OSS ODYSSEY

Purpose of the Application

The **Open-Source Contributor Hub** is designed to empower developers—especially beginners—to discover and contribute to open-source projects. By aggregating beginner-friendly issues from platforms like GitHub and GitLab, the application enables users to:

- Browse and filter issues (e.g., by labels such as "good first issue", programming languages, etc.).
- Refresh issue data to always have up-to-date tasks.
- Manage their personal contributions by creating projects and tracking the progress of issues they choose to work on (from "in progress" to "completed" with PR/patch submissions).

This clearly defines key business domains—issue aggregation, user/project management, and data synchronization—which can be isolated into separate microservices for scalability and maintainability.

Main Features / Modules

1. **Issue Aggregation Service**
    - **Responsibilities:**
        - Integrate with GitHub and GitLab APIs (using GraphQL and/or REST) to fetch beginner-friendly issues.
        - Apply filters such as labels and programming languages.
        - Cache frequently accessed data (using Redis) to reduce external API calls.
    - **Key Endpoints:**
        - `GET /issues`: Retrieve all aggregated issues.
        - `GET /issues/{id}`: Retrieve a specific issue.
        - `POST /refresh`: Trigger a refresh (fetching updated issues from external APIs).
2. **User Management & Authentication Service**
    - **Responsibilities:**
        - Handle user registration, login, and profile management.
        - Manage authentication via JWT and OAuth (e.g., GitHub OAuth).
    - *(This module can evolve separately if you decide to add more user-centric functionality.)*
3. **Project & Issue Management**

    - **Responsibilities:**
        - Allow users to create or join open-source projects (e.g., Linux, Kubernetes, Apache).
        - Enable users to add issues they plan to work on within these projects, track progress, update status, and ultimately mark issues as completed once a PR/patch is submitted.
4. **GraphQL API Layer**
    - **Responsibilities:**
        - Provide a unified, typed interface for querying and mutating data.
        - Utilize FastAPI with Strawberry to expose both query resolvers (for reading issue data) and mutation resolvers (for refreshing issues and managing user-specific actions).
5. **Background Task Scheduler & Caching**
    - **Responsibilities:**
        - Use Celery to schedule periodic background jobs that refresh issue data.
        - Employ Redis for caching frequently accessed data, improving performance and reducing load on external APIs.
6. **Data Persistence**
    - **Responsibilities:**
        - Persist issue data, user information, and project details in a PostgreSQL database.
        - Use SQLAlchemy ORM to manage database interactions.

Technology Preferences & Constraints

- **Backend:**
    - **FastAPI** (Python 3.10) with **Strawberry** for GraphQL.
    - **Celery** for background tasks.
- **Database:**
    - **PostgreSQL** for persistent storage.
    - **SQLAlchemy** for ORM support.
- **Caching:**
    - **Redis** for caching frequently accessed data.
- **External Integrations:**
    - GitHub GraphQL API and GitLab API.
- **Deployment:**
    - Docker for containerization.
    - Kubernetes for orchestration (if scaling requires).
    - Cloud hosting via AWS, GCP, or similar.
- **Frontend (if applicable):**
    - **React** with **Apollo Client** (or Next.js for SSR/SSG) to consume the GraphQL API.

Communication Between Microservices

- **Synchronous Communication:**
  - Use RESTful APIs or GraphQL endpoints for direct, immediate interactions (e.g., a client querying the GraphQL API).
- **Asynchronous Communication:**
  - Utilize messaging queues like RabbitMQ or Kafka (in conjunction with Celery) for background tasks such as periodic issue refreshes and sending notifications.
- **API Gateway (Optional):**
  - Implement an API gateway (using NGINX, Kong, or a service mesh) to route client requests to the appropriate microservice, manage authentication, and handle logging and rate limiting.

Scaling & Reliability Requirements

- **High Availability:**
  - The system is designed to scale horizontally. Use Kubernetes to manage containers, deploy multiple replicas, and load balance incoming requests.
- **Low Latency:**
  - Implement Redis caching to ensure rapid responses to frequently requested queries.
- **Performance SLAs:**
  - Use auto-scaling and monitoring (with Prometheus and Grafana) to meet performance targets.
- **Resilience:**
  - Implement circuit breakers, retries, and fallback strategies to handle external API outages or service failures gracefully.

