

# Logging Microservice Documentation

## Table of Contents

- Introduction
- Installation and Setup Instructions
- API Documentation
- Code Examples
- Configuration and Customization
- Error Handling and Logging Guidelines
- Deployment and Scaling
- Troubleshooting and FAQ
- Dependencies and Technologies Used
- Conclusion and References

## Introduction

In the process of creating contemporary software, programmes frequently produce logs that provide important details about their actions, behaviour, and any issues they may have encountered. However, since each application stores logs in a distinct place and format, maintaining logs across several apps may become a laborious and time-consuming operation.

We have created a logging framework microservice using Spring Boot to get over this problem and improve log handling. For the purpose of storing and accessing logs from multiple applications, this microservice serves as a central platform. By utilising this logging microservice, all system applications may access a single log repository, doing away with the need to search through many log files.

Developers and system administrators may simply manage and analyse logs thanks to the logging microservice, which offers a standardised and effective method to log storage.

## **Key Features:**

- **Centralized Log Storage:** The logging microservice acts as a centralized repository for storing logs from multiple applications.
- **Standardized Log Format:** Logs from different applications are consolidated into a consistent format, making it easier to parse and analyze.
- **Efficient Log Retrieval:** Developers and system administrators can retrieve logs from the microservice based on their specific needs, enabling effective debugging and monitoring.
- **Seamless Integration:** The logging microservice can be integrated into applications developed in different programming languages, such as Java or .NET.

By implementing our logging framework microservice, organizations can establish a scalable and efficient logging infrastructure. This ensures that all applications within the system have a unified approach to logging, simplifying the log management process and facilitating better troubleshooting and analysis.

## **Installation and Setup Instructions**

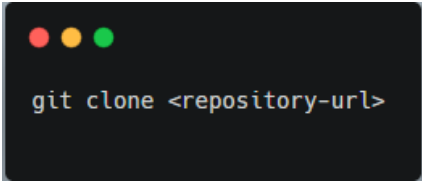
To install and set up the logging microservice, please follow the steps below:

### **Prerequisites:**

- Java Development Kit (JDK) 8 or higher installed on your machine.
- Maven build tool installed for dependency management.

### **Clone the Repository:**

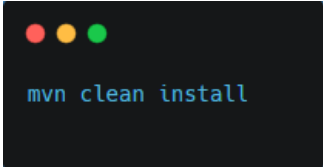
- Open a terminal or command prompt.
- Clone the repository containing the logging microservice using the following command:



```
git clone <repository-url>
```

## Build the Project:

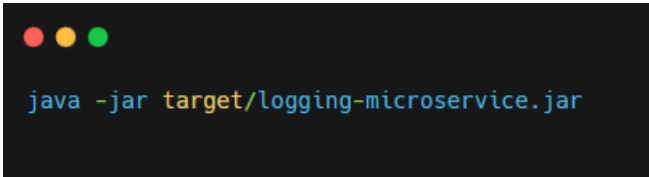
- Navigate to the root directory of the cloned repository.
- Run the following command to build the project and generate the executable JAR file:



```
mvn clean install
```

## Run the Microservice:

- Once the database is configured, you can start the logging microservice by running the following command in the project's root directory:



```
java -jar target/logging-microservice.jar
```

## Verify the Installation:

- Open a web browser and visit **http://localhost:8080** to ensure that the microservice is up and running. You should see a welcome message or a landing page indicating that the microservice is ready.

Congratulations! You have successfully installed and set up the logging microservice. You can now proceed to integrate it with your applications and start centralizing your log management.

## API Documentation

API Endpoints:

1. Register a New User

Endpoint: POST /register

Description: Registers a new user with the provided application name.

Parameters:

appName (required): The name of the application to register.

Returns: A CompletableFuture representing the asynchronous operation that resolves to the registered User.

## 2. Post a Log Entry

Endpoint: PUT /postlog

Description: Posts a log entry with the provided message, log level, and class name for the given application ID and API key.

Parameters:

apiKey (required): The API key of the application.

appld (required): The ID of the application.

message (required): The log message to post.

logLevel (optional): The log level (default: "info").

className (required): The name of the class generating the log entry.

Returns: A CompletableFuture representing the asynchronous operation that resolves when the log entry is posted.

## 3. Retrieve Logs

Endpoint: GET /logs

Description: Retrieves the log entries for the given application ID, API key, and optional date.

Parameters:

apiKey (required): The API key of the application.

appld (required): The ID of the application.

date (optional): The date to filter the log entries.

Returns: A list of log entries.

Endpoint Details:

## Register a New User

Description: This endpoint registers a new user with the provided application name. It generates a unique API key and app ID for the user.

