

SDD (Project detailed documentation)

1- Introduction

Taskaty is a task management application designed to help users organize their daily tasks efficiently. Taskaty empowers users to create, track, delete, and manage their tasks seamlessly to stay organized and productive.

1.1 Purpose :

This document aims to comprehensively outline the design aspects of Taskaty, focusing on TaskMicroservice and UserMicroservice applications. It provides insights into the system's architecture, components, and interactions.

1.2 Scope :

The document covers the design and implementation details of TaskMicroservice and UserMicroservice for the Taskaty app. It includes architecture, validation mechanisms, Aspect-Oriented Programming (ASOP) implementations, microservices design principles, API Documentation, and deployment on cloud infrastructure.

1.3 Overview :

TaskMicroservice and UserMicroservice are built using Spring Boot, emphasizing scalability, maintainability, and reliability.

2- Architectural Overview

2.1 High-Level Architecture :

Taskaty follows a three-tier architecture: :

- 1- Presentation Layer: Frontend components (HTML, CSS, JavaScript).
- 2- Application Layer: Backend services (Java, Spring Boot, Spring Data JPA, Spring Cloud).
- 3- Data Layer: Database (MySQL) for user and task data.

2.2 Design Principles

The architecture follows the MVC design pattern for separating concerns and ensuring maintainability and scalability.

3- System Description(Requirements)

3.1 Task Microservice :

Manages tasks for users with functionalities including task creation, retrieval, update, and deletion:

- Creating a task for a user.
- Retrieving tasks for a user.
- Updating task status.
- Updating task title.
- Deleting a task.

3.2 User Microservice :

Manages user authentication and profile management, including sign-up, login, and profile updates:

- User sign-up.
- User login.
- Updating user email.
- Updating user password.

4- Architecture

4.1 Validation :

Both microservices utilize validation mechanisms to ensure data integrity and enforce business rules. Validation is performed using annotations such as @NotBlank, @Size, @Pattern and @Email from the Jakarta Bean Validation API.

- The Validation used :
 - a. Email addresses must not be null, blank, and must be unique within the system.
 - b. Password must not be null or blank and must meet complexity requirements:
 - i. Minimum length of 6 characters and maximum length of 20 characters.
 - ii. Must contain at least one uppercase letter, one lowercase letter, and one digit.
 - c. Task title length (between 1 and 100 characters).

4.2 ASOP (Aspect-Oriented Programming) :

Aspect-Oriented Programming is implemented in both microservices to address cross-cutting concerns such as logging and measuring method execution time.

Two aspects are defined:

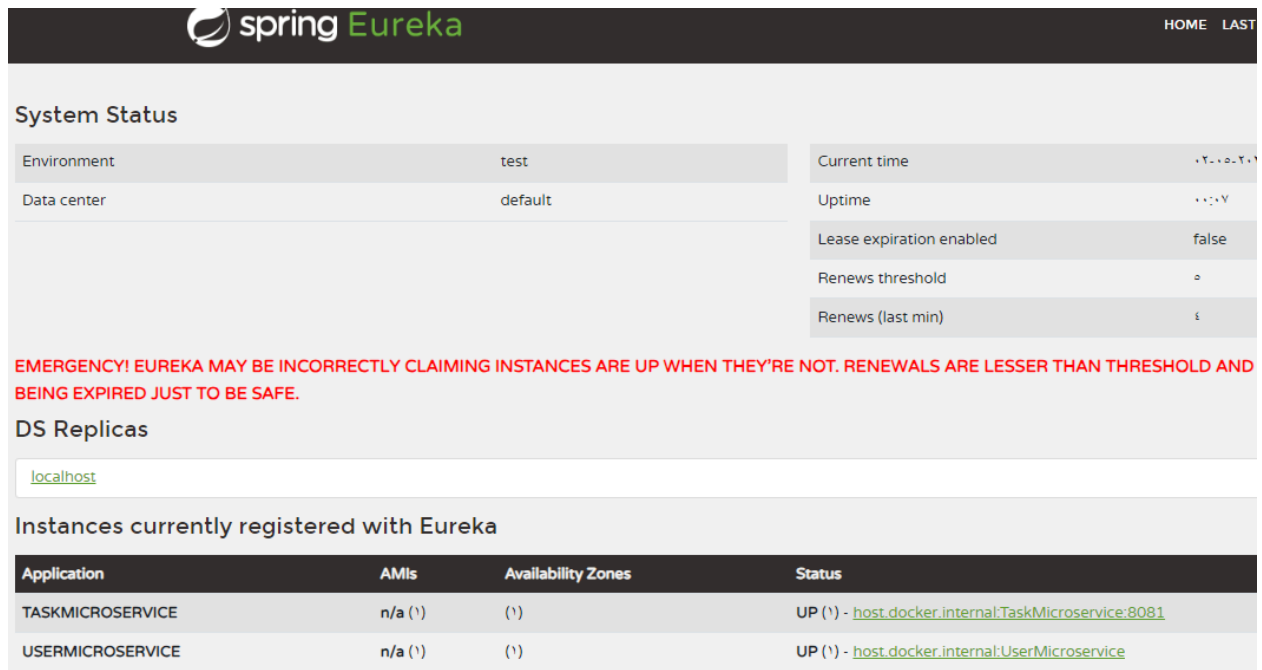
- LoggingAspect: Logs method execution before and after invocation.
- MethodExecutionTimeAspect: Measures and logs method execution time.

