, it is important that engineers are aware of common pitfalls that can impact the usability and security of the service [32]. OAuth2 is becoming a ubiquitous solution for authentication, authorisation, and security when implementing RESTful architectures. OAuth2 provides the service with a means of segregating user actions by assigning roles. These roles are typically managed by a third party, such as KeyCloak, that will validate the user on every call. This validation makes use of a token that contains authentication details, meaning plain text usernames and passwords do not need to be transmitted to the service. This is an effective means of protecting service actions and preventing common vulnerabilities such as cross-site request forgery, cross-site scripting, IP spoofing and denial of service [33].

3.3 REST vs SOAP

REST has become ubiquitous within web service development, with most greenfield projects opting to use it over other available protocols. This has been a monumental shift and may largely be attributed to the rise of mobile and embedded devices. Hardware-constrained devices, such as those in the internet of things, require speed and fault tolerance. REST also benefits from interoperability with many transport protocols, which is important when working with devices where HTTP has too much overhead [34].

Comparing SOAP and REST performance shows RESTful architectures benefit from a ten times performance boost in mobile environments. Like-for-like requests have also been shown to carry a payload two to three times smaller when using REST over SOAP [1]. This can lead to very real cost savings not only for the business transporting the data but also for the user who may be on a data-limited mobile network.

It may seem like SOAP is a thing of the past with RESTful architectures taking over across the industry, but this is not the case. There are many use cases for SOAP, and many of the standardised security protocols that SOAP benefits from have not yet made their way into the RESTful sphere [30, 29]. REST also struggles

to achieve the level of discoverability achieved by SOAP due to the lack of a registry and non-conformance with HATEOAS in Restful architectures [2]. This makes pathfinding in machine-to-machine interactions with no human interference difficult in RESTful architectures; however, there is research being done in this area [35].

In the case of existing SOAP providers, it is not always a straightforward process to transition to a RESTful architecture. There is not always a logical mapping to HTTP verbs, and much of the discoverability afforded by SOAP is lost in the process. The semantic web, providing service endpoints with semantic machine-readable information, may help but is still an active field of research [36].

3.4 Web Service Architectures

There are many web service architectures available, with each providing solutions to different problem domains. The most common of these are the monolithic architecture and the microservices architecture.

Monolithic architectures are what is generally considered a typical web service. All, or most, of the business logic will be contained within a single application that contains multiple services. These services will each deal with a single responsibility and, in the case of REST architectures, will typically concern themselves with a single resource [32].

Microservices architectures are a development approach where a single application is developed as a suite of small services [37]. Each of these web services will exist within its own bounded context and will be independently deployable. They will communicate with one another via web protocols, typically REST, each dealing with a single business concern within the system [7].

The choice of web service architecture is a nuanced and difficult problem domain. Monolithic architectures can be much easier to reason and understand than microservices due to the emergent complexity that exists when dealing with many distributed services [37]. They are also generally faster to develop for simpler applications and are easier to test, as dependencies can be easily stubbed and mocked. Monolithic architectures can also benefit from vertical scaling, increasing hardware performance, although this is limited and has diminishing returns [7].

Microservices, on the other hand, despite being far more complex, can work well in large organisations, as each development team can benefit from owning a service. This can speed up delivery where fewer communication channels are required, preventing blocking changes and conflicts within the code base [38]. Microservices also benefit from horizontal scaling, where more deployed service instances are produced to deal with increasing load [39]. Scaling can be targeted at the specific services that are dealing with increased load due to the shared-nothing approach of this architecture style [40]. This provides fault tolerance, as any one service failing can

quickly be replaced with a new instance or its load can be sent to an instance already in use. Microservices have been successfully deployed in major applications such as Netflix to handle billions of concurrent requests daily [7, 37].

There is also nuance in architectural choice in the costs accrued, especially in the cloud. Beyond development time, there are also compute considerations due to pricing models from services such as Amazon Web Services (AWS). Choice of cloud provider is important for minimising costs, as each provider will offer products that benefit in different contexts. For example, AWS Lambda, a serverless solution for building microservice architectures, has been shown to reduce costs by more than 50% in the cloud when compared to monolithic solutions. This is not true in all cases, however, with monoliths actually improving development speed and reducing cloud costs for smaller user bases [10].

