Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.067127

ARTICLE

Secure Development Methodology for Full Stack Web Applications: Proof of the Methodology Applied to Vue.js, Spring Boot and MySQL

Kevin Santiago Rey Rodriguez, Julián David Avellaneda Galindo, Josep Tárrega Juan, Juan Ramón Bermejo Higuera*, Javier Bermejo Higuera and Juan Antonio Sicilia Montalvo

School of Engineering and Technology, International University of La Rioja, Avda. de La Paz, 137, Logroño, 26006, La Rioja, Spain

*Corresponding Author: Juan Ramón Bermejo Higuera. Email: juanramon.bermejo@unir.net

Received: 25 April 2025; Accepted: 17 July 2025; Published: 29 August 2025

ABSTRACT: In today's rapidly evolving digital landscape, web application security has become paramount as organizations face increasingly sophisticated cyber threats. This work presents a comprehensive methodology for implementing robust security measures in modern web applications and the proof of the Methodology applied to Vue.js, Spring Boot, and MySQL architecture. The proposed approach addresses critical security challenges through a multi-layered framework that encompasses essential security dimensions including multi-factor authentication, fine-grained authorization controls, sophisticated session management, data confidentiality and integrity protection, secure logging mechanisms, comprehensive error handling, high availability strategies, advanced input validation, and security headers implementation. Significant contributions are made to the field of web application security. First, a detailed catalogue of security requirements specifically tailored to protect web applications against contemporary threats, backed by rigorous analysis and industry best practices. Second, the methodology is validated through a carefully designed proof-of-concept implementation in a controlled environment, demonstrating the practical effectiveness of the security measures. The validation process employs cutting-edge static and dynamic analysis tools for comprehensive dependency validation and vulnerability detection, ensuring robust security coverage. The validation results confirm the prevention and avoidance of security vulnerabilities of the methodology. A key innovation of this work is the seamless integration of DevSecOps practices throughout the secure Software Development Life Cycle (SSDLC), creating a security-first mindset from initial design to deployment. By combining proactive secure coding practices with defensive security approaches, a framework is established that not only strengthens application security but also fosters a culture of security awareness within development teams. This hybrid approach ensures that security considerations are woven into every aspect of the development process, rather than being treated as an afterthought.

KEYWORDS: Web security methodology; secure software development lifecycle; DevSecOps; security requirements; secure development; Full Stack Web applications

1 Introduction

Corporations increasingly adopt technological services. This adoption optimizes corporate efficiency but also increases susceptibility to damage from vulnerabilities and attacks. These attacks can be deliberate or accidental, originating from both users and deficiencies in system design [1].

Among the solutions adopted in a digitalization process, web applications stand out for their robustness and adoption in the corporate environment, establishing the necessity of integrating rigorous security

measures from the initial stages of development [2]. This approach aims to mitigate the types of risks that these developments may present [3].

Although protocols have been established to mitigate risks or at least consider them during development, the difficulty in anticipating vulnerabilities from the moment requirements are defined poses a significant challenge [4]. This requires programming practices and technological tools to adopt a defensive perspective, necessitating the involvement of specialized professionals. This situation assigns information security managers to a crucial role within development teams, where constant attention to security can be compromised by the complexity of anticipating failures, the availability and accuracy of tools, among other factors [5].

Within the development process, engineering requirements support system design by providing frameworks used to mitigate known security flaws. The areas of these frameworks are known as domains [6]. These domains allow for the segregation of requirements to make them more specific and provide detailed information on the area in which they apply. Security requirements enable the adoption of specific safeguards by domains and consideration of security from an early stage of software solution development [7]. For this reason, initiatives have been created that attempt to provide a structured process with measures to mitigate risks and measure security maturation in software development.

Despite the existence of various security frameworks and SSDLC methodologies, there remains a significant gap in practical security approaches for full-stack web applications using Vue.js, Spring Boot, and MySQL. Current methodologies often separate security concerns from development, leaving security validation as a divided phase performed often only by specialists. This creates an inefficient cycle where vulnerabilities are discovered late in the process, increasing remediation costs. Our research addresses this gap by proposing a methodology that integrates security from the beginning through clear guidelines, patterns, and practices that can be followed by all team members—regardless of their security expertise. Rather than relying solely on defensive programming and vulnerability detection at later stages, this approach emphasizes secure coding principles that can be applied by every developer throughout the development lifecycle. By democratizing security knowledge and embedding it into standard development practices, the methodology allows teams without dedicated security specialists to still produce secure applications, while ensuring that security becomes an integral part of the development process rather than an afterthought.

Numerous errors can be identified through systematic methods, such as static code analysis or penetration testing [8]. However, design flaws or the implementation of insecure programming patterns represent a more challenging category of vulnerabilities to analyze. The existence of conceptual errors in design can not only trigger unforeseen security breaches, thus causing an increase in costs and delays in project execution but also complicate the implementation of corrective solutions in advanced phases of development, jeopardizing the project's feasibility. For this reason, the critical importance of integrating software security from the design stage is emphasized, adopting a development methodology that empowers the entire team in the production of secure code, suitable for use cases and aligned with best practices from the start [9].

Adopting a meticulous planning strategy from the design phase facilitates not relying exclusively on reviewing already developed components to identify potential vulnerabilities [10]. Instead, this approach ensures that each developer has the ability from the beginning to generate high-quality code, minimizing the likelihood of incorporating design flaws in the development of new functionalities or in software maintenance. This is considered secure code development that precedes defensive development. This secure code generation approach ensures that, in later stages, the active search for vulnerabilities is considered an additional reinforcement and not the only way to discover security deficiencies [11]. This improves software quality from an early stage [12].

When discussing the technological transition of companies to web environments, one of the most used technologies is Java with its development and execution environment [13]. A significant number of applications still run under the Java virtual machine today, as it is a robust development language that has taken care of the backward compatibility of each of its versions. This characteristic has provided companies with confidence in migrating their services [14]. Within the JVM ecosystem, numerous frameworks are available, focused on maximizing development performance; among web-focused frameworks, Spring Framework has taken great relevance within the Java web application development environment. Although this framework maximizes development performance, its meta-framework "Spring Boot" has become an industry standard for Java-based web application development [15]. This area of development constitutes the central focus of this study [16].

While Spring Boot can create embedded graphical interfaces managed by the same web server, modern development favors distributed architectures that separate business logic, persistence components, and UI—the direct layer where users interact through browsers. In the browser, code execution can be performed using various technologies such as Web-Assembly, though JavaScript remains the most used tool for developing graphical interfaces. Among JavaScript frameworks, Vue.js stands out due to its widespread adoption and open-source nature, making it compelling for professional development [17]. Given that this section of the code is hosted within users' browsers, it is necessary to consider that the security processes needed for this type of development are essential [18].

Within a full-stack development, the division of the persistence layer from other components is considered a good practice [19]. This component is considered critical as it is where information linked to the component is stored. This makes it an essential asset of the system. Due to the importance and separation of this component, it requires an adequate procedure to ensure its secure operation in terms of communication and information transmission. It is also necessary to consider measures in the form of data persistence and internal functioning of the persistence layer, such as data redundancy with strict policies for data recovery [20]. Currently, MySQL is a quite robust solution that is widely known in the current field of software development [21].

The widespread adoption of these technologies in web application development necessitates a methodology that adopts a hybrid security approach. This approach combines secure coding practices that involve the entire development team with defensive strategies that enable security specialists to implement targeted processes. This model ensures multi-layered security across all software development phases.

Modern methodologies incorporate containerization optimized for cloud environments throughout all phases—from analysis and design through implementation to deployment. This approach enables continuous integration and efficient deployment, ensuring both functional robustness and security throughout the application lifecycle. This hybrid approach facilitates early vulnerability detection while promoting a culture where all team members understand their role in software protection.

Based on the identified gap in security approaches for full-stack applications, we formulate the following research questions:

- How can security requirements be effectively integrated into the development process of full-stack web applications using Vue.js, Spring Boot, and MySQL in a way that is accessible to all team members, regardless of security expertise?
- What specific security measures and design patterns are necessary for each component of this technological stack that can be implemented by developers without specialized security knowledge?

- How can a DevSecOps pipeline be designed to automate and validate security requirements throughout the development lifecycle, while providing clear guidance for remediation that empowers developers to address security issues themselves?

This work hypothesizes that a comprehensive methodology that democratizes security knowledge through accessible secure coding practices and clear guidelines, tailored specifically for this technology stack, will significantly reduce vulnerabilities compared to approaches that rely primarily on security specialists at later stages of development.

To address these research questions and validate our hypothesis, this work makes the following specific contributions:

- A comprehensive security methodology specifically designed for full-stack web applications using Vue.js, Spring Boot, and MySQL architecture, accessible to developers without specialized security expertise.
- A structured model of security requirements based on OWASP SAMM and ASVS frameworks, adapted for this specific technology stack and presented as implementable design patterns and guidelines.
- A practical validation through a proof-of-concept implementation of a ticket management system demonstrating the application of security principles by non-security specialists.
- A DevSecOps pipeline design that automates security validation processes throughout the development cycle with clear remediation guidance for developers.
- Quantitative and qualitative metrics for evaluating the effectiveness of the proposed methodology in reducing vulnerabilities when implemented by teams without dedicated security resources.

The rest of this work is structured as follows. Section 2 presents the context of digital transformation and its implications for security, exploring traditional development models and the evolution towards secure development methodologies. Also, it examines related work, analyzing the different types of existing SSDLCs and security requirements engineering approaches. Section 3 details the proposed methodology for secure web development, establishing a structured model of security requirements and defining specific processes for each security domain. Also, it describes the implementation and validation of the methodology through a ticket management system, detailing both the system architecture and the security validation tools used. Section 4 presents the analysis and discussion of the results obtained during the evaluation of the methodology. Finally, Section 5 offers the conclusions of the work and proposes future lines of research to continue improving security in web application development.

2 Background and Relative Work

Digital transformation is a fundamental pillar for growth and competitiveness, both for companies and countries. Digitalization is utilized by companies to create external opportunities and to improve internal processes, thus obtaining significant benefits for both organizations and users [22]. Digitalization is even used as an economic indicator that describes a country's competitiveness in international markets, therefore the cybersecurity of the systems becomes more relevant [23]. Countries' support for digital transformation of their industries promotes direct and indirect economic growth [24]. For many countries, the importance of industrial transformation is prioritized to the extent that it forms part of their long-term objectives, as indicated by the European Commission [25]. The continuous rise and evolution of digital transformation affecting all business processes of companies and organizations, amply justifies cybersecurity in the context of digital transformation [26].

However, this digitalization, or software development is a complicated task, requiring a combination of technical and soft skills. Systems are increasingly complex and require constant learning and adaptation

to changes [27]. When executed collaboratively, as in some type of project, its complexity increases, involving factors of communication, coordination, integration among other aspects [28]. Additionally, at an organizational level, it becomes not only a complex but costly task, where infrastructure, equipment, and other factors involved in development influence costs. On the other hand, software development often involves uncertainty, meaning it may not be clear how the desired product will be carried out, which in many cases causes projects to fail, by not meeting expectations or by encountering too many problems that delay deliveries [29]. Multiple factors contribute to project failure, one of the most common being inadequate requirements, which result in underestimated projects and trigger a domino effect of issues [30]. With all these factors when executing projects, organizations commonly implement a type of "software development life cycle", which helps increase the chances of success [31]. SDLC can be defined as a formal or informal methodology that comprises the necessary steps or aspects for the design, construction and maintenance of software [32].

While digitalization is increasingly relevant for companies and industries, cybersecurity has also become a critical concern. When services and processes are migrated to a digital environment, cyberattacks emerge as an increasingly significant risk. These cyberattacks have occurred with increasing frequency, resulting in the loss or degradation of essential assets [33]. These attacks are expected to occur more frequently each year and become more sophisticated; it is already an industry that has a current growth of at least 1.5 trillion per year. With more services and projects online, attackers gain a larger attack surface, and entities that develop and maintain software also face a wider surface to audit [34,35]. In traditional approaches, security errors in software, such as bugs and vulnerabilities, are mostly discovered through tests, such as penetration testing, user reports, and stakeholder reports that validate the behaviour of the software when it is in its final phases. The characteristics of these bug detection processes allow these steps to be characterized as reactive actions, which evidence shortcomings in the artifacts of the previous phases in the development cycle in very late stages of the software [36]. Since the digitalization process is fundamentally linked to the creation of software, the development cycle of how it is created takes on vital importance for software security.

The most traditional model is known as Waterfall, it is a sequential model, where it goes through different phases and where greater detail is given to documentation and planning, additionally each stage must wait for the previous one to finish before starting the next [37]. Another model, or rather meta-model, is the spiral, which can be used as a basis in other models, it focuses on dividing the project into small segments and carrying out each phase of development with these fragments, it presents the opportunity to constantly evaluate risks [37]. Agile development is a combination of iterative and incremental processes, where the main motivation is to adapt to changes, always seeking customer satisfaction by making rapid deliveries, which are improved with each new iteration. Today, these traditional development cycles, such as Waterfall and Spiral, remain widely used in software creation companies [38]. For this reason, a deep understanding of their stages is valuable to understand the security problems that can occur when using traditional models.

Secure software development was not traditionally considered within development life cycles. Security was typically applied at the network or infrastructure level, for example, through the implementation of perimeter systems, firewalls, or by physically securing equipment [39], for at the end of the development, attempts were made to breach the system in order to identify and address errors; in the worst cases, solutions were only sought once an attack had already occurred. This approach is known as "penetrate and patch" [40]. However, this method fails to incorporate security into the system during its initial construction.

This traditional method of securing systems presents several problems: security breaches are always costly, either directly or because of lost organizational trust. There is also the possibility that the system is being attacked or exploited imperceptibly. Because of security being included during the final phases of

development, attackers are given the advantage of always being one step ahead. When security patches are implemented, new vulnerabilities may be introduced, and furthermore, users may not even apply the security patches [41].

With these issues in mind, a solution arises from a straightforward premise: errors and security breaches originate from flaws in the code; therefore, security must be rigorously integrated during the software development process, rather than at its final stage [40]. SDLC was originally described as a methodology for software development aimed at ensuring the creation of software that meets its intended needs. Addressing the challenges outlined, the concept now known as the "Secure Software Development Lifecycle", abbreviated as SSDLC, emerged. SSDLC seeks to incorporate security principles and practices more proactively within software development processes, in contrast to traditional approaches [42].

Microsoft's Security Development Lifecycle (SDL) represents one of the first comprehensive implementations of SSDLC principles. SDL consists of specific security activities organized into seven phases: training, requirements, design, implementation, verification, release, and response [43]. Each phase includes mandatory security activities, such as threat modeling during design, static code analysis during implementation, and penetration testing during verification. This approach marked a paradigm shift from reactive security measures to a proactive security-by-design approach, establishing security as a non-functional requirement that must be considered from the earliest stages of development [41].

Building upon similar principles but with different approaches, the Open Web Application Security Project (OWASP) introduced the Software Assurance Maturity Model (SAMM), which focuses on evaluating and improving security practices across an organization rather than within individual projects [44]. SAMM is organized around five business functions: Governance, Design, Implementation, Verification, and Operations, with each function containing three security practices. Unlike SDL, which prescribes specific activities, SAMM provides a flexible framework that organizations can adapt based on their risk profile and business context. This model emphasizes progressive improvement through three maturity levels for each practice, allowing organizations to evolve their security posture gradually [45].

The Building Security In Maturity Model (BSIMM) takes yet another approach, focusing on observing and analyzing real-world security initiatives rather than prescribing ideal practices. BSIMM is a descriptive model derived from studying actual security activities in over 100 organizations across various industries [46]. Organized into four domains (Governance, Intelligence, SSDL Touchpoints, and Deployment) with 12 practices, BSIMM serves as a measuring stick for organizations to benchmark their security programs against industry peers. Unlike SDL and SAMM, which are prescriptive, BSIMM offers insights into what organizations are doing, providing realistic expectations for security program evolution.

