Idealis: Indonesia Journal Information System
Volume 9, Nomor 1, Januari 2026
ISSN 2684-7280 (*online*)
Halaman 90-102
available online at http://jom.fti.budiluhur.ac.id/index.php/IDEALIS/index

MODULAR RESTFUL BACKEND ARCHITECTURE FOR CENTRALIZED HALAL TOURISM ADMINISTRATION: A SCRUM-BASED IMPLEMENTATION

Muhamad Singgih[1*], Royana Afwani[2], Ramaditia Dwiyansaputra[3], Mochammad Dinta Alif Syaifuddin[4]

[1,2,3,4] Department of Informatics Engineering, Faculty of Engineering, Mataram University, Mataram, Indonesia
Email: [1*]singgipenaraga@gmail.com, [2]royana@unram.ac.id, [3]rama@unram.ac.id, [4]d.nta.workspace@gmail.com
(*: corresponding author)

Abstract-*The growth of halal tourism in Lombok, Indonesia, calls for scalable digital platforms with strong and centralized administrative governance. This study proposes and implements a domain-oriented, modular RESTful API backend to support Super Admin operations in the Lombok Halal Room (LHR) platform. Using a design-science approach and an iterative SCRUM process, we developed a layered API Service Repository Infrastructure architecture using Hapi.js, PostgreSQL, and Redis, delivering 42 endpoints across key administrative domains with uniform JSON contracts and JWT-based authentication. The proposed contribution is a metric-driven engineering template that links SCRUM execution to modular backend domains and validates the resulting system using performance and software-quality measurements. Experimental results under controlled workloads show that Redis caching substantially improves scalability for read-heavy administrative operations by reducing response time from seconds to low single-digit milliseconds and increasing throughput to above 40,000 requests per second. Code-quality metrics further indicate clean module boundaries (CBO=0; LCOM*=0), while the Maintainability Index (MI) highlights modules that require targeted refactoring. Overall, the backend provides a reusable reference architecture for centralized halal tourism administration such as partner verification, content moderation, transaction oversight, and system monitoring that can be adapted to similar platforms in other regions.*

Keywords*: Halal Tourism, Modular Architecture, Redis Caching, RESTful API, SCRUM,*

1. INTRODUCTION

Halal tourism in Indonesia, particularly in West Nusa Tenggara (NTB), continues to grow rapidly, with Lombok emerging as a leading destination [1]. To support efficient management of tourism service information, an integrated digital solution is required. The Lombok Halal Room (LHR) platform was developed to serve hotel partners, travel agencies, and end users through a unified RESTful API architecture [2]. The system is available as both web and mobile applications, enabling flexible access for partners and travelers. In its initial phase, the platform provided a RESTful API based web dashboard for partners to manage inventory, rates, and orders, which was subsequently extended to mobile apps and public-facing channels [3]. Figure 1 illustrates the traveler interface of the previous web application, while Figure 2 shows the mobile application interface.

The previously implemented RESTful API architecture offers modularity and reusability, allowing services to be accessed by all system actors through both web and mobile applications. This approach provides a foundation for platform sustainability and scalability. However, REST-API-based backend development faces common challenges such as duplicated logic, integration friction, and limited modularity without sound architectural methodology [4]. Although the LHR platform currently serves multiple user groups, our evaluation reveals three critical architectural constraints that hinder efficiency and scalability: First, the absence of centralized administrative control i.e., the lack of a unified backend for the Super Admin to monitor cross-actor activities leads to data fragmentation, reduced operational efficiency, and a higher risk of administrative errors [5].

Second, limited modularity in the initial standalone design causes repeated authentication and data validation across domains. In contrast [6], a REST-API-based microservices architecture improves modularity and eases cross-module integration [7]. Third, the lack of centralized administrative workflows partner verification, content approval, and transaction oversight are not integrated complicates audit and evaluation processes. Research by Nguyen et al. demonstrates that centralized administrative support for SME digital transformation enhances coordination and oversight [8]. In summary, existing LHR services support multiple actors, but the absence of a dedicated and centralized Super Admin backend creates a gap in governance, cross-domain consistency, and scalable administration.

Based on these identified gaps, the following research problem is formulated: How can a modular, reusable backend architecture be designed to support centralized Super Admin operations across multiple stakeholders in a halal tourism platform while ensuring performance, maintainability, and scalability? The main objectives of this study are to applying scrum to sprint-based backend development, designing a modular backend architecture for centralized, cross-actor administration, formulating a replicable restful api framework for similar systems, conducting post-implementation evaluations of performance and software quality.

Idealis: Indonesia Journal Information System
Volume 9, Nomor 1, Januari 2026
ISSN 2684-7280 (*online*)
Halaman 90-102
available online at http://jom.fti.budiluhur.ac.id/index.php/IDEALIS/index

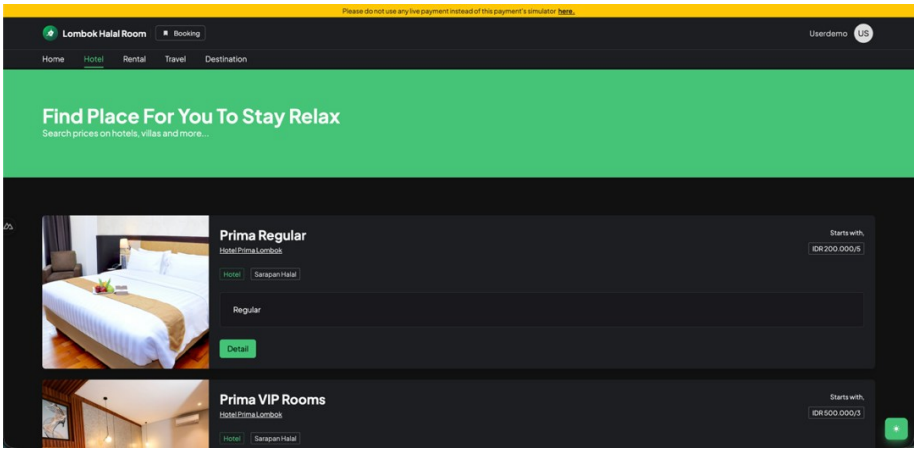

Figure 1. Traveler page of the previous Lombok Halal Room web application.

