Assignment2: Online meeting booking system

G33 Members

Lekun.Liu	2255714
Yiyang.Han	2252155
Junhao.Huang	2256792
Yichuan.Zhang	2251668
Tiantian.Ye	2251663
Ruiyang.Wu	2257387
Daiyan.Wu	2257475
Weizhe.Sun	1927150
Xuning.Hu	2251992

Overview

Part 1
Introduction & Background

Part 2
Architectural Design

Part 3
Software Design

Part 4
Software Testing

Part 5
Project Conclusion

Introduction & Background

The development of a web-based online meeting booking system for a university's meeting room management

Aims of our project

A high-quality, efficient, user-friendly, and secure system that enables convenient management and meeting room reservations

Meet all kinds of customer’s specification

Introduction & Background

User Characteristics

Staff and student users

Regular Access

- Efficient meeting room reservation process
- Real-time access to room availability
- Clear room descriptions
- Photos of meeting rooms
- Email confirmation and notification for registration

Administrator

Superior Access

- Manage feedback and maintenance Requests
- Manage meeting rooms, equipment, booking approvals, and user permissions

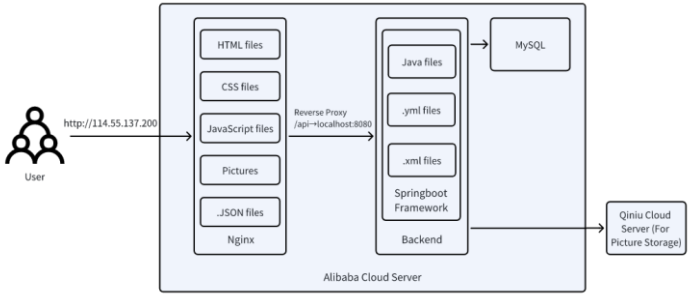
Project Risks

Integration Complexity: Compatibility risks with Spring Boot, MySQL, Qiniu, etc.
Usability Challenges: Unintuitive filters, calendar, or email flow may reduce adoption
Data Breach & Unauthorized Access: Weak session or password handling can expose user data
Injection Vulnerabilities: No rate limit on email code requests may lead to spamming
Ethical Considerations: Mishandling student data may violate university or privacy laws

Architectural Design

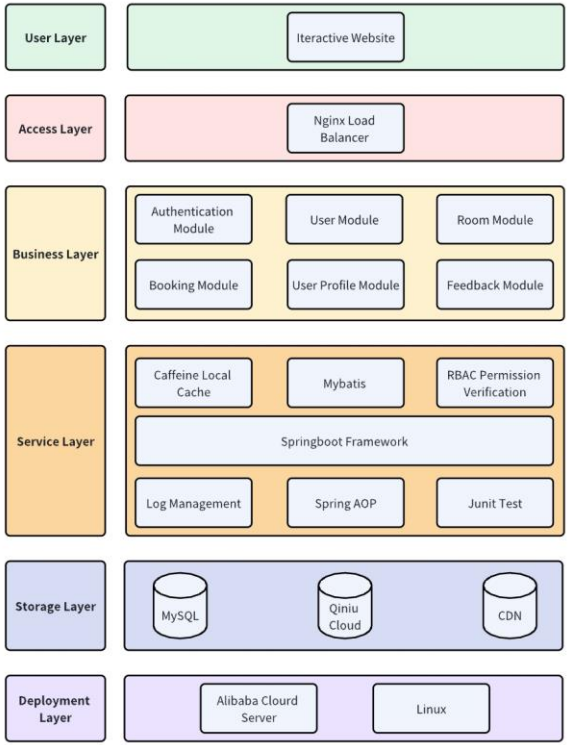
Five key requirement

- **Reliability**
- **Performance**
- **Usability**
- **Security**
- **Continuous Improvement**



System Components

Frontend: Vue 2, HTML, CSS, JavaScript, Node.js 18.16.0
Backend: Spring Boot 3.0.2, MyBatis-Plus 3.5.3, Java 17.0.8
Database: MySQL 8.0+
OSS: Qiniu Cloud
Deployment: Nginx, Ali Cloud
Communication: Axios HTTP Client



Architectural Design

Frontend Technology & Justification

- HTML, CSS, JavaScript
- Component-Based Architecture
  - Manages stateful user interactions
  - Enhances frontend scalability and maintainability
- Axios HTTP Client
  - Promise-based HTTP client
  - Handles backend requests more efficiently
  - Request and response interception
  - Streamlined error handling and better code organization

Architectural Design

Backend Technologies & Dependencies

Spring Boot
- Project foundation
- Centralized config.

