Rikugan

Architecture and Design Document

Version 1.0
Printing Date: December 22, 2025

Group ID: Group 9

Group Members:

1155211784	Fok Chun Yin
1155211541	HO Sum Ming
1155211779	Chi Ho KWOK
1155211479	NG Ka Long
1155214295	SIN Lok Man

Revision History:

Version	**Date**	**Author**	**Description**
1.0	December 13, 2025	Group 9	Initial release

Contents

1 Introduction and Goals

Rikugan is a gamified project management web application combining Kanban functionality with bounty-based task rewards for software development teams.

1.1 Requirements Overview

Core Features:

- Role-based management: Goons (task workers), Hashira (task creators), Oyakatasama (admins)
- Bounty system with monetary task rewards and deadline penalties
- Real-time notifications and license-based access control

1.2 Quality Goals

Priority	Goal	Target
1	Usability	Intuitive interface, <10min learning curve
2	Security	JWT auth, RBAC, data protection
3	Performance	<500ms API response, 50 concurrent users
4	Maintainability	Modular architecture, 70% test coverage
5	Scalability	Support 200 users, 1000 tasks

Table 1: Quality Goals

1.3 Stakeholders

The primary stakeholders include:

- **Goons**: Junior developers who complete tasks
- **Hashira**: Senior developers who create and assign tasks
- **Oyakatasama**: System administrators with full access

2 Architecture Constraints

2.1 Technical Constraints

- React 18+
- Node.js
- MySQL 8.0+
- Docker
- Web-based access

2.2 Organizational Constraints

- 3-4 student developers
- One semester timeline
- Git version control
- 70% test coverage requirement

2.3 Conventions

- RESTful API architecture
- arc42 documentation standard
- ESLint code standards
- Naming conventions:
  - snake_case for database
  - camelCase for JavaScript code
  - PascalCase for React components

2.4 Source Control

The project uses **GitHub** for version control and collaboration with the following workflow:

Repository: Centralized GitHub repository for all source code, documentation, and configuration files

Branching Strategy: Feature branching workflow

- `main`: Production-ready stable branch

- Feature branches: Descriptive names for development work (e.g., `frontend`$_{b}uildup$, `backend`$_{t}esting$

Workflow:
1. Create feature branch from `main`
2. Develop and commit changes locally
3. Push feature branch to GitHub
4. Merge to `main` after group approval

Commit Standards: Descriptive commit messages following conventional commits format

Code Review: All changes require peer review before merging

3 System Scope and Context

3.1 Business Context

User Roles:

Goons: Browse and complete tasks, earn bounties

Hashira: Create tasks, manage teams, includes all Goon functions

Oyakatasama: Full system administration, user and license management

3.2 Technical Context

Technical Components:

- **Frontend**: React 18+ with HeroUI (browser-based UI)
- **Backend**: Node.js/Express.js (RESTful API, authentication, business logic)
- **Database**: MySQL 8.0+ (users, tasks, bounties, notifications, licenses)
- **Deployment**: Docker containerization

4 Solution Strategy

Architecture: Role-based hierarchy (Goons → Hashira → Oyakatasama) + bounty incentive system

Stack: React 18+ (HeroUI), Node.js/Express.js (MVC), MySQL (normalized schema), Docker

Security: JWT authentication, RBAC at API/component levels

Key Decisions: Bounty-first design, license-controlled access, API-first development, audit trails

5 Building Block View

5.1 System Architecture

The system follows a three-tier architecture pattern as illustrated in Figure 1:

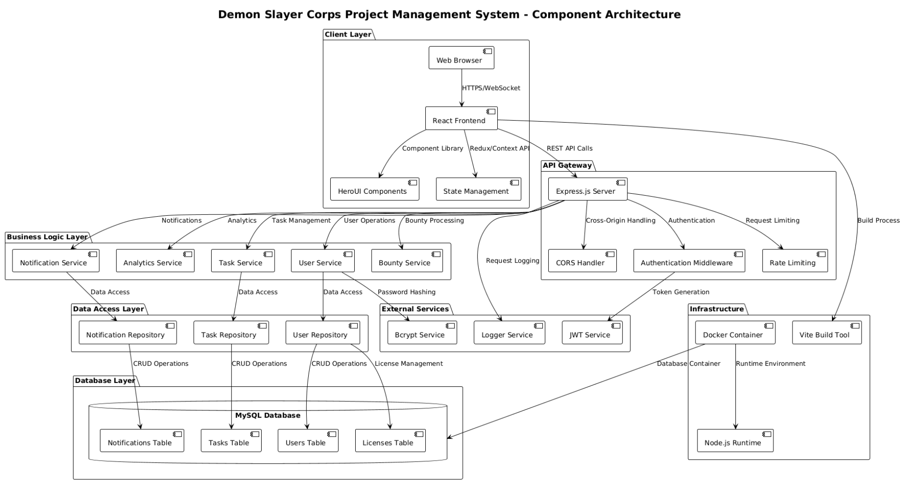


Figure 1: System Architecture - Three-Tier Pattern

- **Client Layer**: React UI with HeroUI components, state management
- **API Gateway**: Express.js server, JWT authentication, routing, validation
- **Database**: MySQL with normalized schema, connection pooling

5.2 Use Case View

The system supports various use cases for different user roles, including task management, bounty processing, and team administration.

5.3 Backend MVC Architecture

Model: Data entities, business logic, database interactions

Controller: HTTP request handling, route management

View: JSON response formatting

5.4 Backend Black Box View

The backend system acts as a RESTful API server, exposing interfaces to the frontend and managing all business logic and data persistence.