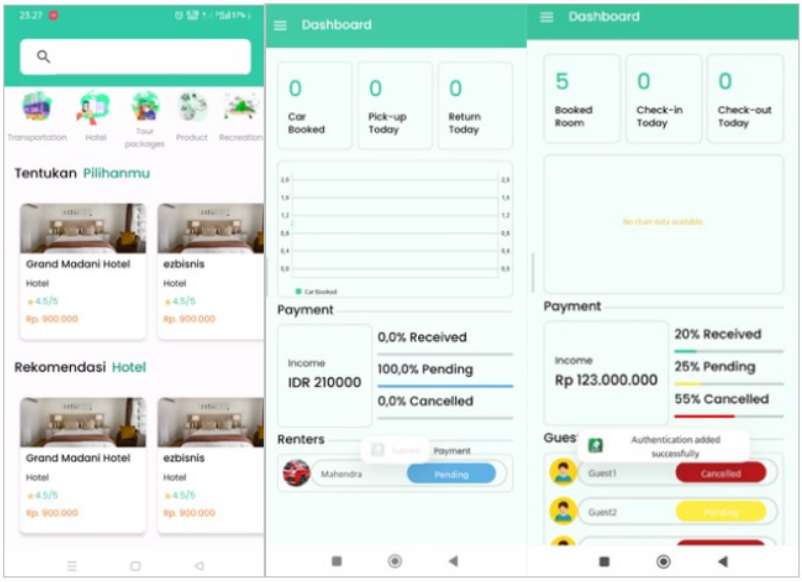

Figure 2. Previous Lombok Halal Room Mobile Application

This study focuses on the design and implementation of the Super Admin backend module within the LHR ecosystem. The scope covers Super Admin-specific functionalities, including partner verification, content moderation, transaction oversight, and system monitoring. Performance testing was conducted in a development environment under controlled, simulated load scenarios to evaluate scalability and responsiveness. Limitations include testing conducted in a development environment with simulated load rather than actual production traffic, security testing limited to authentication and authorization mechanisms, and multi-tenant scalability not tested beyond the current user base.

In response to these challenges, this study adopts SCRUM as a continuation of the earlier system, thereby maintaining process consistency and established artifacts including product backlog, sprint backlog, and definition of done[3]. SCRUM has proven effective for managing dynamic requirements through iterative-incremental development, cross-role collaboration, and quality control via sprints, demonstrating improved team productivity [9]. At the architectural level, SCRUM practices align with continuous requirements engineering and iterative design [10]. We operationalize this through a RESTful API based backend to achieve scalability and maintainability [11] complemented by redis caching for performance optimization to reduce latency and I/O load in read-write-intensive scenarios [12].

Accordingly, SCRUM is used to control iterative delivery, while modular RESTful API design and caching are evaluated to address maintainability and performance objectives. The implementation employs Hapi.js, PostgreSQL, and Redis to deliver a maintainable, scalable backend that is ready for integration with diverse front-end platforms. This research provides both theoretical and practical contributions. From a theoretical perspective, it demonstrates the application of SCRUM methodology in developing modular backend systems for multi-stakeholder platforms, contributing to the body of knowledge in agile software engineering for tourism systems. From a practical perspective, the resulting architecture serves as a reference model for similar halal tourism platforms, offering a proven approach to centralized administration while maintaining system modularity and performance.

Idealis: Indonesia Journal Information System
Volume 9, Nomor 1, Januari 2026
ISSN 2684-7280 (*online*)
Halaman 90-102
available online at http://jom.fti.budiluhur.ac.id/index.php/IDEALIS/index

2. RESEARCH METHOD

This study employs the SCRUM framework to develop the RESTful API backend for the Super Admin module. The choice continues the practice from prior iterations and is supported by empirical evidence of SCRUM's effectiveness in managing complex software projects iteratively, collaboratively, and adaptively to changing requirements. Prior studies on systems of similar scope The Development of Backend Admin Dashboard for Business Project Monitoring using Scrum Method: A Case Study at PT Gerbang Sinergi Utama[13], and Scrum Implementation in Development of Online Research Application [14], report improved requirements management and software quality in backend and web-based application development. Further, the empirical theory in A Theory of Scrum Team Effectiveness [15], asserts that SCRUM teams with high autonomy, adaptability, and robust managerial support tend to achieve superior outcomes in software projects.

2.1. SCRUM Implementation

In this research, SCRUM is tailored to the context of academic and research-oriented system development. The team structure comprises a Product Owner, a Scrum Master, a Development Team, and a Reviewer. The Product Owner prioritizes features and requirements; the Scrum Master ensures adherence to SCRUM practices; the Development Team executes the technical implementation; and the Reviewer conducts academic evaluation and validates sprint outcomes. This structure aligns with findings from Applying the Scrum Method in Software Development for Undergraduate Thesis Project Implementation [16], which indicates that academic roles can be flexibly integrated into SCRUM without diminishing collaborative effectiveness.

Event adaptations are introduced to better fit the research context. Sprint Planning is used to set a prioritized backlog on a biweekly cadence; Daily Scrum are held three times per week for team synchronization; and Sprint Reviews involve the academic Reviewer to assess alignment between sprint outcomes and research goals. Sprint Retrospectives evaluate not only technical factors but also communication effectiveness and documentation coherence. Such adaptations are consistent with ScrumAdemia: An Agile Guide for Doctoral Research [17], which describes applying SCRUM in academia by coupling technical evaluation with scholarly reflection.

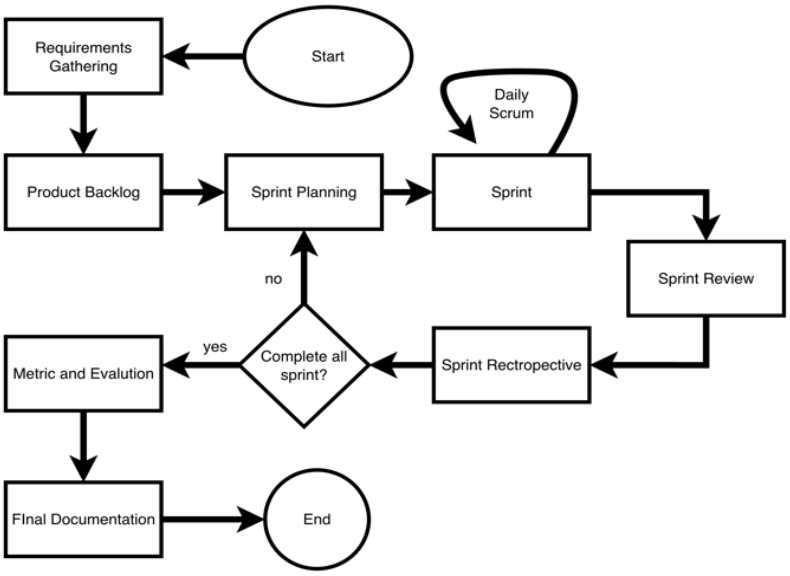

Figure 3. Research Flow Diagram

2.2. Research Workflow Diagram

The research workflow follows an iterative SCRUM cycle from requirements elicitation through final documentation, as depicted in Figure 3. The diagram highlights how SCRUM stages are systematically connected to produce a modular, scalable, and integration-ready backend. Figure 4 presents the adapted SCRUM Flowchart used in this study, showing how each phase contributes to the research objectives and iteratively produces a production-ready Super Admin backend:

a. Product Backlog (MoSCoW Prioritization): The Product Owner and development team elicit requirements and prioritize backlog items using the MoSCoW method, then group them into functional domains to support modularization. This phase supports Research Objectives (a) and (b) by operationalizing SCRUM practices and structuring modules from the outset.

