Dodgy Geezers [FileFlow]

Daniel Ilyin <18327256>

Anthony Salib <20341603>

Synopsis:

Describe the system you intend to create:

FileFlow is a distributed file storage and management system that provides scalable, and fault-tolerant file storage and sharing services. It uses a microservices architecture to handle file operations across distributed storage nodes, as well as allowing for the sharing of files with other users.

What is the application domain?

FileFlow is a distributed file storage and sharing system focusing on providing a distributed, tolerant and scalable solution.

What will the application do?

FileFlow provides a secure, scalable, and efficient platform for managing files and their associated metadata. It enables users to upload, store, share, delete and retrieve files while offering flexible metadata querying capabilities using tags. With high availability and fault tolerance in mind, FileFlow uses a JWT-based authentication for secure user access and provides a TypeScript-based frontend for an intuitive user experience. It handles metadata management, such as file name, size, owner, and upload date, using GraphQL APIs backed by a PostgreSQL database. Built with SpringBoot microservices, the application uses Netflix Eureka for service discovery, allowing for dynamic scaling and communication between services.

Technology Stack

List of the main distribution technologies you will use

- ***Netflix Eureka**:*
  - Used for service discovery and registration of microservices
  - Allows for dynamic scaling and load balancing of the microservices
  - Provides service discovery, service registry and health monitoring

- ***Spring Boot Microservices**:*
  - Provides REST APIs for uploading and managing files, built using Spring Boot.
  - Allows us to create the REST APIs

- ***MinIO:***
  - Used for the storage of user uploaded files
  - Distributed across multiple nodes (2 nodes in our setup – but can very easily be horizontally scaled)
  - Each node has multiple drives for redundancy (fault tolerant)
  - Gives us high availability and data replication

- ***GraphGL:***
  - Used for communication between the File Management services and Metadata service
  - Allows for fetching and manipulating metadata using flexible and efficient queries.

- ***PostgreSQL:***
  - Our chosen database used for the storage of metadata of uploaded files
  - Has structured data querying for efficient metadata retrieval and management

- ***Docker:***
  - Packages each service (File Management, Metadata, Authentication) into independent containers.

- ***JWT Authentication***
  - Manages user authentication and security and protects endpoints and services by enforcing user authentication.

- ***Frontend (TypeScript)***
  - Provides the user interface for interacting with the system.
  - Communicates with the backend services via REST and GraphQL APIs.

Highlight why you used the chosen set of technologies and what was it about each technology that made you want to use it.

- ***Netflix Eureka***:
  - Facilitates service discovery and registration of microservices.
  - Its ability to enable seamless inter-service communication and monitor service health made it a perfect choice for managing the distributed microservices architecture.
- ***Spring Boot Microservices***:
  - Simplifies the creation of REST APIs for file management and backend services.
  - Its modularity, scalability, and ecosystem allowed us to quickly develop and integrate microservices like File Management, Metadata and Authentication services.
  - Java SpringBoot is used in enterprise applications too for its scalability and robustness.
- ***MinIO:***
  - Ensures efficient and fault-tolerant file storage.
  - Its distributed architecture, high availability, and compatibility with AWS S3 APIs made it an excellent choice for managing user-uploaded files in a scalable manner.
- ***GraphGL:***
  - Its ability to fetch specific data fields, perform complex queries, and reduce over-fetching of data suited the requirements for metadata management.
- ***PostgreSQL:***
  - PostgreSQL has reliable and powerful relational database capabilities for metadata storage.
  - Its support for complex queries, data integrity, and scalability fit perfectly with the need to store and retrieve metadata efficiently.
- ***Docker:***
  - Was used for its ability to containerize each microservice independently ensured portability, scalability, and ease of orchestration.
- ***JWT Authentication***
  - Its scalability and simplicity in securing APIs and user sessions made it the best choice for handling user authentication across services.
  - Ensures secure and stateless user authentication.

- ***Frontend (TypeScript)***
  - TypeScript's strong typing, combined with React, allowed for building a robust, maintainable, and user-friendly frontend for interacting with the system.

System Overview

Describe the main components of your system.

Netflix Eureka

Netflix Eureka acts as the service discovery mechanism in the system, allowing microservices to dynamically register and discover each other. This removes the need for hardcoded service URLs, facilitates load balancing, and ensures fault tolerance by monitoring service health and rerouting traffic as necessary.

Spring Boot Microservices

Spring Boot is used to build the backend microservices, including the File Management and Metadata services. Its lightweight framework and extensive ecosystem make it ideal for creating REST APIs and GraphQL endpoints, enabling modular development and seamless integration between components.