Purpose and Responsibility

The backend serves as the central business logic and data management layer with the following responsibilities:

- **Authentication & Authorization**: Validate user credentials, generate JWT tokens, enforce role-based access control
- **Data Management**: CRUD operations for all entities (users, tasks, teams, bounties, licenses)

- **Business Logic**: Implement bounty calculations, task lifecycle rules, deadline penalties
- **Notification Processing**: Generate and deliver real-time notifications for system events
- **License Validation**: Enforce team access control and license expiration policies

Interfaces

HTTP REST API Endpoints:

Endpoint	Methods	Purpose
/api/auth	POST	Login, logout, token refresh
/api/users	GET, POST, PUT, DELETE	User management
/api/tasks	GET, POST, PUT, DELETE	Task CRUD and assignment
/api/bounties	GET, POST	Bounty payments and history
/api/teams	GET, POST, PUT, DELETE	Team management
/api/licenses	GET, POST, PUT	License validation
/api/notifications	GET, PUT	Notification retrieval and updates

Table 2: Backend API Endpoints

External Dependencies:

- MySQL database connection (port 3306)
- Environment configuration (.env file)
- JWT secret keys for token signing

Quality Attributes

- **Availability**: 99% uptime during business hours
- **Performance**: <500ms average response time for API calls
- **Scalability**: Support up to 200 concurrent users
- **Security**: JWT authentication, input validation, SQL injection prevention
- **Maintainability**: Modular architecture, 70% test coverage

5.5 Backend White Box View

The backend internal structure follows the Model-View-Controller (MVC) pattern with clear separation of concerns across multiple layers.

Internal Structure

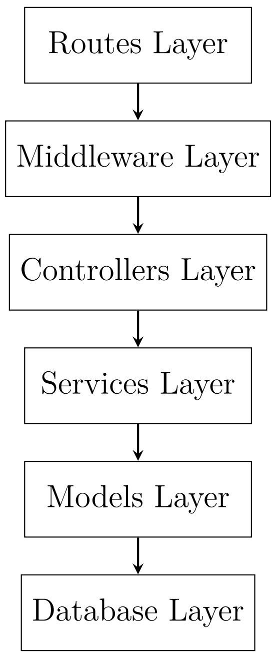


Figure 2: Backend Layered Architecture

Component Descriptions

Routes Layer (routes/)
Defines API endpoints and maps HTTP requests to controllers. Registers all REST routes and applies route-specific middleware.

Middleware Layer (middleware/)
Provides cross-cutting concerns:

- `auth.js`: JWT token validation and extraction
- `roleCheck.js`: Role-based access control enforcement
- `validation.js`: Request payload validation
- `errorHandler.js`: Global error handling and formatting
- `logger.js`: Request/response logging

Controllers Layer (controllers/)
Handles HTTP requests and responses, orchestrates service calls:

- `AuthController`: Login, logout, token management
- `UserController`: User CRUD operations
- `TaskController`: Task management and assignment
- `BountyController`: Bounty processing and payment
- `TeamController`: Team operations
- `LicenseController`: License validation
- `NotificationController`: Notification delivery

Services Layer (services/)

Implements core business logic:

- `AuthService`: Authentication, JWT generation/validation
- `UserService`: User management, balance updates
- `TaskService`: Task lifecycle, status transitions, deadline checks
- `BountyService`: Payment calculations, penalty processing
- `NotificationService`: Event-based notification generation
- `LicenseService`: License validation, expiration checks

Models Layer (models/)

Data access objects (DAOs) for database operations:

- `User`: User entity CRUD
- `Task`: Task entity CRUD
- `Team`: Team entity CRUD
- `License`: License entity CRUD
- `Notification`: Notification entity CRUD
- `Transaction`: Transaction entity CRUD (immutable)

Database Layer (database/)

MySQL connection management:

- `connection.js`: Connection pool configuration
- `migrations/`: Database schema version control

Data Flow

A typical API request flows through the system as follows:

1. **Request Reception**: HTTP request arrives at Express server
2. **Route Matching**: Router matches URL pattern to controller method
3. **Middleware Processing**: Sequential execution of middleware chain
    - Logger records request details
    - Auth middleware validates JWT token
    - Role check enforces permissions
    - Validator checks request payload
4. **Controller Execution**: Controller receives validated request
5. **Service Invocation**: Controller delegates to appropriate service
6. **Business Logic**: Service executes business rules
7. **Model Interaction**: Service calls model methods for data access

8. **Database Operation**: Model executes SQL queries via connection pool
9. **Response Formation**: Results propagate back through layers
10. **JSON Response**: Controller formats and sends HTTP response
11. **Error Handling**: Any errors caught by global error handler

Key Design Patterns

- **Layered Architecture**: Clear separation between routes, controllers, services, and models
- **Dependency Injection**: Services injected into controllers, models into services
- **Repository Pattern**: Models act as data repositories abstracting database access
- **Middleware Chain**: Composable request processing pipeline
- **Service Layer Pattern**: Business logic isolated from HTTP concerns
- **Error Handling Strategy**: Centralized error handling with custom error types

5.6 Domain Model

The domain model (Figure 3) defines the core entities and their relationships within the system, including Users, Tasks, Teams, Licenses, Transactions, and Notifications.