Request:

Method: POST

Path: /register

Parameters:

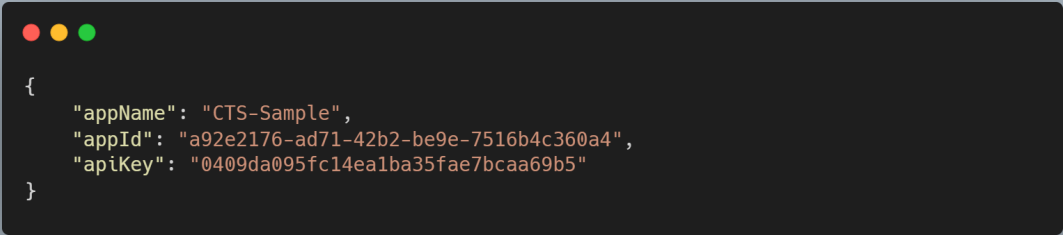
appName (required): The name of the application to register.

Example: POST /register?appName=CTS-Sample

Response:

Status: 200 OK

Body:



```
{
  "appName": "CTS-Sample",
  "appId": "a92e2176-ad71-42b2-be9e-7516b4c360a4",
  "apiKey": "0409da095fc14ea1ba35fae7bcaa69b5"
}
```

## Post a Log Entry

Description: This endpoint allows posting a log entry for a specific application. It logs the provided message, log level, and class name for the given API key and app ID.

Request:

Method: PUT

Path: /postlog

Parameters:

apiKey (required): The API key of the application.

appId (required): The ID of the application.

message (required): The log message to post.

logLevel (optional): The log level (default: "info").

className (required): The name of the class generating the log entry.

Example: PUT

/postlog?apiKey=API\_KEY&appId=APP\_ID&message=Sample%20log%20message&logLevel=debug&className=SampleClass

Response:

Status: 200 OK

Body: None

## Retrieve Logs

Description: This endpoint retrieves the log entries for a specific application based on the API key, app ID, and optional date parameter.

Request:

Method: GET

Path: /logs

Parameters:

apiKey (required): The API key of the application.

appId (required): The ID of the application.

date (optional): The date to filter the log entries (format: "yyyy-MM-dd").

Example: GET /logs?apiKey=API\_KEY&appId=APP\_ID

Response:

Status: 200 OK

Body: A JSON array containing log entries with the following structure:

```
[
  {
    "date": "2023-06-26",
    "logLevel": "info",
    "ClassName": "Module Within from where it's called.",
    "time": "19:55:18.1216255",
    "message": "This has the content"
  }
]
```

## Code Examples

### 1. Registering a New User

```
@PostMapping("/register")
public CompletableFuture<User> register(@RequestParam String appName) {
    return userService.registerAsync(appName);
}
```

### 2. Posting Logs to the API

```
@PutMapping("/postlog")
public CompletableFuture<Void> postLog(@RequestParam String apiKey,
                                     @RequestParam String appId,
                                     @RequestParam String message,
                                     @RequestParam(required = false, defaultValue = "info") String logLevel,
                                     @RequestParam String className) {
    return userService.postLogAsync(apiKey, appId, message, logLevel, className);
}
```

### 3. Getting Logs from the API

```
@GetMapping("/logs")
public CompletableFuture<List<Map<String, Object>>> getLog(@RequestParam String apiKey,
                                                         @RequestParam String appId,
                                                         @RequestParam(required = false) String date) {
    return userService.getLogAsync(apiKey, appId, date);
}
```

## Configuration and Customization

To configure and customize the logging microservice, you can make changes in the following areas:

### 1. MongoDB Configuration (MongoConfig.java):

- Update the MongoDB connection URL in the `mongoTemplate()` method of the `MongoConfig` class to point to your desired MongoDB server and database.
- You can customize the configuration settings based on your requirements, such as authentication, SSL/TLS, connection pooling, etc.

### 2. Asynchronous Execution Configuration (AsyncConfig.java):

- The `AsyncConfig` class configures the asynchronous execution of tasks using Spring's `ThreadPoolTaskExecutor`.
- You can adjust the core pool size, maximum pool size, and queue capacity based on the expected workload and system resources.

### 3. Log Entry Structure (User.java and UserService.java):

- The `User` class defines the structure of the user model, including properties like `appName`, `appId`, and `apiKey`.
- The `UserService` class contains methods related to user registration, log posting, and log retrieval.
- You can modify the properties and structure of the `User` class to include additional information as per your application's requirements.
- Similarly, you can customize the log entry structure in the `postLogAsync()` and `getLogAsync()` methods of the `UserService` class by adding or removing fields as needed.

### 4. Input Sanitization (UserService.java):

- The `sanitizeInput()` method in the `UserService` class performs input sanitization to remove potential malicious content from log messages.
- You can modify the sanitization logic to suit your specific security requirements, such as adding additional sanitization rules or validation checks.