Migration from a monolithic architecture to a distributed microservices architecture is exceptionally difficult. In many cases, it takes highly skilled engineers and a total business shift into the DevOps development style to achieve. Where it has been successful, a staggered approach has been recommended, where parts of the system are strangled and moved to the cloud incrementally, minimising risk and impact to the user base [9]. Because of the difficulty and specialised skill required, many monolithic services in use opt to simply move to the cloud and reap the benefits there, or remain unchanged.

3.5 Quality Metrics and Design Principles

Web services, as software systems, benefit from many of the same quality metrics that can be applied to plain object-oriented systems. These quality measures are reliability, reusability, maintainability, loose coupling, etc. In web services these qualities are measured at the service level [38].

Loose coupling in particular is a critical design principle, especially in microservices architectures that are loosely coupled by design. By keeping services loosely coupled from one another, they benefit from increased scalability and fault tolerance. This also allows changes to be made to services without cascading impact throughout the system [38, 7].

In contrast, highly coupled services rely heavily on one another, making the system fragile and difficult to maintain. For example, SOAP-based architectures often involve tighter coupling due to their use of service registries and complex metadata [1]. RESTful architectures, with their emphasis on statelessness and resource addressability, are better suited for achieving loose coupling [2].

Another key principle for web services is separation of concerns. By maintaining logically cohesive services you can create modules that are easier to maintain, test, and reason about [7]. This is especially important

in REST architectures due to the difficulties in service discoverability. Having logical service boundaries and contexts allows greater ease of use of the system [2].

3.6 Security Challenges

Cybersecurity is a significant concern in web services, as they are exposed to various attack vectors. SQL injection attacks have largely been mitigated by modern frameworks, but phishing and Distributed Denial of Service (DDoS) attacks remain prevalent. DDoS attacks are particularly impactful in cloud environments, where costs are incurred per invocation or compute time [41]. Zero trust architectures are increasingly adopted to mitigate security risks, emphasising the principle of "never trust, always verify" [42].

Engineers are encouraged to use built-in framework validation mechanisms, parameterised database queries, and logging to enhance security [31]. Logging is especially important in microservice architectures due to the difficulty in debugging and tracking errors in a distributed environment [37]. These logs can incur extra costs, however, so should be carefully planned for maximum effect.

4 Spring Boot Design Principles and Quality Metrics

The choice of framework can significantly impact the performance, cost, and sustainability of web services. Java Spring Boot is one of the most widely adopted frameworks for Restful architectures, known for its reliability and ease of use [37, 43]. However, Java-based frameworks generally exhibit higher energy consumption compared to alternatives like .NET, which perform better in low-power scenarios [7]. For JavaScript, frameworks like Express, Nest, and Fastify are commonly used, while Python developers often rely on Django, Flask, or FastAPI [43].

Case studies show that different frameworks excel in different scenarios. Express and Node.js scale well, while Spring Boot is more reliable under intensive workloads. Developers should consider non-functional requirements such as power efficiency, cost, and scalability when selecting a framework, rather than relying solely on personal preference [43].

Java Spring Boot is a high-quality backend framework that has been shown to be resilient and reliable in high-stress scenarios. It is the most widely used backend framework for Java, with few other languages exhibiting the same level of ubiquity in framework choice [43]. This has led to a massive ecosystem of support within the Spring Boot environment, furthering the capabilities of the framework. The following sections will discuss specific object-oriented principles within the Spring Boot framework and their relevance in modern software engineering.

4.1 Dependency Injection

Spring Boot is an opinionated framework that claims to provide rapid delivery with minimal fuss. Many third-party modules exist to reduce the boilerplate required to get started with Spring Boot. Spring Boot relies on two key principles that have briefly been touched upon. These are inversion of control (IoC) and dependency injection (DI) [44].

IoC provides control flow of the application to the Spring Boot framework, allowing the framework to determine required dependencies. DI is then used to provide these dependencies throughout the application. Data annotations such as **@Autowired** provide hints to the framework and allow the correct dependency to be injected at runtime [45]. For example, consider a **UserService** that depends on a **UserRepository** to access user data. Using **@Autowired**, Spring Boot can automatically provide an instance of **UserRepository** when creating a **UserService** without explicit injection required.