While these methodologies have advanced software security significantly, they present limitations when applied to modern web development environments, particularly for specific technology stacks like Vue.js, Spring Boot, and MySQL. Traditional SSDLC methodologies often lack granular guidance for specific technologies, leaving development teams to interpret general principles without clear implementation directions. Additionally, these methodologies typically assume the presence of security specialists, creating barriers for teams without dedicated security resources [47]. Furthermore, the rapid evolution of web technologies often outpaces the update cycles of established methodologies, creating gaps in coverage for emerging threats and vulnerabilities specific to modern frameworks and libraries [42].

When examining security considerations specifically for Vue.js applications, additional challenges emerge beyond those addressed by general SSDLC frameworks. Single Page Applications (SPAs) built with Vue.js face unique security threats, including Cross-Site Scripting (XSS) vulnerabilities through unsafe template binding, insecure state management in Vuex stores, and client-side routing vulnerabilities [48].

Traditional security frameworks rarely address these frontend-specific concerns, leaving developers without clear guidance. Vue's reactivity system, while powerful for development, introduces specific security challenges such as prototype pollution and injection attacks if not properly managed. Hellquist's (2024) comprehensive framework comparison reveals that Vue.js applications face unique security issues such as Format String Errors, which were identified during dynamic testing using OWASP ZAP. While Vue demonstrated a clean dependency check with no vulnerable packages in its base configuration, it still showed significant security misconfigurations related to Content Security Policy, CORS settings, and anti-clickjacking protections—vulnerabilities shared across modern JavaScript frameworks but requiring framework-specific mitigations. The Vue.js security ecosystem itself is still evolving, with gaps in automated security tooling specifically designed for Vue's component architecture and lifecycle hooks. These frontend-specific security challenges require specialized knowledge and approaches that extend beyond traditional SSDLC frameworks, supporting Hellquist's finding that 'developers must gain knowledge and implement secure coding practices without relying solely on first-party mitigations.

The main contributions of this work are:

- Analyze and investigate existing SSDLC (Secure Software Development Life Cycle) methodologies, evaluating their strengths, limitations, and applicability in modern web development environments.
- Define a methodology that integrates both secure coding practices and defensive approaches, creating a hybrid framework that involves the entire development team in project security while allowing the security team to implement specific and targeted processes.
- Validate the effectiveness of the proposed methodology through the development of a ticket management system based on Vue.js, Spring Boot, and MySQL architecture that incorporates the identified security requirements, demonstrating its practical applicability in a real scenario.
- Design and implement a DevSecOps pipeline that automates security processes, integrating dependency analysis tools (OSV-Scanner, Trivy, OWASP Dependency-Check), static code analysis (SonarQube, Bearer, Fortify), and dynamic analysis (OWASP ZAP) throughout the development cycle.
- Establish metrics to evaluate the effectiveness of the proposed methodology, providing objective indicators that allow measuring the level of security achieved and the efficiency of the implemented process.

The Secure by Design approach presented by [36] offers a promising paradigm shift toward democratizing security, yet it contains significant gaps in addressing practical implementation. Their framework suggests that developers without security expertise can naturally create secure software through domain-focused design principles; however, the literature fails to provide methodological frameworks for making security accessible to non-specialist developers. The authors emphasize that software design is inherently central to most developers' competencies, but their work lacks structured guidance on training programs or knowledge transfer mechanisms that would effectively bridge the expertise gap. This limitation becomes particularly evident in the absence of empirical research on security education models tailored to development teams without specialized security backgrounds. While Deogun and Johnsson demonstrate how design-driven approaches can implicitly solve security issues—as in their domain primitive example that prevents XSS vulnerabilities—they do not offer a comprehensive educational framework that would systematically empower ordinary developers to identify and implement such patterns independently. This oversight perpetuates a dependency on security expertise in an industry where dedicated security resources remain scarce for many organizations.

3 Methods

3.1 Security Requirements Model

The OWASP Software Assurance Maturity Model (SAMM) [49] is an initiative that provides a method for assessing and implementing a secure development lifecycle through a series of security practices applicable at any phase. This model is structured around business functions, each of which contains certain security practices. These practices consist of a set of activities, each divided into maturity levels. This work focuses on the design phase of application development. In this phase, an organization defines the objectives and determines how to create software that can fulfil them, incorporating security requirements and security architecture. In OWASP SAMM, this phase includes practices for threat assessment, security requirements, and security architecture. Within the design activities, this work centers on security requirements. SAMM's security requirements activities define processes both for the security of the software being developed and for third-party services used.

SAMM maintains a direct approach to software development and application security, rather than focusing on organizational-level security. The framework allows security to be incorporated in stages, according to the maturity level at which the organization currently operates. To develop a methodology that formulates a model serving as security requirements during application development, a framework is sought that permits the application of recommendations solely during the design stage. As SAMM demonstrates flexibility, it allows the adoption of recommendations according to the business function of interest, enabling isolation and utilization of its recommendations as a foundation. Additionally, its focus on software development and application security rather than at an organizational level aligns with the objective of the methodology focused solely on security requirements, whilst ensuring the methodology can integrate into a Secure Software Development Lifecycle (SSDLC).

Our methodology is specifically designed for full-stack web architectures and functions as a complementary tool that integrates within established SSDLC frameworks such as SAMM (Software Assurance Maturity Model), Microsoft SDL (Security Development Lifecycle), or BSIMM (Building Security In Maturity Model). While these frameworks provide general guidelines and organizational processes for implementing security in software development, our proposal offers detailed and granular technical specifications for a specific architectural domain. This integration allows development teams to achieve a completer and more accurate outcome in defining and implementing security requirements for full-stack web applications, using our methodology as a specific technical guide within the broader process established by the SSDLC framework adopted by the organization.

The specificity of our methodology for full-stack web architectures enables addressing vulnerabilities and security requirements that general SSDLC frameworks do not cover with the level of technical detail necessary for practical implementation. For example, while SDL establishes the need to "implement secure authentication controls", our methodology specifies the 35 concrete requirements organized by layers (authentication, session management, authorization, input validation, secure communications), providing precise technical guides such as controls SR-AU-01 to SR-AU-06 for authentication, SR-SM-01 to SR-SM-04 for session management, and SR-AC-01 to SR-AC-05 for authorization. This technical granularity is essential for developers to effectively implement the security principles established by SSDLC frameworks in the specific context of modern web applications.

For applications that commence without requirements, SAMM recommends the use of ASVS, which proves highly practical as its recommendations are based on the application's risk level as well as the data it will handle. This approach makes it ideal for selecting recommendations that suit the software to be developed, allowing proportionate efforts to be made relative to the potential impacts of security

failures. This risk-based approach ensures that security controls are appropriately scaled to match the criticality and sensitivity of the application being developed. Another advantage of ASVS is its adaptation to all aspects of the development lifecycle, encompassing design principles, implementation guidance, and security considerations in operations. This comprehensive coverage makes it ideal for applications that are beginning construction, as it provides structured security verification standards across three defined levels of security assurance.

Based on this model, the methodology aims to compile a set of clear and straightforward requirements to help achieve an acceptable level of application security. It also provides direct documentation of the security aspects it intends to cover, supported by models developed by OWASP ASVS, CWE, and other sources, thus offering a starting point for further development. These requirements are intended to be generic, so they can serve as a base for constructing any type of application, with the goal of ensuring a minimum level of security.

The security requirements support compliance with data protection regulations, specifically GDPR Articles 25 and 32. Requirements SR-SC-01 and SR-SC-02 address Article 25's data protection by design mandate and Article 32(1)(a)'s encryption requirements. Data protection controls SR-DP-01 and SR-DP-02 implement Article 25's safeguarding provisions, while communication security requirements SR-COM-01 and SR-COM-02 satisfy Article 32(1)(b)'s confidentiality and integrity mandates. This alignment enables systematic compliance demonstration through technical implementation.

3.2 Defining a Secure Methodology

The definition of specific security requirements for the application is conducted through the identification of potential threats and the evaluation of vulnerabilities. This includes the development of a specification document detailing the assets to be protected, the identified threats, possible vulnerabilities, analysed risks, and recommended practices for mitigating these risks.

The identification of security requirements is based on the OWASP Application Security Verification Standard (ASVS) model, which provides a structured framework to ensure the security of web applications. This model encompasses several key areas of security, including:

- Authentication: Implement multi-factor authentication to ensure that only authorized users can access the system. This aims to verify the identity of the user or entity and to ensure that the verification process is not vulnerable to impersonation or credential interception.
- Access Control: The application must correctly manage authorization, controlling and specifying access to each resource based on roles and permissions, following the principle of least privilege.
- Session Management: Guarantee proper interaction between the user and the application, ensuring that sessions remain individual, non-transferable, and are invalidated when no longer necessary.
- Validation, Sanitization and Encoding: Secure all information processed and entered into the system by validating encoding levels, values, ranges, filters, and sanitizing any stored data.
- Database Security: Ensure secure access to data storage, covering configuration, application integration, and secure communication.
- Stored Cryptography: Implement cryptographic modules within the application to prevent uncontrolled errors and to securely manage access to sensitive data. Ensure data encryption both in transit and at rest and use ORM and prepared statements to prevent SQL injection.
- Logging and Auditing: Provide relevant and adequate information to stakeholders without exposing unnecessary or sensitive data. Control log volumes by reducing noise and performing regular cleanups. Implement secure logs for activity monitoring and auditing.

- Data Protection: Ensure data is protected across all dimensions, meaning it is reliable, protected from unauthorized inspection, maintains integrity (cannot be maliciously altered), and is available whenever the user requires access.
- Communication: Ensure secure communication between application components by using robust encryption methods, appropriate configurations, and adhering to the latest industry standards.
- Malicious Code: Ensure the application source code must meet security requirements, protecting it from malicious or unintended instructions, backdoors, or inherited malicious code.
- API and Webservices: Ensure the API layer must incorporate proper authentication, session management, authorization, effective security controls, and input validation for all parameters.
- Configuration: Ensure production environment configurations must be designed to protect the system from common attacks, securing the system once it is online.

All security requirements are included and explained in Tables 1–35.

3.2.1 Authentication

Table 1 specifies the requirement for the password security verification.

Table 1: Requirement SR-AUT-01: password security verification

SR-AUT-01:	**Verify that the user's password meets appropriate length and complexity requirements**
Description	The user's password must meet a minimum required length and include certain special characters. The system should notify the user whether the password they are attempting to create meets these requirements.
Recommendation	An identity and access management solution should be Implemented. Configure this service with security policies that enforce appropriate password length and complexity.

Table 2 specifies the requirement for the Multi-Factor Authentication (OTP).

Table 2: Requirement SR-AUT-02: Multi-Factor Authentication (OTP)

SR-AUT-02:	**Ensure the availability of multi-factor authentication via OTP**
Description	The system must implement multi-factor authentication using one-time passwords (OTP) as a strategy. Each time the user attempts to access the system, they must provide this password.
Recommendation	Enable the multi-factor authentication functionality provided by the identity and access management solution with OTP.

Table 3 specifies the requirement for the user credential management.

Table 3: Requirement SR-AUT-03: user credential management

SR-AUT-03:	Verifying that user credentials are stored securely
Description	It is required that user credentials be stored securely, incorporating an appropriate salt to enhance security. The system must allow for the secure updating of credentials, ensuring compliance with best practices in cryptographic storage and access control.
Recommendation	It is preferable not to store credentials within the application's database. Instead, this responsibility can be delegated to an identity and access management solution, which securely manages credential storage, ensuring robust authentication and access control mechanisms.

3.2.2 Session Management

Table 4 specifies the requirement for the session token management.

Table 4: Requirement SR-SM-01: session token management

SR-SM-01:	Verifying that the session token is generated upon authentication and not exposed
Description	When a user successfully authenticates, the application must request a session token that is not exposed in any request parameters. This token should be securely used throughout the client's interaction with the application, preventing potential security risks such as token leakage or interception.
Recommendation	Generate a JWT token, which should be stored securely in the frontend application. Every request must include this token for authentication purposes. Additionally, the application should be configured to integrate with identity and access management solution, enabling method security authorizations to secure all application endpoints effectively.

Table 5 specifies the requirement for the session token invalidation.

Table 5: Requirement SR-SM-02: session token invalidation

SR-SM-02:	Ensuring proper session invalidation upon expiry or logout
Description	When a session is terminated, whether due to expiration or explicit logout, a request must be sent to invalidate it, preventing unauthorized reuse.
Recommendation	Configure token policies to ensure they are invalidated after a defined period. The application should explicitly call the session invalidation function and ensure that the session cannot be reused. This enhances security by preventing session hijacking and unauthorized access.

Table 6 specifies the requirement for the session token configuration.

Table 6: Requirement SR-SM-03: session token configuration

SR-SM-03:	Ensuring session generation through a secure token
Description	Access to the application must be restricted so that users cannot gain entry without a valid token. This token defines the user session and must be securely stored using a cookie to prevent unauthorized access and potential security breaches.
Recommendation	Implement a validation mechanism within the application that utilizes a cookie to verify the user's session. The cookie must be configured with the following security attributes http-only true, secure true, same-site strict.

3.2.3 Access Control

Table 7 specifies the requirement for the role-based access.

Table 7: Requirement SR-AC-01: role-based access

SR-AC-01:	Ensuring resources are associated with role-based access control
Description	All resources must be associated with specific roles, adhering to the principle of least privilege. It must be ensured that users can only access the resources explicitly permitted for their assigned roles.
Recommendation	The user's role should be extracted from the JWT token and validated within the application. All application controllers must be associated with a role, enforce the usage of role-based access control for each resource. Additionally, within the web application, each interface functionality must be linked to a corresponding role to ensure proper authorization management.

Table 8 specifies the requirement for the Cross Site Request Forgery (CSRF) protection.

Table 8: Requirement SR-AC-02: CSRF protection

SR-AC-02:	Ensuring the application is protected against CSRF
Description	The technologies used in the application must enforce anti-CSRF mechanisms to prevent Cross-Site Request Forgery (CSRF) attacks, ensuring that malicious actors cannot perform unauthorized actions on behalf of authenticated users.
Recommendation	To mitigate CSRF risks, implement an anti-CSRF token in all application forms and proper validation of request origins. Additionally, for API security, consider adopting an stateless architecture, which reduces CSRF risks by eliminating reliance on session-based authentication.

3.2.4 Validation, Satinization, Encoding

Table 9 specifies the requirement for the entry points protection.

Table 9: Requirement SR-VSE-01: entry points protection

SR-VSE-01:	Ensuring all application entry points are protected against malicious data
Description	It is required to validate the structure of input parameters to protect the application from maliciously crafted data and HTTP parameter pollution attacks that could compromise security.
Recommendation	Implement input validation for all incoming data objects, ensuring proper formatting and structural integrity. Utilize language and framework-specific validation features to define expected data types and enforce strict validation rules. Reject any invalid or unexpected data to prevent it from being processed or stored within the application.

Table 10 specifies the requirement for the input data sanitation.

Table 10: Requirement SR-VSE-02: input data sanitation

SR-VSE-02:	Ensuring input data is sanitized before execution in the application
Description	It is required that special fields, such as WYSIWYG editors, do not allow the execution of dynamic code. The application must validate or execute user-uploaded files and expressions in a controlled manner to prevent XSS (Cross-Site Scripting) and SSRF (Server-Side Request Forgery) attacks.
Recommendation	Front-end frameworks usually provide built-in protection against the execution of malicious code. It is essential to follow the framework's best security practices to ensure proper data escaping and blocking of unauthorized executions. Additionally, sanitization libraries and security headers should be implemented to reinforce protection.

Table 11 specifies the requirement for the output data sanitation.

Table 11: Requirement SR-VSE-03: output data sanitation

SR-VSE-03:	Ensuring output data is sanitized
Description	The encoding of output data must be properly escaped to match the required context interpreter, preventing injection attacks. Additionally, the application must maintain the user's regional settings to ensure data consistency and proper localization.
Recommendation	Front-end frameworks usually provide built-in protection against malicious code execution. It is essential to follow the framework's best practices to ensure correct data escaping and prevent unintended executions. Additionally, always enforce proper encoding techniques to match the expected output format and avoid security vulnerabilities.

3.2.5 Data Base

Table 12 specifies the requirement for the data queries protection.