4.3 Microservices:

The microservices architecture follows a distributed design pattern where each service is independent, loosely coupled, and focused on a specific domain. They communicate via RESTful APIs, allowing for scalability and flexibility. Service discovery and registration are achieved using Spring Cloud Netflix Eureka.

4.4 Cloud Deployment :

Both microservices are designed to be deployed on a cloud infrastructure. They leverage Spring Cloud dependencies for seamless integration with cloud services. Configuration properties are externalized to allow for easy configuration in cloud environments.



The screenshot displays the Spring Eureka web interface. At the top, the "spring Eureka" logo is visible on the left, and "HOME" and "LAST" links are on the right. The main content area is titled "System Status" and contains two tables. The left table shows "Environment" as "test" and "Data center" as "default". The right table shows "Current time" as "2020-10-20 10:10:10", "Uptime" as "10:10:10", "Lease expiration enabled" as "false", "Renews threshold" as "0", and "Renews (last min)" as "1". Below these tables, a red warning message states: "EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND BEING EXPIRED JUST TO BE SAFE." Under the "DS Replicas" section, a link for "localhost" is shown. The "Instances currently registered with Eureka" section contains a table with the following data:

Application	AMIs	Availability Zones	Status
TASKMICROSERVICE	n/a (1)	(1)	UP (1) - host.docker.internal:TaskMicroservice:8081
USERMICROSERVICE	n/a (1)	(1)	UP (1) - host.docker.internal:UserMicroservice

5- API documentation

TaskService :

1- **POST** (CreateTask) :

[http://host.docker.internal:8081/api/users/tasks/\\${userId}](http://host.docker.internal:8081/api/users/tasks/${userId})

The screenshot displays a REST client interface with a POST request configured. The request is sent to the URL `http://host.docker.internal:8081/api/users/tasks/1`. The request body is a JSON object: `{ "title": "API Documentation" }`. The response is a 201 status code, indicating successful creation, with a response time of 849 ms and a body size of 318 B. The response body is shown in a pretty-printed JSON format: `{ "id": 1, "title": "API Documentation", "done": false, "userId": 1 }`.

```
POST CreateTask
TaskService / CreateTask
POST http://host.docker.internal:8081/api/users/tasks/1
Body
{
  "title": "API Documentation"
}
201 Created 849 ms 318 B
Pretty
{
  "id": 1,
  "title": "API Documentation",
  "done": false,
  "userId": 1
}
```

2- GET (GetTasks) :

[http://host.docker.internal:8081/api/users/tasks/\\${userId}](http://host.docker.internal:8081/api/users/tasks/${userId})

The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** `http://host.docker.internal:8081/api/users/tasks/1`
- Status:** 200 OK
- Response Time:** 403 ms
- Response Size:** 315 B

The response body is displayed in JSON format:

```
1 [
2   {
3     "id": 1,
4     "title": "API Documentation",
5     "done": false,
6     "userId": 1
7   }
8 ]
```

3- PUT(UpdateTitle) :

[http://host.docker.internal:8081/api/users/tasks/\\${userId}/\\${taskId}](http://host.docker.internal:8081/api/users/tasks/${userId}/${taskId})

The screenshot displays a REST client interface with the following details:

- Request Method:** PUT
- URL:** `http://host.docker.internal:8081/api/users/tasks/1/1`
- Body:** A JSON object with the key `"title"` and value `"Finish"`.

```
1 {  
2   "title": "Finish"  
3 }
```
- Response:** A 200 OK status with a response time of 78 ms and a body size of 302 B. The response is displayed in a pretty-printed JSON format:

```
1 {  
2   "id": 1,  
3   "title": "Finish",  
4   "done": false,  
5   "userId": 1  
6 }
```

