**Objective**

The goal is to handle **client requests** to query and create a "cork" object (essentially a maintenance request) for a device, ensuring proper **authentication**, **validation**, and seamless interaction between APIs.

**Key Points in the Workflow**

1. **Client Request:**
   * The client initiates a request (payload).
   * Payload includes basic information (e.g., client details, device identifier).
2. **Authentication:**
   * Perform **two levels of authentication**:
     + For now, use **hard-coded values** for client authentication (time-constrained solution).
     + In the future, implement a dynamic authentication mechanism.
   * Validate the **client identity** based on the payload.
3. **API Interaction:**
   * **Access Token Generation:**
     + Use the credentials to generate an access token behind the scenes.
   * **Cork Management APIs**:
     + **Query Cork API**: Check if the device is in maintenance.
     + **Create Cork API**: Create a maintenance request for the device if eligible.
4. **Decision-Making:**
   * Should the **client query the device status** before creating the request themselves?
     + If yes, provide API documentation for querying.
     + If no, automate the query within the system workflow and provide a single interaction point for the client.

**Real-World Context**

* **Why Query First?**
  + Maintenance on devices (like laptops or wireless equipment) must be pre-checked to ensure **no conflicts** with ongoing activities.
  + For example:
    - If Alex is working on Device ABCD, others cannot perform overlapping tasks. Hence, querying the device ensures it is not under maintenance before creating a new request.
* **Compliance Requirements:**
  + Creating a maintenance request is mandatory to:
    - Log the activity in a traceable system (similar to creating a JIRA).
    - Maintain a record for audits and compliance.

**Proposed Workflow**

1. **Client Interaction:**
   * Client submits a request payload with required details (e.g., device, activity type).
2. **Query Device Status:**
   * System checks if the device is under maintenance.
   * Returns status to the client (e.g., "Available" or "Under Maintenance").
3. **Create Cork Request:**
   * If the device is free, proceed to create the cork request automatically or prompt the client to confirm.
4. **Feedback/Response:**
   * System provides a response with the cork details (if created) or the maintenance status.

**Next Steps**

* **For Immediate Implementation:**
  1. Add a **hard-coded authentication check** for the single client.
  2. Implement basic **payload validation**.
  3. Integrate the **Query Cork API** to check device status.
  4. Provide a workflow where the client queries first, or the system handles it automatically based on requirements.
  5. Return meaningful responses to guide the client.
* **For Future Enhancements:**
  1. Replace hard-coded authentication with a robust mechanism.
  2. Add dynamic support for multiple clients.
  3. Develop a more detailed logging system to track activities for compliance.

**Questions for the Team**

1. **Query vs. Create Integration:**
   * Should we give the client control over the query step, or integrate it into the system workflow for simplicity?
2. **API Authentication:**
   * Should the system handle token management transparently, or require the client to supply tokens?
3. **Response Expectations:**
   * What level of detail should we return in responses (e.g., exact device status, cork request details)?

**Key Objectives**

1. **Focus Areas**:
   * **Create**, **Query**, **Close**, and **Correct/Conduct** APIs for managing maintenance activities (e.g., cork requests).
   * Ensure the project structure is **clean, readable, and scalable**.
2. **Critical Components**:
   * **Payload Definitions**:
     + Define **request** and **response payloads** clearly using Python classes for better understanding by clients and the team.
   * **Unit Testing**:
     + Integrate unit tests early to validate functionality and ensure the project remains robust.
   * **Swagger Documentation**:
     + Generate Swagger/OpenAPI specs to document the APIs, ensuring clarity for client teams.
   * **Log Requests**:
     + Push logs to **ELK Stack** instead of storing data locally to save time.
3. **Scaffold and Skeleton**:
   * You’ve set up a basic structure under the **December feature**, including:
     + **Config**, **Models**, and **Source** folders.
     + Skeletons for APIs (with placeholder methods and class definitions).
     + A **Dockerfile** and basic requirements.txt.

**Immediate Action Items**

Here’s how the team should proceed step-by-step:

**1. Review and Fix the Skeleton**

* **Config**: Review the settings and ensure environment-specific configurations are included.
* **Models**: Start defining models based on the payload structure you outlined.
* **Source**:
  + **Cork API**:
    - Implement the **Create** and **Query** methods as per your payload specifications.
    - Ensure modular design (e.g., separate concerns for request validation, processing, and response).
  + **Main API**:
    - Set up the entry points for the application (e.g., Flask or FastAPI routing).

**2. Define Payloads and Validation**

* Use Python **dataclasses** or **Pydantic models** to define:
  + **Request Payloads**:
    - Include client ID, device ID, and any activity metadata.
  + **Response Payloads**:
    - Define what information will be returned (e.g., cork status, request IDs).
* Implement payload validation to handle errors gracefully.

**3. Logging**

* Set up the system to log API requests and responses to the **ELK Stack**.
* For now, avoid local storage or database integration unless necessary.

**4. Docker Setup**

* Finalize the **Dockerfile** to ensure the project can run seamlessly in containers.
* Include necessary environment variables in the config or .env file.

**5. Unit Testing**

* Use a testing framework like **pytest** to:
  + Validate payload definitions.
  + Test individual API functions (mock external dependencies if required).