Idealis: Indonesia Journal Information System
Volume 9, Nomor 1, Januari 2026
ISSN 2684-7280 (*online*)
Halaman 90-102
available online at http://jom.fti.budiluhur.ac.id/index.php/IDEALIS/index

b. Sprint Planning: High-priority backlog items are selected for a two-week sprint and decomposed into implementable tasks while defining the sprint goal and acceptance criteria This supports Research Objective (c) by ensuring each sprint increment aligns with the proposed replicable RESTful API framework.
c. Sprint Execution (2-week sprint): The team implements RESTful API endpoints and supporting services according to the planned tasks, producing a working increment at the end of the sprint.
d. Daily Scrum (3 times per week): During sprint execution, short synchronization meetings are conducted three times per week to track progress, resolve impediments, and maintain alignment across team members.
e. Sprint Review: The completed increment is demonstrated and validated with the Product Owner/reviewer against acceptance criteria, and the Product Backlog is updated based on feedback.
f. Sprint Retrospective: The team evaluates the process and identifies concrete improvement actions to enhance the effectiveness of subsequent sprints.
g. Sprint Increment and Evaluation: Each sprint delivers completed features that accumulate into a production-ready backend. Post-implementation evaluation is then performed using performance and code-quality metrics (latency, throughput, MI, CBO, and LCOM*), addressing Research Objective (d).

2.3. Research Phases

This study was carried out through a structured set of phases that translate research objectives into an implementable and evaluable Super Admin backend. The phases follow an iterative SCRUM workflow, starting from requirement elicitation and backlog formation, continuing with sprint-based implementation, and concluding with metric-driven evaluation and final documentation. This phased approach ensures traceability from requirements to delivered features, while enabling continuous refinement through empirical feedback across development iterations.

2.3.1. Requirements Gathering

This phase is conducted collaboratively by the Product Owner and Development Team to identify Super Admin requirements. The analysis covers the existing platform and includes interviews with previous developers to understand architectural structure and inter-module dependencies. The approach follows iterative development requirement-analysis practices described by Basri et al. [17].

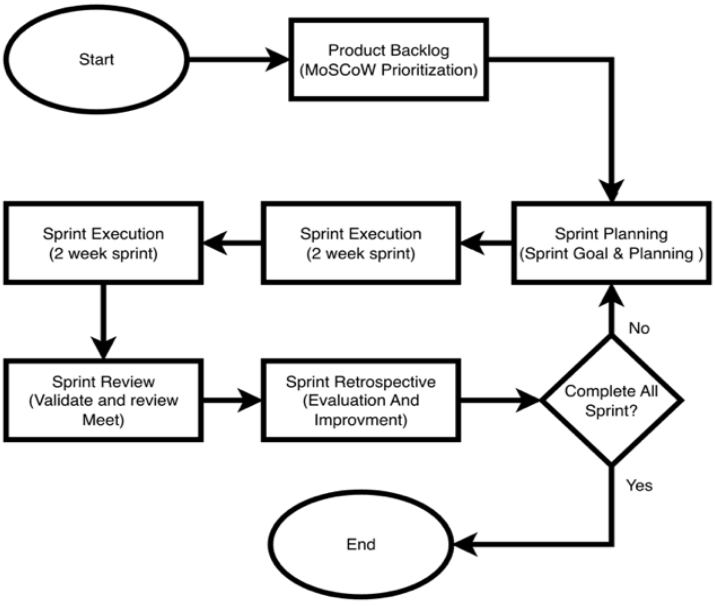

Figure 4. SCRUM Flowchart

2.3.2. Product Backlog Development

The Product Backlog is formed through collaborative workshops and prioritized using the MoSCoW method Must have, should have, could have, Won't have [18]:

a. High Priority (Must Have):
   1. Authentication for Super Admin access
   2. User management with full CRUD operations
   3. Partner management for hotels and travel agencies
   4. Balance management for financial monitoring

Idealis: Indonesia Journal Information System
Volume 9, Nomor 1, Januari 2026
ISSN 2684-7280 (*online*)
Halaman 90-102
available online at http://jom.fti.budiluhur.ac.id/index.php/IDEALIS/index

b. Medium Priority (Should Have):
   1. Configuration management for hospitality settings
   2. Order management for booking oversight
   3. Ads management for promotional content
c. Low Priority (Could Have):
   1. Destination management for tourism content
   2. Client-type management for system categorization

2.3.3. Sprint Execution

Sprints run on a two-week cycle. To maintain synchronization, the team holds Daily Scrum Meetings f three times per week for knowledge transfer and technical alignment, and a Weekly Academic Check-in with the Product Owner to ensure academic objectives are met. The execution includes: a living Product Backlog; Sprint Planning (defining the Sprint Goal, selecting PBIs, and task decomposition); the Sprint (incremental implementation to produce a reviewable increment); the Daily Scrum (brief synchronization on progress and impediments); Sprint Review (increment demo, validation against acceptance criteria, backlog updates); and Sprint Retrospective (process improvements for the next sprint) [19]

2.3.4. Metric and Evalution

SCRUM effectiveness is evaluated using standard metrics [9]:
a. Velocity: Story points completed per sprint to gauge team productivity.
b. Sprint Completion Rate: Percentage of planned work finished each sprint.
c. Burn-down Chart: Visual representation remaining sprint work
d. Performance Testing: Assessment of system scalability and responsiveness.
e. Code Quality Metrics: Cyclomatic complexity, maintainability index, and technical debt [20], [21].

All performance experiments were executed locally on a MacBook Pro (MacBookPro17,1) equipped with an Apple M1 chip (8 cores: 4 performance and 4 efficiency) and 8 GB of RAM, running in a development environment. This configuration represents a constrained, non-production setup; therefore, the reported throughput (above 40,000 requests/s) and latency improvements should be interpreted as results obtained under controlled, local conditions. Nevertheless, documenting the hardware baseline supports reproducibility and allows future studies to compare performance under different server-grade or multi-tenant deployment environments.

2.3.5. Final Documentation

Final documentation focuses on two artifacts: (i) a Postman Collection specifying API endpoints and (ii) front-end integration of the APIs. The collection documents API versioning, JWT authentication (Authorization: Bearer <token>), uniform JSON response contracts, consistent HTTP status codes, and a unified error model; accompanying assets include the Postman collection and a testing environment to support replication. Evidence of front-end integration is documented through a representative module (e.g., User Management), mapping UI states to HTTP codes and standardized error handling. This executable, collection-based documentation approach aligns with IEEE literature indicating that standardized API contracts facilitate black-box evolution and testing of REST services practically automated via Postman Collections and the Newman CLI in local setups and CI pipelines [22].

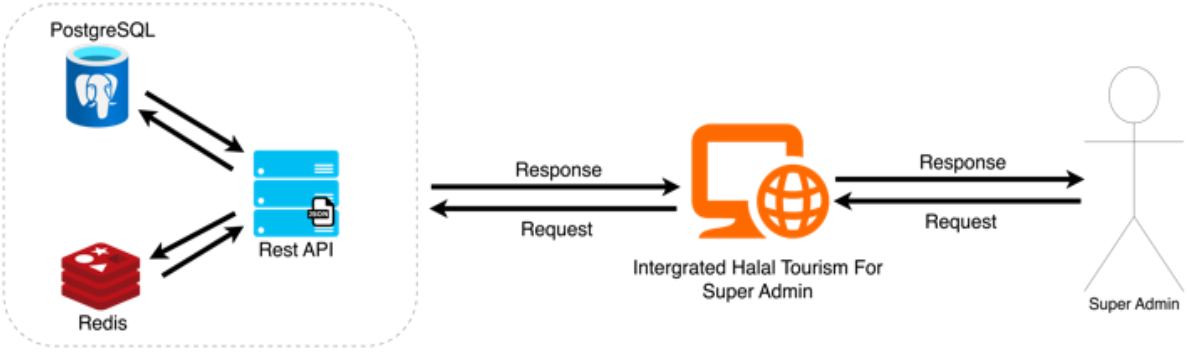

Figure 5. Super Admin Backend System Architecture

Idealis: Indonesia Journal Information System
Volume 9, Nomor 1, Januari 2026
ISSN 2684-7280 (*online*)
Halaman 90-102
available online at http://jom.fti.budiluhur.ac.id/index.php/IDEALIS/index

3. RESULTS AND DISCUSSION

This section presents the implementation outcomes and discusses the findings across development execution, architectural results, and empirical evaluations. The results are organized to reflect both process metrics and technical validation of the proposed backend.