By leveraging the power of DI provided by Spring Boot out of the box, developers reap many benefits. This is especially true if dependency inversion is employed, relying on abstractions and interfaces over concretions. Benefits range from increased ease of testing, loose coupling that minimises the ripple effect of change, and increased agility allowing rapid response to changing requirements through component swapping [46].

Systems that employ DI and the factory pattern, as Spring Boot does, have been shown to exhibit a 10% increase in testability [47]. DI has also been shown to enable faster development, allowing teams to work independently by employing stubs for incomplete modules [48].

While DI offers numerous benefits, it is not a universal solution and can introduce pitfalls if misused. Developers should avoid common anti-patterns, such as expecting default constructors and providing default implementations in method signatures, as these violate dependency inversion principles. Instead, dependencies should be injected via interfaces to maintain flexibility and minimise coupling [48].

4.2 Object-oriented Principles

Spring Boot's existence as a Java framework necessitates its deep integration with object-oriented design principles. Services and providers within any given application present themselves as classes, and all of the benefits of object-oriented principles are in effect. These include abstraction, encapsulation, inheritance, and polymorphism. Dependency injection (DI) relies on polymorphism to provide just-in-time dependencies to modules within the system [20, 46]. Applications making use of Spring Boot will typically provide abstraction and encapsulation in the form of layered architecture.

Making use of application layering provides separation of concerns and increases testability. Typical layers

will include the controller layer, the service layer, and the repository layer [49]:

The controller layer is the exposed public API of the web service application. This is where URI and HTTP verb mappings will be defined. The controller layer should typically only deal with input and output translation, such as JSON or XML into Java objects and vice versa. Returned types and status codes will also be determined here. For example, a controller method may receive a JSON payload representing a new user, translate this into a **User** object, and then pass this down into the service layer. Based on the response from the service layer, the controller layer will return a relevant HTTP status code indicating success or failure.

The service layer will receive data transfer objects from the controller layer in order to perform business logic, such as data validation. This will then perform necessary data transformation to prepare data to be passed into the repository layer. For example, the service layer could validate the format of an email address or ensure that a username is unique before allowing the repository layer to persist the new user.

Finally, the repository layer will handle interaction with the database. This is where transaction logic should be, handling thread-safe retrieval and updating of data from the database layer. Spring Data, a plugin for Spring Boot services, makes use of the repository pattern to interact with the database. The repository pattern leverages object-relational mapping, abstracting database interaction and reducing the need for low-level database code such as SQL [3].

4.3 Object Relational Mapping and the Repository Pattern

Object-Relational Mapping (ORM) is a programming technique that facilitates interaction with a database by abstracting database tables and rows into objects and classes. ORM tools such as Hibernate allow developers to perform database operations using object-oriented paradigms, reducing the need to write raw SQL queries [3]. They also reduce boilerplate code for generic CRUD operations and improve code readability through the use of design patterns such as the Data Mapper pattern [4]. This abstraction simplifies database interactions, increases portability, and enhances maintainability by allowing the underlying database to be changed with minimal impact on the application code [4].

The repository pattern, as implemented by Spring Data, abstracts database interactions and simplifies development through naming conventions and built-in methods. By leveraging this pattern, developers can build scalable and maintainable applications while reducing boilerplate code [50].

Data access objects are Plain Old Java Objects (POJOs) that are mapped to relational entities using Spring Data annotations [51]. This can be taken even further in simple CRUD services, where the repository layer can be defined as a controller, providing public URI mappings to database interactions. This massively reduces the

amount of code required, allowing for rapid development [3].

By abstracting database interactions and adhering to object-oriented principles, ORM and the repository pattern enhance maintainability and scalability in modern web applications.

4.4 Advantages and Limitations of Object Relational Mapping

ORM simplifies and accelerates development by abstracting database operations and providing caching capabilities which reduces the need for specialised database knowledge. This makes ORM a useful tool for large-scale applications that require maintainable and scalable database interactions. ORM frameworks also provide built in security against common threats such as SQL injection [4].