Regulatory & Security Considerations

- **Data Protection:**
  - Secure endpoints with HTTPS.
  - Encrypt sensitive data (e.g., tokens, user data) and store secrets in a secure manner (e.g., using cloud-managed secrets or Vault).
- **Authentication & Authorization:**
  - Use robust OAuth flows and JWT for stateless session management.
- **Compliance:**

    - Follow data privacy laws (like GDPR) by limiting personal data storage and providing users with control over their data.
- **API Rate Limiting:**
    - Protect against abuse and ensure fair usage of both your endpoints and third-party APIs.

Additional Questions to Consider

Monitoring & Logging

- **How will the system health and performance be monitored?**
    - Deploy Prometheus to collect metrics and Grafana to visualize performance.
    - Use a centralized logging solution like ELK/EFK (Elasticsearch, Logstash/Fluentd, Kibana) to aggregate and analyze logs.

Deployment & CI/CD

- **What does the deployment pipeline look like?**
    - Implement automated testing (unit, integration, E2E) and continuous integration using tools like GitHub Actions, Jenkins, or CircleCI.
    - Containerize your microservices with Docker and deploy them using Kubernetes to support seamless updates, rollback, and scaling.

Service Discovery & Configuration

- **How will microservices discover each other and manage configuration?**
    - Use a service registry (such as Consul) or a service mesh (like Istio) for service discovery.
    - Manage configuration centrally via environment variables, configuration files, or a dedicated configuration service.

Data Consistency & Synchronization

- **How will we ensure that data fetched from GitHub and GitLab stays up-to-date?**
    - Schedule periodic background refreshes using Celery tasks.
    - Employ caching strategies with Redis to quickly serve data while reducing load on external APIs.
    - Consider implementing eventual consistency and data reconciliation processes.

API Versioning & Extensibility

- **How will the future changes to the API be handled without breaking existing clients?**
    - Use semantic versioning (e.g., `/v1/`) in your API endpoints.
    - Maintain backward compatibility with clear documentation and versioned API schemas.
    - Adopt a modular architecture to allow easy updates and addition of new features.

Fault Tolerance & Recovery

- **What mechanisms will be in place for error handling and recovery in case of service failure?**
    - Implement robust error handling with retries and circuit breakers for external API calls.
    - Use comprehensive logging and alerting systems to quickly detect and recover from failures.
    - Plan for graceful degradation so that non-critical features can fail without affecting core functionality.

User Experience & Feedback

- **How will user feedback be gathered and incorporated for iterative improvement?**
    - Integrate user feedback forms or surveys directly into the application.
    - Monitor user behavior and engagement metrics to identify pain points.
    - Use agile development practices with regular sprint reviews and retrospectives to continuously improve the product.

1. Back-end

Micro services

Below is a comprehensive list of microservices

1. Issue Aggregation Service

Responsibilities:

Integrates with GitHub and GitLab APIs (via GraphQL/REST) to fetch beginner-friendly issues.
Applies filters such as labels (e.g., good first issue, help wanted) and programming languages.
Caches frequently accessed data to reduce external API calls.

Technology:

Framework: FastAPI
GraphQL: Ariadne or Graphene (if opting for GraphQL endpoints)
Task Scheduling: Celery (for periodic data refresh tasks)
Data Storage: PostgreSQL (for persistent issue data)
Caching: Redis (for frequently fetched data)

Key Endpoints:

GET /issues: Returns a list of aggregated issues.
GET /issues/{id}: Returns detailed information for a specific issue.
(Optional) POST /refresh: Triggers an immediate data refresh (secured endpoint).

Documentation Notes:

Document the API schema using OpenAPI/Swagger (automatically generated with FastAPI).
Include sample responses for success and error cases.
Detail caching strategy and refresh intervals.