Table 12: Requirement SR-DB-01: data queries protection

SR-DB-01:	Ensuring data queries are protected against injections
Description	All database queries must be secured to prevent SQL injection attacks. This can be achieved by implementing ORMs (Object-Relational Mappers) or Entity Frameworks, depending on the technologies used.
Recommendation	Use prepared statements to prevent direct SQL injection vulnerabilities and avoid executing queries with unvalidated user inputs. It is highly recommended to implement an ORM to manage database interactions securely.

Table 13 specifies the requirement for the secure database connection.

Table 13: Requirement SR-DB-02: secure database connection

SR-DB-02:	Ensuring a secure database connection
Description	By default, databases are not configured with strict security measures. It is essential to enable security controls, ensure authenticated access, and enforce encrypted communication to protect sensitive data.
Recommendation	The database must be isolated, restricting connections to authorized hosts only. Configure the database to accept only encrypted connections to prevent data interception. Ensure the database requires authentication, using strong credentials. The database user account must have restricted privileges, following the principle of least privilege. The database must not run as the root system user. Instead, a dedicated user account should be implemented for enhanced security.

3.2.6 Stored Cryptography

Table 14 specifies the requirement for the classifying information.

Table 14: Requirement SR-SC-01: classifying information

SR-SC-01:	Classifying information types and ensuring proper handling
Description	It is essential to identify sensitive information within the system, such as personal, financial, or medical data, and apply appropriate encryption mechanisms to protect it from unauthorized access.
Recommendation	All sensitive data must be securely stored using encryption techniques. Strong cryptographic algorithms should be implemented to ensure data confidentiality and integrity, preventing exposure in case of unauthorized access or data breaches.

Table 15 specifies the requirement for the cryptographic modules.

Table 15: Requirement SR-SC-02: cryptographic modules

SR-SC-02:	Ensuring the use of proper cryptographic modules
Description	The security module must handle errors without revealing sensitive information, utilise the latest encryption standards, deprecate outdated algorithms, authenticate data using digital signatures, and prevent unauthorized modifications.
Recommendation	When generating random data, use only secure implementations to ensure cryptographic strength. Always utilize the latest versions of the framework and libraires used. Implement proper exception handling, providing a generic exception controller to sanitize error messages and prevent information leaks.

Table 16 specifies the requirement for the secure storage of keys.

Table 16: Requirement SR-SC-03: secure storage of keys

SR-SC-03:	Ensuring secure storage of keys
Description	A dedicated security module must be implemented to handle key management securely. This should involve the use of key vaults or secure key managers, ensuring that sensitive information is not exposed outside these secure storage solutions.
Recommendation	Do not store credentials within the application; they should be provided at runtime instead, using environment variables or dedicated secrets management systems. Use identity management system for user credential management, ensuring proper security controls and encryption mechanisms.

3.2.7 Error Handling and Logging

Table 17 specifies the requirement for the secured logging.

Table 17: Requirement SR-EHL-01: secured logging

SR-EHL-01:	Ensuring logs contain appropriate content and are securely accessible
Description	Application logs must not expose sensitive information; instead, sensitive data should be masked or omitted. Logs should contain relevant details, such as successful operations, failures, and validation errors, to aid in monitoring and troubleshooting.
Recommendation	Configure logging levels appropriately, set audit logging to INFO, set database framework logging to ERROR to log only critical database-related issues.

Table 18 specifies the requirement for the synchronized timestamps.

Table 18: Requirement SR-EHL-02: synchronized timestamps

SR-EHL-02:	Ensuring logged timestamps are synchronized
Description	The application must correctly handle time and time zones by using UTC as the standard for timestamps. Additionally, it should support displaying information in the user's local time for better usability.
Recommendation	Use timestamps in UTC for all logged events to maintain consistency across different environments.

Table 19 specifies the requirement for the appropriate error messages.

Table 19: Requirement SR-EHL-03: appropriate error messages

SR-EHL-03:	Ensuring error messages are appropriate and do not reveal sensitive information
Description	The application must implement generic error messages for unexpected failures, ensuring that no sensitive information is exposed. All expected errors should be properly handled to provide clear but secure responses.
Recommendation	Configure the controllers to centrally manage exceptions and enforce consistent error handling. Ensure that detailed stack traces are logged internally but never displayed to users.

3.2.8 Data Protection

Table 20 specifies the requirement for the client-side information protection.

Table 20: Requirement SR-DP-01: client-side information protection

SR-DP-01:	Ensuring client-side information is properly protected
Description	Sensitive information must not be stored on the client side, including in the browser's localStorage, sessionStorage, cookies, or the DOM. Additionally, all stored data should be cleared upon session termination to prevent unauthorized access.
Recommendation	Restrict client-side storage to prevent the persistence of sensitive data. Conduct a security review to ensure that no confidential information is exposed in the client environment. Automatically clear stored data when the user logs out.

Table 21 specifies the requirement for the sensitive information protection.

Table 21: Requirement SR-DP-02: sensitive information protection

SR-DP-02:	Ensuring sensitive information is always protected
Description	Sensitive information must always be encrypted during transmission. Additionally, once the data is no longer needed, it should be overwritten to prevent unauthorized access. Any access to this information should be logged for auditability.
Recommendation	Implement secure-by-design data structures that explicitly define sensitive data handling.

3.2.9 Communication

Table 22 specifies the requirement for the secure client-originated communication.

Table 22: Requirement SR-COM-01: secure client-originated communication

SR-COM-01:	Ensuring secure client-originated communication
Description	All communication initiated by the client must use TLS (Transport Layer Security) with an updated version, preferably TLS 1.2 or TLS 1.3, to ensure data integrity and protection against interception or tampering.
Recommendation	The client-facing application must enforce TLS to secure all communications. Disable insecure protocols (e.g., TLS 1.0, TLS 1.1, and unencrypted HTTP) to prevent vulnerabilities.

Table 23 specifies the requirement for the secure server-originated communication.

Table 23: Requirement SR-COM-02: secure server-originated communication

SR-COM-02:	Ensuring secure server-originated communication
Description	All server-originated communication must use TLS certificates issued by trusted Certificate Authorities (CAs). Additionally, internal communication across all ports and protocols must be encrypted, and any TLS failures should be logged and reported.
Recommendation	Enforce TLS for all API connections to prevent unencrypted data transmission.

3.2.10 Code Base

Table 24 specifies the requirement for the defensive programming.

Table 24: Requirement SR-CB-01: defensive programming

SR-CB-01:	Implementing defensive programming
Description	Security-oriented programming paradigms and design patterns must be implemented to create a secure code layer within the application. Defensive programming ensures that potential vulnerabilities are mitigated at the code level.
Recommendation	Use a language-specific security model to enforce best practices in secure development. Ensure safe usage of JavaScript utilities, preventing misuse or unintended vulnerabilities.

Table 25 specifies the requirement for the secure data structures.

Table 25: Requirement SR-CB-02: secure data structures

SR-CB-02:	Secure data structures
Description	Data structures must be designed with built-in security mechanisms, ensuring their safe usage when handling sensitive information.
Recommendation	Implement defensive programming principles to enforce secure data handling, creating data structures that restrict direct access to sensitive values, allowing only controlled usage. Associate each piece of data with a specific type and enforce strict validation rules. Use DTOs (Data Transfer Objects) to filter and expose only relevant information at each layer of the application, preventing unnecessary data leakage.

Table 26 specifies the requirement for the avoid hardcoded credentials in the source code.

Table 26: Requirement SR-CB-03: avoid hardcoded credentials in the source code

SR-CB-03:	Verify absence of hardcoded credentials in source code
Description	Connection strings typically contain sensitive data that must not be embedded within the source code.
Recommendation	Centralize all connection strings to ensure no hardcoded strings containing sensitive information are inadvertently exposed within the codebase. This centralization facilitates comprehensive verification that no data leakage occurs through embedded credentials or configuration parameters.

3.2.11 Web Services and API

Table 27 specifies the requirement for the role-based access control.

Table 27: Requirement SR-WS-01: role-based access control

SR-WS-01:	Ensuring services require authentication for access
Description	All services must be secured so that controllers require authorization before execution. Additionally, access should be reinforced using a permission-based model to restrict actions based on user roles.
Recommendation	Use Spring Security to configure and enforce method-level security, ensuring that all application endpoints are protected. Require a valid authentication token for all controller interactions. The token must include user roles, which should be validated and enforced at each endpoint to restrict access based on permissions.

Table 28 specifies the requirement for the security in web services.

Table 28: Requirement SR-WS-02: security in web services

SR-WS-02:	Ensuring exposed services comply with security standards
Description	All exposed services must follow consistent encoding and data parsing standards across components, avoid exposing sensitive information, and return appropriate error codes to enhance security and maintain best practices.
Recommendation	Define a global encoding standard and enforce uniform data parsing utilities across all components. Filter responses using DTOs (Data Transfer Objects) to expose only relevant data while preventing sensitive information leaks. Implement a Global Exception Handler to map and return appropriate HTTP response codes based on the exceptions generated, ensuring meaningful but secure error messages.

Table 29 specifies the requirement for the service schemas validation.

Table 29: Requirement SR-WS-03: service schemas validation

SR-WS-03:	Ensuring service schemas are validated
Description	JSON schemas must be validated before processing within an isolated layer to detect missing or extra fields and ensure data integrity.
Recommendation	Implement input object validations to verify that received data matches the expected schema. Use whitelists for fields that should only accept specific values, ensuring controlled data input. Apply regular expressions to each data field to filter out invalid or potentially harmful input, preventing injection attacks or malformed data.

Table 30 specifies the requirement for the CSRF protection.

Table 30: Requirement SR-WS-04: CSRF protection

SR-WS-04:	Ensuring requests are protected against CSRF
Description	It is required to validate that cookies used for session management are protected against Cross-Site Request Forgery (CSRF) attacks through the implementation of appropriate security mechanisms.
Recommendation	Implement CSRF protection by validating the session cookie, ensuring it cannot be exploited in cross-origin requests. Enforce origin and referrer header checks to verify that requests come from trusted sources. Configure the SameSite attribute for cookies to Strict or Lax. Consider using CSRF tokens alongside cookie-based authentication to further mitigate CSRF risks.

Table 31 specifies the requirement for the secure content.

Table 31: Requirement SR-WS-05: secure content

SR-WS-05:	Ensuring received content is secure
Description	It is required to explicitly validate the expected content and ensure that all data transmission is encrypted using TLS to maintain secure communication.
Recommendation	Validate request headers to enforce Content-Type: application/json, ensuring that only properly formatted data is processed. Ensure that all headers and messages are transmitted over TLS encryption to prevent interception or tampering.

3.2.12 Configuration

Table 32 specifies the requirement for the secure production mode.

Table 32: Requirement SR-CFG-01: secure production mode

SR-CFG-01:	Ensuring the application is running in production mode
Description	All system components must have debugging modes disabled, with no access to development consoles or non-production tools to prevent security risks and performance issues.
Recommendation	Verify that deployment configurations use the correct production profiles. Ensure debug logs and developer tools are disabled in production. Automate the enforcement of production-ready settings to avoid misconfigurations.

Table 33 specifies the requirement for the secure headers to add.

Table 33: Requirement SR-CFG-02: secure headers

SR-CFG-02:	Ensuring security headers are properly configured
Description	It is required to enforce security headers to mitigate common web vulnerabilities. This includes correctly setting the Content-Type, implementing a Content Security Policy (CSP), preventing XSS attacks, ensuring responses include X-Content-Type-Options, and configuring the Referrer-Policy header.
Recommendation	Implement security policies to enable, XSS Protection to prevent script injection attacks and Content Security Policy (CSP) to restrict allowed content sources.

Table 34 specifies the requirement for the Cross Origin Resource Sharing (CORS) configuration.

Table 34: Requirement SR-CFG-03: CORS configuration

SR-CFG-03:	Ensuring response headers are validated
Description	The application must validate HTTP methods, ensuring it only accepts the expected request types while blocking unauthorized ones. Additionally, CORS (Cross-Origin Resource Sharing) must be properly configured to prevent security risks.
Recommendation	Configure CORS with the following security policies, Allowed Origin: "https://unir-sc.com" (Restrict access to trusted sources). Allowed Methods: GET, POST, PATCH, DELETE (Only permit necessary HTTP methods). Allowed Headers: Content-Type, Origin, Accept, Authorization.

Table 35 specifies the requirement for the secure deployment.

Table 35: Requirement SR-CFG-04: secure deployment

SR-CFG-04:	Ensuring deployment validation
Description	The authenticity of deployment images generated must be verified to prevent tampering and security vulnerabilities. Additionally, the build process must follow secure compilation practices.
Recommendation	Use digitally signed and verified images to ensure authenticity. Implement minimal-privilege container images, restricting unnecessary permissions and reducing the attack surface. Regularly scan deployment images for vulnerabilities before deployment.

Table 36 demonstrates the direct alignment between methodology requirements and GDPR provisions, enabling systematic compliance verification through technical implementation.

Table 36: GDPR compliance mapping

Security requirement	GDPR article	Compliance provision
SR-SC-01	Article 25(1)	Data minimisation by design
SR-SC-02	Article 32(1)(a)	Encryption of personal data
SR-DP-01	Article 25(2)	Data protection by default
SR-DP-02	Article 32(1)(b)	Confidentiality safeguards
SR-COM-01	Article 32(1)(b)	Integrity of processing systems
SR-AUT-01-03	Article 32(4)	Authorization controls
SR-DB-01	Article 32(2)	Prevention of data alteration

3.3 Implementation and Validation of the Methodology

In this section, the development process of the proof of concept (PoC) implementation is detailed. The primary objective was to validate the proposed security requirements through practical application and testing scenarios. The decision to proceed with this approach was driven by the need to effectively test and validate the proposed security requirements in a real-world environment.

The selection of this methodology allows for a thorough validation of the proposed framework and its underlying principles. To build an effective proof of concept, it was determined that developing a web application would be the most suitable approach. This web application needed to incorporate at least the minimal functional requirements necessary to enable the implementation and testing of the non-functional requirements described in the previous section.

Given these considerations, the implementation of a ticket tracking application on Vue.js, Spring Boot, and MySQL architecture was proposed. This choice was strategic, as ticket management systems typically require robust security measures while maintaining user-friendly interfaces. Such an application provides an ideal testing ground for the security requirements, offering practical utility that closely mirrors real-world business needs and challenges.

The suggested application shall implement:

- Different screens for various functionalities.
- Management of multiple users with specific roles.
- Access restriction levels to information based on user roles.
- Main functionality for ticket management.
- Ability to generate and assign reports to support staff.

If implementation is required, the project repository can be found at the following URL. https://gitlab.com/paper_project/secure_development_methodology (accessed on 11 July 2025). with the following commit hash 1620f6e028c079f82ec54a54ab613f827946964b.

3.3.1 Implementation

Authentication Process

This sequence diagram illustrates the comprehensive authentication and session management process implemented in our secure web application using Keycloak as the identity provider (see Fig. 1). The workflow begins when a user accesses the application, triggering a secure authentication flow where credentials

are validated against Keycloak's secure user store. For enhanced security, the system implements Multi-Factor Authentication (MFA) through One-Time Password (OTP) verification when enabled, adding an additional layer of protection. Once authenticated, the application employs JWT tokens stored exclusively in memory (never in localStorage or sessionStorage) to prevent XSS vulnerabilities. When accessing protected resources, the system performs robust role-based authorization checks (SR-AC-01) before permitting data retrieval from the MySQL database. The diagram also demonstrates the secure session termination process (SR-SM-02), where logging out triggers proper token invalidation in Keycloak and complete removal of authentication data from memory, ensuring no session remnants can be exploited. This implementation aligns with industry best practices for secure authentication flows and session management in modern web applications.

Use Case—Report Generation

In the proposed ticket management system implements a role-based access control architecture with three distinct user levels, as shown in Fig. 2. Regular users can create and track their own support tickets, providing detailed descriptions of issues and monitoring resolution progress. Administrators have comprehensive system oversight, including the ability to assign tickets to appropriate support staff and manage user accounts. Support personnel focus exclusively on resolving their assigned tickets, updating status and providing solutions within their designated workload. This hierarchical structure ensures clear accountability, prevents unauthorized access to sensitive operations, and maintains an efficient workflow from initial ticket submission through final resolution. The system's design promotes scalability while maintaining security boundaries between different organizational roles.