**6. Swagger/OpenAPI Integration**

* Use a framework like **FastAPI** or **Flask-Swagger** to auto-generate Swagger documentation from the codebase.
* This ensures the API is easy to understand and use for external clients.

**7. Priority Tasks**

* You will focus on providing:
  + The **payload definitions** (request and response) for **Create** and **Query APIs**.
  + Skeleton implementations for **Close** and **Conduct APIs**.
* The team will:
  + Complete the implementation of **Create** and **Query APIs**.
  + Structure the project for scalability and readability.

A screenshot of a computer program

Description automatically generated

**Next Steps for the Team**

1. Review the provided skeleton and payload definitions.
2. Begin implementation of the **Create** and **Query APIs**.
3. Set up Swagger documentation for the exposed APIs.
4. Push logs to ELK Stack for all API requests/responses.

**Key Updates & Decisions**

1. **Persistence (MongoDB)**:
   * **No Immediate Persistence**:
     + The system is transient for now, so **storing data** is not a priority.
     + This ensures the API functions without blockers even if storage mechanisms fail.
   * **Future Integration**:
     + **MongoDB** will be used to persist **JSON objects** in the long term.
     + The design will prioritize simplicity by avoiding unnecessary transformations of the data structure (JSON in, JSON stored).
     + This approach minimizes time spent understanding or converting attributes.
2. **Error Handling**:
   * The system should handle failures in storage gracefully (e.g., log the issue but continue processing requests).
   * Storage-related issues must not become blockers for the main flow.
3. **Wish List for Future Enhancements**:
   * Persisting data (MongoDB integration).
   * Reporting and analytics functionalities.
   * A robust error-handling and notification mechanism for failures.
4. **Registration Process**:
   * A **registration process** needs to be completed (possibly related to client or API access).
   * You will handle this registration, but the team should remain prepared for instructions.

**Action Items**

1. **Immediate Tasks**:
   * Finalize and implement the **Create** and **Query APIs** without persistence.
   * Ensure the system is transient and does not break due to storage unavailability.
   * Continue logging requests and responses to the **ELK Stack**.
2. **Future Planning**:
   * Add MongoDB integration to the **wish list** with clear goals for:
     + Storing JSON objects for easier access and future scalability.
     + Simplifying reporting and analytics by leveraging the stored data.
3. **Follow-Up**:
   * You will provide details on the registration process and any related dependencies.
   * Once registered, ensure the APIs are tested in the registered environment.

**Next Steps for the Team**

1. **Focus on API Implementation**:
   * Continue with the transient design.
   * Use logging for traceability in place of persistence.
2. **Prepare for MongoDB**:
   * Plan how to structure the data (JSON object-based).
   * Keep integration modular to add persistence later without disrupting the flow.
3. **Action Wishlist**:
   * Maintain a prioritized list of enhancements for reporting, error handling, and persistence.

**Development Tools**

1. **Python Frameworks**:
   * **FastAPI** or **Flask**: For creating lightweight and scalable REST APIs.
   * **Pydantic**: For payload validation and schema definitions.
2. **Unit Testing**:
   * **pytest**: For creating and running test cases.
   * **mock**: For mocking external dependencies during testing.
3. **Code Quality**:
   * **Black**: For automatic code formatting.
   * **flake8**: For linting and ensuring PEP8 compliance.
   * **mypy**: For optional static typing checks.

**Logging and Monitoring**

1. **ELK Stack**:
   * **Elasticsearch, Logstash, Kibana**: For centralized logging, analysis, and visualization of logs.
2. **Application Monitoring**:
   * **Prometheus and Grafana**: For monitoring system health, API performance, and resource utilization.

**Database and Storage**

1. **Primary Database**:
   * **MongoDB**: For persisting JSON objects and enabling seamless querying of unstructured data.
2. **Future Considerations**:
   * **PostgreSQL** or **MySQL**: If relational database support becomes necessary for structured data.
   * **Redis**: For caching frequently accessed data to improve API response times.

**Containerization and Deployment**

1. **Docker**:
   * For containerizing the application and ensuring consistency across environments.
2. **Kubernetes**:
   * For managing and orchestrating containers at scale (if required later).
3. **CI/CD Tools**:
   * **GitLab CI/CD** or **Jenkins**: For automating build, test, and deployment pipelines.

**API Documentation**

1. **Swagger / OpenAPI**:
   * To auto-generate and maintain API documentation for ease of use by client teams.
2. **Postman**:
   * For testing APIs and sharing test collections with the team.

**Authentication and Security**

1. **OAuth2 / JWT**:
   * For secure client authentication and token-based authorization.
2. **HashiCorp Vault**:
   * For managing sensitive credentials like API keys and environment secrets.

**Future Enhancements**

1. **Reporting and Analytics**:
   * **Tableau** or **Power BI**: For advanced data visualization and reporting.
2. **Error Tracking**:
   * **Sentry**: For real-time error tracking and debugging.
3. **Event Streaming**:
   * **Apache Kafka**: For handling real-time data pipelines and event-driven systems.

**Collaboration and Communication**

1. **GitLab**:
   * For source control, code review, and collaboration.
2. **JIRA**:
   * For task tracking and sprint planning.
3. **Confluence**:
   * For documenting processes, workflows, and decisions.