MyBatis Plus
- Simplified ORM for SQL mapping and CRUD
- Wrapper-based query construction, improving readability.

Spring AOP
- Allows the specified method action to be performed before or after method execution. It is used for log management and permission verification.

Caffeine
- Lightweight local cache to enhance system performance
- High concurrency and low latency for frequent data

Qiniu OSS
- Image storage integration via secure object storage
- Supports file upload and CDN delivery with minimal setup

Knife4j
- Knife4j provides API documentation UI for RESTful services
- Allows interactive API exploration with parameter editing and live response

Architectural Design

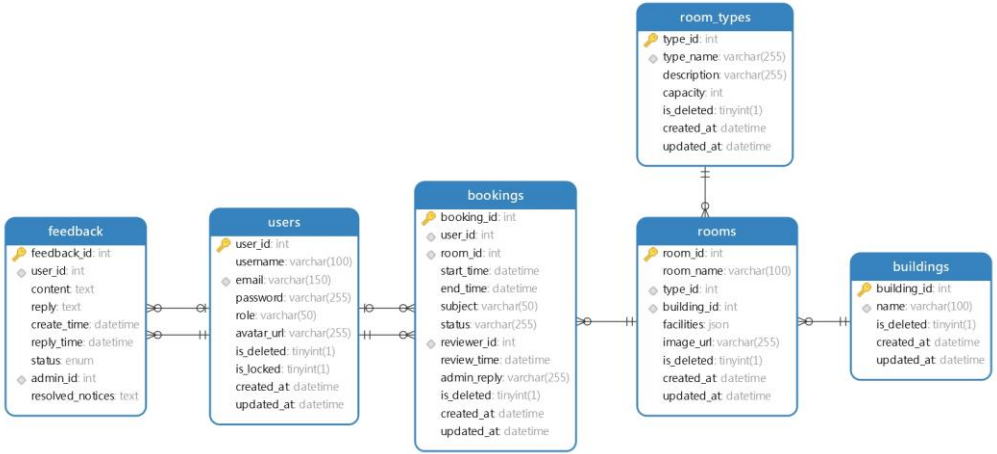
High-level Database Design

Table Relationships

users to bookings: 1:M
users to feedback: 1:M
rooms to bookings: 1:M
room_types to rooms: 1:M
buildings to rooms: 1:M