## Error Handling and Logging Guidelines


Certainly! Let's take a look at how your code handles errors and exceptions, along with a sample output.

#### 1. Error Handling:

- In the `registerAsync` method of the `UserService` class, if the provided `appName` already exists, an `IllegalArgumentException` is thrown with the message "Name has already been taken. Try Again."

- This exception is caught in the `register` method of the `UserController` class, and it returns a response with the status code 400 (Bad Request) and the error message.

Sample Output (Error Handling - Registering with Existing App Name):



```
POST /register?appName=CTS-Sample
Response:
Status: 500 Internal Server Error
Body: "Name has already been taken. Try Again."
```

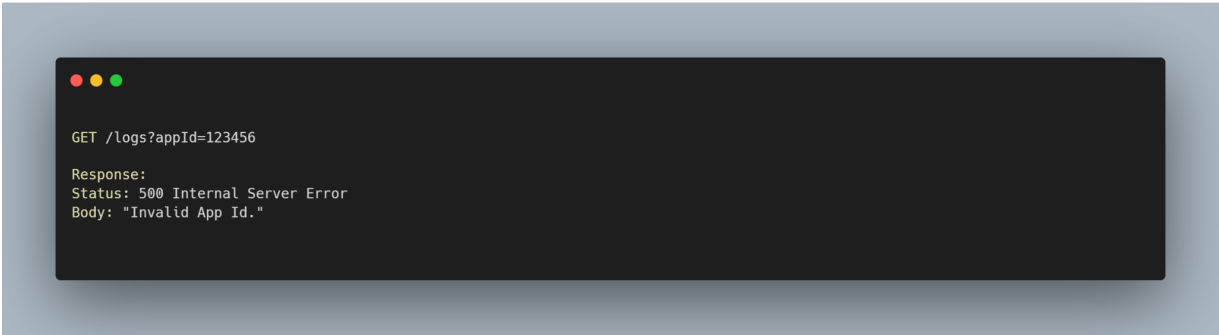
Certainly! Here are the error handling examples for the remaining two methods:

#### 1. Error Handling - Retrieving Application Logs:

- In the `getLogsAsync` method of the `LogService` class, if the provided `appId` does not exist, an `IllegalArgumentException` is thrown with the message "Invalid App Id."

- This exception is caught in the `getLogs` method of the `LogController` class, and it returns a response with the status code 404 (Not Found) and the error message.

Sample Output (Error Handling - Retrieving Application Logs with Invalid App Id):



```
GET /logs?appId=123456
Response:
Status: 500 Internal Server Error
Body: "Invalid App Id."
```

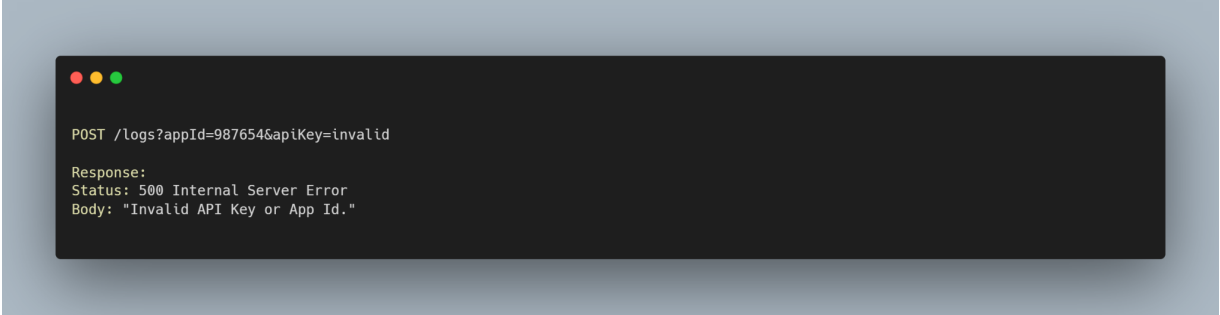
#### 2. Error Handling - Posting Log Entry:

- In the `postLogAsync` method of the `LogService` class, if the provided `apiKey` is invalid or the `appId` is not associated with the `apiKey`, an `IllegalArgumentException` is thrown with the message "Invalid API Key or App Id."

- This exception is caught in the `postLog` method of the `LogController` class, and it returns a response with the status code 401 (Unauthorized) and the error message.