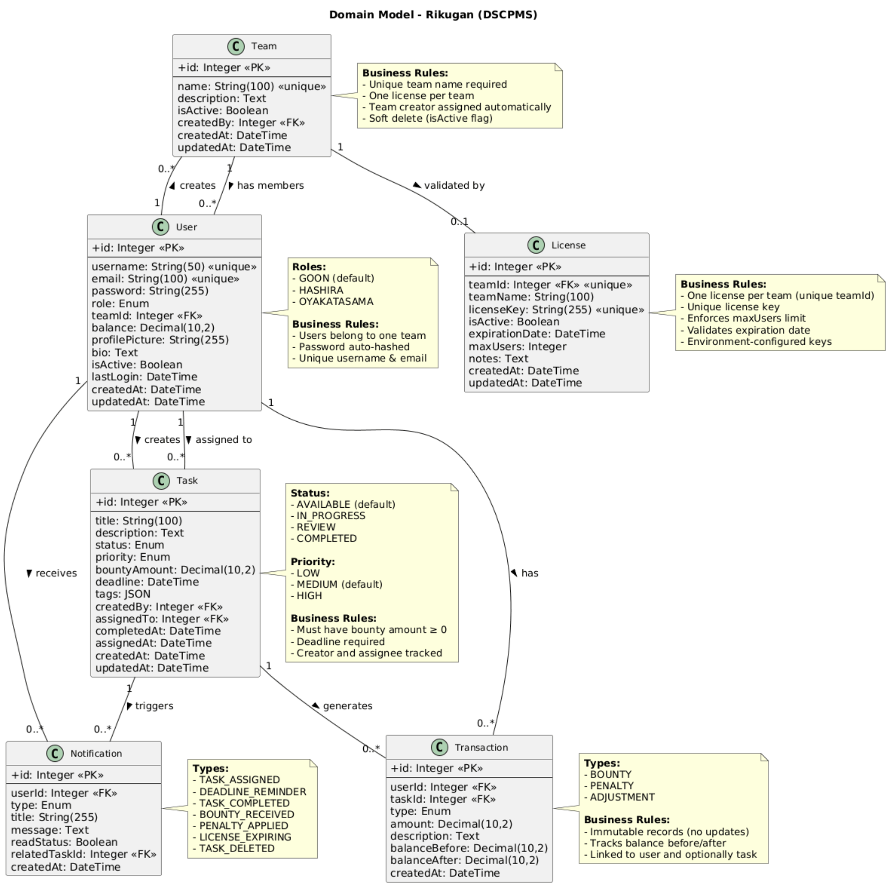


Figure 3: Domain Model - Core Entities and Relationships

5.7 Backend Modules

Module	Responsibilities
auth	JWT authentication, login/logout, token management
users	User CRUD, profiles, team membership, earnings
tasks	Task lifecycle, assignment, status tracking
bounties	Payment processing, balance management, penalties
notifications	Event notifications, delivery, preferences
licenses	License validation, team access control
teams	Team management, member operations, statistics

Table 3: Backend Modules and Responsibilities

6 Runtime View

6.1 User Authentication Flow

The authentication flow implements secure JWT-based authentication as shown in Figure 4. The process follows these steps:

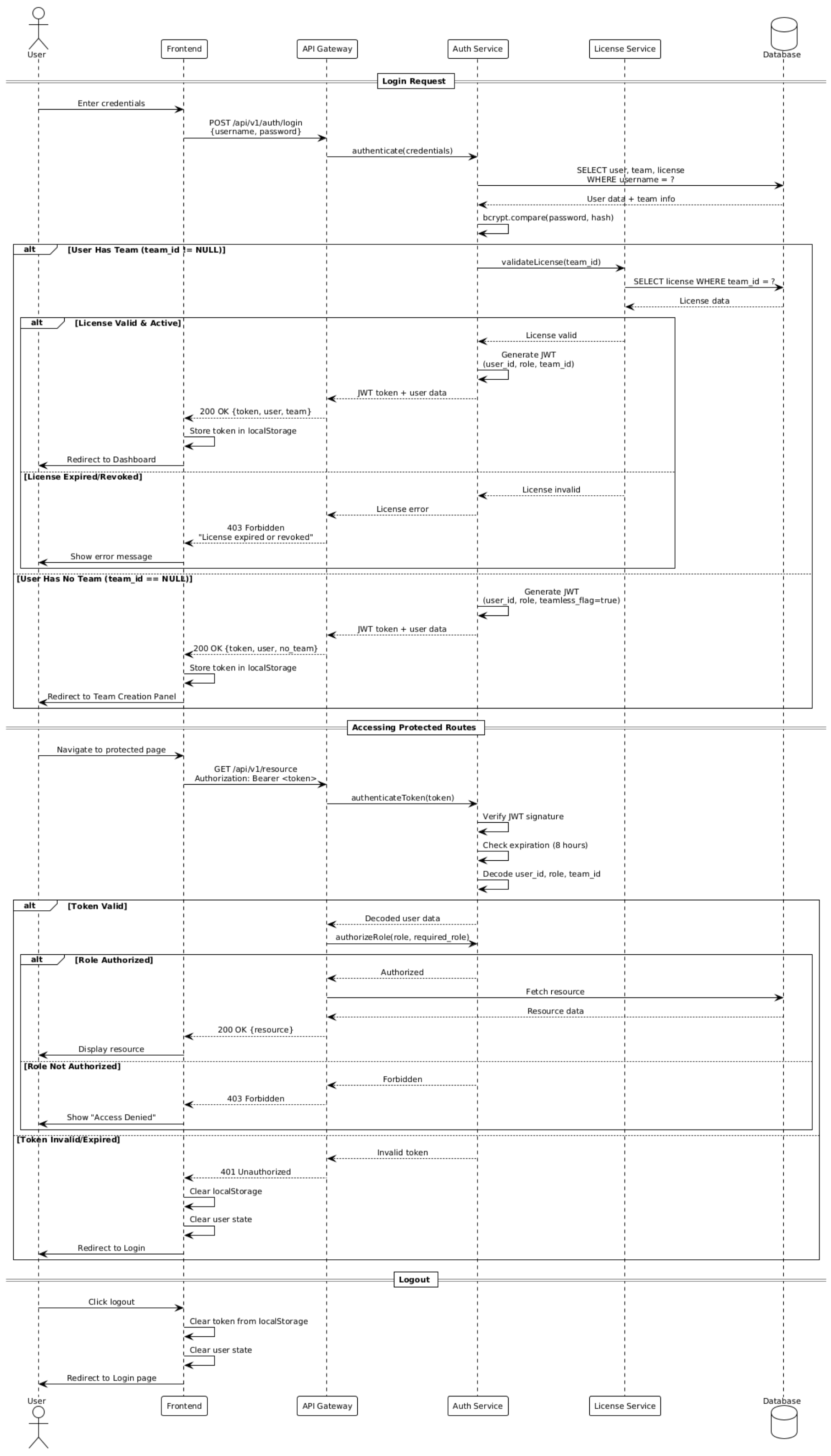

User
Frontend
API Gateway
Auth Service
License Service
Database
Login Request
Enter credentials
POST /api/v1/auth/login {username, password}
authenticate(credentials)
SELECT user, team, license WHERE username = ?
User data + team info
bcrypt.compare(password, hash)
alt [User Has Team (team_id != NULL)]
validateLicense(team_id)
SELECT license WHERE team_id = ?
License data
alt [License Valid & Active]
License valid
Generate JWT (user_id, role, team_id)
JWT token + user data
200 OK {token, user, team}
Store token in localStorage
Redirect to Dashboard
[License Expired/Revoked]
License invalid
License error
403 Forbidden "License expired or revoked"
Show error message
[User Has No Team (team_id == NULL)]
Generate JWT (user_id, role, teamless_flag=true)
JWT token + user data
200 OK {token, user, no_team}
Store token in localStorage
Redirect to Team Creation Panel
Accessing Protected Routes
Navigate to protected page
GET /api/v1/resource Authorization: Bearer <token>
authenticateToken(token)
Verify JWT signature
Check expiration (8 hours)
Decode user_id, role, team_id
alt [Token Valid]
Decoded user data
authorizeRole(role, required_role)
alt [Role Authorized]
Authorized
Fetch resource
Resource data
200 OK {resource}
Display resource
[Role Not Authorized]
Forbidden
403 Forbidden
Show "Access Denied"
[Token Invalid/Expired]
Invalid token
401 Unauthorized
Clear localStorage
Clear user state
Redirect to Login
Logout
Click logout
Clear token from localStorage
Clear user state
Redirect to Login page

Figure 4: User Authentication Flow - JWT Token-Based Authentication

1. User submits credentials
2. Backend validates credentials
3. JWT token is generated and returned

4. Token is stored on client for subsequent requests
5. Token is validated on protected endpoints

6.2 User Registration and Team Creation Flow

Figure 5 illustrates the complete workflow for user registration and team formation, showing the interaction between users, the backend system, and the database during the team creation process.