4- PUT(UpdateStatus) :

[http://host.docker.internal:8081/api/users/tasks/\\${userId}/toggle/\\${taskId}](http://host.docker.internal:8081/api/users/tasks/${userId}/toggle/${taskId})

The screenshot shows a REST client interface with the following details:

- Request Method:** PUT
- URL:** `http://host.docker.internal:8081/api/users/tasks/1/toggle/1`
- Response Status:** 200 OK, 45 ms, 301 B
- Response Body (JSON):**

```
{
  "id": 1,
  "title": "Finish",
  "done": true,
  "userId": 1
}
```

The interface includes tabs for Params, Authorization, Headers (7), Body, Pre-request Script, Tests, Settings, and Cookies. The Body tab is selected, showing the response in JSON format.

GET GetTasks

No environment

TaskService / GetTasks

GET

http://host.docker.internal:8081/api/users/tasks/1

Send

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

Key	Value	Description	...	Bulk Edit
Key	Value	Description		

Body

Cookies

Headers (8)

Test Results

200 OK 26 ms 303 B Save as example

Pretty

Raw

Preview

Visualize

JSON

```
1 [
2   {
3     "id": 1,
4     "title": "Finish",
5     "done": true,
6     "userId": 1
7   }
8 ]
```

5- DELETE(DeleteTask) :

[http://host.docker.internal:8081/api/users/tasks/\\${userId}/\\${taskId}](http://host.docker.internal:8081/api/users/tasks/${userId}/${taskId})

The screenshot shows a REST client interface for a service named 'TaskService'. The selected endpoint is 'DeleteTask' with a DELETE method. The URL is 'http://host.docker.internal:8081/api/users/tasks/1/1'. The interface includes tabs for Params, Authorization, Headers (6), Body, Pre-request Script, Tests, and Settings. Below these is a 'Query Params' table with columns for Key, Value, and Description. The response status is '204 No Content' with a response time of '78 ms' and a size of '201 B'. The response body is empty, and the 'Text' tab is selected.

TaskService / **DeleteTask** Save

DELETE http://host.docker.internal:8081/api/users/tasks/1/1 Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

Key	Value	Description	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (6) Test Results 204 No Content 78 ms 201 B Save as example

Pretty Raw Preview Visualize Text 1

The screenshot shows the same REST client interface for 'TaskService', but with the 'GetTasks' endpoint selected. The method is GET and the URL is 'http://host.docker.internal:8081/api/users/tasks/1'. The response status is '200 OK' with a response time of '28 ms' and a size of '255 B'. The response body is an empty JSON array '[]', and the 'JSON' tab is selected.

TaskService / **GetTasks** Save

GET http://host.docker.internal:8081/api/users/tasks/1 Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

Key	Value	Description	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (8) Test Results 200 OK 28 ms 255 B Save as example

Pretty Raw Preview Visualize JSON 1 []

UserService :

1- **POST** (SignUp) :

<http://host.docker.internal:8080/api/users/signup>

The screenshot displays a REST client interface with a dark theme. At the top, the request is identified as a **POST** to `SignUp`. The URL bar shows `http://host.docker.internal:8080/api/users/signup`. The **Body** tab is selected, showing a JSON payload:

```
{
  "email": "a@gmail.com",
  "password": "12345aA"
}
```

. Below the request, the response is shown in the **Body** tab, displaying a **200 OK** status with a response time of **772 ms** and a size of **304 B**. The response body is a JSON object:

```
{
  "id": 1,
  "email": "a@gmail.com",
  "password": "12345aA"
}
```

. The interface includes tabs for Params, Authorization, Headers (8), Body, Pre-request Script, Tests, Settings, Cookies, and Beautify. The response status bar at the bottom indicates a successful **200 OK** response.

2- POST (Login) :

<http://host.docker.internal:8080/api/users/login>

The screenshot displays a REST client interface for a service named 'UserService' with the endpoint 'Login'. The request is a POST to 'http://host.docker.internal:8080/api/users/login'. The request body is a JSON object with 'email' and 'password' fields. The response is a 200 OK status with a JSON body containing 'id', 'email', and 'password' fields.