System Architecture

The diagram presented in Fig. 3 illustrates the implementation of Clean Architecture integrated with Domain-Driven Design (DDD) principles, establishing an organizational structure that ensures separation of concerns and isolation of business rules. The system architecture for the ticket management proof of concept consists of four main components, each operating independently to support specific business logic tasks, resulting in a distributed system.

The architecture is divided into three concentric layers: the Domain layer at the core, containing Entities, Value Objects, and Repository interfaces that define the essential behavior of the system; the Application layer, which implements Use Cases that orchestrate the flow of business logic; and the Infrastructure layer, which houses technical components. This latter layer includes the User Interface developed with Vue.js, providing an interactive interface; the Business Logic (Backend) implemented in Spring Boot, managing application logic and handling frontend requests; the Persistence Component using MySQL for storing and managing data; and the Authentication and Security System based on Keycloak, handling authentication and user management.

To achieve system decoupling and simplify deployment, Docker was used to create an image for each component. Using component images enhances security across different development phases, such as release and testing stages. During the deployment phase, each component utilizes Nginx configured as a reverse proxy, designed to effectively route requests to the various components being managed by Docker. The reverse proxies contain the path where SSL certificates are hosted, which are managed by Certbot. This architectural organization ensures that dependencies always point inward, allowing the domain to remain completely independent of technical details.

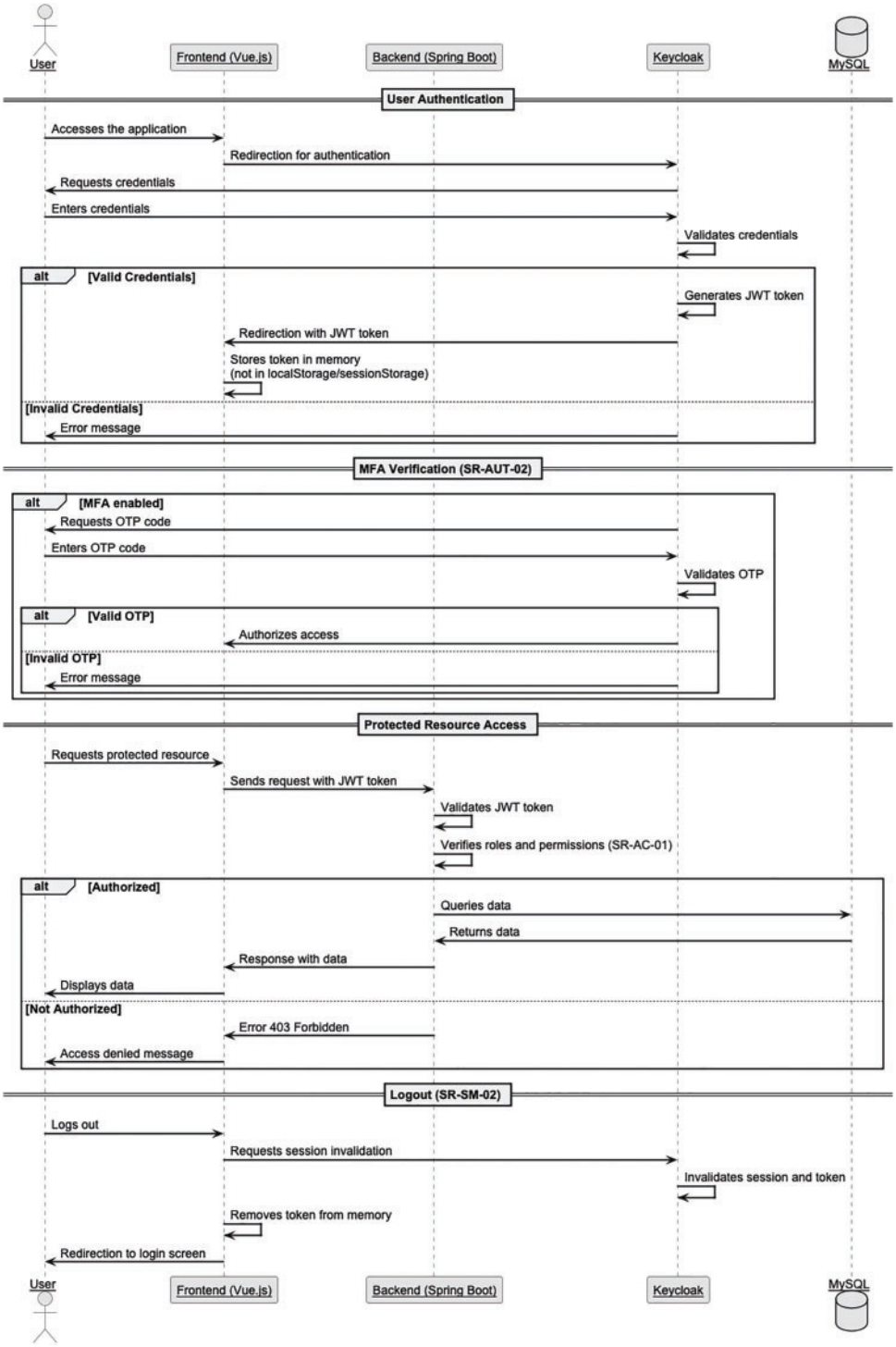

Figure 1: Keycloak authentication

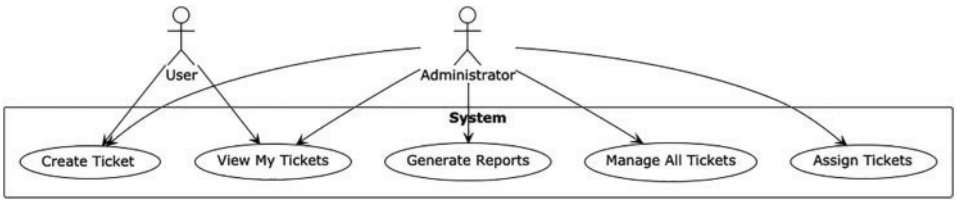

Figure 2: Use case of the proposed system

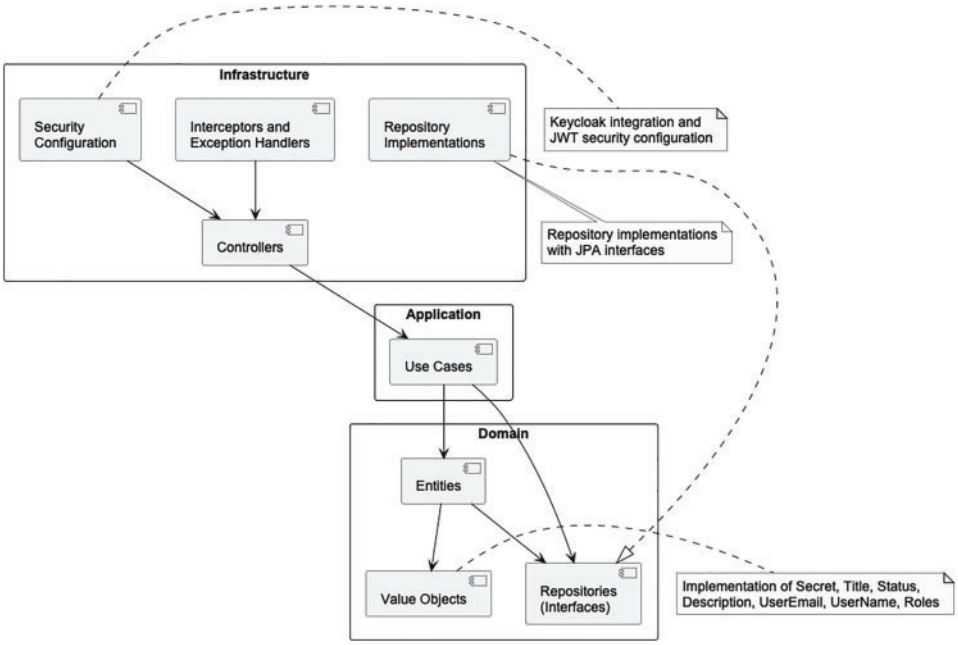

Figure 3: Components diagram for the proposed system

3.3.2 Security Requirements Implementation for VUE, Spring, MySQL Architecture

Authentication

Requirement SR-AUT-01 Password Security Verification implementation

The implementation addresses Table 1 by leveraging Keycloak's robust identity and access management capabilities for both Vue frontend and Spring Security backend.

The solution delegates password security verification to Keycloak's built-in security policies, which are configured to enforce:

- Minimum password length requirements;
- Character complexity rules (uppercase, lowercase, numbers, special characters);
- Real-time validation feedback to users during password creation.

Requirement SR-AUT-02 Multi-Factor Authentication (OTP) Implementation

The implementation is addressed Table 2. It is configured directly in Keycloak's graphical interface, which serves as the authentication service. Multi-factor authentication with OTP is enabled through Keycloak's administration console by activating:

- OTP policies using HMAC-SHA1 algorithm;
- Mandatory two-factor authentication in the authentication flow;
- Time-based OTP (TOTP) with 30-s validity.

Keycloak manages the entire process of OTP generation, validation, and delivery as an external service without requiring additional development.

Requirement SR-AUT-03 User Credential Management Implementation

The implementation addresses Table 3: It fully delegates credential management to Keycloak as a centralized IAM (Identity and Access Management) service.

The application does not store credentials in its own database, with Keycloak handling:

- Password storage using PBKDF2/SHA-256 hashing with unique salt per user;
- Security policy management (length, complexity, history);
- Automatic secure credential generation and storage.

All credential-related operations (authentication, updates, recovery) are exclusively managed by Keycloak as a specialized external service.

Session Management

Requirement SR-SM-01: Session Token Management Implementation

The implementation addresses Table 4 through the integration of Spring Security with Keycloak for session management. The configuration shown establishes a proper OAuth2/OIDC setup where:

- The Spring Security OAuth2 client is configured to use Keycloak as the identity provider.
- The authorization-grant-type: authorization_code ensures proper token exchange following secure authentication protocols.
- The JWT resource server configuration links to the same Keycloak realm, ensuring consistent token validation.
- The scope: OpenId parameter enforces standard OpenID Connect protocols for token management.

This configuration ensures that when a session is terminated (through expiration or logout), Keycloak's token management automatically handles invalidation at the server level, preventing token reuse in accordance with SR-SM-02 requirements. The Spring Security framework respects these invalidation events and enforces them across the application, protecting against session hijacking attempts. Listing 1 shows the Keyloack integration with Spring Security.

```
spring:
 security:
  oauth2:
   client:
    registration:
     keycloak:
      scope: openid
      authorization-grant-type: authorization_code
      client-id: security-concept
    provider:
     keycloak:
      issuer-uri: https://<keycloak.url>/realms/security-concept
   resourceserver:
    jwt:
     issuer-uri: https://<keycloak.url>/realms/security-concept...
```

Listing 1: Keycloak integration with Spring Security

Requirement SR-SM-02: Session Token Invalidation Implementation

The implementation addresses Table 5 through the integration of Keycloak with the Vue application. As outlined in Table 37, requirement SR-SM-02 defines how session tokens must be invalidated upon expiration or logout. The code initializes Keycloak with the configuration onLoad: 'login-required' and flow: 'standard', which ensures that:

- Keycloak automatically manages the token lifecycle, including invalidation upon expiration
- The standard authentication flow implements the security mechanisms recommended by OpenID Connect
- When logging out, Keycloak invalidates the token on the server, preventing its reuse
- The configuration checkLoginIframe: false avoids potential security risks related to continuous validation in iframes.

Table 37: Requirement SR-SM-02: session token invalidation

SR-SM-02:	Ensuring proper session invalidation upon expiry or logout
Description	When a session is terminated, whether due to expiration or explicit logout, a request must be sent to invalidate it, preventing unauthorised reuse.
Recommendation	Configure token policies to ensure they are invalidated after a defined period. The application should explicitly call the session invalidation function and ensure that the session cannot be reused. This enhances security by preventing session hijacking and unauthorised access.
Artifact/resource	Keycloak, Vue

Listing 2 shows this integrated approach. It ensures that sessions are properly invalidated according to security standards, preventing session hijacking attacks.

```
...import keycloak from './utilities/keycloak.ts';
keycloak.init({
  onLoad: 'login-required',
  enableLogging: true,
  checkLoginIframe: false,
  flow: 'standard'
}).then((authenticated: boolean) => {
  if (authenticated) {
    createApp(App).use(router).mount('#app');
  } else {
    window.location.reload();
  }
})...
```

Listing 2: Keycloak integration and authentication flow in the Vue Application

Requirement SR-SM-03: Session Token Configuration Implementation

The implementation addresses Table 6 requisite by configuring Spring Security to use secure cookies for session management. The solution builds upon the OAuth2/OIDC setup already described in Table 38 (SR-SM-01) by adding specific cookie security properties.

The implementation secures session tokens by:

- Creating HTTP-only cookies that cannot be accessed via JavaScript, preventing XSS attacks;

- Setting the Secure flag to ensure cookies are only transmitted over HTTPS connections;
- Configuring SameSite = Strict to prevent cross-site request attacks;
- Integrating with the session invalidation mechanism described in Table 39 (SR-SM-02).

Table 38: Requirement SR-COM-02: secure server-originated communication

SR-COM-02:	Ensuring secure server-originated communication
Description	All server-originated communication must use TLS certificates issued by trusted Certificate Authorities (CAs). Additionally, internal communication across all ports and protocols must be encrypted, and any TLS failures should be logged and reported.
Recommendation	Enforce TLS for all API connections to prevent unencrypted data transmission.
Artifact/resource	Spring security

Table 39: Cookies analyzed from ZAP

Type of alerts	Context	Risk	False positive
SQL Injection	https://auth.unir-sc.com	HIGH	YES
Open Redirect	https://auth.unir-sc.com	HIGH	YES
Absence of anti CSRF tokens	https://auth.unir-sc.com	MEDIUM	NO
CSP: Wildcard Directive	https://unir-sc.com	MEDIUM	NO
CSP: Script-src unsafe-inline	https://auth.unir-sc.com	MEDIUM	NO
CSP: Style-src unsafe-inline	https://auth.unir-sc.com	MEDIUM	NO
ELMAH Information Leak	https://unir-sc.com	MEDIUM	YES
Lack of header Anti-Clickjacking	https://auth.unir-sc.com	MEDIUM	YES
Cookie No HttpOnly Flag	https://auth.unir-sc.com	LOW	NO
Cookie Without Secure Flag	https://auth.unir-sc.com	LOW	NO
Cookie with attribute SameSite set to None	https://auth.unir-sc.com	LOW	NO
Cookie without Same—Site attribute	https://auth.unir-sc.com	LOW	NO
Time stamp disclosure—UNIX	https://unir-sc.com	LOW	NO
Server Leaks Version Information	https://unir-sc.com	LOW	NO
Strict-Transport-Security Header	https://auth.unir-sc.com	LOW	NO

This configuration ensures that session tokens are properly protected throughout their lifecycle, from generation through transmission to invalidation, aligning with the security controls implemented for other authentication mechanisms in the system.

Access Control

Requirement SR-AC-01: Role-Based Access Implementation

The implementation addresses Table 7 requisite by applying method-level security through Spring Security's @PreAuthorize annotations. This approach ensures that resources are only accessible to users with appropriate roles.

The solution implements role-based access control at multiple levels:

- Backend controllers restrict method execution to specific roles using Spring Security annotations;
- Role information is extracted directly from the JWT token, integrating with the token management described in Table 4 (SR-SM-01);
- Frontend components dynamically adjust their visibility based on user roles, leveraging the Vue application's integration with Keycloak described in Table 5 (SR-SM-02).

This implementation follows the principle of the least privilege by ensuring that each endpoint and UI component explicitly checks for required roles before allowing access, creating a cohesive authorization system across both backend and frontend.