MinIO

MinIO provides a distributed, fault-tolerant storage solution for user-uploaded files. Its compatibility with S3 APIs, support for horizontal scaling, and built-in redundancy through data replication ensure high availability and reliability for file storage needs.

PostgreSQL

PostgreSQL serves as the database for storing metadata about uploaded files, such as file name, size, owner, and upload date. Its ability to handle complex queries and maintain data integrity makes it a robust choice for structured metadata management.

GraphQL

GraphQL is used for flexible metadata querying and updates between the File Management and Metadata services. It allows clients to fetch exactly the data they need, reducing over-fetching and under-fetching while supporting advanced queries like filtering and searching.

Spring Security with JWT

Spring Security with JWT provides secure user authentication and authorization. By issuing and validating JSON Web Tokens, it enables stateless, scalable authentication and ensures protected access to file operations and metadata management.

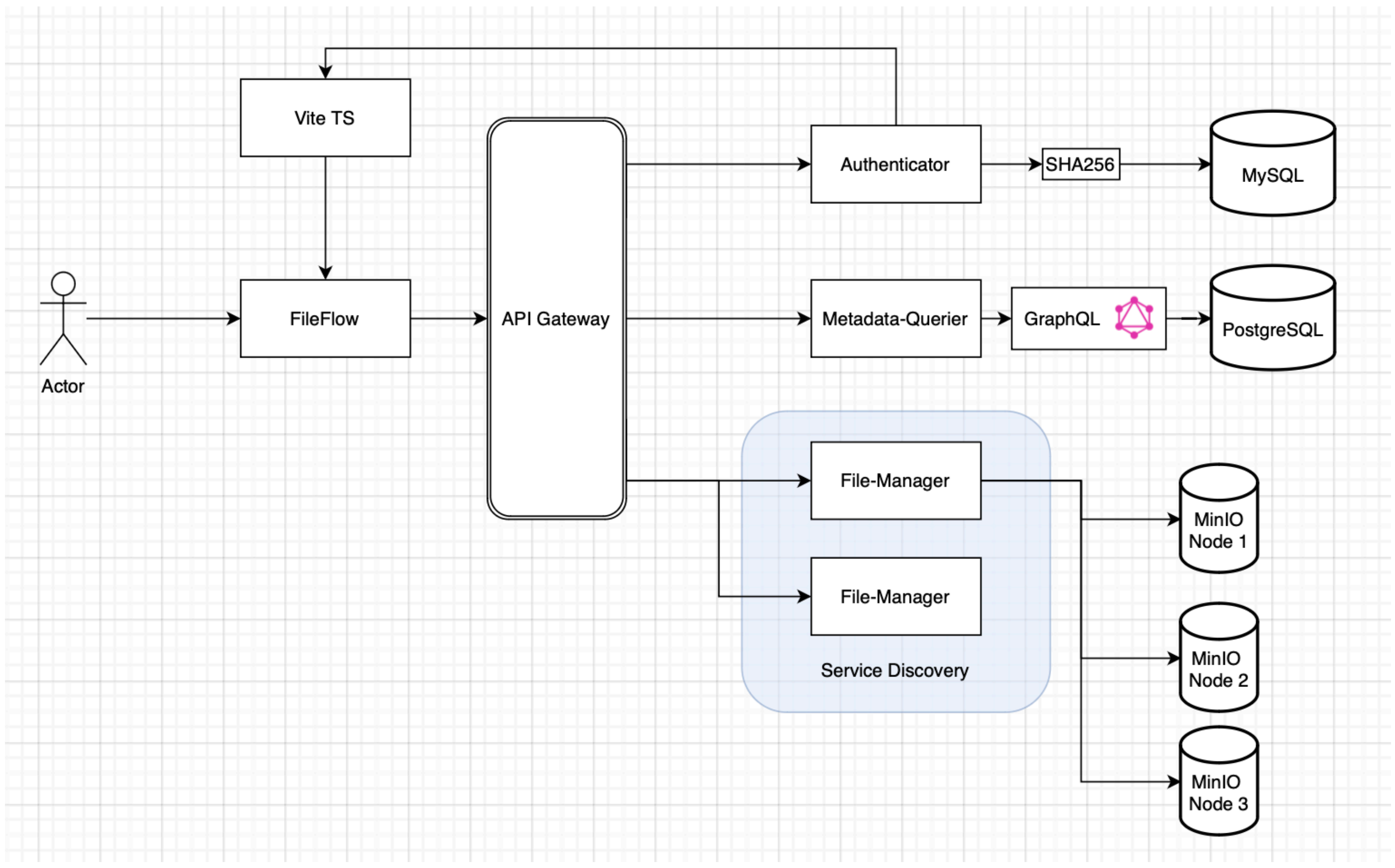
TypeScript Frontend

The frontend, built with TypeScript and React, provides an intuitive interface for interacting with the system. It offers users the ability to upload files, search metadata, and manage accounts through a modern, responsive, and type-safe design.