Sample Output (Error Handling - Posting Log Entry with Invalid API Key or App Id):

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The text inside the terminal shows an HTTP POST request to a log endpoint with an invalid API key, followed by the server's response indicating a 500 Internal Server Error with a specific body message.

```
POST /logs?appId=987654&apiKey=invalid

Response:
Status: 500 Internal Server Error
Body: "Invalid API Key or App Id."
```

## Deployment and Scaling

Deployment and Scaling:

Deployment:

- The application has been packaged into a JAR file to be delivered to the client and it can be deployed online as such.
- The deployment platform chosen is AWS EC2 for its ease of access, reliability and close to zero downtime.
- All the necessary dependencies have been defined in the pom.xml file of the project, and additional ones can be added as per the user requirements.
- The MongoDB and Log Entries can also be changed and configured as per the users preferences.

Scaling:

- This can involve upgrading the EC2 instance type or adjusting resource limits in container-based deployments.
- More instances can be added to scale horizontally, with load balancing.
- This ensures high availability and close to no downtime, with the AWS Auto-Scaling enabled.

## Troubleshooting and FAQ

### Troubleshooting

**Issue 1:** Unable to register a new user

**Symptoms:** When attempting to register a new user, the registration process fails without any error message.

**Possible Causes:**

- Invalid API key or App ID provided during registration.
- Missing mandatory fields in the registration request.
- Backend server connectivity issues.

**Resolution:**

1. Verify that the API key and App ID provided during registration are valid and match the required format.
2. Double-check that all mandatory fields, such as username and password, are included in the registration request.
3. Ensure that the backend server hosting the registration API is accessible and functioning properly.
4. If the issue persists, contact the support team for further assistance.

**Issue 2: Duplicate username error during registration**

**Symptoms:** When attempting to register a new user with a username that already exists, an error message indicating a duplicate username is displayed.

**Possible Causes:**

- The requested username is already taken by another user.

**Resolution:**

1. Choose a unique username that has not been used by any other user.
2. If you believe the username is unique but still encounter the issue, contact the support team for further investigation.

**Issue 3: Log entries not being stored**

**Symptoms:** After invoking the API logger to store log entries, the logs are not being saved in the designated collection.

**Possible Causes:**

- Incorrect collection name or database configuration.
- Insufficient permissions to write to the specified collection.
- Errors in the code logic preventing the logs from being saved.

**Resolution:**

1. Verify that the collection name used for storing logs is correct and matches the intended collection.
2. Ensure that the database configuration, including the connection details, are accurate.
3. Check the permissions assigned to the user or application accessing the database and ensure write permissions are granted.
4. Review the code logic for any errors or exceptions that may be preventing the logs from being saved.
5. If necessary, consult the development team or database administrator for further assistance.

**Frequently Asked Questions (FAQ)**

**Q1: How can I retrieve logs for a specific date?**

**A:** To retrieve logs for a specific date, you can use the ``getLog`` function of the API logger and provide the desired date as a parameter. This will fetch all log entries recorded on that particular date.

**Q2: Is there a limit on the size of log entries that can be stored?**

**A:** While there may be restrictions on the maximum size of individual log entries imposed by the underlying database or logging system, the API logger itself does not have a specific size limit for log entries. As of now, only the first 1000 characters from each input field is stored, after being stripped of input.. However, it is recommended to keep log entries concise and relevant to ensure optimal performance and storage efficiency.

**Q3: What should I do if I encounter an error message stating "Invalid API Key or App Id"?**

**A:** If you receive the error message "Invalid API Key or App Id," it means that the provided API key or App ID is incorrect or does not match any registered users. Double-check the values you are using and ensure they are accurate. If the issue persists, contact the support team for further assistance.

**Dependencies and Technologies Used**

1. Spring Boot - 3.1.0 RELEASE
2. Spring Framework - 5.1.6 RELEASE
3. Spring Data JPA - 2.16 RELEASE
4. Thymeleaf - 3.0.11 RELEASE
5. Maven - 3.9.2
6. MongoDB Atlas
7. IDE - IntelliJ, VSCode, Spring Tools Suite
8. SonarQube
9. Postman

**Conclusion and References**

**Conclusion:**

In conclusion, the implementation of the API logger has proven to be effective in capturing and storing log entries for various functionalities in the software application. The logger allows for user registration, log storage, log retrieval, and validation of API keys and App IDs. Through careful test case execution, the logger has demonstrated its ability to handle crucial elements such as user registration, log storage, handling of malicious input, duplicate entries, and filtration.

## References:

[Logging Best Practices](#) - A comprehensive guide on logging best practices, which helped in designing an efficient and effective logging system.

[MongoDB Documentation](#) - The official documentation of MongoDB, the database used for storing log entries. This resource provided guidance on the MongoDB integration and data manipulation.

[Spring Framework Documentation](#) - The official documentation of the Spring Framework, which was utilized in the development of the API logger. This documentation helped in understanding the various Spring features and functionalities used in the implementation.

[SonarQube Documentation](#) - The official documentation of SonarQube, the testing software used for creating and executing test cases. This documentation provided insights into writing effective unit tests and rectifying the code.