Request:

```
1 {
2   "email": "a@gmail.com",
3   "password": "12345aA"
4 }
```

Response:

```
1 {
2   "id": 1,
3   "email": "a@gmail.com",
4   "password": "12345aA"
5 }
```

3- PUT(UpdateEmail) :

[http://host.docker.internal:8080/api/users/\\${userId}/email?newEmail=\\${newEmail}](http://host.docker.internal:8080/api/users/${userId}/email?newEmail=${newEmail})

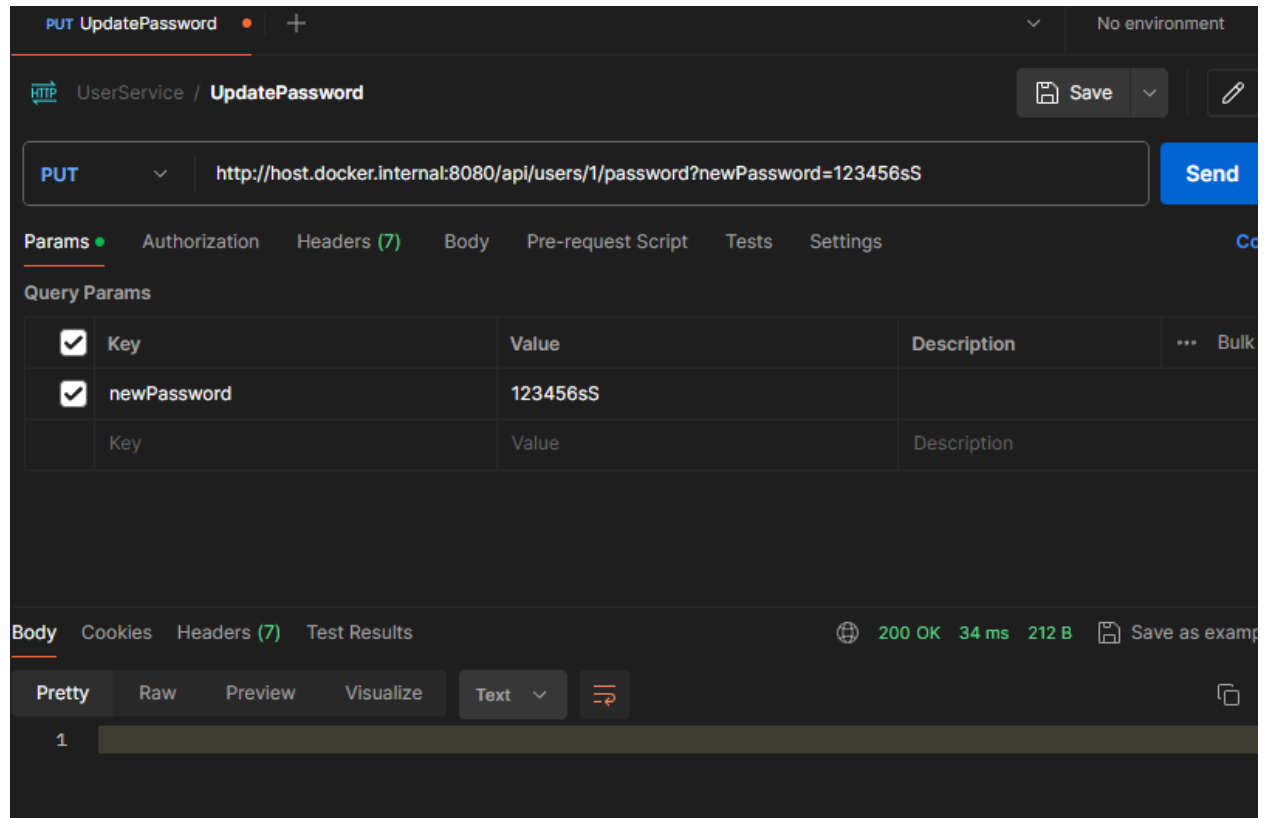
The screenshot shows a REST client interface with the following details:

- Request Method:** PUT
- Endpoint:** `http://host.docker.internal:8080/api/users/1/email?newEmail=ah@gmail.com`
- Query Params:**

Key	Value	Description
newEmail	ah@gmail.com	
- Response:** 200 OK, 127 ms, 212 B
- Body:** Pretty (Raw, Preview, Visualize, Text)

4- **PUT**(UpdatePassword) :

[http://host.docker.internal:8080/api/users/\\${userId}/password?newPassword=\\${newPassword}](http://host.docker.internal:8080/api/users/${userId}/password?newPassword=${newPassword})



6- Conclusion

This Software Design Document provides a detailed overview of Taskaty app with TaskMicroservice and UserMicroservice applications, including their purpose, scope, architecture, APIs, and key functionalities. It serves as a guide for developers and stakeholders involved in the development and deployment of these microservices.