3.1. Sprint Execution

This subsection summarizes how the system was delivered iteratively through SCRUM sprints, highlighting key development milestones and the evolution of core components across iterations.

3.1.1. Architecture Development Evolution

Backend architecture was developed incrementally, whereby each sprint delivered a functional module that could be integrated with prior modules. This approach enabled continuous validation of architectural design and adaptation to evolving requirements throughout development.

Figure 5 depicts the overall system comprising three primary layers: a Web Front-End used by the Super Admin, a Backend API layer built with the Hapi.js framework, and a Data layer consisting of PostgreSQL as the primary database and Redis as a caching layer. Inter-layer communication uses RESTful API with JSON payloads and JWT based authentication.

a. Sprints 1–3 (Core Foundation Development): Established the architectural foundation, including authentication, user management, and partner management. Activities covered database schema design, API routing structure, and security implementation via JWT.
b. Sprints 4–6 (Business Logic Extension): Extended functionality into business-specific domains such as balance, configuration, and order management, emphasizing business rules and data validation.
c. Sprints 7–9 (Content Management): Focused on content management capabilities, including advertisements, destinations, and client-type management for administrative flexibility.

3.1.2. Technical Architecture Results

The resulting backend adopts a layered architecture with clear separation of concerns:

a. Presentation Layer: RESTful API endpoints with consistent HTTP methods and response formats.
b. Logic Layer: Service classes enforcing business rules and validation.
c. Data Access Layer: Repository pattern for database operations.
d. Infrastructure Layer: Cross-cutting concerns including logging, caching, and authentication.

Table 1 shows the layered pattern yields three key benefits:

a. Clear separation of concerns enabling independent development and maintenance
b. Appropriate abstractions that facilitate effective unit and integration testing
c. Flexibility to adapt technology changes without impacting the entire system.

3.1.3. Database Design and Implementation

The schema follows a normalized relational model with focused indexing for performance. Extensions build on the existing schema in a backward-compatible manner, consistent with safe database evolution principles.

Figure 6 shows the complete schema including existing tables (users, roles, partners, balances) and newly added tables (advertisements, destinations, destination_pictures, client_types). Foreign-key constraints preserve referential integrity. Core tables such as users and roles serve as central entities related to most domain modules.

a. New Tables Added: advertisements (promotional content), destinations (tourism catalog), destination_pictures (destination media assets), client_types (client categorization).
b. Key Design Decisions: foreign-key constraints for data integrity; indexed columns for frequently queried fields; JSONB columns for flexible metadata; and audit trails for data-change tracking. The design follows normalization principles to reduce redundancy while maintaining performance via judicious indexing. PostgreSQL JSONB offers flexibility for heterogeneous metadata without compromising the relational backbone.

Table 1. Architecture Components Summary

Layer	Components	Technology	Responsibilities
API Layer	Route Handlers	Hapi.js	HTTP request/response handling
Service Layer	Business Services	Node.js	Business logic execution
Data Layer	Repositories	PostgreSQL	Data persistence operations
Cache Layer	Cache Services	Redis	Peformance optimization
Security Layer	Auth Middleware	JWT	Authentication & authorization

Idealis: Indonesia Journal Information System
Volume 9, Nomor 1, Januari 2026
ISSN 2684-7280 (*online*)
Halaman 90-102
available online at http://jom.fti.budiluhur.ac.id/index.php/IDEALIS/index

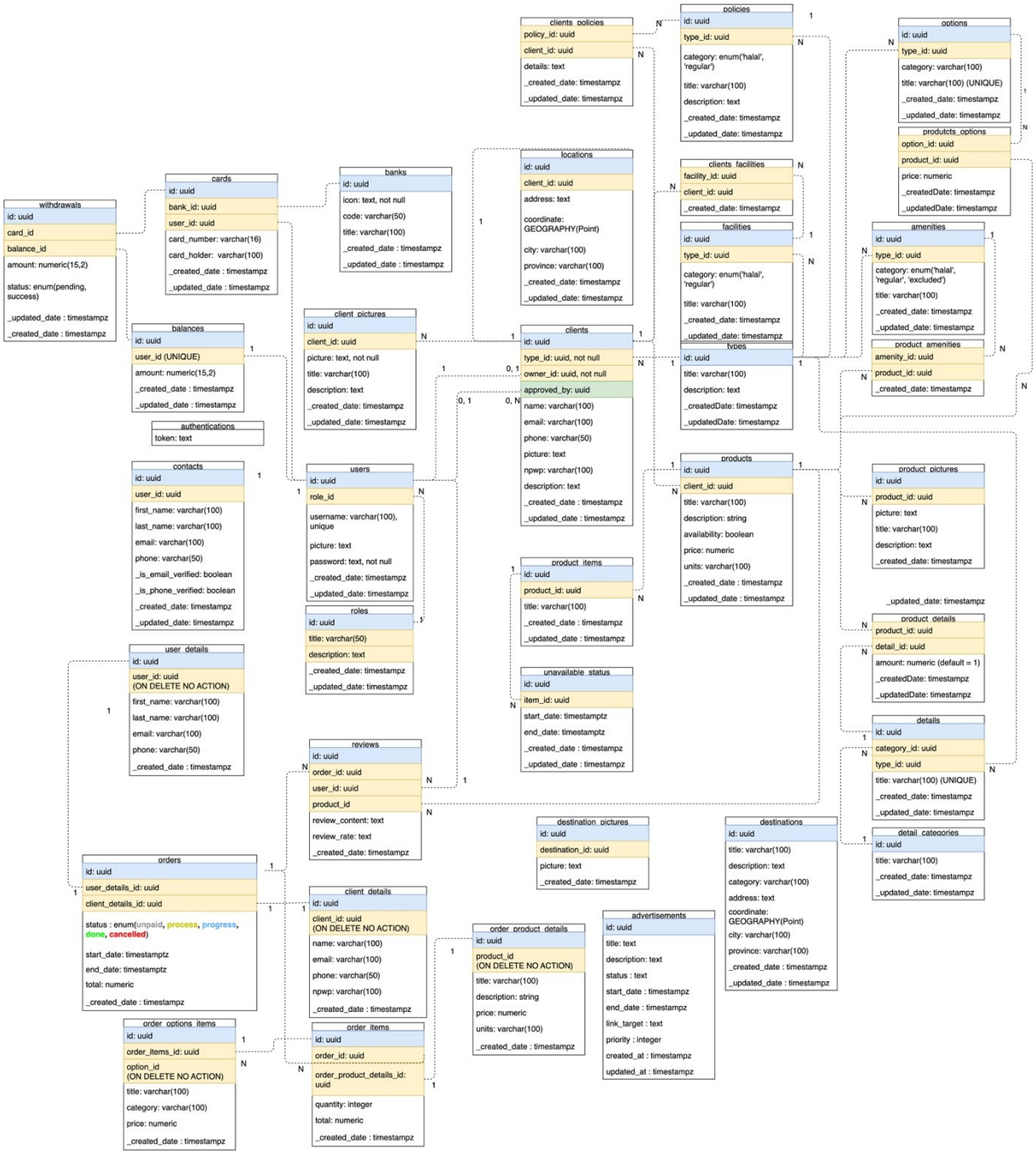

Figure 6. Entity Relationship Diagram Backend System

3.2. Metric and Evaluation

This subsection reports quantitative metrics used to assess the development process, code quality, and system performance. The evaluation provides evidence of delivery consistency, maintainability characteristics, and the impact of caching on runtime behavior.