Listing 3 shows how to apply method-level security through Spring Security's @PreAuthorize annotations.

```
@PreAuthorize ("hasRole('user')")
public ResponseEntity<ReportResponseDTO> createPost(
    @RequestBody ReportRequest reportRequest,
    @AuthenticationPrincipal Jwt jwt)
{...
```

Listing 3: Role-based access control implementation using Spring Security

Requirement SR-AC-02: CSRF Protection Implementation

The implementation addresses the CSRF protection requirement (Table 8) through Spring Security's built-in CSRF defense mechanisms. The solution configures Spring Security to use a Cookie-based CSRF token repository that allows JavaScript access to the tokens.

This implementation ensures that all state-changing requests are protected against Cross-Site Request Forgery attacks. The system generates unique CSRF tokens that must be included in every non-GET request. The tokens are stored in cookies accessible to the frontend Vue application, which can retrieve and include them in subsequent requests.

Spring Security automatically validates these tokens on the server side for each request, rejecting any requests without valid tokens. This prevents malicious websites from tricking authenticated users into performing unwanted actions, as those external sites cannot access the CSRF token required to make valid requests. Listing 4 shows how to add CSRF token.

```
...@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http)throws Exception
{
  http.csrf(csrf -> csrf
      .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
       )...
  return http.build();
}...
```

Listing 4: CSRF protection configuration using Spring Security with Cookie-based token repository

Validation, Satinization, Encoding

Requirement SR-VSE-01: Entry Points Protection Implementation

The implementation addresses Table 9 requisite through a robust domain validation approach using Java records with built-in validation logic. The UserEmail class is designed as an embeddable JPA component that enforces structural validation of email addresses before they can be used within the application.

This implementation satisfies the requirement by:

- Validating input structure through explicit checks (null/empty, presence of '@' and '.');
- Throwing exceptions for invalid data, preventing malformed inputs from entering the system;
- Using Java's record feature to create immutable data objects that cannot be modified after validation;
- Leveraging JPA's @Embeddable annotation to ensure validated objects are used consistently in entity mappings.

The approach implements the recommended input validation for incoming data objects, ensuring proper formatting and preventing HTTP parameter pollution attacks by rejecting any invalid or unexpected email formats before they can be processed by the application. Listing 5 shows Domain-level email validation using Java records and JPA embeddable components.

```
@Embeddable
public record UserEmail(String email) {
  public UserEmail
  {
    if (email == null || email.isEmpty())
    {
      throw new IllegalArgumentException("Email cannot be null or empty");
    }
    if (! email.contains("@"))
    {
      throw new IllegalArgumentException("Email must contain @");
    }
    if (! email.contains("."))
    {
      throw new IllegalArgumentException("Email must contain .");
    }
  }
}
```

Listing 5: Domain-level email validation using Java records and JPA embeddable components

Requirement SR-VSE-02: Input Data Sanitation Implementation

The implementation of this requisite (Table 10) leverages Vue.js's built-in security features for preventing malicious code execution. Vue automatically escapes HTML content in templates and interpolations, making it resistant to most XSS attacks by default.

The solution takes advantage of Vue's key security mechanisms:

Vue's template system automatically escapes HTML content when using double curly braces ({{ }}) for data binding, preventing injection of executable code.

For WYSIWYG editors and rich text input fields, the application employs Vue-compatible sanitization libraries that strip potentially dangerous HTML tags and attributes while preserving legitimate formatting.

All user-uploaded files are processed through validation pipelines that enforce strict MIME type checking and never directly execute or include uploaded content in sensitive contexts.

The implementation follows Vue's security best practices by avoiding the use of dangerous directives like v-html with untrusted content, and when rich HTML rendering is required, content is pre-sanitized using dedicated libraries.

Server-side validation complements the frontend protections, ensuring that even if client-side validations are bypassed, malicious input is detected and neutralized before processing.

Requirement SR-VSE-03: Output Data Sanitation Implementation

The implementation addresses Table 11 requisite by leveraging Vue's built-in output sanitization features. Vue automatically escapes all data interpolated through double curly braces ({{ }}), preventing XSS attacks without additional code. The application follows Vue's security best practices by:

- Using Vue's automatic HTML escaping for all dynamic content display
- Implementing Vue-i18n for proper localization that respects user regional settings
- Avoiding raw HTML rendering (v-html) with untrusted content
- Using content security policies to provide additional defense-in-depth

This approach ensures all output data is properly sanitized before rendering while maintaining appropriate localization based on user preferences, satisfying both security and usability requirements.

Requirement SR-DB-01: Data Queries Protection Implementation

The implementation satisfies Table 12 requisite by utilizing Spring Data JPA's repository pattern. By extending JpaRepository, the interface automatically implements SQL injection protection through parameterized queries and Hibernate's ORM capabilities. This approach eliminates direct query manipulation and ensures all database interactions—including the custom finder method—are secure by default, meeting the requirement for protected data queries without requiring manual query sanitization. Listing 6 shows how to secure data query handling using Spring data JPA and ORM-based injection prevention.

```
public interface CustomerInterfaceJPA extends JpaRepository<Customer, UUID> {
   Optional<Customer> findCustomerByEmail(UserEmail email);
}
```

Listing 6: Secure data query handling using Spring data JPA and ORM-based injection prevention

Requirement SR-DB-02: Secure Database Connection Implementation

The implementation of this requisite (Table 13) utilizes Nginx as a reverse proxy to enforce secure database connections. Nginx is configured to act as a secure gateway, only allowing database connections from known and authorized URLs/IP addresses.

The solution implements multiple layers of security:

- Nginx reverse proxy restricts database access to only pre-approved application servers by IP filtering;
- All database connections are forced through TLS/SSL encryption;
- The MySQL database is configured to reject non-encrypted connection attempts;
- Database user accounts operate with minimal required privileges following least privilege principles;
- A dedicated non-root database service account is used for running the MySQL instance;

The Nginx configuration limits MySQL traffic to specific known sources "whitelisted", functioning as an additional security layer beyond MySQL's native access controls. Listing 7 shows how to Secure database access via Nginx Reverse Proxy with IP whitelisting and TLS enforcement.

```
location /mysql-proxy {
   allow 192.168.1.100;
   allow 192.168.1.101;
   deny all;
   proxy_pass 127.0.0.1:3306;
   proxy_ssl on;
}
```

Listing 7: Securing database access via Nginx Reverse Proxy with IP whitelisting and TLS enforcement

Stored Cryptography

Requirement SR-SC-01: Classifying Information Implementation

The implementation addresses Table 14 requisite by configuring MySQL with transparent data encryption through the keyring_file plugin. This command-line configuration enables database-level encryption for sensitive data, implementing server-side protection for stored information. By setting up the keyring file storage location and enabling native password authentication, the system ensures that sensitive information is automatically encrypted at rest without requiring application-level changes. This approach satisfies the requirement for applying appropriate encryption mechanisms to protect classified information from unauthorized access. Listing 8 shows how configuring MYsql with Transparent data encryption in MySQL using keyring file plugin.

```
command: --default-authentication-plugin=mysql_native_password --early-plugin-load=keyring_file.so --
keyring_file_data=/var/lib/mysql-keyring/keyring
```

Listing 8: Transparent data encryption in MySQL using keyring file plugin

Requirement SR-SC-02: Cryptographic Modules Implementation

The implementation of this requisite (Table 15) delegates core cryptographic operations to Keycloak as the primary identity and access management solution. For user authentication and credential management, Keycloak handles all sensitive cryptographic operations using industry-standard algorithms.

For application-specific cryptographic needs outside of identity management, the system leverages Spring Security's built-in cryptographic modules:

- Password encoding utilizes Spring Security's BCryptPasswordEncoder for any local password handling requirements, with configurable work factor to balance security and performance;
- Secure random data generation is implemented through Java's SecureRandom class as recommended by Spring Security, ensuring cryptographically strong entropy for tokens and keys.

While Keycloak manages the majority of security-critical operations, these Spring Security cryptographic modules provide robust protection for application-specific security requirements, ensuring the use of modern, secure cryptographic standards throughout the system.

Requirement SR-SC-03: Secure Storage of Keys Implementation

The implementation of this requisite (Table 16) addresses the secure storage of keys by leveraging Keycloak as the dedicated security module for credential and key management. This approach ensures that no sensitive credentials or keys are stored within the application itself.

Keycloak provides a robust key management system that handles the secure storage of all cryptographic materials, including OAuth tokens, signing keys, and encryption keys. The application retrieves necessary authentication tokens at runtime through secure OAuth/OIDC flows, never directly accessing or storing the underlying cryptographic keys.

For application-specific secrets (such as API keys and database credentials), the system uses environment variables supplied at runtime through the deployment pipeline, completely avoiding hardcoded secrets in the codebase or configuration files.

This implementation fully satisfies the requirement by ensuring all sensitive keys remain within specialized secure storage solutions and are never exposed to the application layer.

Error Handling and Logging

Requirement SR-EHL-01: Secured Logging Implementation

The implementation satisfies Table 17 requisite through a security-focused class that prevents sensitive data exposure in logs. By overriding toString() to return only masked content ("******") and implementing secure memory handling, the code ensures sensitive information is never inadvertently logged. This implementation directly addresses the requirement to protect sensitive data while maintaining useful application logs. Listing 9 shows how to Prevent sensitive data exposure through secure logging and memory handling.

```
...@Override
  protected void finalize() throws Throwable
  {
    Arrays.fill(value, (byte) 0);
    super.finalize();
  }
  public <T> T use(Function<? Super byte[], ? extends T> f)
  {
    byte[] copy = value.clone();
    try
    {
      return f.apply(copy);
    } finally
    {
      Arrays.fill(copy, (byte) 0);
    }
  }
  @Override
  public String toString()
  {return "******"; }...
```

Listing 9: Preventing sensitive data exposure through secure logging and memory handling

Requirement SR-EHL-02: Synchronized Timestamps Implementation

The implementation satisfies Table 18 requisite by utilizing Java's Instant class, which represents a precise moment on the UTC timeline. By storing timestamps as Instant objects in the database (marked as non-nullable), the application ensures all event timestamps are consistently recorded in UTC regardless of server location or local settings.

This approach provides several key benefits:

- Eliminates timezone ambiguity in logged events;
- Ensures timestamp consistency across distributed systems;
- Simplifies server migrations across different regions;
- Facilitates accurate chronological sorting and comparison.

When displaying timestamps to users, the application can convert these UTC instants to local time zones as needed, balancing backend consistency with frontend usability. This implementation directly addresses the requirement for synchronized timestamps while maintaining flexibility for user-friendly time representation. Listing 10 shows UTC-based timestamp management using Java instant for consistent event logging.

```
import java.time.Instant;
@Column (nullable = false) private Instant creationDate;
```

Listing 10: UTC-based timestamp management using Java instant for consistent event logging

Requirement SR-EHL-03: Appropriate Error Messages Implementation

The implementation addresses Table 19 requisite by establishing a centralized exception handling system using Spring's @ControllerAdvice. It categorizes exceptions into three types (unexpected, validation, and business) with distinct handling strategies for each. The system generates unique error references for unexpected exceptions, enabling support teams to trace issues while exposing only generic messages to users. Detailed error information including stack traces is logged internally but never revealed in responses. This approach balances security with usability by providing appropriate feedback based on error type while ensuring sensitive implementation details remain protected. Listing 11 shows Centralized exception handling with @ControllerAdvice for secure and structured error management.

Data Protection

Requirement SR-DP-01: Client-Side Information Protection Implementation

The implementation of this requisite (Table 20) uses Keycloak JavaScript adapter (keycloak-js v24.0.4) to protect client-side information in the Vue application. The configuration:

- Stores authentication tokens in memory only, not in localStorage or sessionStorage;
- Uses standard Authorization Code flow to prevent token exposure in URLs;
- Requires authentication before mounting the application;
- Automatically clears all tokens upon logout or session expiration.

```
@ControllerAdvice
@Slf4j
public class ExceptionHandlerConfig {
  private static final String GENERIC_ERROR_MESSAGE = "An unexpected error occurred. Please contact support.";
  private static final String VALIDATION_ERROR_MESSAGE = "The provided data is invalid.";
  @ExceptionHandler(Exception.class)
  public ResponseEntity<ErrorResponse> handleException(Exception ex, HttpServletRequest request) {
    String errorRef = generateErrorReference();
    log.error("Error reference: {} - Request URI: {} - Exception: {}",
        errorRef, request.getRequestURI(), ex.getMessage(), ex);
    var error = new ErrorResponse("ERR-GENERAL", GENERIC_ERROR_MESSAGE, errorRef);
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(error);
  }
  @ExceptionHandler({MethodArgumentNotValidException.class, BindException.class})
  public ResponseEntity<ErrorResponse> handleValidationExceptions(Exception ex) {
    log.warn("Validation error: {}", ex.getMessage());
    var error = new ErrorResponse("ERR-VALIDATION", VALIDATION_ERROR_MESSAGE, null);
    return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(error);
  }
  @ExceptionHandler(BusinessException.class)
  public ResponseEntity<ErrorResponse> handleBusinessException(BusinessException ex) {
    log.info("Business exception: {}", ex.getMessage());
    var error = new ErrorResponse(ex.getErrorCode(), ex.getUserMessage(), null);
    return ResponseEntity.status(ex.getStatus()).body(error);
  }
  private String generateErrorReference() {
    return UUID.randomUUID().toString().substring(0, 8).toUpperCase();
  }
  public record ErrorResponse(String errorCode, String message, String reference) {
  }
}
```

Listing 11: Centralized exception handling with @ControllerAdvice for secure and structured error managements

These measures ensure no sensitive information persists in browser storage or the DOM, satisfying the requirement for client-side information protection. Listing 12 shows how to Secure frontend sessions with Keycloak-JS and authorization code flow in Vue.

Requirement SR-DP-02: Sensitive Information Protection Implement

The implementation addresses Table 21 requisite through Spring's comprehensive security features:

- All sensitive data transmission occurs over TLS/HTTPS, enforced through Spring Security configuration as detailed in Table 23 (SR-COM-02)
- Sensitive data objects implement memory-clearing techniques similar to those shown in Table 17 (SR-EHL-01), automatically overwriting data when no longer needed
- Access to sensitive information is tracked through AOP-based logging aspects that record details in a secure audit log, aligning with the logging requirements in Table 17 (SR-EHL-01)
- Database interactions for sensitive fields leverage the encryption capabilities configured in Table 14 (SR-SC-01), ensuring data is protected at rest

```
import Keycloak from 'keycloak-js';
import {url_auth} from "./varMode.ts";
const keycloak = new Keycloak({
  url: url_auth,
  realm: 'security-concept',
  clientId: 'security-concept',
});
export default keycloak;
```

Listing 12: Securing frontend sessions with Keycloak-JS and authorization code flow in Vue

This implementation ensures sensitive information is encrypted during transmission, properly cleared when no longer needed, and all access is logged for audit purposes.

Communication

Requirement SR-COM-01: Secure Client-Originated Communication Implementation

The implementation addresses Table 22 requisite through Certbot's automated TLS certificate management integrated with an Nginx reverse proxy. This configuration enforces TLS 1.2/1.3 exclusively while blocking older protocols. The reverse proxy acts as a security layer, handling certificate termination and implementing strict cipher policies. All client traffic passes through this proxy, ensuring encryption, proper redirection from HTTP to HTTPS, and HSTS enforcement—creating a robust security barrier that protects communications from interception or tampering.

Requirement SR-COM-02: Secure Server-Originated Communication Implementation

The implementation addresses Table 23 requisite by configuring Spring Security to enforce TLS for all server-originated communications. The system uses trusted CA-issued certificates for external APIs and mutual TLS authentication for internal service communication. All microservices are configured to reject non-encrypted connections, with automatic TLS failure logging to a centralized monitoring system. The implementation applies consistent security policies across all environments through infrastructure-as-code, ensuring that development, staging, and production maintain identical TLS configurations.

Code Base

Requirement SR-CB-01: Defensive Programming Implementation

The implementation addresses Table 24 requisite through functional programming techniques that enhance security. The code demonstrates defensive programming by:

- Using immutable data structures with @FieldDefaults(makeFinal = true) to prevent state manipulation;
- Implementing the Either monad pattern (isRight()) for explicit error handling without exceptions;

- Employing a declarative method of chaining without intermediate variables, eliminating state-based vulnerabilities;
- Leveraging pure functions and immutability to create predictable execution flows resistant to tampering.