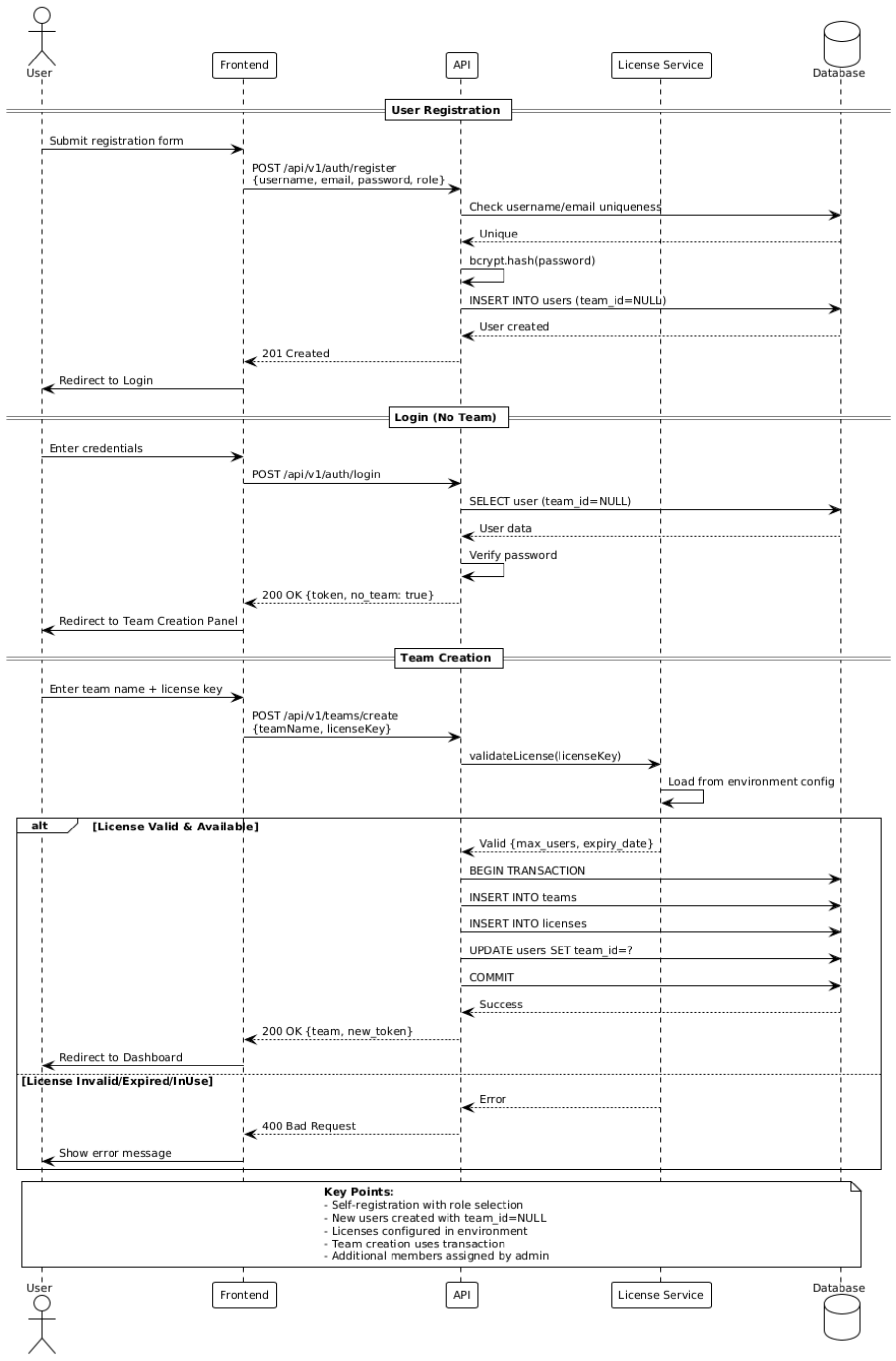

User
Frontend
API
License Service
Database
User Registration
Submit registration form
POST /api/v1/auth/register {username, email, password, role}
Check username/email uniqueness
Unique
bcrypt.hash(password)
INSERT INTO users (team_id=NULL)
User created
201 Created
Redirect to Login
Login (No Team)
Enter credentials
POST /api/v1/auth/login
SELECT user (team_id=NULL)
User data
Verify password
200 OK {token, no_team: true}
Redirect to Team Creation Panel
Team Creation
Enter team name + license key
POST /api/v1/teams/create {teamName, licenseKey}
validateLicense(licenseKey)
Load from environment config
alt
[License Valid & Available]
Valid {max_users, expiry_date}
BEGIN TRANSACTION
INSERT INTO teams
INSERT INTO licenses
UPDATE users SET team_id=?
COMMIT
Success
200 OK {team, new_token}
Redirect to Dashboard
[License Invalid/Expired/InUse]
Error
400 Bad Request
Show error message
Key Points:
- Self-registration with role selection
- New users created with team_id=NULL
- Licenses configured in environment
- Team creation uses transaction
- Additional members assigned by admin
User
Frontend
API
License Service
Database

Figure 5: User Registration and Team Creation Workflow

6.3 Task Assignment and Completion

The complete task lifecycle is depicted in Figure 6, showing the progression from task creation through completion and bounty distribution.