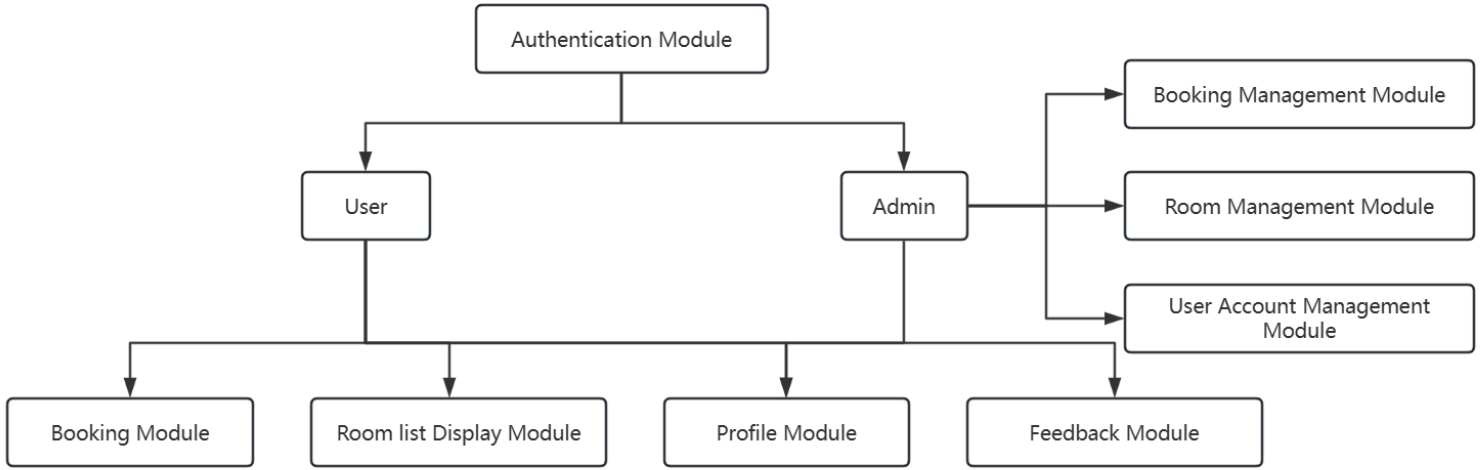
Table Foreign Keys

users.user_id => bookings.user_id
bookings.room_id => rooms.room_id
users.user_id => bookings.reviewer_id
users.user_id => feedback.user_id
users.user_id => feedback.admin_id
room_types.type_id => rooms.type_id
buildings.building_id => rooms.building_id



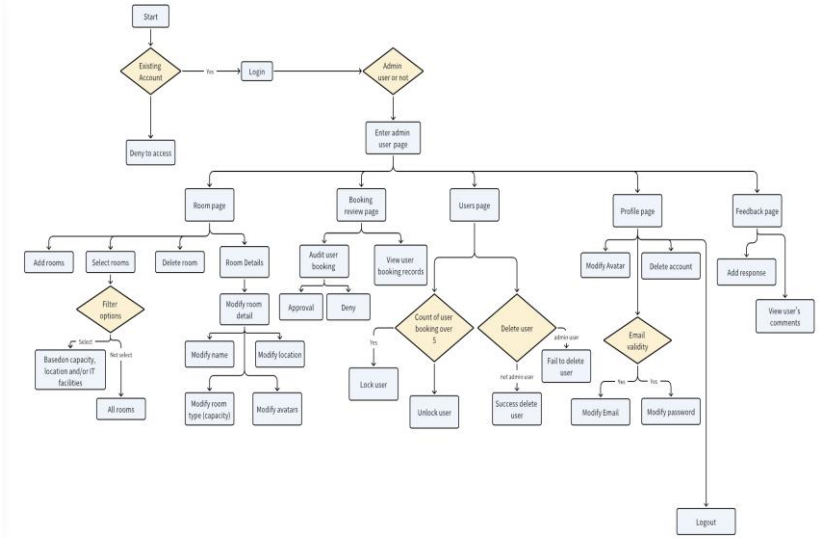
Software Design

Software Modules

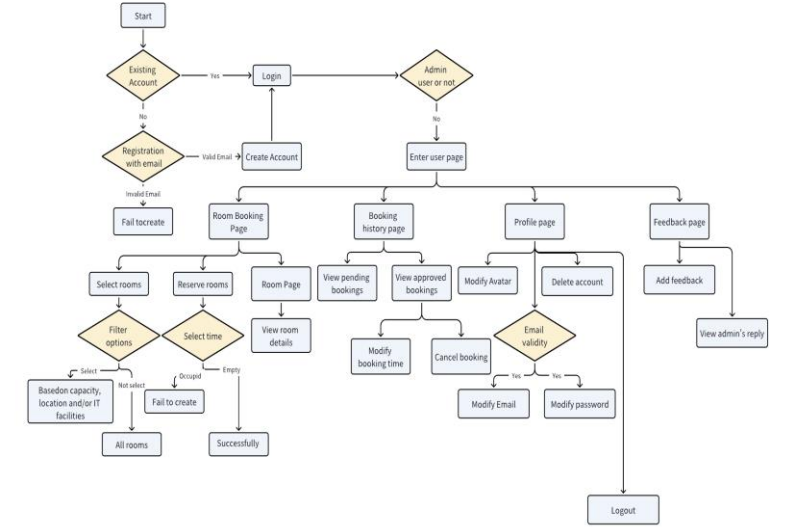


Software Design

High-level process flow



Admin Activity Diagram



User Activity Diagram

Software Design

Software Support Service

Database Services

MySQL provides relational data storage
MyBatis-Plus simplifies ORM with built-in CRUD
QueryWrapper supports dynamic query building in Java

Caching & Utility Services

Caffeine Cache stores short-lived data
Hutool Toolkit enables string, file, and object utilities

Security & Email Services

Passwords hashed via MD5 + Salt, rate-limited Captcha
Email verification sent via NetEase 163 SMTP

Cloud Storage & CDN

Qiniu Cloud stores user avatars and room images
Supports CDN acceleration & lifecycle management