This functional approach eliminates common vulnerability vectors by removing mutable state, preventing null reference errors, and enforcing clear separation between success and failure paths, all without relying on variable state that could be compromised. Listing 13 shows how to Secure functional programming practices in Java using immutability and the either monad pattern.

Requirement SR-CB-02: Secure Data Structures Implementation

The implementation addresses Table 25 requisite through secure data structures by:

- Using immutable record type for DTOs (CustomerResponse) to prevent unauthorized modification of data after creation;
- Implementing functional programming with monadic operations (map().map().getOrElse()) that provide safer data transformation without mutable state;
- Applying Domain-Driven Design with value objects (customer names and emails are accessed through domain methods like .name() and .email());
- Creating controlled data mapping through a dedicated static factory method (getResponseFrom) that filters sensitive data;
- Enforcing authorization boundaries with Spring Security (@PreAuthorize("hasRole('admin')")), adding a layer of access control.

```
@FieldDefaults (makeFinal = true, level = AccessLevel.PRIVATE)
public class CreateUserController {
  UserCreationUseCase userCreationUseCase;
  @PostMapping ("/create")
  public ResponseEntity<String> createUser(
      @RequestBody Customer customer)
  {
    return userCreationUseCase.createCustomer(customer)
                    .isRight() ? ResponseEntity.ok()
                                  .build()
                          : ResponseEntity.badRequest()
                                  .body("no_created");
  }
}
```

Listing 13: Secure functional programming practices in Java using immutability and the either monad pattern

This approach creates a secure pipeline that transforms domain objects into limited-exposure DTOs through pure functions, ensuring that sensitive data is properly filtered, and only authorized users can access the information. Listing 14 shows how to accomplish a functional and domain-driven secure data mapping with authorization enforcement in Java.

Requirement SR-CB-03: Avoid Hardcoded Credentials in the Source Code Implementation

The requisite of Table 26 is addressed by centralizing all connection strings to ensure no hardcoded strings containing sensitive information are inadvertently exposed within the codebase. This centralization facilitates comprehensive verification that no data leakage occurs through embedded credentials or configuration parameters. Implementation is performed by Spring, Vue artifacts.

```
@FieldDefaults (makeFinal = true, level = AccessLevel.PRIVATE)
public class GetAllUserController {
  UserGetAllUseCase userGetAllUseCase;
  @GetMapping ("")
  @PreAuthorize ("hasRole('admin')")
  public ResponseEntity<List<CustomerResponse>> getAllUser(
      @RequestParam String role)
  {
    return userGetAllUseCase.getAllCustomers(role.isEmpty() ? null : "ROLE_" + role)
                .map(this :: createResponseObject)
                .map(this :: createResponseEntity)
                .getOrElse(ResponseEntity.badRequest()
                        .build());
  }
  private List<CustomerResponse> createResponseObject(List<Customer> customers)
  {
    return customers.stream()
            .map(CustomerResponse :: getResponseFrom)
            .toList();
  }
  private ResponseEntity<List<CustomerResponse>> createResponseEntity(List<CustomerResponse>
customerResponses)
  {
    return ResponseEntity.ok()
                .body(customerResponses);
  }
  record CustomerResponse(
      UUID id,
      String firstName,
      String email,
      List<Roles> roles
  ) {
    public static CustomerResponse getResponseFrom(Customer customer)
    {
      return new CustomerResponse(customer.getUuid(),
                    customer.getName()
                        .name(),
                    customer.getEmail()
                        .email(),
                    customer.getRoles());
    }
  }
}
```

Listing 14: Functional and domain-driven secure data mapping with authorization enforcement in Java

Web Services and API

Requirement SR-WS-01: Role-Based Access Control Implementation

The implementation addresses Table 27 requisite through a basic Spring Security configuration that enforces authentication for all requests. While this establishes the foundation for secure access, it requires additional role-based authorization rules and method-level security to fully implement the permission-based model specified in the requirement. The current implementation satisfies the authentication requirement but needs enhancement to complete the role-based access control functionality. Listing 15 shows how to accomplish a baseline authentication enforcement using Spring Security's global request protection.

Requirement SR-WS-02: Security in Web Services Implementation

The implementation addresses Table 28 requisite by establishing consistent security standards across all exposed web services. The solution leverages several components already implemented in other parts of the system:

- Response filtering is implemented through the DTO pattern demonstrated in Table 25 (SR-CB-02), where CustomerResponse records filter sensitive information from domain objects before returning data to clients;
- Secure error handling is provided through the global exception handler shown in Table 19 (SR-EHL-03), which categorizes exceptions into three types (unexpected, validation, and business) with distinct handling strategies for each.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{
  http .authorizeHttpRequests(authorize -> authorize
          .anyRequest()
          .authenticated())
}
```

Listing 15: Baseline authentication enforcement using Spring Security's global request protection

This implementation ensures all exposed services follow consistent security standards by reusing proven patterns established throughout the application, providing a unified approach to data encoding, information filtering, and error handling across all web services.

Requirement SR-WS-03: Service Schemas Validation Implementation

Implement input object validations are implemented to verify that received data matches the expected schema. Use whitelists for fields that should only accept specific values, ensuring controlled data input. Apply regular expressions to each data field to filter out invalid or potentially harmful input, preventing injection attacks or malformed data. Implementation is performed by Spring Web artifact.

Requirement SR-WS-04: CSRF Protection Implementation

Our implementation utilizes Spring Security's comprehensive CSRF protection framework to defend against cross-site request forgery attacks. The configuration establishes multiple defensive layers including CORS policy enforcement, secure HTTP headers, and referrer policy validation. By configuring frame options to same origin, enabling XSS protection in block mode, and implementing strict Content Security Policy directives, we protect session cookies from unauthorized cross-origin requests. Additionally, the implementation enforces HTTP Strict Transport Security with subdomains inclusion and preloading, creating a robust security posture against CSRF vulnerabilities in alignment with modern web security best practices. Listing 16 shows how to configure an advanced CSRF mitigation in Spring Security with CORS, CSP, and Secure HTTP header configuration.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception
{
  http .cors(httpSecurityCorsConfigurer ->
httpSecurityCorsConfigurer.configurationSource(corsConfigurationSourceLocal())).headers(headers -> {
        headers.frameOptions(HeadersConfigurer.FrameOptionsConfig :: sameOrigin);
        headers.xssProtection(xss ->
xss.headerValue(XXssProtectionHeaderWriter.HeaderValue.ENABLED_MODE_BLOCK));
        headers.contentSecurityPolicy(csp -> csp.policyDirectives(POLICY_DIRECTIVES));
        headers.referrerPolicy(referrer -> referrer.policy(ReferrerPolicyHeaderWriter.ReferrerPolicy.SAME_ORIGIN));
        headers.httpStrictTransportSecurity(hsts -> hsts.includeSubDomains(true).preload(true)
.maxAgeInSeconds(MAX_AGE_IN_SECONDS));
    })
}
```

Listing 16: Advanced CSRF mitigation in Spring Security with CORS, CSP, and Secure HTTP header configuration

Requirement SR-WS-05: Secure Content Implementation

Request headers are validated to enforce Content-Type: application/json, ensuring that only properly formatted data is processed. Ensure that all headers and messages are transmitted over TLS encryption to prevent interception or tampering. This requisite es implemented by Spring Security artifact.

Configuration

Requirement SR-CFG-01: Secure Production Mode Implementation

The implementation addresses Table 3 requisite through environment-specific configuration variables embedded in the CI/CD pipeline artifacts. This approach ensures:

- Production-specific variables are automatically injected during the image/artifact creation phase of the pipeline;
- Development and debugging features are programmatically disabled when deploying to production environments;
- Each technology stack (Spring, Vue, MySQL) receives appropriate production configurations:
  - Spring: Production profiles with disabled debug logging and dev tools;
  - Vue: Minified builds with disabled Vue Devtools and source maps.

By integrating these controls directly into the build pipeline rather than relying on manual configuration, the system enforces production-ready settings automatically, eliminating human error and potential security vulnerabilities from misconfiguration.

Requirement SR-CFG-02: Secure headers Implementation

As specified in requisite of Table 33, our Spring Security implementation comprehensively addresses the requirement for protecting requests against Cross-Site Request Forgery (CSRF) attacks. The configuration implements multiple defensive mechanisms to ensure that cookies used for session management are properly protected.

Our implementation follows the recommendations in Table 33 by:

- Origin Validation: The CORS configuration restricts which domains can make requests to our application, working alongside the referrer policy set to SAME_ORIGIN to verify that requests come only from trusted sources.
- Cookie Protection: While not shown in the code snippet, our implementation configures cookies with the SameSite attribute set to Lax, preventing them from being sent in cross-origin requests, a key recommendation from Table 33.
- CSRF Tokens: Our implementation uses Spring Security's CSRF token mechanism which generates and validates tokens for each session, ensuring that requests cannot be forged from external sites.
- Additional Security Layers: The implementation also includes complementary security headers that help mitigate related vulnerabilities, creating a defense-in-depth approach against various cross-origin attacks.

Requirement SR-CFG-03: CORS Configuration Implementation

According to Table 33, CORS is configured with the following security policies, Allowed Origin: "https://unir-sc.com" (Restrict access to trusted sources). Allowed Methods: GET, POST, PATCH, DELETE (Only permit necessary HTTP methods). Allowed Headers: Content-Type, Origin, Accept, Authorization. Spring security artifact is used.

Requirement SR-CFG-04: Secure deployment Implementation

The implementation addresses Table 35 through a comprehensive container security pipeline. All deployment images are built using multi-stage builds with distroless base images to minimize attack surface. The build process implements:

- Digital signature verification using Docker Content Trust to ensure image authenticity;
- Rootless containers running with non-privileged users and minimal capabilities;
- Automated vulnerability scanning with Trivy integrated in the CI/CD pipeline;
- Immutable infrastructure patterns preventing runtime modifications;
- Container image verification with admission controllers before deployment.

This approach ensures deployment images are validated for authenticity, follow secure compilation practices, and maintain minimal privilege principles throughout the container lifecycle.

3.3.3 Applied Paradigms

Although Java was conceived as an object-oriented language, this project incorporated functional programming concepts such as immutability and pure functions, reflected in the class structure. The data flow was designed to handle requests and behaviours as streams, leveraging the language's functional capabilities. Spring Boot supports this through automatic dependency injection. Communication between aggregates is implemented through well-defined interfaces, providing a detailed description of behaviour and ensuring coherent and flexible interaction between system components.

To implement code patterns that follow secure programming principes and could be reused and implemented in a holistic way for any team members, classes have been created as Secret. Such as class provides a default secure approach, as the data which is saved in this data structure has a closed scope, which precludes the accidentality lack of data by mistakes or misuses.

The Secret class implements a secure pattern for handling sensitive data using an immutable private byte array. It provides controlled access through the 'use()' method which creates a temporary data copy that is automatically cleared after use. The class prevents memory leaks through the 'finalize()' method that overwrites data with zeros and protects against accidental exposure via a 'toString()' that masks the actual content. Being a final class with immutable members ensures sensitive data remains encapsulated and is properly cleaned up. Listing 17 shows the secret class implementation for secure programming.

3.3.4 Domain-Specific Language in VUE Requests

The requests made by the Front-end component with TypeScript define a Domain-Specific Language (DSL) to perform HTTP requests with GET, POST, and DELETE methods to the protected Backend API. The DSL focuses on providing a fluid and secure interface to build these requests, implementing defensive programming (Requirement SR-CB-01). Keycloak is imported to handle authentication, ensuring that all requests include a valid token in the authorization header. This provides the ability to use the token centrally in a highly restricted scope. The backend base URL is obtained from an environment variable to facilitate configuration in different environments.

As shown in example 9, the MET constant defines the allowed HTTP methods, and the METType type ensures that only these methods are used, providing type safety. It's interesting to note that the requests handle request bodies appropriately by serializing to JSON only when necessary. Additionally, the use of TypeScript ensures that any type errors are caught at compile time, reducing the possibility of runtime errors. This provides developers with an abstraction layer for API usage that they can use simply, allowing typing errors and misuse of sensitive data, such as the access token, to be reduced.

```java
import java.util.Arrays;
import java.util.function.Function;

public final class Secret {
  private final byte[] value;
  public Secret(byte[] value)
  {
    this.value = value;
  }
  @SuppressWarnings ("removal")
  @Override
  protected void finalize() throws Throwable
  {
    Arrays.fill(value, (byte) 0);
    super.finalize();
  }
  public <T> T use(Function<? super byte[], ? extends T> f)
  {
    byte[] copy = value.clone();
    try
    {
      return f.apply(copy);
    } finally
    {
      Arrays.fill(copy, (byte) 0);
    }
  }
  @Override
  public String toString()
  {
    return "******";
  }
}
```

Listing 17: Secret class implementation for secure programming

The implementation creates a typed HTTP client that centralizes requests and authorization token management in a single point. Using a builder pattern, it reuses base configuration (headers, Bearer authentication, JSON handling) while allowing customization of path, HTTP method, and body in a chained manner. This prevents code duplication in requests and keeps authentication logic centralized, ensuring all requests follow the same security and error handling standards. Listing 18 shows DSL implementation in TypeScript and Listing 19 shows a DSL usage example.

3.4 Security Validation

3.4.1 Pipeline Implementation

The CI/CD pipeline implemented for the ticket management system emphasizes security throughout the development lifecycle, incorporating multiple security scanning stages before any code reaches production. The pipeline is structured into two main stages: Quality Assurance (QA) and Build, with each stage containing specialized tasks designed to ensure both code quality and security (see Fig. 4).

In the QA stage, the pipeline begins with concurrent code quality and security analysis. For the backend, SonarCloud performs static code analysis using Maven, generating code coverage reports with JaCoCo to identify potential vulnerabilities and ensure high-quality code. Similarly, the frontend undergoes SonarCloud scanning using the dedicated scanner CLI, with results cached to optimize future scans.

Security scanning is enhanced through multiple specialized tools. Trivy performs comprehensive filesystem scans of both frontend and backend codebases, specifically targeting critical severity vulnerabilities and generating standardized reports in GitLab SAST format. This is complemented by Bearer scanning, which focuses on identifying data security and privacy issues in both components, providing additional security insights through detailed SAST reports.

The pipeline is configured to run automatically on merge requests and main branch commits, with certain jobs allowing manual triggering. Strategic use of artifacts ensures that security reports are preserved for future reference. The entire process implements security-by-design principles, with multiple scanning tools providing overlapping coverage to minimize the risk of vulnerabilities reaching production. This comprehensive approach to DevSecOps integrates security testing directly into the development workflow, enabling early detection and remediation of potential security issues.

3.4.2 Software Composition Analysis (SCA)

The dependency on external libraries and components in software development introduces security risks that, if not properly managed, can compromise the integrity of the system. This study presents the use of dependency analysis tools such as OSV-Scanner, Trivy and OWASP Dependency-Check for the detection and mitigation of vulnerabilities in software projects. These tools allow early identification of risks and are effectively integrated in CI/CD environments, contributing to the security of the development life cycle.

```
const MET = ['GET', 'POST', 'DELETE'] as const;
type METType = typeof MET[number];
export const requestwithPath = (path: string) => {
  return {
    withMethode: (methode: METType) => {
      return {
        withBody: async (body: unknown | null | undefined) => {
          const resp: Response = await fetch(url_back + path, {
            method: methode,
            headers: {
              "Authorization": `Bearer ${keycloak.token}`,
              "Content-Type": "application/json",
              "Accept": "application/json",
            },
            body: body ? JSON.stringify(body) : null,
          })
          if (!resp.ok) {
            return;
          }
          return {
            resp: resp.json()
          }
        }
      }
    }
  }
}
```

Listing 18: DSL implementation in TypeScript

```
const baseRequest = requestwithPath('/api')
                                    .withMethode('POST')
                                    .withBody({ /* data */ })
```

Listing 19: DSL usage example

Modern applications often include external dependencies that facilitate development but can also bring critical vulnerabilities. Dependency validation with specialized tools offers a quick and effective strategy to identify vulnerabilities in third-party libraries and mitigate the associated risks.

