Design Document: Centralized Logging REST API

1. Overview

The Centralized Logging REST API is designed to handle log entries from various components of a distributed system, including services written in different technologies like .NET, Power BI, and Databricks. This API uses the `ArcCommonLogger` library to process and forward logs to an upstream logging system that integrates with Azure HDInsight and Big Panda for alerts and notifications.

1.1 Objectives

- Provide a single RESTful API endpoint to accept both simple and detailed log entries.
- Dynamically process the logs, converting simple entries into detailed entries where necessary.
- Use the `ArcCommonLogger` for consistent log processing and forwarding.
- Ensure compatibility with enterprise-level requirements like security, scalability, and maintainability.

2. Architecture

2.1 System Architecture

- **Client Components**: Various services and applications (e.g., .NET, Power BI) send logs to the `/api/logs` endpoint using HTTP POST requests.
- **Spring Boot Logging API**: A REST API built using Spring Boot that:
    - Accepts log entries.
    - Dynamically parses the incoming requests to determine the type of log (basic or detailed).
    - Converts simple logs into detailed logs.
    - Uses `ArcCommonLogger` to forward logs to the upstream logging processor.
- **Upstream Processor**: External system that `ArcCommonLogger` communicates with for further processing, including sending logs to Azure HDInsight and Big Panda.

2.2 Workflow

1. **Client sends a POST request** to `/api/logs` with JSON data containing log details.

2. **API parses the request** into a Java `Map<String, Object>`.
3. **API determines the type of log**:
    - If detailed fields (`processName`, `metrics`) are present, it processes as a `LogEntry`.
    - If those fields are absent, it treats the request as a `BasicLogEntry`.
4. **Basic logs are converted to detailed logs** with default values.
5. **API uses `ArcCommonLogger`** to forward the log.
6. **Response** is returned to the client, indicating success or failure.

3. API Endpoints

3.1 POST `/api/logs`

- **Description**: Accepts both basic and detailed log entries.
- **Request Body**: JSON object that can represent either a `BasicLogEntry` or a `LogEntry`.
- **Response Codes**:
    - `200 OK`: Log successfully processed and forwarded.
    - `400 Bad Request`: Validation error in the input data.
    - `500 Internal Server Error`: Errors while processing the log.

3.2 Example Requests

- **Basic Log Request**:
  ```
  json
  {
    "logLevel": "INFO",
    "message": "Service started successfully.",
    "componentName": "DotNetServiceA"
  }
  ```
- **Detailed Log Request**:
  ```
  json
  {
    "timestamp": "2024-10-07T14:30:00Z",
    "logLevel": "ERROR",
    "componentName": "DotNetServiceA",
    "processName": "ARC_DRF_LOAD_CALC",
    "subSystem": "DATA_MODELS",
    "message": "An error occurred during processing.",
    "metrics": {
      "metricName": "CALC_EVENT",
      "status": "FAILED",
      "eventId": "12345-abcde-67890",
  ```

```
      "context": "Sample context message"
    },
    "severityLevel": 2,
    "exception": {
      "message": "Null pointer exception",
      "stackTrace": "java.lang.NullPointerException: at ..."
    },
    "customProperties": {
      "userId": "user-123",
      "operation": "data load"
    }
  }
```

4. Data Models

4.1 BasicLogEntry.java

- **Purpose**: Represent minimal log data.
- **Fields**:
    - `String logLevel`
    - `String message`
    - `String componentName`

4.2 LogEntry.java

- **Purpose**: Represent detailed log data.
- **Fields**:
    - `String timestamp`
    - `String logLevel`
    - `String componentName`
    - `String processName`
    - `String subSystem`
    - `String message`
    - `Metrics metrics`
    - `int severityLevel`
    - `ExceptionDetails exception`
    - `Map<String, String> customProperties`

4.3 Metrics.java

- **Purpose**: Store metric details.
- **Fields**:

- `String metricName`
- `String status`
- `String eventId`
- `String context`

4.4 ExceptionDetails.java

- **Purpose**: Represent exception information.
- **Fields**:
  - `String message`
  - `String stackTrace`

5. Service Design

5.1 LogService.java

- **Methods**:
  - `void processLog(Map<String, Object> logRequest)`: Determines the type of log and processes it.
  - `void processLog(LogEntry logEntry)`: Processes detailed logs using `ArcCommonLogger`.
  - `void processLog(BasicLogEntry basicLogEntry)`: Converts a `BasicLogEntry` into a `LogEntry` with default values and processes it.
  - `LogEntry convertToLogEntry(Map<String, Object> logRequest)`: Converts the incoming request to a `LogEntry`.
  - `BasicLogEntry convertToBasicLogEntry(Map<String, Object> logRequest)`: Converts the incoming request to a `BasicLogEntry`.

5.2 Example: Converting BasicLogEntry to LogEntry

```
java
public void processLog(BasicLogEntry basicLogEntry) {
    LogEntry logEntry = new LogEntry();
    logEntry.setLogLevel(basicLogEntry.getLogLevel());
    logEntry.setMessage(basicLogEntry.getMessage());
    logEntry.setComponentName(basicLogEntry.getComponentName());
    logEntry.setTimestamp(Instant.now().toString());
    logEntry.setProcessName("DEFAULT_PROCESS");
    logEntry.setSubSystem("DEFAULT_SUBSYSTEM");
    logEntry.setMetrics(new Metrics("DEFAULT_METRIC", "N/A", UUID.randomUUID().toString(), "No context provided"));
    logEntry.setSeverityLevel(1);
    logEntry.setCustomProperties(Collections.emptyMap());

    processLog(logEntry);
}
```

6. Security Requirements

- **Authentication**: Use OAuth2 or JWT-based authentication for secure access.
- **Authorization**: Implement role-based access control (RBAC) for access management.
- **Input Validation**: Validate all incoming request data to prevent injection attacks.
- **TLS/HTTPS**: Ensure all communication is encrypted using TLS.

7. Deployment Requirements

- **Environment-specific Configuration**: Use Spring Profiles for `dev`, `qa`, and `prod` configurations.
- **Containerization**: Package the API as a Docker image.
- **Kubernetes**: Deploy on AKS with scaling and load balancing.
- **CI/CD**: Use GitLab pipelines for automated builds and deployments.

8. Error Handling

- **400 Bad Request**: Returned when required fields are missing or improperly formatted.
- **500 Internal Server Error**: Returned for unexpected errors during log processing.
- **Detailed Logging of Errors**: Use `ArcCommonLogger` to log internal errors while handling user requests.

9. Example Integration for Clients

9.1 Sample .NET Client

```
csharp
using System.Net.Http;
using System.Text;
using Newtonsoft.Json;

var httpClient = new HttpClient();
var logEntry = new
{
    logLevel = "INFO",
    message = "Service started successfully.",
    componentName = "DotNetServiceA"
};

var json = JsonConvert.SerializeObject(logEntry);
var content = new StringContent(json, Encoding.UTF8, "application/json");

var response = await httpClient.PostAsync("http://logging-service/api/logs", content);
```

10. Additional Resources

- **ArcCommonLogger Documentation**: Refer to the provided documentation for details on integrating with `ArcCommonLogger`.
- **Spring Boot Reference**: For guidance on REST API implementation and security.
- **Docker and Kubernetes Guides**: For containerization and deployment practices.