ORM frameworks are not without limitations, however. They can suffer from performance antipatterns, such as inefficient loops and eager loading, which increase database load and reduce efficiency. Additionally, poorly designed relational models, often a result of misuse of ORM, can lead to tightly coupled service models and database schemas that are difficult to change [40].

Active Record antipatterns are another concern in ORM-based applications. These occur when CRUD operations are exposed to derived classes of DAOs, reducing cohesion and encouraging anaemic data classes that lack business logic [40]. In large-scale systems, fine-tuned SQL queries may outperform ORM-generated queries in specific scenarios, necessitating careful consideration of when and how to use ORM [3, 52].

4.5 Best Practices in Spring Boot Applications

Designing scalable and efficient web services with Spring Boot requires adherence to best practices. Services should be loosely coupled and highly cohesive to ensure fault tolerance and maintainability [38]. Spring Boot's support for modular architectures allows developers to build microservices that can be independently deployed and scaled, making it suitable for cloud-based applications [7].

Authentication and authorisation mechanisms such as OAuth2 can be integrated with Spring Boot to protect services from vulnerabilities like Cross-Site Request Forgery (CSRF) and Distributed Denial of Service (DDoS) attacks [33]. Additionally, developers should make use of built-in validation mechanisms and avoid directly controlling dependencies to minimise risks [48]. When using ORM with Spring Boot, developers should carefully tune database queries and avoid performance antipatterns to maximise efficiency [40]. If absolute performance and complex queries are required at the database level, then developers may consider incorporating low-level static SQL queries. In these cases careful use of parameterised queries should be implemented to alleviate common database vulnerabilities [31]. This is generally not recommended, however, as it is likely to tightly couple

the application to the underlying database, introducing tedious error-prone code that reduces maintainability and portability [3].

Making use of Spring Boot’s opinionated design principles when possible will likely lead to the greatest success. Especially in the case of migration from a monolithic architecture to microservices. Spring Boot has been proven to increase the rate of success during such migrations [9]. This is in part due to Spring’s built-in cloud configuration and other cloud features that developers can lean on. Leveraging existing tools is crucial for web security. By writing the least code to achieve the desired outcome, developers also avoid introducing bugs and vulnerabilities into their systems [53, 37, 3].

5 Challenges and Future Perspectives

One key challenge in web development is overreliance on tools without consideration for the application context. Taking a cloud-first approach, as the industry is leaning towards, will mean truly considering the choice of tool in use. Frameworks such as Spring Boot provide great fault tolerance under load but may incur greater costs compared to more sustainable or performant solutions [43]. In some cases, teams can move away from frameworks and decrease costs by creating a serverless solution [10].

An interesting paradigm that is seeing greater uptake now is GraphQL. GraphQL is an alternative transport protocol to REST and SOAP, introducing handling of complex data types, type schemas, and flexibility in interactions. Clients are able to select and deselect desired fields of the resources they query. Resolvers then manage transmitting only the desired data to the client, potentially reducing bandwidth usage [8].

In practice, GraphQL has not yet been shown to outperform REST in its intended contexts. GraphQL has been shown to achieve greater efficiency and resource utilisation, which may introduce cost savings in the cloud [8]. Unfortunately, REST has thus far been shown to achieve greater performance than GraphQL, with GraphQL queries often suffering from inefficient data retrieval due to eager loading and caching difficulties [31]. REST has also been shown to provide greater data throughput in high-demand scenarios [8].

Despite these challenges, GraphQL continues to be an interesting area of research in web development. This young technology shows much promise. Spring Boot’s flexibility allows integration with emerging technologies like GraphQL through plugins such as `graphql-spring-boot-starter`, enabling developers to explore hybrid solutions that combine REST and GraphQL. As more frameworks and plugins begin to integrate GraphQL natively and developers begin to make use of it, it is sure to evolve and improve rapidly.