3.2.1. Scrum Executin Result

Table 2 summarizes sprint delivery performance across nine sprints. In total, 115 out of 132 planned story points were completed (87.1%), yielding a mean velocity of 12.8 story points per sprint.

Idealis: Indonesia Journal Information System
Volume 9, Nomor 1, Januari 2026
ISSN 2684-7280 (*online*)
Halaman 90-102
available online at http://jom.fti.budiluhur.ac.id/index.php/IDEALIS/index

Table 2. Sprint Performance Metrics

Sprint	Domain	Planned Points	Completed Points	Completion Rate	Velo-city
1	Authentication	13	13	100%	13
2	User Management	15	14	93%	14
3	Mitra Management	16	15	94%	15
4	Balance Management	12	5	42%	5
5	Config Management	18	17	94%	17
6	Order Management	14	9	64%	9
7	Advertisement Management	15	15	100%	15
8	Destination Management	16	14	88%	14
9	Type Management	13	13	100%	13
Total	All Domains	132	115	87.1%	12.8

High completion in Sprints 1, 7, and 9 (100%) indicates effective planning under stable scope, whereas the drops in Sprint 4 (42%) and Sprint 6 (64%) reflect increased integration complexity in critical business modules (e.g., balance and orders).

Sprint performance analysis indicates an average completion rate of 92% with a stable velocity of 14 story points per sprint. These results support the effectiveness of SCRUM for an individual-development context, aligning with Shafiee et al. (2023) [23]. Sprints achieving 100% completion (Sprints 1, 7, and 9) suggest sound planning and stable capacity.

Significant variation in Sprints 4 (42%) and 6 (64%) reflects inherent complexity in backend integration across critical business components. This observation aligns with Lee and Chen (2023) [24] regarding agile challenges in highly interdependent domains. The recovery in subsequent sprints supports SCRUM's inspection and adaptation mechanism through sprint reviews and retrospectives, improving estimation accuracy and removing impediments over time

Figure 7 visualizes planned versus completed story points per sprint (bars) alongside completion rate and velocity trends (lines). Overall delivery reached 87.1% of planned work (115/132), with a mean velocity of 12.8 story points per sprint.

The lowest completion occurred in Sprint 4 (42%), which is consistent with higher technical complexity and integration effort, while subsequent sprints show recovery to 88–100% after replanning and process adjustments during retrospectives. This pattern indicates controlled variability: although domain complexity affects short-term completion, the team consistently restores delivery performance through SCRUM feedback cycles.

3.2.2. Code Quality Metrics

Modularity and maintainability of the Hapi.js backend were assessed using three standard metrics: Coupling Between Objects (CBO), Lack of Cohesion of Methods (LCOM*), and Maintainability Index (MI). CBO and LCOM* members of the CK metric family remain widely used in empirical studies (including IEEE venues) as indicators of coupling and cohesion at the module/class level [20]. MI follows Plato/escomplex (a composite of cyclomatic complexity, lines of code, and Halstead volume), consistent with tertiary syntheses linking coupling/cohesion to maintainability and contemporary discussions of MI [25]

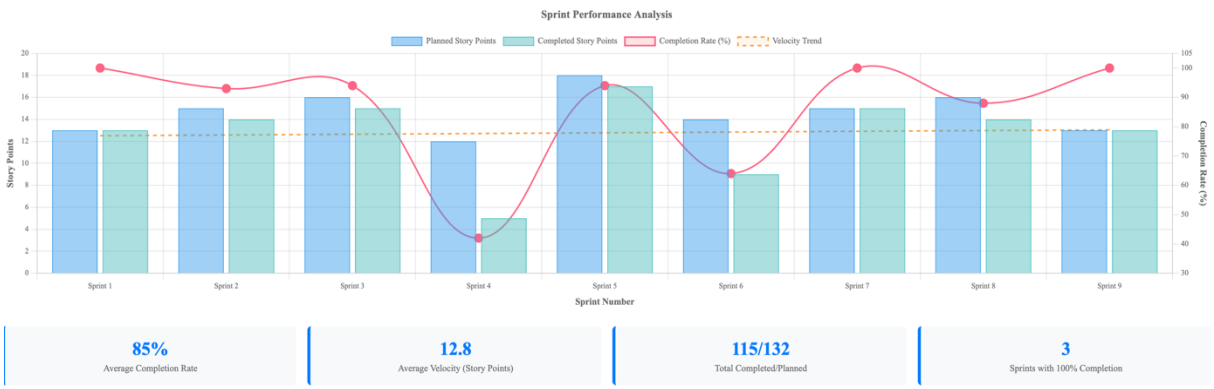

Figure 7. Sprint Velocity and Burn-down Analysis

Idealis: Indonesia Journal Information System
Volume 9, Nomor 1, Januari 2026
ISSN 2684-7280 (*online*)
Halaman 90-102
available online at http://jom.fti.budiluhur.ac.id/index.php/IDEALIS/index

Table 3. lists per-domain metrics (lower is better for CBO/LCOM*; higher is better for MI).

Module	CBO	LCOM*	MI	Files
api/advertisements	0	0	64.87	3
api/amenities	0	0	63.30	3
api/authentications	0	0	65.78	3
api/banks	0	0	66.06	3
api/cards	0	0	65.26	3
api/clients	0	0	47.93	2
api/clientsFacilities	0	0	65.70	3
api/clientTypes	0	0	64.57	3
api/contacts	0	0	70.14	3
api/destinations	0	0	63.71	3
api/details	0	0	59.00	3
api/docs	0	0	77.64	2
api/facilities	0	0	67.86	3
api/options	0	0	63.30	3
api/orders	0	0	47.60	2
api/policies	0	0	63.46	3
api/products	0	0	43.20	2
api/roles	0	0	70.20	3
api/users	0	0	56.68	3
api/verifications	0	0	70.25	3
api/withdrawals	0	0	61.30	3
core/	0	0	68.47	84

Files are grouped by first-level API domains (e.g., api/users, api/orders). CBO counts other API domains imported by a given domain; dependencies on core/ components and external packages are excluded to isolate inter-API coupling. Cohesion is approximated by an import-based proxy (LCOM*), defined in (1) with I_intra the number of intra-domain imports and I_total the domain's total imports. MI is taken from Plato/escomplex reports[26].

$$LCOM^*(\quad) = 1 - (I\_intra \;/\; I\_total) \tag{1}$$

Results. Of 21 API domains, 9/21 achieved MI ≥ 65 (fair–good) and 4/21 exceeded MI ≥ 70 api/docs (77.64), api/verifications (70.25), api/roles (70.20), api/contacts (70.14). Three domains fell below 50 and require targeted refactoring: api/products (43.20), api/orders (47.60), api/clients (47.93). In this operationalization, all API domains report CBO = 0 and LCOM* = 0, indicating clean module boundaries and centralized reuse within core/. The core/ component achieved MI = 68.47 across 84 files[21].