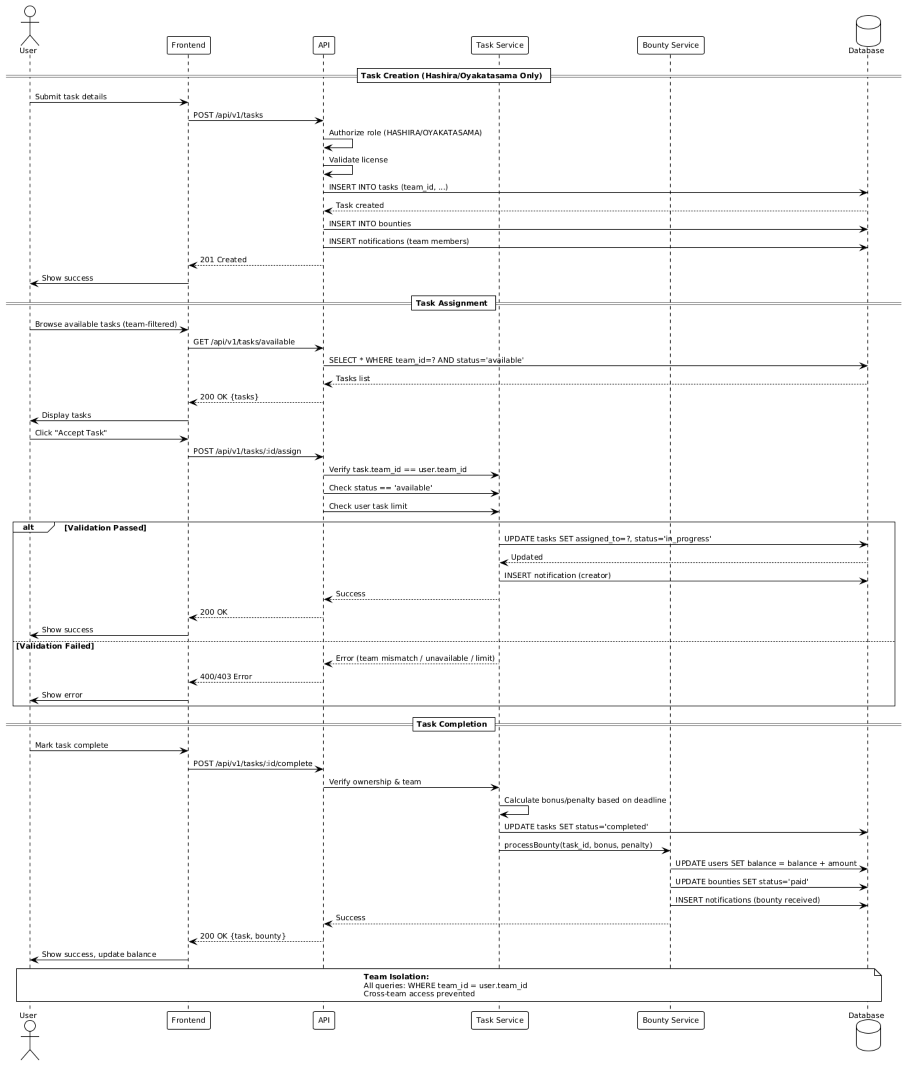

User
Frontend
API
Task Service
Bounty Service
Database
Task Creation (Hashira/Oyakatasama Only)
Submit task details
POST /api/v1/tasks
Authorize role (HASHIRA/OYAKATASAMA)
Validate license
INSERT INTO tasks (team_id, ...)
Task created
INSERT INTO bounties
INSERT notifications (team members)
201 Created
Show success
Task Assignment
Browse available tasks (team-filtered)
GET /api/v1/tasks/available
SELECT * WHERE team_id=? AND status='available'
Tasks list
200 OK {tasks}
Display tasks
Click "Accept Task"
POST /api/v1/tasks/:id/assign
Verify task.team_id == user.team_id
Check status == 'available'
Check user task limit
alt
[Validation Passed]
UPDATE tasks SET assigned_to=?, status='in_progress'
Updated
INSERT notification (creator)
Success
200 OK
Show success
[Validation Failed]
Error (team mismatch / unavailable / limit)
400/403 Error
Show error
Task Completion
Mark task complete
POST /api/v1/tasks/:id/complete
Verify ownership & team
Calculate bonus/penalty based on deadline
UPDATE tasks SET status='completed'
processBounty(task_id, bonus, penalty)
UPDATE users SET balance = balance + amount
UPDATE bounties SET status='paid'
INSERT notifications (bounty received)
Success
200 OK {task, bounty}
Show success, update balance
Team Isolation:
All queries: WHERE team_id = user.team_id
Cross-team access prevented
User
Frontend
API
Task Service
Bounty Service
Database

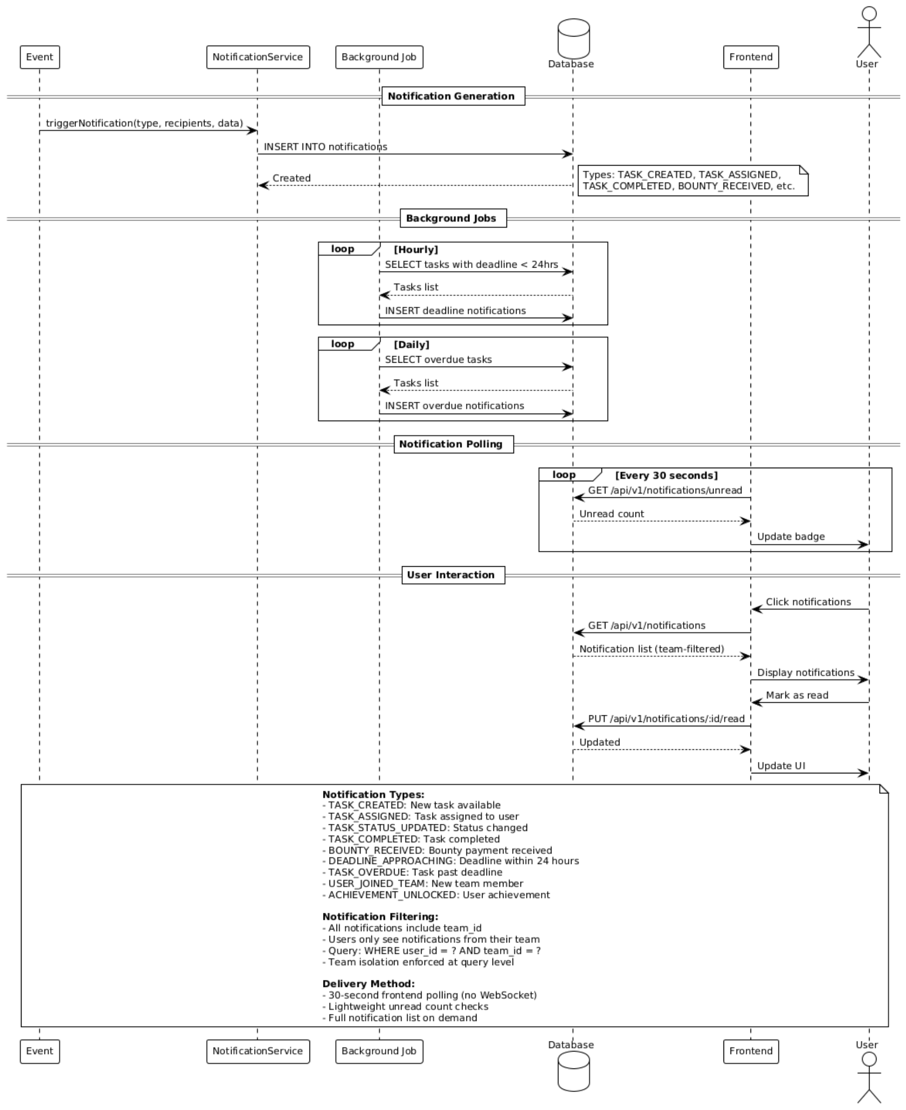
Figure 6: Task Lifecycle - From Creation to Completion

The workflow follows these key stages:

1. Task creation by Hashira
2. Task assignment to Goon
3. Task status progression: AVAILABLE → IN_PROGRESS → REVIEW → COMPLETED
4. Bounty distribution upon completion

6.4 Notification System Flow

The notification system (Figure 7) provides real-time updates to keep users informed of important events:


Event
NotificationService
Background Job
Database
Frontend
User
Notification Generation
triggerNotification(type, recipients, data)
INSERT INTO notifications
Created
Types: TASK_CREATED, TASK_ASSIGNED, TASK_COMPLETED, BOUNTY_RECEIVED, etc.
Background Jobs
loop [Hourly]
SELECT tasks with deadline < 24hrs
Tasks list
INSERT deadline notifications
loop [Daily]
SELECT overdue tasks
Tasks list
INSERT overdue notifications
Notification Polling
loop [Every 30 seconds]
GET /api/v1/notifications/unread
Unread count
Update badge
User Interaction
Click notifications
GET /api/v1/notifications
Notification list (team-filtered)
Display notifications
Mark as read
PUT /api/v1/notifications/:id/read
Updated
Update UI

Notification Types:
- TASK_CREATED: New task available
- TASK_ASSIGNED: Task assigned to user
- TASK_STATUS_UPDATED: Status changed
- TASK_COMPLETED: Task completed
- BOUNTY_RECEIVED: Bounty payment received
- DEADLINE_APPROACHING: Deadline within 24 hours
- TASK_OVERDUE: Task past deadline
- USER_JOINED_TEAM: New team member
- ACHIEVEMENT_UNLOCKED: User achievement

Notification Filtering:
- All notifications include team_id
- Users only see notifications from their team
- Query: WHERE user_id = ? AND team_id = ?
- Team isolation enforced at query level

Delivery Method:
- 30-second frontend polling (no WebSocket)
- Lightweight unread count checks
- Full notification list on demand

Figure 7: Notification System Flow - Real-Time Event Processing

Notification Types:

- Task assignments
- Status changes
- Bounty payments
- Deadline reminders

6.5 License Validation and Team Access Control

The license validation system (Figure 8) enforces team access control and ensures compliance with licensing requirements:

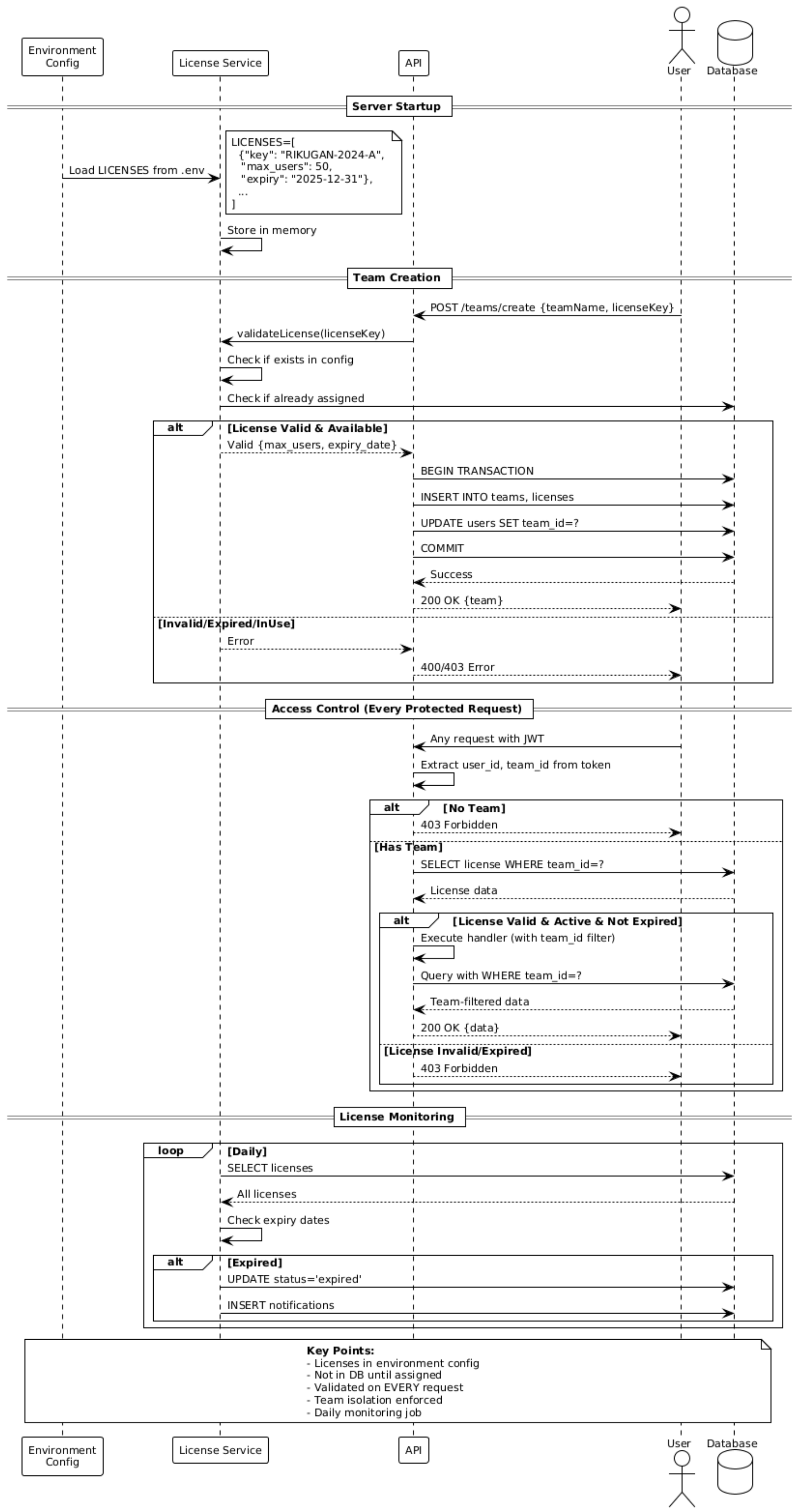

Environment Config
License Service
API
User
Database
Server Startup
Load LICENSES from .env
LICENSES=[{"key": "RIKUGAN-2024-A", "max_users": 50, "expiry": "2025-12-31"}, ...]
Store in memory
Team Creation
POST /teams/create {teamName, licenseKey}
validateLicense(licenseKey)
Check if exists in config
Check if already assigned
alt
[License Valid & Available]
Valid {max_users, expiry_date}
BEGIN TRANSACTION
INSERT INTO teams, licenses
UPDATE users SET team_id=?
COMMIT
Success
200 OK {team}
[Invalid/Expired/InUse]
Error
400/403 Error
Access Control (Every Protected Request)
Any request with JWT
Extract user_id, team_id from token
alt
[No Team]
403 Forbidden
[Has Team]
SELECT license WHERE team_id=?
License data
alt
[License Valid & Active & Not Expired]
Execute handler (with team_id filter)
Query with WHERE team_id=?
Team-filtered data
200 OK {data}
[License Invalid/Expired]
403 Forbidden
License Monitoring
loop
[Daily]
SELECT licenses
All licenses
Check expiry dates
alt
[Expired]
UPDATE status='expired'
INSERT notifications
Key Points:
- Licenses in environment config
- Not in DB until assigned
- Validated on EVERY request
- Team isolation enforced
- Daily monitoring job

Figure 8: License Validation and Access Control Flow

Validation Checks:

- Valid team licenses
- User count within limits
- Active license status
- Expiration monitoring

7 Deployment View

The system deployment architecture is illustrated in Figure 9, showing the containerized setup with Docker.

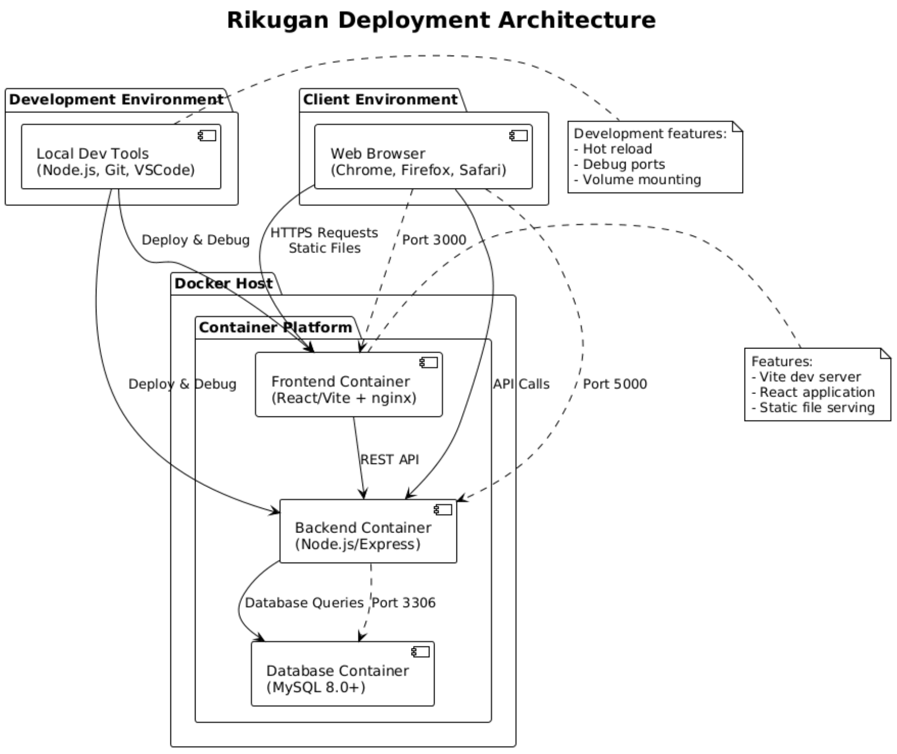


Figure 9: Deployment Architecture - Docker Containerization

Components:

- **Frontend Container**: React 18+/Vite (production build)
- **Backend Container**: Node.js 18+/Express.js (API server)
- **Database Container**: MySQL 8.0+ (persistent storage)
- **Reverse Proxy**: nginx (load balancing, SSL)

Requirements:

- Docker 20.10+
- Docker Compose 2.0+

- 4GB RAM
- 20GB disk space

8 Quality Scenarios

8.1 Quality Tree

Rikugan Quality Goals

Usability

- *Learning Curve (High Priority)*
  New users can complete basic tasks within 10 minutes
- *Interface Consistency (Medium Priority)*
  All pages follow consistent navigation patterns

Performance

- *Response Time (High Priority)*
  API calls respond within 500ms for 95% of requests
- *Concurrent Users (Medium Priority)*
  Support 50 simultaneous users without degradation

Security

- *Authentication (High Priority)*
  Secure login with role-based access control
- *Data Protection (High Priority)*
  All user data encrypted and validated

Maintainability

- *Code Quality (Medium Priority)*
  70% test coverage with clean architecture
- *Documentation (Medium Priority)*
  Complete API and component documentation

Document: Architecture design per arc42 template
Version: 1.0
Updated: December 2025