6 Conclusion

The development of high-quality web services relies on a strong foundation of Object-Oriented Programming (OOP) principles and careful architectural decisions. Encapsulation, cohesion, loose coupling, and modularity ensure that software systems are maintainable, reusable, and adaptable to changing requirements [5, 6]. Quality metrics, such as cohesion and coupling, provide measurable insights into the reliability and scalability of these systems, while design principles like separation of concerns and dependency inversion promote robust implementations [16, 17].

Frameworks like Spring Boot exemplify the practical application of OOP principles in web service development. By leveraging features such as dependency injection, layered architecture, and object-relational mapping, Spring Boot enables developers to build scalable, secure, and efficient systems [48, 3]. Meanwhile, the comparison between SOAP and REST highlights the evolution of web services, with RESTful architectures emerging as the industry standard due to their simplicity and scalability [1, 2].

Looking ahead, emerging paradigms like GraphQL and cloud-first approaches continue to challenge the status quo, offering new possibilities for efficiency and flexibility. While REST remains dominant, the growing adoption of GraphQL demonstrates the industry’s interest in highly customisable data retrieval mechanisms [8]. As these technologies evolve, developers must balance innovation with practicality, ensuring that tools and frameworks are chosen based on the specific needs of their applications [43, 10].

Ultimately, the successful development of web services depends on a holistic approach that integrates foundational principles, practical tools, and forward-thinking strategies. By adhering to best practices and embracing emerging trends, developers can build systems that are not only reliable and maintainable but also adaptable to the ever-changing landscape of modern software engineering.

References

- [1] F. Belqasmi, J. Singh, S. Y. Bani Melhem, and R. H. Glitho, “Soap-based vs. restful web services: A case study for multimedia conferencing,” *IEEE Internet Computing*, vol. 16, no. 4, pp. 54–63, Jul. 2012.
- [2] A. Neumann, N. Laranjeiro, and J. Bernardino, “An analysis of public rest web service apis,” *IEEE Transactions on Services Computing*, vol. 14, no. 4, pp. 957–970, Jul. 2021.
- [3] 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,” *Applied Sciences*, vol. 14, no. 7, p. 2743, Mar. 2024.
- [4] G. Vial, “Lessons in persisting object data using object-relational mapping,” *IEEE Software*, vol. 36, no. 6, pp. 43–52, Nov. 2019.
 - [5] B. Stroustrup, “What is object-oriented programming?” *IEEE Software*, vol. 5, no. 3, pp. 10–20, May 1988.
 - [6] M. Fowler, “Data access routines,” *IEEE Software*, vol. 20, no. 6, pp. 96–98, Nov. 2003.
 - [7] G. Blinowski, A. Ojdowska, and A. Przybylek, “Monolithic vs. microservice architecture: A performance and scalability evaluation,” *IEEE Access*, vol. 10, pp. 20 357–20 374, 2022.
 - [8] 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, p. 138, Oct. 2021.
 - [9] A. Balalaie, A. Heydarnoori, and P. Jamshidi, “Microservices architecture enables devops: Migration to a cloud-native architecture,” *IEEE Software*, vol. 33, no. 3, pp. 42–52, May 2016.
 - [10] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, “Cost comparison of running web applications in the cloud using monolithic, microservice, and aws lambda architectures,” *Service Oriented Computing and Applications*, vol. 11, no. 2, pp. 233–247, Apr. 2017.
 - [11] R. Dromey, “A model for software product quality,” *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 146–162, 1995.
 - [12] Shipra, “Cohesion measurements between variables and methods using component-based software systems,” *International Journal of System Assurance Engineering and Management*, vol. 15, no. 7, pp. 3242–3250, May 2024.
 - [13] R. Plösch, J. Bräuer, C. Körner, and M. Saft, “Measuring, assessing and improving software quality based on object-oriented design principles,” *Open Computer Science*, vol. 6, no. 1, pp. 187–207, Jan. 2016.
 - [14] L. Etzkorn and H. Delugach, “Towards a semantic metrics suite for object-oriented design,” in *Proceedings. 34th International Conference on Technology of Object-Oriented Languages and Systems - TOOLS 34*, ser. TOOLS-34. IEEE Comput. Soc, pp. 71–80.