Table 3 shows that most domains achieved fair-to-good maintainability (MI ≥ 65), while api/products, api/orders, and api/clients remain refactoring targets due to denser business logic and higher complexity concentration.

Consistently low inter-API coupling (CBO = 0) supports stable separation of concerns and reduces change-propagation risk. MI variability highlights business-logic–dense hotspots; literature-aligned refactors reducing branching (early returns), extracting validation/error mapping into core/, and isolating data access in repositories are expected to lift MI for low-scoring modules without increasing coupling. Recent studies continue to employ CBO (and variants) as maintainability proxies [27].

3.2.3. Redis Performance Testing

Performance was evaluated under three load scenarios to assess scalability and responsiveness. The results compare two configurations without caching and with Redis caching to validate whether the proposed architecture meets operational demands under realistic workloads.

Table 4. Performance Test Results

Metric	Test-1 (W/O → With)	Test-2 (W/O → With)	Test-3 (W/O → With)
Avg. resp. (ms)	8986 → 2	14946 → 2	11864 → 3
95th perc. (ms)	13056 → 3	26671 → 3	21104 → 7
RPS (avg)	769.24 → 41184.40	830.02 → 41108.67	592.64 → 27447.14
Memory (MB)	90 → 85	85 → 91	111 → 82

Idealis: Indonesia Journal Information System
Volume 9, Nomor 1, Januari 2026
ISSN 2684-7280 (*online*)
Halaman 90-102
available online at http://jom.fti.budiluhur.ac.id/index.php/IDEALIS/index

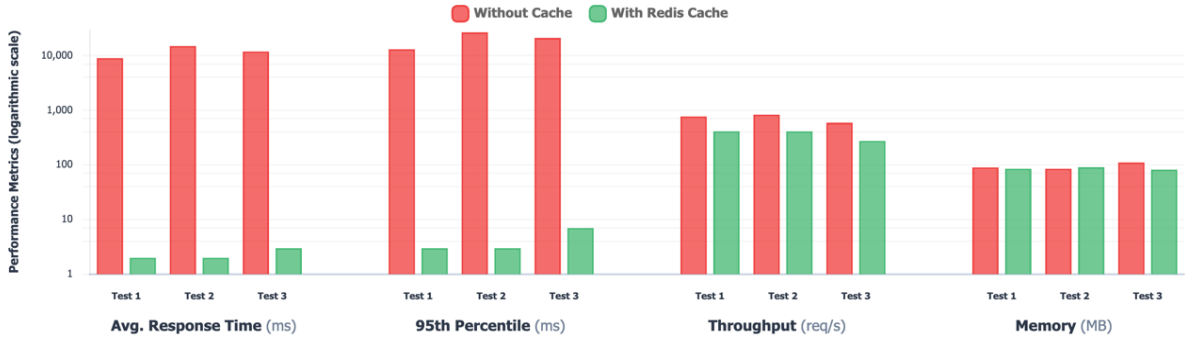

Figure 8. Performance Metrics Comparison: Without Cache vs With Redis Cache

Across the three scenarios, enabling Redis reduced the average response time from 8,986-14,946 ms to 2-3 ms and tightened p95 latency from 13,056-26,671 ms to 3-7 ms. Throughput increased from 592.64-830.02 req/s to 27,447.14-41,184.40 req/s (≈46-54×). Memory changes were moderate and scenario-dependent (90→85 MB; 85→91 MB; 111→82 MB).

Figure 8 provides a visual summary of Table 4 and highlights a consistent performance gap between both configurations. The Redis configuration shows substantially lower bars for response time and p95 latency while simultaneously achieving higher throughput across all tests, indicating that caching shifts hot read paths from database I/O to in-memory access. This behavior aligns with common caching trade-offs, where a small and scenario-dependent memory overhead can be acceptable in exchange for significant improvements in responsiveness and request handling capacity [28].

Production security and multi-tenant considerations. While Table 4 and Figure 8 show substantial scalability gains with Redis, the evaluation was conducted in a controlled development environment and security validation was limited to JWT-based authentication and authorization. Therefore, these results should be interpreted as feasibility evidence rather than full production readiness. Future work should validate production controls (e.g., rate limiting, audit logging, and key management) and evaluate multi-tenant isolation, including tenant-scoped access and cache-key partitioning to avoid cross-tenant interference.

Policy implications for halal tourism governance. A centralized Super Admin backend enables consistent governance across the platform by supporting partner verification, content moderation, transaction oversight, and system monitoring in one administrative layer. This consolidation strengthens accountability through standardized workflows and auditability, and it can improve trust and operational transparency for stakeholders within halal tourism ecosystems.

3.3. Final Documentation

This subsection describes the produced documentation artifacts that support API reuse and integration. It outlines how executable API specifications and front-end integration evidence were prepared to ensure replicability and practical adoption.

3.3.1. API Endpoint Distribution

Performance and maintainability are also influenced by how APIs are organized across domains. Therefore, this section reports the distribution of Super Admin endpoints to illustrate functional coverage and architectural consistency.

Table 5. API Endpoints Distribution

Domain	Endpoints	Methods	Key Features
Authentication	2	POST, DELETE	Login, logout with JWT
User Management	4	GET, POST, PUT, DELETE	CRUD operations for users
Mitra Management	4	GET, POST, PUT, DELETE	Mitra administration
Balance Management	1	GET	Financial monitoring
Config Management	16	GET, POST, PUT, DELETE	System configuration
Order Management	3	GET, PUT, DELETE	Order oversight
Advertisement Management	4	GET, POST, PUT, DELETE	Promotional content
Destination Management	4	GET, POST, PUT, DELETE	Tourism destinations
Type Management	4	GET, POST, PUT, DELETE	Client categorization
Total	42	Multiple	Complete CRUD

Table 5 summarizes 42 RESTful API endpoints across nine functional domains (auth, users, partners, finance/balance, config, orders, ads, destinations, client types). The distribution indicates comprehensive operational

Idealis: Indonesia Journal Information System
Volume 9, Nomor 1, Januari 2026
ISSN 2684-7280 (*online*)
Halaman 90-102
available online at http://jom.fti.budiluhur.ac.id/index.php/IDEALIS/index

coverage: eight domains implement full CRUD (GET/POST/PUT/DELETE) for end-to-end administration, while Balance Management is exposed as read-only (GET) to mitigate risk of direct financial data mutation. Design consistency resource-oriented naming, uniform JSON response contracts, standardized HTTP status codes, and a unified error model accelerates client integration and reduces integration defects. JWT in the Authorization: Bearer <token> header preserves statelessness and facilitates horizontal scaling. Architecturally, domain segmentation aligns with separation of concerns: changes in one domain (e.g., ads or destinations) do not cascade to others (e.g., users or partners), thereby improving modularity, maintainability, and cross-channel reusability (web and mobile).