Development & Deployment Tooling

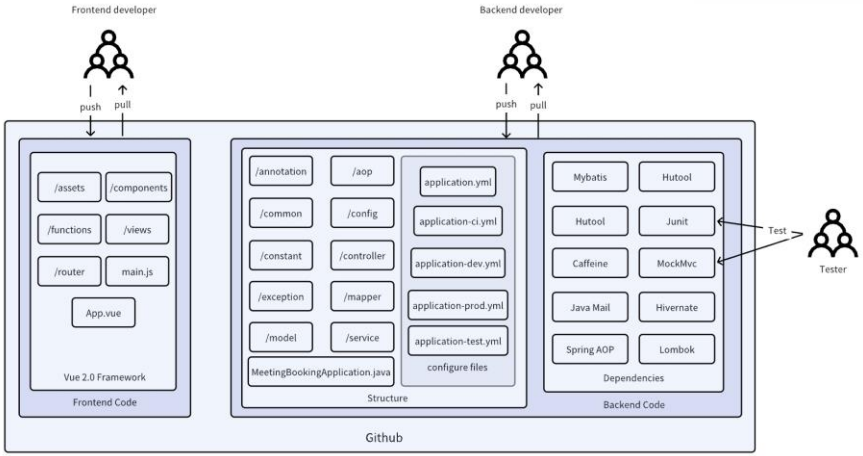
IntelliJ IDEA (backend) & VS Code (frontend)
Version control with Git + GitHub, CI pipelines for pre-merge validation

Documentation & Support

RESTful APIs documented via Swagger / Knife4j
Community & troubleshooting support via Stack Overflow and official docs

Software Design

Coding Structure



5 Core Principles:

- **Separation of concerns**
- **High cohesion & Low coupling**
- **Maintainability**
- **Testability**
- **RESTful API design**

Layered Architecture:

- **Controller Layer**: Handles HTTP requests
- **Service Layer**: Contains business logic and Mapper Layer SQL abstraction via MyBatis interfaces
- **Model Layer**: Entity classes mapped to DB; DTOs/VOs used for I/O transformation

Software Design

Shared Functional Packages

Shared functional packages help organize reusable logic, simplify configuration, and ensure consistency across the system — making the codebase cleaner, more modular, and easier to maintain.

- common/ – Shared tools for response, email, and rate limiting
- config/ – Centralized project configuration
- constant/ – Global enums for roles and status codes
- exception/ – Unified error handling structure
- annotation + aop/ – Declarative access control with AOP(Aspect-Oriented Programming)

Coding Conventions

- Unified naming
- Strict API contracts
- Typed errors
- Javadoc, logging, and pre-commit linting
- Agile workflow with short sprints, CI, code reviews, and daily stand-ups

Software Testing – Unit Test

Unit Testing Process

Focus of Unit Testing: Verifying the **functionality, correctness, and accuracy** of individual code units.

1. **Identify Target Unit:** Pinpoint the specific method or class to test.
2. **Understand Expected Behavior:** Define expected results for various inputs and conditions.
3. **Isolate the Unit:** Use mocks (Mockito) to simulate external dependencies.
4. **Write Test Case (AAA):** Arrange inputs, act by calling code, assert expected outcomes.
5. **Run the Test:** Execute the test using the framework and check results.

Email format verification (Common Layer): By testing multiple valid and invalid email addresses, ensure that the **EmailValidator** utility class can accurately verify the email format in accordance with the RFC 5322 standard.

```java
@Test
void testValidEmails() {
    assertTrue(EmailValidator.isValidEmail("user@example.com"));
    assertTrue(EmailValidator.isValidEmail("user.name+tag+sorting@example.com"));
    assertTrue(EmailValidator.isValidEmail("x@example.co.uk"));
    assertTrue(EmailValidator.isValidEmail("customer/department=shipping@example.com"));
    assertTrue(EmailValidator.isValidEmail("user_name@example.travel"));
    assertTrue (EmailValidator.isValidEmail("user@sub-domain.example.com"));
}

MushihimePepsi
@Test
void testInvalidEmails() {
    assertFalse(EmailValidator.isValidEmail(null));            // null
    assertFalse(EmailValidator.isValidEmail(""));              // empty
    assertFalse(EmailValidator.isValidEmail("plainaddress"));  // missing @
    assertFalse(EmailValidator.isValidEmail("@no-local-part.com"));
    assertFalse(EmailValidator.isValidEmail("user@.invalid.com"));
    assertFalse(EmailValidator.isValidEmail("user@invalid..com"));
    assertFalse(EmailValidator.isValidEmail("user@@example.com"));
    assertFalse(EmailValidator.isValidEmail("user@ example.com")); // space
}
```

Software Testing – Unit Test

Unit Testing Process

Focus of Unit Testing: Verifying the **functionality, correctness, and accuracy** of individual code units.

