Minimum requirements for implementing the server part to obtain a passing grade

For successful defense of your thesis, the server part of your application must meet the following minimum requirements:

1. Use of a modern framework

The application must be developed using a high-level framework designed for server development. For example, ASP.NET Core, Spring, Django, Node.js, and others.

1.1 Exception: If your work involves the development of a console application or other type of software that logically does not require the use of frameworks, you are required to independently implement the functionality provided by such tools. This includes:

- Logging;
- Error handling;
- Serialization and data binding;
- Authorization and authentication.

2. Working with data state

If your application stores the state of objects or data, you need to use a database. The choice between a relational (e.g., PostgreSQL, MySQL) and a non-relational (e.g., MongoDB) database depends on your project requirements.

3. Using ORM

Interaction with the database should be implemented using ORM (Object-Relational Mapping). For example, Entity Framework, Hibernate, Sequelize, and others.

4. Application architecture

The application should follow the principles of multi-layer architecture (e.g., separation into layers: presentation, business logic, data access). This will ensure modularity and ease of code maintenance.

5. Design principles (SOLID)

The application code must be written in accordance with SOLID principles:

- Single Responsibility Principle;
- Open/Closed Principle;
- Liskov Substitution Principle;
- Interface Segregation Principle;
- Dependency Inversion Principle.

6. Description of the interaction interface

The interface for interacting with the server side (API) must be clearly described. This can be done using documentation (e.g., OpenAPI/Swagger) or a text description if the project does not involve standard tools.

7. Global error handling

A centralized error handling mechanism should be implemented in the application. This will increase its reliability and simplify debugging.

8. Logging

The application should log key events, errors, and other significant operations. Logs should be structured and accessible for analysis.

9. Deployment in a working environment

The finished application should be deployed in a production environment. This can be:

- A cloud solution (e.g., AWS, Azure, Google Cloud, Heroku);

- Private server resources (e.g., Docker containers on a local server).

Important: you must demonstrate that the application is working and accessible to external users.

10. Test coverage

The application's business logic must be covered by unit tests. The minimum requirement for code coverage is **70%**. Use code coverage analysis tools to confirm this metric.

Specific restrictions, technologies, and approaches

When developing your thesis, you must adhere to the following restrictions and recommendations to ensure the quality, security, and availability of your application:

General restrictions

1. **Prohibition on the use of paid libraries**
   Do not use paid libraries or tools that may limit the ability of other users to test and use your application. Your solution must be accessible and reproducible.
2. **Avoid vulnerable packages**
   Check the libraries and dependencies you use for known critical vulnerabilities. Use security analysis tools such as **npm audit**, **pip-audit**, or similar tools for your technology stack.
3. **Working in a version control system**
   All development should be done in an environment that supports version control, such as Git. The repository should contain a history of changes so that your progress and approach to solving problems can be tracked.
4. **Use JSON**
   JSON should be the primary data format used for transferring and processing

information. This will ensure compatibility and ease of use with most modern technologies.

5. **Storing secrets and confidential data**
   All secret data (e.g., API keys, passwords, tokens) should not be stored in the repository. They should be moved to secure storage or environment variables. Make sure your deployment process takes this into account.

6. **Automatic deployment from the repository**
   The application should be able to deploy and run from the latest version in the repository without the need for manual changes or additional "crutches." This demonstrates the maturity of your approach to development.

What not to do

1. **Inventing your own encryption protocols**
   Do not create your own encryption protocols or custom implementations of standard solutions such as JWT. Use proven, standardized technologies.

2. **Using outdated or educational libraries**
   Do not use libraries that are outdated or intended solely for educational purposes. For example:

   - **xmlrpc**;
   - Built-in **http.server** in Python without modifications;
   - Other tools not intended for production use.

3. **Creating placeholders instead of APIs**
   The application should not return static JSON or other dummy data instead of fully interacting with the database or other components. This is considered an unacceptable simplification.

Recommendations for modularity

- The application should be modular. The code should be structured according to the principles of modularity to avoid a monolithic approach (e.g., all code in one file). This simplifies application support, testing, and scaling.

Maximum requirements for the implementation of the server part of the thesis project

To receive the highest grade (10 points), the server part of your thesis project must demonstrate advanced practices in application development, deployment, and support. Meeting the following requirements will allow you to demonstrate in-depth knowledge and skills:

1. Implementation of CI/CD practices

- Automation of application build, testing, packaging, and deployment using CI/CD (Continuous Integration/Continuous Deployment).
- Sample process:
    - Create a Docker image with your application;
    - Store the image in a container registry (e.g., Docker Hub, AWS ECR, GitHub Container Registry);
    - Automatically deploy the image to a production environment (e.g., in the cloud: AWS, Azure, GCP, or on a local server).
- During the deployment phase, secrets (keys, passwords, tokens) and settings must be implemented through a system of environment variables or secret stores. All stages must be performed through a build and deployment system pipeline (e.g., GitHub Actions, GitLab CI/CD, Jenkins).