- [15] D.-L. Miholca, “New conceptual cohesion metrics: Assessment for software defect prediction,” in *2021 23rd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, Dec. 2021, pp. 163–170.
- [16] E. N. Haner Kirğıl and T. Ergelebi Ayyıldız, “Predicting software cohesion metrics with machine learning techniques,” *Applied Sciences*, vol. 13, no. 6, p. 3722, Mar. 2023.
- [17] Y. Xiang, W. Pan, H. Jiang, Y. Zhu, and H. Li, “Measuring software modularity based on software networks,” *Entropy*, vol. 21, no. 4, p. 344, Mar. 2019.
- [18] A. Marcus and D. Poshyvanyk, “The conceptual cohesion of classes,” in *21st IEEE International Conference on Software Maintenance (ICSM’05)*. IEEE, 2005, pp. 133–142.
- [19] S. Brunswicker, E. Almirall, and A. Majchrzak, “Optimizing and satisficing: The interplay between platform architecture and producers’ design strategies for platform performance,” *MIS Quarterly*, vol. 43, no. 4, pp. 1249–1277, Dec. 2019.
- [20] K. Lieberherr and I. Holland, “Assuring good style for object-oriented programs,” *IEEE Software*, vol. 6, no. 5, pp. 38–48, Sep. 1989.
- [21] M. J. LaMantia, Y. Cai, A. MacCormack, and J. Rusnak, “Analyzing the evolution of large-scale software systems using design structure matrices and design rule theory: Two exploratory cases,” in *Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*. IEEE, Feb. 2008, pp. 83–92.
- [22] M. Hitz and B. Montazeri, *Measuring coupling and cohesion in object-oriented systems*, 1995.
- [23] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, “Decoupling level: a new metric for architectural maintenance complexity,” in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE ’16. ACM, May 2016.
- [24] D. NICOLAU, “Demitizarea interfețelor în programarea orientată obiect,” *Revista Română de Informatică și Automatică*, vol. 30, no. 4, pp. 73–84, Dec. 2020.
- [25] F. M. Al Obisat, Z. T. Alhalhouli, T. I. Alrawashdeh, and T. E. Alshabat, “Review of literature on software quality,” *World of Computer Science & Information Technology Journal*, vol. 8, no. 5, 2018.
- [26] O. Bjuhr, K. Segeljakt, M. Addibpour, F. Heiser, and R. Lagerstrom, “Software architecture decoupling at ericsson,” in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, Apr. 2017, pp. 259–262.

- [27] A. K. Yadav and C. L. P. Gupta, “Quantification of reliability in object-oriented software: Functionality perspective,” in *2024 IEEE 1st Karachi Section Humanitarian Technology Conference (KHI-HTC)*. IEEE, Jan. 2024, pp. 1–6.
- [28] S. S. Kumar, P. K. M. Kumar, S. A. Panimalar, and J. Vinoj, “Qos based soft computing techniques for evaluating efficient web service recommendation,” *International Journal of System Assurance Engineering and Management*, vol. 15, no. 1, pp. 205–215, Jul. 2022.
- [29] L. Lo Iacono, H. V. Nguyen, and P. L. Gorski, “On the need for a general rest-security framework,” *Future Internet*, vol. 11, no. 3, p. 56, Feb. 2019.
- [30] M. I. Beer and M. F. Hassan, “Adaptive security architecture for protecting restful web services in enterprise computing environment,” *Service Oriented Computing and Applications*, vol. 12, no. 2, pp. 111–121, Nov. 2017.
- [31] D. V. Kornienko, S. V. Mishina, S. V. Shcherbatykh, and M. O. Melnikov, “Principles of securing restful api web services developed with python frameworks,” *Journal of Physics: Conference Series*, vol. 2094, no. 3, p. 032016, Nov. 2021.
- [32] J. Bogner, S. Kotstein, and T. Pfaff, “Do restful api design rules have an impact on the understandability of web apis?” *Empirical Software Engineering*, vol. 28, no. 6, Sep. 2023.
- [33] A. Chatterjee and A. Prinz, “Applying spring security framework with keycloak-based oauth2 to protect microservice architecture apis: A case study,” *Sensors*, vol. 22, no. 5, p. 1703, Feb. 2022.
- [34] Z. Shelby, “Embedded web services,” *IEEE Wireless Communications*, vol. 17, no. 6, pp. 52–57, Dec. 2010.
- [35] J. Bellido, R. Alarcón, and C. Pautasso, “Control-flow patterns for decentralized restful service composition,” *ACM Transactions on the Web*, vol. 8, no. 1, pp. 1–30, Dec. 2013.
- [36] S. Kennedy, R. Stewart, P. Jacob, and O. Molloy, “Storhm: a protocol adapter for mapping soap based web services to restful http format,” *Electronic Commerce Research*, vol. 11, no. 3, pp. 245–269, Feb. 2011.
- [37] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, “Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study,” *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, Feb. 2021.
- [38] V. Raj and R. Sadam, “Evaluation of soa-based web services and microservices architecture using complexity metrics,” *SN Computer Science*, vol. 2, no. 5, Jul. 2021.