1. **Identify Target Unit:** Pinpoint the specific method or class to test.
2. **Understand Expected Behavior:** Define expected results for various inputs and conditions.
3. **Isolate the Unit:** Use mocks (Mockito) to simulate external dependencies.
4. **Write Test Case (AAA):** Arrange inputs, act by calling code, assert expected outcomes.
5. **Run the Test:** Execute the test using the framework and check results.

Obtain the user list (Service layer): Use Mockito to simulate dependencies such as Mapper, verify whether the business logic of **AdminUserService** for obtaining the user list is correct, and correctly assemble and return the data.

```java
@Test
void testGetUserLists_success() {
    // Prepare simulated returned user data
    UserVO userVO = new UserVO();
    userVO.setUser_id(123);
    userVO.setUsername("testUser");
    userVO.setAvatar_url("avatar/test.png");  // simulated database returns key, not complete URL

    // mock method actions
    when(usersMapper.getUserWithBookingCount()).thenReturn(Collections.singletonList(userVO));
    when(qiniuConfig.getDomain()).thenReturn( t: "cdn.qiniu.com/");
    // call adminUserService method
    List<UserVO> result = adminUserService.getUserLists();

    // Verification results
    assertEquals( expected: 1, result.size());
    assertEquals( expected: 123, result.get(0).getUser_id());
    assertEquals( expected: "testUser", result.get(0).getUsername());
    assertEquals( expected: "http://cdn.qiniu.com/avatar/test.png", result.get(0).getAvatar_url());
}
```

Software Testing – Unit Test

Unit Testing Process

Focus of Unit Testing: Verifying the **functionality, correctness, and accuracy** of individual code units.

1. **Identify Target Unit:** Pinpoint the specific method or class to test.
2. **Understand Expected Behavior:** Define expected results for various inputs and conditions.
3. **Isolate the Unit:** Use mocks (Mockito) to simulate external dependencies.
4. **Write Test Case (AAA):** Arrange inputs, act by calling code, assert expected outcomes.
5. **Run the Test:** Execute the test using the framework and check results.

Obtain the reservation by ID (Controller layer): Use **MockMvc** to simulate HTTP GET requests, and combine **Mockito** to simulate the Service layer to verify whether the Controller interface for administrators to query reservation information based on reservation numbers can correctly handle and return the expected results

```
@Test
void testGetBookingWithNameByBookingID() throws Exception {
    BookingListWithNameVO vo = new BookingListWithNameVO();
    vo.setBooking_id(123);
    vo.setRoom_name("Conf Room");
    Mockito.when(bookingService.getBookingWithNameByBookingID(123)).thenReturn(vo);

    mockMvc.perform(get( urlTemplate: "/admin/bookings/{bookingId}",  ...uriVariables: 123)
                    .session(adminSession()))
            .andExpect(status().isOk())
            .andExpect(jsonPath( expression: "$.code").value( expectedValue: 200))
            .andExpect(jsonPath( expression: "$.data.booking_id").value( expectedValue: 123))
            .andExpect(jsonPath( expression: "$.data.room_name").value( expectedValue: "Conf Room"));
}
```

Software Testing – Integration Test

Integration Testing Process

Focus of Integration Testing: Iterations between different modules. Identify interaction level issues that may be missed when unit testing is performed alone.

1. **Define Scope:** Define modules and their interactions for testing together.
2. **Prepare Environment:** Set up the test environment, possibly including real dependencies.
3. **Design Scenarios:** Create test cases covering key inter-module interactions.
4. **Execute Tests:** Run test cases, triggering actual component interactions.
5. **Verify Results:** Check outcomes, ensuring correct interactions and data states.
6. **Report & Fix:** Log failures; locate and fix discovered integration issues.