2. User Management & Authentication Service

Responsibilities:

Handles user registration, login, and profile management.
Issues JWT tokens for session management.
Manages OAuth integration (e.g., GitHub OAuth) for social login.

Technology:

Framework: FastAPI
Authentication: FastAPI-JWT-Auth or PyJWT for JWT handling
OAuth Integration: Authlib or python-social-auth
Data Storage: PostgreSQL (for user profiles, credentials)
Caching (optional): Redis (for session storage or temporary token data)

Key Endpoints:

POST /register: Registers a new user.
POST /login: Authenticates a user and returns a JWT.
GET /profile: Retrieves user profile information (JWT-protected).
POST /logout: (Optional) Invalidates a user session.

Documentation Notes:

Provide flow diagrams for registration and OAuth login.
Include JWT token structure and security considerations.
Outline error responses for invalid credentials or unauthorized access.

3. Bookmarking & Progress Tracking Service

Responsibilities:

Enables users to bookmark issues for later reference.
Tracks the progress of issues (e.g., “To Do,” “In Progress,” “Completed”).
Associates user data with specific issues and progress statuses.

Technology:

Framework: FastAPI
Data Storage: PostgreSQL (with relational models linking users, issues, and progress)
Caching (optional): Redis (for quick retrieval of user bookmarks and statuses)

Key Endpoints:

GET /bookmarks: Retrieves all bookmarked issues for the logged-in user.
POST /bookmarks: Adds a new bookmark.
PUT /bookmarks/{id}: Updates the status of a bookmarked issue.
DELETE /bookmarks/{id}: Removes a bookmark.

Documentation Notes:

Define the data models (User, Issue, Bookmark/Progress) with entity relationships.
Provide example payloads for creating and updating bookmarks.
Detail any caching or indexing strategies used to optimize queries.

4. Filtering & Search Service

Responsibilities:

Provides fast search and filtering functionalities across issues.
Supports filtering by language, difficulty, project category, and other metadata.
Optionally leverages Elasticsearch for enhanced search capabilities.

Technology:

Framework: FastAPI
Search Engine: Elasticsearch or PostgreSQL full-text search
Caching: Redis (for caching common search queries and results)

Key Endpoints:

GET /search: Accepts query parameters (e.g., ?language=Python&label=good%20first%20issue) and returns filtered issue results.

Documentation Notes:

Document available query parameters and filtering options.
Provide examples of search queries and expected JSON responses.
Describe indexing strategies if using Elasticsearch.

5. Dashboard & Analytics Service

Responsibilities:

Aggregates user activity data, such as bookmarked issues and contribution progress.
Displays statistics and visualizations like trending repositories and personal progress.
Provides insights into overall platform activity.

Technology:

Framework: FastAPI
Data Storage: PostgreSQL (for persistent user analytics data)
Caching: Redis (for real-time data aggregation and quick responses)
Visualization: Data can be consumed by a React front-end using charting libraries (e.g., Chart.js, Recharts)

Key Endpoints:

GET /dashboard: Returns aggregated metrics for the authenticated user.
GET /analytics: Provides overall platform statistics (possibly segmented by time periods).

Documentation Notes:

Include sample dashboard data and visualization descriptions.
Detail how real-time data is aggregated and cached.
Document the metrics collected and their calculations.

6. Notification & Webhook Service

Responsibilities:

Listens for webhooks from GitHub/GitLab when issues are updated (e.g., closed, commented on).
Sends notifications (email, in-app, or push notifications) based on these events.
Processes background tasks for notification delivery.

Technology:

Framework: FastAPI
Task Queue: Celery (for asynchronous processing of notifications)
Notification Libraries: Flask-Mail equivalent in FastAPI (or external services like SendGrid)
Data Storage: PostgreSQL (for notification logs)
Caching: Redis (for managing rate-limiting or transient notification states)

Key Endpoints:

POST /webhooks: Receives webhook events from external services.
GET /notifications: Retrieves a list of notifications for the authenticated user.
(Optional) POST /notify: Endpoint to trigger manual notifications (secured)

Documentation Notes:

Document expected webhook payloads and security measures (e.g., signature validation).
Provide examples of notification payloads and delivery statuses.
Explain the retry mechanism for failed notifications and the role of the message queue.