- [39] S. Telenyk, V. Voinalovych, and D. Smakovskiy, “Web-application architecture for the kubernetes cluster at the google cloud platform with horizontal automatic scaling,” *Adaptive automatic control systems*, vol. 2, no. 39, pp. 98–105, Dec. 2021.
- [40] B. Alshemaimri, R. Elmasri, T. Alsahfi, and M. Almotairi, “A survey of problematic database code fragments in software systems,” *Engineering Reports*, vol. 3, no. 10, Jul. 2021.
- [41] H. S. Lallie, A. Thompson, E. Titis, and P. Stephens, “Analysing cyber attacks and cyber security vulnerabilities in the university sector,” *Computers*, vol. 14, no. 2, p. 49, Feb. 2025.
- [42] B. Alotaibi, “Cybersecurity attacks and detection methods in web 3.0 technology: A review,” *Sensors*, vol. 25, no. 2, p. 342, Jan. 2025.
- [43] S. D. Meglio and L. Libero Lucio Starace, “Evaluating performance and resource consumption of rest frameworks and execution environments: Insights and guidelines for developers and companies,” *IEEE Access*, vol. 12, pp. 161 649–161 669, 2024.
- [44] Spring. (2025) Why Spring? Accessed on: 2025-05-01. [Online]. Available: <https://spring.io/why-spring/>
- [45] ——. (2025) Using @Autowired. Accessed on: 2025-05-01. [Online]. Available: <https://docs.spring.io/spring-framework/reference/core/beans/annotation-config/autowired.html>
- [46] G. Stubbings and S. Polovina, “Levering object-oriented knowledge for service-oriented proficiency: Enhancing service capability in developers,” *Computing*, vol. 95, no. 9, pp. 817–835, Feb. 2013.
- [47] M. Zakeri-Nasrabadi, S. Parsa, and S. Jafari, “Measuring and improving software testability at the design level,” *Information and Software Technology*, vol. 174, p. 107511, Oct. 2024.
- [48] R. Laigner, D. Mendonça, A. Garcia, and M. Kalinowski, “Cataloging dependency injection anti-patterns in software systems,” *Journal of Systems and Software*, vol. 184, p. 111125, Feb. 2022.
- [49] Baeldung. (2025) Spring Validation in the Service Layer. Accessed on: 2025-05-01. [Online]. Available: <https://www.baeldung.com/spring-service-layer-validation>
- [50] Spring. (2025) Custom Repository Implementations. Accessed on: 2025-05-01. [Online]. Available: <https://docs.spring.io/spring-data/jpa/reference/repositories/custom-implementations.html>
- [51] ——. (2025) Working with Spring Data Repositories. Accessed on: 2025-05-01. [Online]. Available: <https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html>

- [52] D. Zmaranda, L.-L. Pop-Fele, C. Gyorödi, R. Gyorödi, and G. Pecherle, “Performance comparison of crud methods using net object relational mappers: A case study,” *International Journal of Advanced Computer Science and Applications*, vol. 11, no. 1, 2020.
- [53] V. Engström, P. Johnson, R. Lagerström, E. Ringdahl, and M. Wällstedt, “Automated security assessments of amazon web services environments,” *ACM Transactions on Privacy and Security*, vol. 26, no. 2, pp. 1–31, Mar. 2023.