Email send & lock/unlock user function: Test AdminUserService lock and unlock function and checking Service/Mapper interactions and database update. Besides, mocking email sending process.

```java
@BeforeEach
void setUp() {
    // insert test user
    Users user = new Users();
    user.setUsername("testUser");
    user.setUser_id(123);
    user.setPassword("12345678");
    user.setEmail("testUser@example.com");
    user.setRole("user");
    user.setIs_locked(0);
    usersMapper.insert(user);
    testUserId = user.getUser_id();
}
```

```java
@Test
void testUnlockUser_withLockedUser_shouldUnlockSuccessfully() {
    Users user = usersMapper.selectById(testUserId);
    user.setIs_locked(1);
    usersMapper.updateById(user);

    adminUserService.unlockUser(testUserId);

    Users updatedUser = usersMapper.selectById(testUserId);
    assertEquals( expected: 0, updatedUser.getIs_locked());
}
```

Software Testing – Acceptance Test

Acceptance Testing Process

Focus of Integration Testing: Validating the system meets project requirements and user needs, ensuring it's ready for deployment.

1. **Define Acceptance Criteria:** Define clear pass/fail criteria based on user requirements.
2. **Plan Test Scenarios:** Develop real-world user scenarios covering criteria.
3. **Prepare Environment/Data:** Set up production-like environment and realistic test data.
4. **Execute Tests:** End-users run scenarios simulating real-world usage.
5. **Evaluate Results:** Compare actual results against criteria; log defects.

Acceptance Testing Example:
User booking conference room

Objective:
Users can select an available room and book it for a certain period of time. After reservation, the subscriber can view reservation records.

Test steps:
1. Login to the system with a test account
2. Navigate to the "Rooms" page
3. Filter avilable room list
4. Choose one meeting room and set a specific meeting time (tomorrow 1900-2100).
5. Fill in the purpose of the booking ‘Team Meeting’.
6. Click on the ‘Book’ button.

Result:
Booking information should appear in the ‘My Bookings’ list.

Conclusion

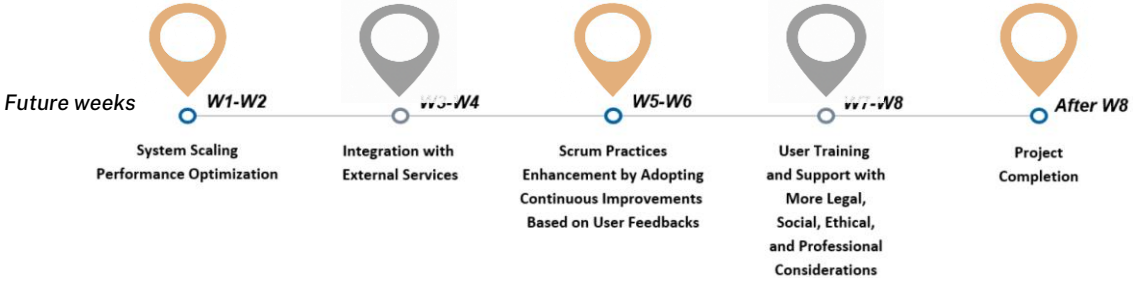
Achievement

- Development of a Web-based Meeting Room Booking System for Universities
- Practical Adoption of the Scrum Framework throughout Iterative Sprints
- Understanding of Legal, Ethical, and Security Considerations in System Design
- Complete System Specification, Design, Implementation, and Evaluation

Problems in our project

- Limited Frontend Styling
- Unoptimized Query and Loading Performance
- Lack of coping strategies for high-concurrency scenarios

Future work



References

- Noor, G., & Thomas, G. (2021). Aspect-Oriented Programming: Transforming Code Modularity in Spring Ecosystems.
- Tran, N. (2020). Applying vue. js framework in developing web applications.
- Zhang, F.; Sun, G.; Zheng, B.; Dong, L. Design and Implementation of Energy Management System Based on Spring Boot Framework. *Information* **2021**, *12*, 457. https://doi.org/10.3390/info12110457.
- Catrina, A. (2023). A Comparative Analysis of Spring MVC and Spring WebFlux in Modern Web Development.
- Gajewski, M., & Zabierowski, W. (2019, May). Analysis and comparison of the Spring framework and play framework performance, used to create web applications in Java. In *2019 IEEE XVth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)* (pp. 170-173). IEEE.
- Zhao, J. (2022, June). Application and practice of MVC architecture pattern in on-the-job internship management system. In *2022 International conference on networks, communications and information technology (CNCIT)* (pp. 25-30). IEEE.
- Grebić, B., & Stojanović, A. (2021). Application of the Scrum framework on projects in IT sector. *Eur. Proj. Manag. J*, *11*(2), 37-46.