**Project Overview**

The project is a Spring Boot file upload and download service. It allows users to:

1. Upload files securely using an API key (Celonis-Auth).
2. Download uploaded files by specifying the filename.
3. Persist uploaded files in a directory (/app/uploads).

The application runs locally, in a Docker container, and can be deployed to Kubernetes for scalability and reliability.

**Components and Their Roles**

**1. Codebase**

**FileController.java**

* **Purpose**: Handles HTTP requests for file upload and download.
* **Endpoints**:
  1. POST /files: Uploads a file.
  2. GET /files/{filename}: Downloads a file.
  3. GET /: Displays a welcome message.
* **Key Features**:
  1. Validates the Celonis-Auth header for authentication.
  2. Saves files to a local directory (storagePath).
  3. Responds with appropriate HTTP statuses (200 for success, 401 for unauthorized, 500 for server errors).

**FileServiceApplication.java**

* **Purpose**: Entry point of the Spring Boot application.
* **Key Features**:
  1. Uses the **@SpringBootApplication** annotation to enable Spring Boot configuration and auto-scanning.
  2. Starts the embedded Tomcat server on a specified port, default 8080

**2. Gradle**

* Gradle is a build automation tool used to compile, test, and package Java applications. It's highly customizable and widely used in modern Java projects.
* **How We Used Gradle:**
  1. **Dependencies**: Defined dependencies in the build.gradle.kts file
  2. **Tasks**:
     1. bootRun: Runs the application locally.
     2. bootJar: Packages the application as a .jar file.
  3. **Java Toolchain**: Ensures compatibility by specifying Java version 17.

**Installation and Use**

1. **Installed via Wrapper**: Used gradlew (Gradle wrapper), which eliminates the need for a pre-installed Gradle version.
2. **Commands**:
   * ./gradlew bootRun: Run the application locally.
   * ./gradlew bootJar: Build a .jar file.

**3. Docker**

**Dockerfile**

* **Purpose**: Defines the steps to containerize the application.
* **Stages**:
  1. **Builder Stage**:
     1. Uses eclipse-temurin:17-jdk-alpine as the base image for building the .jar.
     2. Executes gradlew bootJar to generate the .jar file.
  2. **Runtime Stage**:
     1. Uses eclipse-temurin:17-jre-alpine for a lightweight runtime environment.
     2. Copies the .jar file from the builder stage.
     3. Specifies ENTRYPOINT to run the application.

**Commands**

* docker build -t file-service:1.0 .: Builds the Docker image.
* docker run -p 8080:8080 --name file-service-container file-service:1.0: Runs the container.
* docker ps: Lists running containers.
* docker logs file-service-container: Checks logs for debugging.

**4. Kubernetes**

**Manifests**

**1. Deployment**

* **Purpose**: Ensures the application is running with the desired state.
* **Key Configurations**:
  1. **Replicas**: Sets up multiple replicas for high availability.
  2. **Image**: Specifies the Docker image (file-service:1.0).
  3. **Ports**: Maps container port 9090 to the service.

**2. Service**

* **Purpose**: Exposes the application internally within the cluster.
* **Type**: ClusterIP.
* **Ports**:
  1. targetPort: 8080: Port used by the container.
  2. port: 8080: Exposed port for internal communication.

**3. Ingress**

* **Purpose**: Exposes the application externally to the internet.
* **Host**: Routes requests to http://<your-domain>/files to the service.

**Commands**

* kubectl apply -f deployment.yaml: Deploys the application.
* kubectl get pods: Checks the status of pods.
* kubectl logs <pod-name>: Checks logs of a specific pod.

**How the things are working together:**

**Flow of Operations**

* **User Uploads a File**:
  1. Sends a POST request with the file to http://<host>:8080/files.
  2. API validates the Celonis-Auth header.
  3. File is saved to /app/uploads (in container or local).
* **User Downloads a File**:
  1. Sends a GET request to http://<host>:8080/files/<filename>.
  2. File is fetched from the directory and returned.

**Local vs. Docker vs. Kubernetes**

* **Local**: Runs with gradlew bootRun or java -jar.
* **Docker**: Runs in an isolated environment with docker run.
* **Kubernetes**: Orchestrates multiple replicas for scalability.

**Key Advantages**

1. **Security**: API key validation with Celonis-Auth.
2. **Scalability**: Deployable on Kubernetes with auto-scaling and high availability.
3. **Portability**: Dockerized for consistent behavior across environments.
4. **Persistence**: Files are stored in a persistent directory.

**Potential Improvements**

* **Database Integration**:
  1. Store metadata (e.g., upload timestamps) in a database like MySQL or PostgreSQL.
* **Logging and Monitoring**:
  1. Integrate tools like ELK stack or Prometheus for enhanced observability.
* **Error Handling**:
  1. Add descriptive error messages and retry mechanisms.
* **Cloud Deployment**:
  1. Deploy to AWS, Azure, or GCP for global availability.

**What happened when we run the Gradle:**

When you run Gradle commands, it generates a variety of files and directories based on the build tasks and project configuration. Here’s a breakdown of the files and directories typically created or modified when you build the project using Gradle:

**1. Gradle Wrapper Files**

The following files are used by the Gradle wrapper to ensure consistent build environments across systems:

* gradlew and gradlew.bat: The Gradle wrapper scripts for Unix-based systems and Windows, respectively.
* gradle/wrapper/gradle-wrapper.properties: Specifies the Gradle version and distribution URL.
* gradle/wrapper/gradle-wrapper.jar: The binary for the Gradle wrapper.

**2. Output Files and Directories**

When running ./gradlew build or ./gradlew bootJar, Gradle creates the following key output directories and files:

**a. build/ Directory**

This is the main directory where Gradle outputs the results of its build process:

* **build/classes/**:
  + Contains the compiled .class files for your Java code.
  + Subdirectories:
    - java/main: Compiled main application classes.
    - java/test: Compiled test classes.
* **build/libs/**:
  + Contains the generated .jar file, e.g., file-service-server-1.0.0.jar.
* **build/reports/**:
  + Contains reports for tasks like testing, linting, and build problems.
  + Example: build/reports/tests/ for test reports in HTML format.
* **build/tmp/**:
  + Temporary files used during the build process.
* **build/resources/**:
  + Copies resources from src/main/resources/ into build/resources/main/ for use in the packaged application.

**3. Dependency Cache**

Gradle downloads all project dependencies into a centralized cache:

* **Location**:
  + Default on Linux/Mac: ~/.gradle/caches/
  + Default on Windows: C:\Users\<username>\.gradle\caches\
* **Contents**:
  + Libraries downloaded from Maven Central or other repositories.
  + Metadata and temporary files for dependency resolution.

**4. Test Reports**

When tests are executed:

* **Location**: build/reports/tests/
* **Contents**:
  + HTML files summarizing test results.
  + Detailed information about passed, failed, or skipped tests.

**5. Logs and Debug Information**

Gradle may generate log files during builds:

* **build/logs/**:
  + Build logs, useful for debugging.