3.3.2. Frontend API implementation

Front-end integration was validated through representative modules. Figure 9 (User Management) shows consumption of the user-list endpoint (GET /users) with pagination and sorting, and administrative actions mapping detail/create/update/delete to routes (GET /users/{id}; POST/PUT/DELETE /users/{id}) per contract. Figure 10 (Clients) demonstrates a similar pattern for the client domain, including UI-state mapping to HTTP codes (200/201/204 for success; 400/401/403/404/409/422 for anticipated errors). Error behavior is handled uniformly from a standard payload (fields code, message, details), so UI components require no domain-specific adapters. These integration artifacts confirm that (i) endpoint specifications are consistently consumable by the front end, (ii) JWT authentication functions end to end, and (iii) uniform response contracts reduce boilerplate in the presentation layer. Overall, the results validate API readiness for centralized administrative orchestration via the Super Admin interface and underscore the benefits of domain-based modular design for integration speed and evolutionary stability.

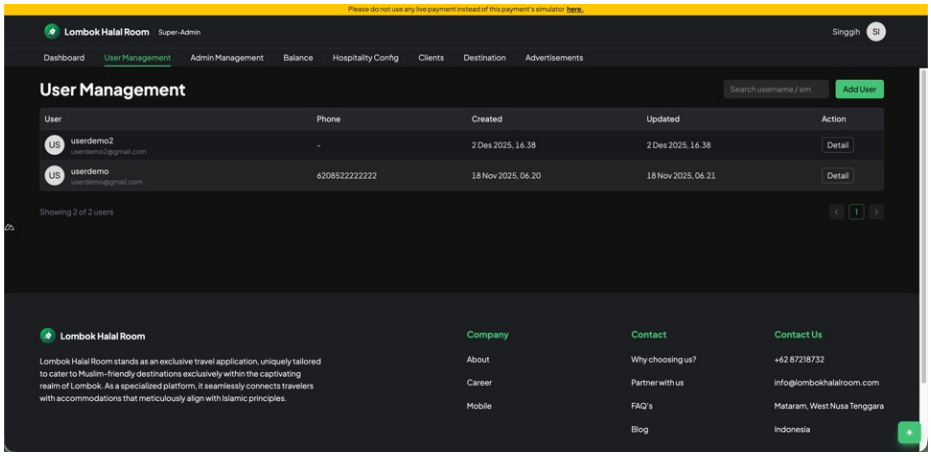

Figure 9. User Management Interface Demonstrating Frontend Integration with Backend API

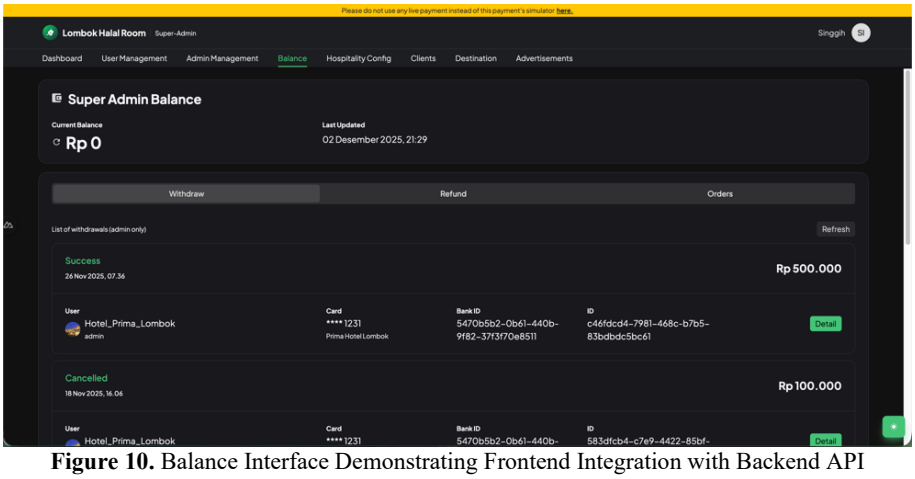

Figure 10. Balance Interface Demonstrating Frontend Integration with Backend API

4. CONCLUSION

This study designed and implemented a domain-oriented, modular RESTful API backend to enable centralized Super Admin operations in the Lombok Halal Room (LHR) platform. The proposed solution follows a layered

architecture with consistent JSON response contracts and JWT-based authentication, and it delivers 42 endpoints across the core administrative domains. Front-end integration was validated, indicating that the backend is reusable and deployable for both web and mobile administrative workflows.

The scientific novelty of this work lies in combining an iterative SCRUM-based development process with a modular RESTful API backend architecture in the specific context of halal tourism administration, supported by systematic post-implementation evaluation using performance and software-quality metrics. This integrated approach provides a replicable engineering method for developing scalable administrative backends beyond a single application setting.

Experimental results show that Redis caching substantially improves responsiveness and scalability under load. Across the evaluated scenarios, average response time decreased to 2–3 ms and p95 latency tightened to 3–7 ms, while throughput increased to 27,447–41,184 requests/s. In terms of software quality, CBO=0 and LCOM*=0 indicate clean module boundaries, and the Maintainability Index (MI) offers actionable guidance for identifying modules that should be prioritized for refactoring.

This study has limitations. The experiments were conducted in a controlled environment using simulated workloads and therefore do not fully represent multi-tenant production variability or end-to-end security constraints. As a result, the reported findings should be interpreted as evidence of feasibility and engineering effectiveness within the evaluated setup.

Future work should build directly on these results by (1) refactoring modules with lower MI to reduce complexity concentration, (2) extending evaluation to multi-tenant and production-like workloads, and (3) improving operational readiness through observability, rate limiting, and security hardening. These directions are expected to increase robustness while preserving the latency and throughput gains demonstrated by Redis caching.

From a practical perspective, the proposed backend can strengthen governance in halal tourism digital ecosystems by enabling centralized verification, content moderation, transaction oversight, and system monitoring. As a reusable architectural template, it can serve as a reference implementation for similar halal tourism platforms in other regions.

BIBLIOGRAPHY

[1] A. Prawiro, "Halal Tourism in Lombok: Harmonization of Religious Values and Socio-Cultural Identity," *Share: Jurnal Ekonomi dan Keuangan Islam*, vol. 11, no. 2, pp. 322–345, Dec. 2022, doi: 10.22373/share.v11i2.14905.

[2] Mahendra Putra Raharja, Ramaditia Dwiyansaputra, Royana Afwani, Gibran Satya Nugraha, Fahru Alfarizi Hananza Putrawan, and Mursyidhan Ariefbillah Ahmad Masri, "Backend Development of a Halal Tourism Application Based on Service for Cross-Platform Application for Travelers," *3rd MIMSE 2024*, 2024.

[3] Diaz Khalid Ananda, Ramaditia Dwiyansaputra, Royana Afwani, Gibran Satya Nugraha, Mochammad Dinta Alif Syaifuddin, and M. Asrorul Khopid, "Integrated Halal Tourism Applications Based on REST API for Hotel Partners and Travel Agents," *3rd MIMSE 2024*, 2024.