Docker

Docker containerizes each microservice, ensuring consistency across environments and simplifying deployment. It allows for independent scaling of services and seamless integration with orchestration tools like Kubernetes.

Include your system architecture diagram in this section.



Explain how your system works based on the diagram.

- The user initiates actions through FileFlow, our frontend interface built with Vite TS - a TypeScript-based framework we used.
- The API gateway handles user requests and communicates with the backend system.
    - It's the intermediary between the user and the core functionalities of the app.
    - It routes requests to the appropriate backend services.
    - It manages communication between the user-facing components and the backend subsystems.
- Requests requiring authentication are processed by the Authenticator service.
    - The SHA256 hashing algorithm is used to secure sensitive information like passwords.
    - User credentials and authentication details are stored in a MySQL database.
- Requests involving file metadata are handled by the metadata service, which interfaces with a PostgreSQL database.
    - The GraphQL API is used to query metadata efficiently, allowing flexible and complex querying and mutation capabilities.
- The file manager component handles file-related operations, including storage and retrieval.
    - It interacts with a distributed MinIO storage system consisting of two+ nodes:
- The system uses Netflix Eureka for service discovery to manage multiple File Manager instances dynamically, ensuring scalability and reliability.

Program Flow:

User interactions flow from FileFlow to the API Gateway, which routes them to:

- The the auth service for authentication,
- The metadata service for metadata-related queries, or
- The File Manager for file operations.

The MinIO nodes ensure distributed and fault-tolerant storage of files, while metadata is managed and queried separately via the PostgreSQL database.

Explain how your system is designed to support scalability and fault tolerance.

Horizontal scalability:

- Since we are using Netflix Eureka for service discovery it is very easy to horizontally scale. All we need to do is deploy more File services, and Eureka and our API gateway will load balance all the instances.
- We can also scale horizontally our MinIO nodes, if needed, increasing availability

Fault tolerance:

- If one of the File services goes down, Eureka takes care of automatic failover with multiple other services available
- We also have fault tolerance in our MinIO clusters with MinIO set up for data replication across the nodes

Load Distribution:

- API Gateway load balancing
- Distributed storage across MinIO nodes
- Multiple service instances

Contributions

Provide a sub section for each team member that describes their contribution to the project. Descriptions should be short and to the point.

Anthony	Daniel
Metadata Service *Authentication Service*	*File Management Service* *File Sharing Service*

Both of us worked on each part of the system but had particular focus on the services above.

Reflections

What were the key challenges you have faced in completing the project? How did you overcome them?

One issue that we had was that both of us have been unfamiliar with technologies like Vite TS or frontend development in general. We first divided the frontend tasks into smaller, manageable chunks. For example, we started by building simple UI components like the form and gradually moved to more complex features like API integration. We also used browser developer tools and debugging features to troubleshoot frontend issues efficiently - like in our metadata tags service.

What would you have done differently if you could start again?

If starting again, we would prioritize better orchestration, monitoring, and testing strategies to ensure smoother development and deployment - using CI/CD principles to automate deployments with tools like GitLab CI/CD and integrate Kubernetes for smoother transitions between development, staging, and production environments..

What have you learnt about the technologies you have used? Limitations? Benefits?

We learnt a lot about MinIO and its high availability and fault tolerance with data replication. For an open-source project it was quite scalable and easy to deploy in a distributed setup. A drawback was that it requires more manual setup and management, compared to paid services like AWS S3 buckets.

We also gained good experience in frontend development. Strong typing in TS reduces runtime errors and improves maintainability of the code compared to vanilla JS. It was also good experience to use modern tooling and ecosystem. The initial setup and learning curve can be steep compared to plain JavaScript but we think it was worth it.

Overall, this project provided us with great insights into building distributed systems, understanding the trade-offs of each technology, and integrating them effectively. If starting again, we would prioritize better orchestration, monitoring, and testing for smoother development and deployment.