2. Microservice architecture

- The application should consist of several microservices, each of which performs its own clearly defined task.
- Microservices must be independent and deployable in a production environment separately from each other.

3. Asynchronous interaction

- Parts of the application (microservices) should interact with each other asynchronously (e.g., via message queues — RabbitMQ, Kafka). This will ensure the scalability and stability of the system.

4. Caching

- Implement caching to optimize performance. For example, use Redis or Memcached to store temporary data, frequently used queries, or calculation results.

5. System monitoring

- Implementation of a system for monitoring the status of the application and infrastructure.
- Use tools such as Grafana and Prometheus to:
    - Collecting metrics (CPU load, number of requests, memory usage, etc.);
    - Visualizing data through dashboards.

6. Logging system

- Application logs must be structured and centralized.
- Ability to filter, search, and analyze logs using specialized tools such as ELK Stack (Elasticsearch, Logstash, Kibana) or Loki.
- Logs should be useful for debugging and analyzing system performance.

7. Tracking requests between microservices

- Implementation of a mechanism for tracking the chain of requests between microservices using **correlation IDs**. This will allow you to track the entire path of a request, even if it passes through several services.

8. Alerting system

- Configure a notification system for certain conditions (e.g., metric thresholds exceeded, errors occurred).
- Use tools such as Alertmanager to send notifications to messengers (Telegram, Slack) or by email.

9. Quality Gates in CI/CD

- Set up quality gates in the CI/CD pipeline. This includes:
    - Automatic code quality checks (e.g., using SonarQube or similar tools);
    - Running tests (unit tests, integration tests);
    - Checking code coverage with tests (e.g., minimum coverage of 80%).

- Deployment should only occur after all checks have been successfully passed.

Major errors and shortcomings affecting the evaluation of the server part of the thesis

Below is a list of errors and oversights that are considered unacceptable and will result in points being deducted from the overall assessment of the thesis. These issues demonstrate a failure to meet modern development standards and reduce the quality of the project.

Security errors

1. **Use of insecure hashing algorithms**
   Example: using MD5 or SHA-1 instead of modern algorithms such as bcrypt, Argon2, or SHA-256.
   Minus 1 point.
2. **Direct storage of confidential data**
   Example: storing passwords or tokens in plain text instead of encrypting or hashing them.
   Minus 1 point.
3. **Lack of input validation**
   Example: lack of user input validation, which can lead to SQL injections, XSS, and other attacks.
   Minus 1 point.

Errors in data processing and error handling

4. **Lack of error handling**
   Example: the application crashes when exceptions occur, instead of handling them correctly and returning clear messages to the user or the system.
   Minus 1 point.
5. **Incorrect use of HTTP codes or their absence**
   Example: The application always returns **200 OK**, even when errors occur (for example, when a resource is missing).
   Minus 1 point.
6. **Inconsistency with the database**
   Example: The API does not work with the declared entities, the fields in the database

do not match the description in the documentation.
Minus 1 point.

Errors in design and code

7. **Lack of API documentation**
   Example: lack of description of API methods, their parameters, and return data (e.g., via Swagger or text documentation).
   Minus 1 point.

8. **Poor code quality**
   Example: unreadable, poorly structured code, lack of comments, use of non-obvious constructs.
   Minus 1 point.

9. **Inefficient database queries**
   Example: use of "greedy" queries, lack of indexes, excessive number of queries instead of optimized operations.
   Minus 1 point.

10. **Hardcoded configurations**
    Example: parameters such as database URLs, API keys, file paths are specified directly in the code instead of using environment variables.
    Minus 1 point.

11. **Lack of modularity**
    Example: All code is written in a single file, with no separation into layers (e.g., controllers, business logic, data access).
    Minus 1 point.

12. **Use of "magic strings"**
    Example: strings or numbers used in the code without explanation of their meaning (e.g., **"admin"** instead of the constant **ROLE_ADMIN**).
    Minus 1 point.

13. **Code duplication**
    Example: Repeated blocks of code instead of moving them to functions or methods.
    Minus 1 point.

Testing errors

14. **Lack of test coverage**
    Example: lack of unit tests or insufficient number of them (less than 70% coverage).
    Minus 1 point.

API implementation errors

15. **Non-compliance with the REST approach**
    Example: incorrect use of HTTP methods (e.g., using **GET** to create a resource), lack of logical URL structure.
    Minus 1 point.