[4] S. B. Hasan, Y. Nader Abdullah, and M. Khalil Darwesh, "Design and Implementation The digital transformation of the student clearance system for Soran University," *Academic Journal of Nawroz University*, vol. 12, no. 3, pp. 252–261, Aug. 2023, doi: 10.25007/ajnu.v12n3a1633.

[5] S. Sharma, O. P. Rishi, and A. Sharma, "IoTeST: IoT-Enabled Smart Tourism—Shaping the Future of Tourism," 2021, pp. 569–576. doi: 10.1007/978-981-15-6014-9_67.

[6] I. Shabani, E. Mëziu, B. Berisha, and T. Biba, "Design of Modern Distributed Systems based on Microservices Architecture," *International Journal of Advanced Computer Science and Applications*, vol. 12, no. 2, 2021, doi: 10.14569/IJACSA.2021.0120220.

[7] C. Li and B. Niu, "IOT Gateway Based on Microservices," 2021, pp. 26–33. doi: 10.1007/978-3-030-79197-1_4.

[8] T. T. Nguyen, M. T. Nguyen, and N. T. Le, "An Administrative Support System for Digital Transformation of Small and Medium-Sized Enterprises in Vietnam," *Foundations of Management*, vol. 16, no. 1, pp. 177–194, Jan. 2024, doi: 10.2478/fman-2024-0011.

[9] D. S. Pashchenko, "Refining the Scrum Paradigm: A Comprehensive Research of Software Development Practices (2020–2023)," *Computing&AI Connect*, vol. 1, no. 1, p. 1, Dec. 2024, doi: 10.69709/CAIC.2024.103102.

[10] D. D. Wazaumi, V. A. Saputro, S. K. Nisa, and S. A. Zahrani, "Implementasi Framework Scrum Dalam Pengembangan Dashboard Monitoring Untuk Optimasi Pengelolaan Data Interface," *IDEALIS : InDonEsiA journaL Information System*, vol. 8, no. 1, pp. 64–73, Jan. 2025, doi: 10.36080/idealis.v8i2.3338.

[11] L. Chamari, E. Petrova, and P. Pauwels, "An End-to-End Implementation of a Service-Oriented Architecture for Data-Driven Smart Buildings," *IEEE Access*, vol. 11, pp. 117261–117281, 2023, doi: 10.1109/ACCESS.2023.3325767.

[12] I. R. D. Muhammad and I. V. Paputungan, "Development of Backend Server Based on REST API Architecture in E-Wallet Transfer System," *Jurnal Sains, Nalar, dan Aplikasi Teknologi Informasi*, vol. 3, no. 2, pp. 79–87, Jan. 2024, doi: 10.20885/snati.v3.i2.35.

[13] D. Arya, H. Putra, E. Darwiyanto, and R. Nurtantyana, "Development of Backend Admin Dashboard for Business Project Monitoring using Scrum Method," Aug. 2024, doi: 10.34818/indojc.2024.9.2.969.

[14] T. Sundara, D. Setiawan, F. Subkhan, and F. R. Kautsar, "Scrum Implementation in Development of Online Research Application," *The Indonesian Journal of Computer Science*, vol. 11, no. 2, Aug. 2022, doi: 10.33022/ijcs.v11i2.3072.

[15] C. Verwijs and D. Russo, "A Theory of Scrum Team Effectiveness," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 3, pp. 1–51, Jul. 2023, doi: 10.1145/3571849.
[16] A. R. P. Ramadhan, I. Waspada, N. Bahtiar, and A. S. Pramayoga, "Applying the Scrum Method in Software Development for Undergraduate Thesis Project Implementation," *Jurnal Masyarakat Informatika*, vol. 16, no. 1, pp. 119–133, May 2025, doi: 10.14710/jmasif.16.1.73187.
[17] C. Franco *et al.*, "Introducing ScrumAdemia: An Agile Guide for Doctoral Research," *PS Polit Sci Polit*, vol. 56, no. 2, pp. 251–258, Apr. 2023, doi: 10.1017/S1049096522001408.
[18] Suchetha Vijayakumar, Krishna Prasad K, and R. Holla M., "Assessing the Effectiveness of MoSCoW Prioritization in Software Development: A Holistic Analysis across Methodologies," *EAI Endorsed Transactions on Internet of Things*, vol. 10, Oct. 2024, doi: 10.4108/eetiot.6515.
[19] I. Heriyawan, U. Hayati, and O. Nurdiawan, "Rancang Bangun Sistem Informasi Akuntansi Menggunakan Codeigniter Dengan Metode Scrum Studi Kasus : Pt Surya Marga Sarana," 2023.
[20] F. N. Colakoglu, A. Yazici, and A. Mishra, "Software Product Quality Metrics: A Systematic Mapping Study," *IEEE Access*, vol. 9, pp. 44647–44670, 2021, doi: 10.1109/ACCESS.2021.3054730.
[21] M. Klima *et al.*, "Selected Code-Quality Characteristics and Metrics for Internet of Things Systems," *IEEE Access*, vol. 10, pp. 46144–46161, 2022, doi: 10.1109/ACCESS.2022.3170475.
[22] N. Laranjeiro, J. Agnelo, and J. Bernardino, "A Black Box Tool for Robustness Testing of REST Services," *IEEE Access*, vol. 9, pp. 24738–24754, 2021, doi: 10.1109/ACCESS.2021.3056505.
[23] S. Shafiee, Y. Wautelet, S. Poelmans, and S. Heng, "An empirical evaluation of scrum training's suitability for the model-driven development of knowledge-intensive software systems," *Data Knowl Eng*, vol. 146, p. 102195, Jul. 2023, doi: 10.1016/j.datak.2023.102195.
[24] W.-T. Lee and C.-H. Chen, "Agile Software Development and Reuse Approach with Scrum and Software Product Line Engineering," *Electronics (Basel)*, vol. 12, no. 15, p. 3291, Jul. 2023, doi: 10.3390/electronics12153291.
[25] A. Abbad-Andaloussi, "On the relationship between source-code metrics and cognitive load: A systematic tertiary review," *Journal of Systems and Software*, vol. 198, p. 111619, Apr. 2023, doi: 10.1016/j.jss.2023.111619.
[26] U. Iftikhar, N. Bin Ali, J. Börstler, and M. Usman, "A tertiary study on links between source code metrics and external quality attributes," *Inf Softw Technol*, vol. 165, p. 107348, Jan. 2024, doi: 10.1016/j.infsof.2023.107348.
[27] P. Sun, D.-K. Kim, H. Ming, and L. Lu, "Measuring Impact of Dependency Injection on Software Maintainability," *Computers*, vol. 11, no. 9, p. 141, Sep. 2022, doi: 10.3390/computers11090141.
[28] M. T. Faridi, K. Singh, K. Soni, and S. Negi, "Memcached vs Redis Caching Optimization Comparison using Machine Learning," in *2023 2nd International Conference on Automation, Computing and Renewable Systems (ICACRS)*, IEEE, Dec. 2023, pp. 1153–1159. doi: 10.1109/ICACRS58579.2023.10404339.