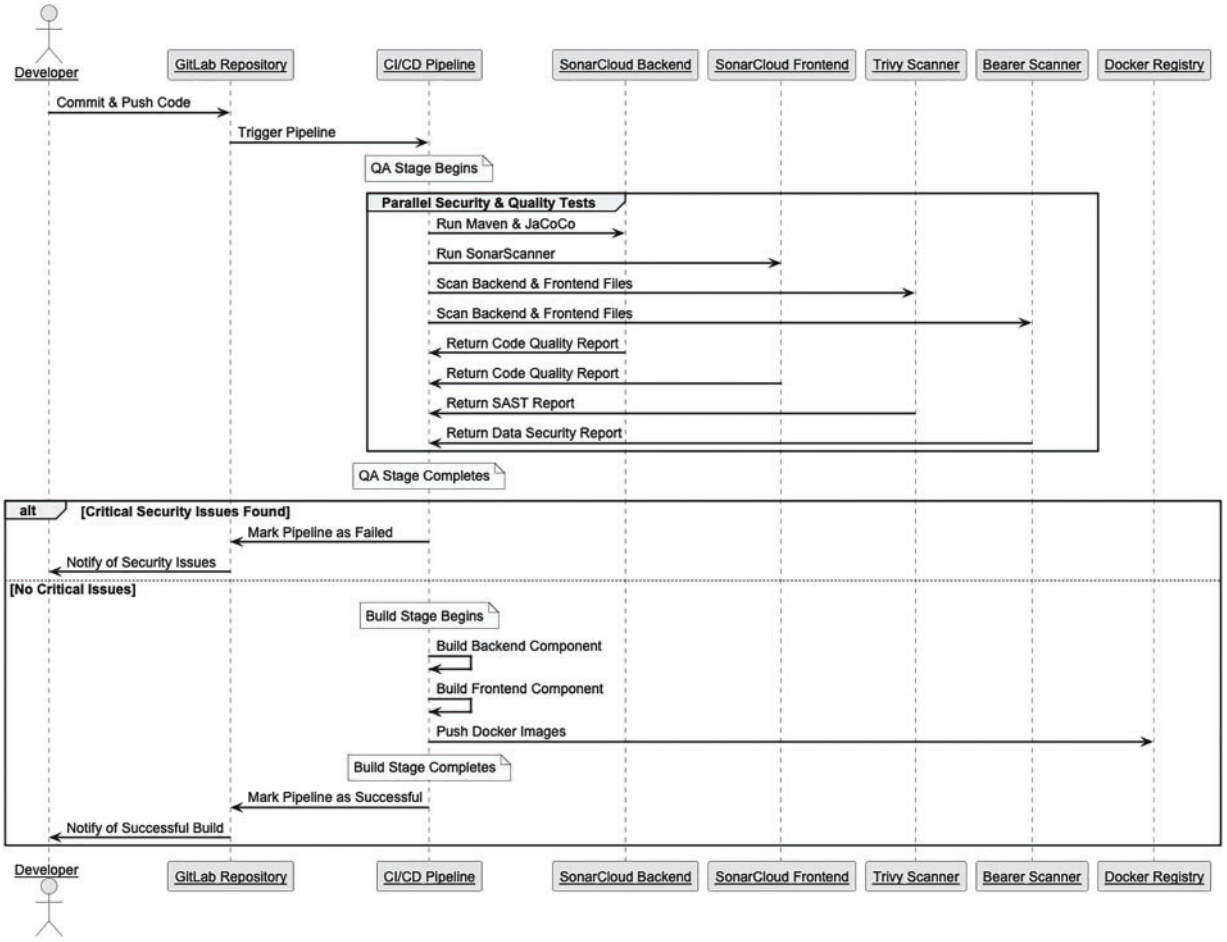

Figure 4: Security CI/CD pipeline sequence diagram

For this study, three open-source tools were selected because of their ease of use and integration:

1. OSV-Scanner: Developed by Google, OSV-Scanner analyses dependencies and generates detailed vulnerability reports based on the OSV database.
2. Trivy: Versatile security analysis tool that evaluates both software dependencies and insecure configurations, compatible with container environments and cloud platforms.
3. OWASP Dependency-Check: Aimed at software composition analysis, it detects vulnerabilities in dependencies using the NIST CVEs database, generating HTML reports.

The validation process included three stages:

1. Dependency detection in the project configuration files (pom.xml for the backend and package.json for the front-end).
2. Vulnerability scanning of the dependencies with OSV-Scanner, Trivy and OWASP Dependency-Check.
3. Reporting and mitigation: The tools produce reports with vulnerabilities classified by severity, facilitating the prioritization of corrective actions.

OSV-Scanner and Trivy scans did not detect vulnerabilities in any of the project's dependencies. OWASP Dependency-Check identified one dependency under review, with no other significant vulnerabilities

reported. The integration of these tools into the CI/CD allows automated scans that strengthen security during the development cycle.

Dependency analysis tools provide a critical layer of security in software development by detecting vulnerabilities early in the lifecycle. Their integration into CI/CD allows maintaining an up-to-date dependency analysis, improving software resilience against emerging threats.

3.4.3 Static Security Analysis (SAST)

Static analysis of code (SAST) is an essential practice in secure software development, enabling early identification of vulnerabilities, code bugs and compliance issues. This project presents the use of SonarQube, Bearer and Fortify as SAST tools to detect and mitigate security and privacy risks in source code. The integration of these tools in CI/CD environments strengthens software security and quality in an automated manner.

SAST analysis tools can detect vulnerabilities in source code before the software goes into production, facilitating early intervention. In this work, SonarQube, Bearer and Fortify are used, three tools that stand out for their accuracy and coverage in the identification of security and privacy risks.

- SonarQube: Configured to analyze each commit in the version control system, SonarQube inspects the code for security vulnerabilities, bugs and technical debt. Its incremental analysis allows problems to be addressed as they are introduced into the code.
- Bearer: Integrated into the CI/CD pipeline, Bearer analyses code for patterns that may compromise the privacy and security of sensitive data, facilitating regulatory compliance.
- Fortify: Configured to assess vulnerabilities in source code, Fortify provides comprehensive coverage through data flow analysis and evaluation of security configurations.

The SonarQube analysis identified 12 issues of varying severity, of which 12 were mitigated by refactoring. One issue was classified as a "false positive" due to its role in protecting sensitive data. Bearer and Fortify detected no additional risks in the project, confirming the robustness of the code in terms of privacy and security. The integration of SonarQube, Bearer and Fortify into the development lifecycle ensures continuous and accurate identification of code risks. These tools enable developers to proactively mitigate vulnerabilities and maintain high standards of quality and compliance. Automating CI/CD analysis reinforces secure and reliable software development.

3.4.4 Dynamic Security Analysis (DAST)

Dynamic application security analysis (DAST) allows the identification of security vulnerabilities in running applications by evaluating their behaviour in real time without the need to inspect the source code. This approach facilitates the detection of problems that are not evident in static analysis. In this study, OWASP ZAP is used to test a web application and document potential security risks.

OWASP ZAP (Zed Attack Proxy) is a popular tool in the field of DAST analysis. Developed by OWASP, ZAP can identify common vulnerabilities such as cross-site scripting (XSS), SQL injections, sensitive data leaks and insecure configurations. The tool acts as an intermediary between the browser and the application, intercepting and analysing HTTP/HTTPS requests to identify potential vulnerabilities.

The scan is divided into several stages:

- Configuration: Includes installation and network settings, as well as the definition of scan parameters, such as target URLs and security policies.
- Execution of the scan:
  - Crawling: Discovery phase where the application structure is mapped, detecting directories and forms.
  - Passive Scanning: Analysis of traffic without altering the behaviour of the application, to identify latent security problems.
  - Active Scanning: Penetration tests that simulate real attacks, injecting malicious payloads to discover exploitable vulnerabilities.
- Report Generation: OWASP ZAP generates detailed reports that categorize vulnerabilities by severity, with explanations and recommendations for mitigation.

The analysis identified 24 vulnerabilities in the application, classified as 2 high, 7 medium and 8 low severity, along with 7 informational alerts. Among the findings, false positives were detected, such as SQL injection and open redirect, which were confirmed to be harmless after manual review. Genuine vulnerabilities were also found in content security settings (CSP), security headers, and cookie handling. High severity vulnerabilities were mitigated following ZAP recommendations, and those in Keycloak-dependent subdomains were escalated and resolved through a system update.

Incorporating OWASP ZAP into the development cycle strengthened application security by enabling runtime vulnerability detection and mitigation. This analysis demonstrated the importance of DAST tools as a fundamental part of secure development, as they reveal potential threats that could compromise both application security and user privacy. Integrating DAST into the CI/CD process ensures that web applications maintain continuously high and robust security standards.

4 Analysis and Discussion

The methodology validation and proof-of-concept analysis process constitutes a critical component in determining whether the proposed approach achieves its intended security objectives. This comprehensive evaluation employed three distinct categories of security testing tools: dependency validation tools, Static Application Security Testing (SAST) tools, and Dynamic Application Security Testing (DAST) tools, each providing unique insights into different aspects of application security.

4.1 Dependency Analysis Results

The dependency analysis revealed no major vulnerabilities in the current application stack. However, this finding should not engender complacency, as modern applications represent living ecosystems in constant evolution. The dynamic nature of software dependencies necessitates continuous monitoring, as new vulnerabilities may emerge in previously secure components. To address this ongoing risk, the integration of dependency scanning tools within CI/CD pipelines provides automated detection capabilities that identify vulnerabilities at the earliest stages of the software development lifecycle, enabling proactive remediation before deployment.

4.2 Static Application Security Testing (SAST) Analysis

Following the identification of issues in SAST-generated reports (Fig. 5), a systematic analysis and comprehensive evaluation of each detected vulnerability was conducted. This process aimed to understand the fundamental nature of security issues and determine appropriate code refactoring strategies for effective mitigation.

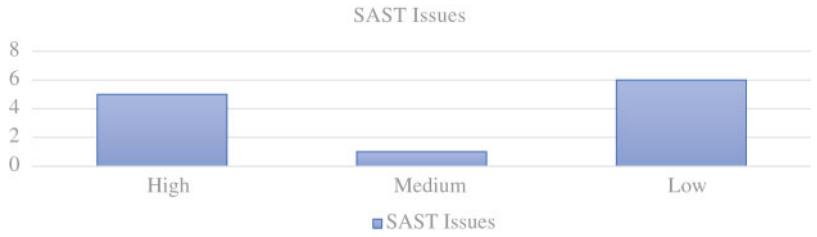

Figure 5: Issues found for SAST component

The analysis methodology involved classifying issues by type, assessing their impact on system security and maintainability, and prioritizing mitigation efforts based on severity and urgency metrics. This structured approach resulted in the development and implementation of targeted refactoring strategies that successfully addressed all but one identified issue.

The remaining unresolved issue involves the overriding of a final method within the secret management design pattern. This override represents an intentional implementation of defensive programming principles, where the variable backup is deliberately replaced with an empty array to prevent sensitive data from persisting in the JVM stack. While SonarQube flags this pattern as problematic, it constitutes a calculated false positive, justified by the proactive protection of sensitive information.

4.3 Dynamic Application Security Testing (DAST) Analysis

The majority of identified alerts were informational in nature, with only two high-severity alerts detected. A detailed analysis of each detection was performed to distinguish between genuine vulnerabilities and false positives, enabling targeted mitigation strategies.

With the DAST analysis tool, the following results were obtained in Fig. 6:

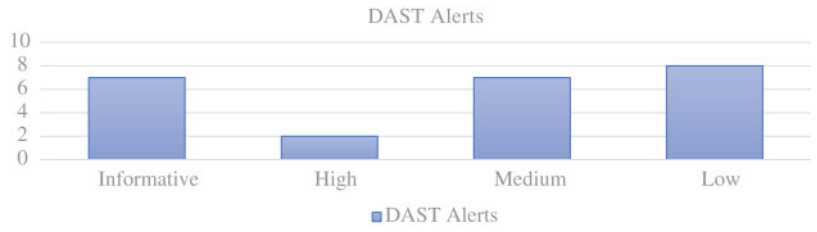

Figure 6: Issues found for DAST component

The vulnerabilities identified in the unir-sc.com domain, classified as true positives, were effectively mitigated following the recommendations detailed in the previous section of the report. The detected vulnerabilities in the auth.unir-sc.com subdomain were escalated to the team responsible for Keycloak, and in response, Keycloak was updated to its most recent version, aligning with best practices in reacting to this

type of issue. This update is crucial in scenarios without direct interference with the source code, allowing to benefit from the latest security patches and improvements.

4.4 Quantitative Security Analysis

The comprehensive security testing methodology employed across the application yielded measurable results that demonstrate effective security management with systematic vulnerability resolution within the integrated security approach. Static Application Security Testing (SonarQube) revealed a manageable severity distribution with five high-severity vulnerabilities (41.7%), one medium-severity issue (8.3%), and six low-severity concerns (50.0%), while complementary SAST tools Bearer and Fortify detected zero additional security risks, indicating consensus on primary vulnerability identification. Dynamic Application Security Testing via OWASP ZAP generated twenty-four distinct security alert categories distributed as two high-severity (8.3%), seven medium-severity (29.2%), eight low-severity (33.3%), and seven informational alerts (29.2%), with manual analysis identifying four false positives (16.7% false positive rate) primarily in Keycloak-managed authentication domains, leaving twenty confirmed vulnerabilities that have been systematically addressed through targeted remediation strategies. The initially concerning high-severity alerts in the authentication module (auth.unir-sc.com), particularly SQL Injection and Open Redirect vulnerabilities detected in Keycloak, were thoroughly analyzed and confirmed as false positives, eliminating the most critical security exposure through proper verification protocols. The confirmed vulnerabilities consisted of seven medium-severity issues including CSP policy deficiencies and absence of anti-CSRF tokens, along with eight low-severity concerns primarily related to cookie configuration and information disclosure, all of which have been successfully resolved through systematic security implementations and Keycloak version updates. Dependency analysis through OSV-Scanner, Trivy, and OWASP Dependency-Check revealed minimal third-party security exposure with zero critical vulnerabilities detected across the external library ecosystem, demonstrating effective dependency management practices that support the overall robust security posture achieved through comprehensive vulnerability remediation.

5 Conclusions and Future Word

The development of this project has shown the critical relevance of integrating security measures in each phase of the software development life cycle. The incorporation of additional security layers from the initial stages of development not only mitigates potential vulnerabilities, but also actively involves each team member in security management, consolidating an organizational culture focused on application protection. The specification of security requirements from the beginning proved to facilitate both the development process and the validation of the application, with observable advantages in the long term. The results obtained during the evaluation of the methodology were positive, underlining the favorable impact of these practices on the security and quality of the application.

The comprehensive security approach implemented, which encompasses authentication, session management, access control, data protection, and secure communication, established a robust defense-in-depth strategy. The implementation of these requirements as priority elements, rather than afterthoughts, resulted in fewer security regressions and a more consistent security baseline across the application. The domain architecture with clean architecture patterns further strengthened the security implementation by creating clear boundaries between system components and reducing the attack surface through proper encapsulation.

The application of modern programming techniques contributed significantly to the robustness of the software, enhancing its ability to resist security threats. The implementation of immutable data structures, functional programming patterns, and defensive coding practices provided inherent protection against common vulnerability classes such as state manipulation and injection attacks. The Secret class demonstrated

how secure design patterns can be systematically applied across an application, ensuring that sensitive data handling follows consistent security practices. Similarly, the domain-specific language developed for API requests in the frontend provided a secure abstraction layer that ensures proper authentication and formatting of requests.

The security requirements have been structured using standard web industry technical terminology and implementation patterns familiar to full-stack developers, facilitating their understanding and adoption by teams without deep security specialization. Each requirement includes clear technical specifications (such as SR-AU-01: "Implement multi-factor authentication for administrative accounts") along with references to technologies and frameworks commonly used in modern web development. The organization by security layers (authentication, sessions, authorization, validation, communications) follows the conceptual structure that web developers already handle in their architectures, reducing the learning curve and facilitating the integration of these controls into their existing workflows.

