This document explains the architecture of **Check-In-Service** built using **SpringBoot (Java)**. The application will provide a single endpoint to record employee check in and check out times. The application will make the process of calling third party APIs and sending emails to users asynchronous by taking an event driven approach. **Apache Kafka** will be used for this purpose. This process will work as follows.

1. Card reader calls check in service's endpoint with **employee ID** as a **path parameter**. The endpoint delegates the employee ID to the **service layer**.
2. The service layer will call the repository layer to retrieve a check in record for the employee against his employee ID and the following actions will take place.
    a. If no record is found, it means that the employee hasn't checked in. A **new record** will be created in Check-In-Service's database with the check in time, employee ID, and PENDING statuses for email delivery, and legacy system notification.
    b. If a record is found for the specified employee ID. The following actions will take place in order.
        i. Check out time will be noted and the **record** will be **updated** in Check-In-Service's database with the **checkout time**.
        ii. The service calculates the shift duration for each employee. If the duration exceeds the maximum allowed shift hours, the system flags it as a 'missed checkout.' In this case, the tracked hours are capped at the maximum allowed limit. Otherwise, the hours are derived directly from the check-in and check-out times. Finally, the service publishes messages to the **Kafka topics** responsible for **email notifications** and **legacy system** synchronization.
        iii. If the service identifies that the employee missed their checkout in the previous step, a new check in record is created for that employee, otherwise the service simply transitions to the next step.
3. The API returns a success message to the caller.

In this way the process of check in and check out are made asynchronous and the caller immediately receives a success response while the processes of notifying the legacy system and emailing the employee are delegated.

Consistency, **retries**, and **error handling** are ensured through the following implementation of the **Kafka** consumers.

- Legacy System API call
    - The topic is made retryable using exponential backoff strategy. Exponential backoff strategy is chosen instead of a fixed delay to tackle rate limiting.
    - The consumer receives a message with employee id, tracked hours and a record id.
    - The consumer calls a mock legacy recording system api call with the employee id and tracked hours.
    - If the api call succeeds, the consumer transitions the legacy system notified status of the record to NOTIFIED from PENDING using the record id.
    - If the api call fails, the system isn't blocked, instead the message is enqueued to a retry topic specified for this event.

    - If the message fails for a specified number of retries, it is enqueued to a Kafka dead letter topic.
    - The dead letter queue consumer receives this message, logs the exception and updates the legacy system notified status of the record to FAILED.
- Emails
    - The topic is made retryable using exponential backoff strategy.
    - The consumer receives a message with employee id, tracked hours, and record id.
    - The consumer calls a mock user service to get the email of the employee.
    - The consumer calls the email service with the employee's email, and tracks hours to send the email.
    - If email is sent successfully, the email sent status of the record is set to SENT using the record id.
    - If the email delivery failed, the message is moved to a retry topic specified for this event.
    - If the message fails for a specified number of retries, it is enqueued to a dead letter topic,
    - The dead letter queue consumer receives this message, logs the exception and updates the email sent status of the record to FAILED.

Circuit Breaker pattern is used using spring cloud's circuit breaker artifact to handle rate limiting and outages of the legacy recording system. It works as follows:

- A handler is implemented to handle circuit breaker state transitions (OPEN, CLOSED) etc.
- When the service goes down, or check in service is being rate limited, the circuit breaker transitions its state to OPEN. A specified number of recent calls, and a specified failure rate percentage are taken into account to move the circuit breaker to OPEN state.
- The handler handles the state transitions as follows:
    - Open
        - Pauses Kafka consumer from listening to messages as it saves retries destined for failures and unnecessary entries to dead letter queue.
    - Half open
        - Resumes the kafka consumer to allow legacy service API calls. A specified number of API calls are made as validation to check if service recovered.
        - If legacy service API calls succeed, the circuit breaker state transitions to CLOSED.
        - If legacy service API calls fail, the circuit breaker state transitions to OPEN.
    - Closed
        - Resumes the kafka consumer.

Monitoring and **Tracing** is implemented using **Apache Log4j 2**, extensive logs are added to produce traces in various steps (request, database updates, message production, consumption, success, retries and failures) during the lifecycle of a single request. A logfile is produced to maintain the logs. SpringBoot actuator is added to provide an endpoint to analyze the logs, and SpringBoot Admin client is added to link to an admin interface to monitor the request lifecycle logs. **Hexagonal Pattern** is implemented to create **loosely coupled components** to enhance **maintainability** and **extensibility**. **Google Gemini** was used to write boilerplate code (i.e. readme file), fix syntax errors, and research spring specific implementation of resilience4j.