7. API Gateway (Optional/Infrastructure Layer)

Responsibilities:

Acts as the single entry point for all client requests.
Routes requests to the appropriate microservice based on URL path or other criteria.
Handles cross-cutting concerns like authentication (JWT verification), rate limiting, and logging.

Technology:

Implementation Options:
Can be implemented as a dedicated FastAPI service or use a third-party solution (e.g., Kong, Nginx).
Deployment: Docker/Kubernetes managed.

Documentation Notes:

Detail the routing logic and how paths map to specific microservices.
Document any authentication, rate limiting, and logging configurations.
Explain how the API Gateway integrates with the CI/CD pipeline for seamless deployment.

Issue Aggregation Service

1. Issue Schema

Purpose:
Stores the details of each aggregated issue from GitHub or GitLab.

Fields:

id: Integer, Primary Key – Internal unique identifier.
external_id: String/Integer – The unique ID provided by GitHub/GitLab.
title: String – Issue title.
description: Text – Issue body or description.
state: Enum/String – Status of the issue (e.g., open, closed).
created_at: Timestamp – When the issue was created.
updated_at: Timestamp – Last update time.

url: String – Direct link to the issue on GitHub/GitLab.
source: Enum/String – Indicates the origin (e.g., github, gitlab).
repository_id: Foreign Key – References the associated repository.

2. Repository Schema

Purpose:
Captures information about the repository where an issue resides.

Fields:

id: Integer, Primary Key – Internal unique identifier.
external_id: String/Integer – Repository ID from GitHub/GitLab.
name: String – Repository name.
full_name: String – Full repository name (including owner, e.g., owner/repo).
description: Text – Brief description of the repository.
url: String – URL of the repository.
owner: String – Owner or organization name.
language: String – Primary programming language used.
created_at: Timestamp – Repository creation time.
updated_at: Timestamp – Last update time.

3. Label Schema

Purpose:
Represents the labels/tags (e.g., good first issue, help wanted) associated with issues.

Fields:

id: Integer, Primary Key – Internal unique identifier.
name: String – Label name.
color: String (Optional) – Color code associated with the label.
description: Text (Optional) – Additional details about the label.

4. Issue-Label Association Schema

Purpose:
Implements the many-to-many relationship between issues and labels.

Fields:

id: Integer, Primary Key – Unique identifier for the association.
issue_id: Foreign Key – References the Issue schema.
label_id: Foreign Key – References the Label schema.

Additional Considerations

Indexes:

Consider adding indexes on fields such as external_id, source, and foreign keys (repository_id) to improve query performance.

Caching Strategy (Redis):

Although not a database schema, define a strategy (e.g., keys like issues:cache) with expiration policies to cache frequently accessed data and reduce API calls.

Timestamps & Audit:

Optionally, include fields for audit purposes (e.g., created_by, updated_by) if you plan on tracking the source of updates within your microservice.

Allowing users to create their own projects and manage issues within those projects adds a valuable layer of engagement and personalization. Here are some suggestions for additional models and schema updates:

5. Project Schema

Purpose:
Define open-source projects (e.g., Linux, Kubernetes, Apache) that users can create or join.

Fields:

id: Unique identifier for the project.

name: The name of the project.
description: A brief description of the project.
external_id: (Optional) The ID of the project on external platforms (if applicable).
created_at: Timestamp when the project was created.
owner_id: (Optional) The user who created or owns the project.

6. UserIssue Schema (or ProjectIssue)

Purpose:
Manage user-specific issues within a project, tracking progress and work status.

Fields:

id: Unique identifier for the user issue.
issue: A reference to the aggregated issue (from the Issue schema) that the user is working on.
project_id: References the project to which the issue belongs.
status: The current state of the issue in the user's workflow. Options could include:
- in_progress
- completed
- dropped (if they decide to stop working on it)

pr_link: (Optional) A URL linking to the submitted pull request or patch.
created_at: Timestamp when the issue was added to the project.
updated_at: Timestamp for the last update.