The methodology provides implementation guides that align with agile development and DevOps practices widely adopted in full-stack web development. The requirements are formulated in a way that they can be directly integrated into user stories, acceptance criteria, and code verification checklists, allowing developers to incorporate security considerations without significantly disrupting their established development processes. For example, input validation controls (SR-VI-01 to SR-VI-04) are expressed in terms of specific technical implementation that can be directly coded and tested, while session management requirements (SR-SM-01 to SR-SM-04) provide technical specifications that can be implemented using standard libraries and middleware from the web ecosystem.

The multi-layered validation approach combining dependency analysis (SCA), static analysis (SAST), and dynamic testing (DAST) proved highly effective at identifying vulnerabilities across different dimensions of the application. This comprehensive testing strategy allowed for the detection of issues that might have been missed by any single testing approach. The inclusion of vulnerability analysis directly integrated into CI/CD pipelines stands out as a particularly effective practice, enabling continuous monitoring of potential security risks without relying exclusively on production environment testing. This "shift-left" approach to security testing ensured that vulnerabilities were identified and remediated much earlier in the development process, significantly reducing the cost and complexity of addressing security issues.

The modular architecture implementation based on Domain-Driven Design (DDD) principles combined with functional programming paradigms demonstrated significant advantages for secure software development. The project architecture consists of two distinct modules—frontend (Vue.js) and backend (Spring Boot)—with clear separation of concerns and secure communication interfaces. While this dual-module approach validates the methodology's effectiveness in distributed component architectures, implementing a comprehensive microservices evaluation would require additional modules representing independent business services (authentication service, ticket service, reporting service, etc.) which was beyond the scope of this research. The clear bounded contexts established within the backend module, the functional programming patterns implemented, and the comprehensive security validation pipeline create a solid foundation that organizations can leverage as a starting point for microservices architectures. Future work should explore the extension of this methodology to multi-service environments to fully validate its applicability in distributed systems contexts.

For future exploration, the orchestration of deployment through platforms such as Kubernetes or OpenShift appears as a key strategy. These platforms enable security policies to be implemented at the network and container level, providing granular traffic control and container isolation. Advanced capabilities such as network policies, pod security contexts, and service meshes could extend the security model to the infrastructure layer, creating a more comprehensive security envelope around the application. The

implementation of secure container image registries with vulnerability scanning and signing would further strengthen the deployment pipeline.

The implementation of service mesh tools promises to increase observability and improve control of communications between microservices, which would help optimize the security and resilience of distributed applications. Technologies such as Istio, Linkerd, or AWS App Mesh could provide additional security capabilities, including mutual TLS between all service communications, fine-grained access control at the request level, enhanced monitoring and anomaly detection, and centralized policy management for distributed applications. Further research into how service mesh architectures can be integrated into the secure SDLC would provide valuable insights for organizations moving toward microservices architectures while maintaining strong security controls.

Acknowledgement: Not applicable.

Funding Statement: The authors received no specific funding for this study.

Author Contributions: The authors confirm contribution to the paper as follows: Conceptualization, Juan Ramón Bermejo Higuera, Kevin Santiago Rey Rodriguez, Julián David Avellaneda Galindo, Josep Tárrega Juan; methodology, Kevin Santiago Rey Rodriguez, Julián David Avellaneda Galindo, Josep Tárrega Juan; software, Kevin Santiago Rey Rodriguez, Julián David Avellaneda Galindo, Josep Tárrega Juan; validation, Juan Ramón Bermejo Higuera, Javier Bermejo Higuera, Juan Antonio Sicilia Montalvo; formal analysis, Juan Ramón Bermejo Higuera, Javier Bermejo Higuera, Juan Antonio Sicilia Montalvo; investigation, Kevin Santiago Rey Rodriguez, Julián David Avellaneda Galindo, Josep Tárrega Juan; resources, Kevin Santiago Rey Rodriguez, Julián David Avellaneda Galindo, Josep Tárrega Juan; data curation, Juan Ramón Bermejo Higuera, Javier Bermejo Higuera, Juan Antonio Sicilia Montalvo; writing—original draft preparation, Kevin Santiago Rey Rodriguez, Julián David Avellaneda Galindo, Josep Tárrega Juan; writing—review and editing, Kevin Santiago Rey Rodriguez, Julián David Avellaneda Galindo, Josep Tárrega Juan; visualization, Juan Ramón Bermejo Higuera, Javier Bermejo Higuera, Juan Antonio Sicilia Montalvo; supervision, Juan Ramón Bermejo Higuera, Javier Bermejo Higuera, Juan Antonio Sicilia Montalvo; project administration, Juan Ramón Bermejo Higuera, Javier Bermejo Higuera, Juan Antonio Sicilia Montalvo; funding acquisition, No funding. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: Data openly available in a public repository. The data that support the findings of this study are openly available in [Secure_Development_Methodolog] at https://gitlab.com/unir8684944/secure_development_methodology (accessed on 11 July 2025).

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest to report regarding the present study.

References

1. Ahmad A, Maulana R, Yassir M. Cybersecurity challenges in the era of digital transformation a comprehensive analysis of information systems. JIEM. 2024;6(1):7–11. doi:10.61992/jiem.v6i1.57.
2. Perera Y. Enhancing the front end web applications performance using design patterns and microservices based architecture [bachelor's thesis]. Kelaniya, Sri Lanka: University of Kelaniya; 2023. doi:10.13140/RG.2.2.36067.53286.
3. Jakimoski K, Stefanovska Z, Stefanovski V. Optimization of secure coding practices in SDLC as part of cybersecurity framework. J Comput Sci Res. 2022;4(2):31–41. doi:10.30564/jcsr.v4i2.4048.
4. Kainulainen S, Tuunanen T, Vartiainen T. Requirements risk management for continuous development: organisational needs. Australas J Inf Syst. 2024;28:1–44. doi:10.3127/ajis.v28.4441.
5. Rhee HS, Ryu YU, Kim CT. Unrealistic optimism on information security management. Comput Secur. 2012;31(2):221–32. doi:10.1016/j.cose.2011.12.001.

6. Syafrizal M, Selamat SR, Zakaria NA. Analysis of cybersecurity standard and framework components. Int J Commun Netw Inf Secur. 2020;12(3):417–32. doi:10.17762/ijcnis.v12i3.4817.
7. Khanneh S, Anu V. Security requirements prioritization techniques: a survey and classification framework. Software. 2022;1(4):450–72. doi:10.3390/software1040019.
8. Ceccato M, Scandariato R. Static analysis and penetration testing from the perspective of maintenance teams. In: Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement; 2016 Sep 8–9; Ciudad Real, Spain. p. 1–6. doi:10.1145/2961111.2962611.
9. Kudriavtseva A, Gadyatskaya O. Secure software development methodologies: a multivocal literature review. arXiv:2211.16987. 2022. doi:10.48550/arxiv.2211.16987.
10. Dopamu O, Asevameh I, Nwamina P, Adesiyan J, Evah P. Secure software development practices for mitigating cyber vulnerabilities in enterprise applications. Int J Creat Res Thoughts. 2024;12(6):2320–882.
11. Phanireddy S. Threat modeling in web application security: a forward-thinking to secure software development. SSRN J. 2025;3(1):2330–40. doi:10.2139/ssrn.5259056.
12. Alphonce DM. Enhancing software quality through early-phase of software verification and validation techniques. Int J Technol Syst. 2024;8(4):1–15. doi:10.47604/ijts.2268.
13. Torredimare A. Extension of an enterprise web application for top-management reporting: a modular approach to web application development [master's thesis]. Turin, Italy: Politecnico di Torino; 2024 [cited 2025 Jul 11]. Available from: https://webthesis.biblio.polito.it/secure/33954/1/tesi.pdf.
14. Sulír M, Porubän J, Chodarev S. Local software buildability across Java versions (registered report). arXiv:2408.11544. 2024. doi:10.48550/arxiv.2408.11544.
15. Suroju AC. Integration of Zuora billing system in microservices architecture: a spring boot implementation. J Comput Sci Technol Stud. 2025;7(4):1060–8. doi:10.32996/jcsts.2025.7.4.120.
16. Rajput AS, Singh HP, Bang G, Joshi S, Patidar T. Comparing spring boot and ReactJS with other web development frameworks: a study. In: Nanda SJ, Yadav RP, Gandomi AH, Saraswat M, editors. Data science and applications. Singapore: Springer Nature; 2024. p. 149–60. doi:10.1007/978-981-99-7817-5_12.
17. Liu C, Suo Z, Mao Q, Zhu Y. Practice and application of Wiki open source document platform based on VUE. In: Proceedings of the 2023 3rd International Symposium on Computer Technology and Information Science (ISCTIS); 2023 Jul 7–9; Chengdu, China. p. 819–22. doi:10.1109/ISCTIS58954.2023.10213203.
18. Sireteanu NA. Security challenges of modern web applications. SSRN J. 2009. doi:10.2139/ssrn.1529803.
19. Smith D. An exploration of the capability of a relational database management system to encompass business and persistence capabilities within architecturally layered software. Milton Keynes, UK: Open university; 2023 [cited 2025 Jul 12]. Available from: https://oro.open.ac.uk/94299/1/SMITH_T847_VOR.pdf.
20. Sharma S, Sharma R. An innovative solution for data persistence problem in reliable data transmission. In: Proceedings of the 2023 International Conference on Distributed Computing and Electrical Circuits and Electronics (ICDCECE); 2023 Apr 29–30; Ballar, India. p. 1–6. doi:10.1109/ICDCECE57866.2023.10150706.
21. Bello RW, Tobi SJ. Software bugs: detection, analysis and fixing. SSRN J [Internet]. 2024 [cited 2025 Jul 9]. Available from: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=1529803.
22. Parviainen P, Tihinen M, Kääriäinen J, Teppola S. Tackling the digitalization challenge: how to benefit from digitalization in practice. Int J Inf Syst Proj Manag. 2017;5(1):63–77. doi:10.12821/ijispm050104.
23. Haleem KMHRJR, Asim MW, Khan AH, Hussain MIN, Razzaq RA. Cybersecurity in digital transformation applications: analysis of past research and future directions. In: Proceedings of the ICCWS 2023 18th International Conference on Cyber Warfare and Security; 2023 Mar 8–9; Towson, MD, USA. p. 337–47. doi:10.34190/ICCWS.22.028.
24. Mentsiev AU, Engel MV, Tsamaev AM, Abubakarov MV, Yushaeva RS. The concept of digitalization and its impact on the modern economy. In: Proceedings of the International Scientific Conference Far East Con (ISCFEC 2020); 2020 Oct 6–9; Vladivostok, Russia. doi:10.2991/aebmr.k.200312.422.
25. European Commission. A europe fit for the digital age [Internet]. 2024 [cited 2025 Jul 11]. Available from: https://commission.europa.eu/strategy-and-policy/priorities-2019-2024/europe-fit-digital-age_en.

26. Saeed S, Altamimi SA, Alkayyal NA, Alshehri E, Alabbad DA. Digital transformation and cybersecurity challenges for businesses resilience: issues and recommendations. Sensors. 2023;23(15):6666. doi:10.3390/s23156666.
27. Eekelen V, Pieterse M. Which are harder? Soft skills or hard skills? In: Proceedings of the Annual Conference of the Southern African Computer Lecturers' Association (SACLA 2016); 2016 Jul 5–6; Cullinan, South Africa. p. 160–7. doi:10.1007/978-3-319-47680-3_15.
28. Mistrík I, Grundy J, Hoek A, Whitehead J. Collaborative software engineering. 1st ed. Berlin/Heidelberg, Germany: Springer; 2010. doi:10.1007/978-3-642-10294-3.
29. Bonsignour C.Jones O. The economics of software quality [Internet]. 2012 [cited 2025 Jul 11]. Available from: https://ptgmedia.pearsoncmg.com/images/9780132582209/samplepages/0132582201.pdf.
30. Cerpa N, Verner JM. Why did your project fail? Commun ACM. 2009;52(12):130–4. doi:10.1145/1610252.1610286.
31. Saravanos A, Curinga MX. Simulating the software development lifecycle: the waterfall model. Appl Syst Innov. 2023;6(6):108. doi:10.3390/asi6060108.
32. Bahattab A, Abdullah A. A comparison between three SDLC models waterfall model, spiral model, and incremental/iterative model. Int J Comput Sci Issues. 2015;12(1):106–11. [cited 2025 Jul 10]. Available from: https://www.ijcsi.org/papers/IJCSI-12-1-1-106-111.pdf.
33. Moşteanu NR. Challenges for organizational structure and design as a result of digitalization and cybersecurity. Bus Manag Rev. 2020;11(1):278–86. doi:10.24052/bmr/v11nu01/art-29.
34. Manadhata PK, Tan KMC, Maxion JA, Jeannette MW. An approach to measuring a system's attack surface. Pittsburgh, PA, USA: Carnegie Mellon University; 2007 [cited 2025 Jul 11]. Available from: https://www.cs.cmu.edu/~wing/publications/CMU-CS-07-146.pdf.
35. Singh S, Kumar S. The times of cyber attacks. Acta Tech Corviniensis. 2020;13(3):133–7. [cited 2025 Jul 11]. Available from: https://acta.fih.upt.ro/pdf/2020-3/ACTA-2020-3-25.pdf.
36. Johnsson DB, Deogun D, Sawano D. Secure by design. New York, NY, USA: Manning Publications; 2019 [cited 2025 Jun 13]. Available from: https://www.manning.com/books/secure-by-design.
37. Govardhan D. A comparison between five models of software engineering. Int J Comput Sci Issues. 2010;7(5):94–101. [cited 2025 Jun 12]. Available from: https://www.researchgate.net/publication/258959806.
38. Olmsted A. Security-driven software development: learn to analyze and mitigate risks in your software projects. Birmingham, UK: Packt Publishing Ltd.; 2024 [cited 2025 Jul 10]. Available from: https://www.amazon.com/Security-Driven-Software-Development-mitigate-software/dp/1835462839.
39. McGraw G. Testing for security during development: why we should scrap penetrate-and-patch. In: Proceedings of COMPASS '97: 12th Annual Conference on Computer Assurance; 1997 Jun 16–19; Gaithersburg, MD, USA. p. 117–9. doi:10.1109/CMPASS.1997.613270.
40. McGraw W. Software security: a touchstone for reliability. IEEE Secur Priv. 2004;2(1):80–3. doi:10.1109/MSP.2004.1264871.
41. Lipner S. The trustworthy computing security development lifecycle. In: Proceedings of the 20th Annual Computer Security Applications Conference; 2004 Dec 6–10; Tucson, AZ, USA. doi:10.1109/CSAC.2004.41.
42. Otieno M, Odera D, Ounza JE. Theory and practice in secure software development lifecycle: a comprehensive survey. World J Adv Res Rev. 2023;18(3):53–78. doi:10.30574/wjarr.2023.18.3.0944.
43. Shostack A. Threat modeling: designing for security. Nashville, TN, USA: John Wiley & Sons; 2014.
44. OWASP F. Software assurance maturity model (SAMM) (Version 2.0) [Internet]. 2020 [cited 2025 Jul 10]. Available from: https://owaspsamm.org.
45. Baninemeh E, Toomey H, Labunets K, Wagenaar G, Jansen S. An evaluation of the product security maturity model through case studies at 15 software producing organizations. In: Hyrynsalmi S, Münch J, Smolander K, Melegati J, editors. Software business. Cham, Switzerland: Springer Nature; 2024. p. 327–43. doi:10.1007/978-3-031-53227-6_23.
46. Synopsys. BSIMM15: building security in maturity model (BSIMM) [Internet]. 2025 [cited 2025 Jul 10]. Available from: https://www.bsimm.com.
47. Singh S. Secure software development life cycle: implementation challenges in small and medium enterprises (SMEs). Authorea. 2025. doi:10.22541/au.174585836.63395541/v1.

48. Hellquist E. Evaluating security for javascript-based frontend frameworks [master's thesis]. Umeå, Sweden: UMEÅ University; 2024 [cited 2025 Jul 10]. Available from: https://umu.diva-portal.org/smash/get/diva2:1928741/FULLTEXT01.pdf.
49. Foundation TO. Software assurance maturity model. 2024 [cited 2025 Jul 10]. Available from: https://owasp.org/www-project-samm/.