**Exploration of our Open Source project (~30 minutes)**

Gravitee is largely open-source software, which can be found on GitHub here:<https://github.com/gravitee-io>

Gravitee is composed of several repositories (~250); many are plugins, while others are full projects, such as **"Gravitee.io - OpenSource API Management"** and **"Gravitee.io - Access Management - OpenSource IAM Solution"**, which are among Gravitee's flagship products.

Explore the **API Management Open Source** project and answer the questions below. We want to evaluate your ability to explore an existing project, understand the tools and technologies used, as well as your basic development knowledge. Provide brief answers, as we will have the opportunity to discuss them in more detail later.

### **1. Technology stack**

**Q1.** What are the main components or modules you have identified in the Gravitee.io project?  
After browsing the repositories, identify the different repositories, modules, or main sub-projects. 4 or 5 examples are sufficient.

Answer:

1. **gravitee-apim-gateway:**

This is the core runtime engine, the actual API Gateway proxy. It's responsible for receiving client requests, applying policies (like security, rate limiting, transformation), routing requests to backend services, and collecting metrics/logs. It's built using Vert.x for high performance and asynchronous processing. This module contains sub-modules for the reactor, policy execution, connectors, synchronization services, etc.

1. **gravitee-apim-rest-api:**

This module contains the backend REST API used primarily by the Management Console and Developer Portal UIs. It provides endpoints for configuring APIs, managing applications, users, plans, subscriptions, platform settings, viewing analytics, etc. It's typically built using the Spring Framework (Spring Boot, Spring MVC/JAX-RS, Spring Security) and interacts with the repository layer for persistence.

1. **gravitee-apim-repository:**

This module defines the abstraction layer for data persistence and provides concrete implementations. It contains the gravitee-apim-repository-api submodule defining the interfaces (e.g., ApiRepository, ApplicationRepository). It also contains implementation submodules like gravitee-apim-repository-mongodb and gravitee-apim-repository-jdbc, allowing Gravitee to store its configuration and state in different types of databases.

1. **gravitee-apim-console-webui:**

This is the source code for the Management Console web interface. It's an Angular (TypeScript) single-page application (SPA) used by API publishers and administrators to manage the API lifecycle, configure policies, monitor traffic, manage users, and set up platform configurations. It interacts with the gravitee-apim-rest-api module for its data and operations.

1. **gravitee-apim-portal-webui:**

This is the source code for the Developer Portal web interface. Also an Angular (TypeScript) SPA, this is the frontend for API consumers. Developers use it to discover available APIs, read documentation, register their applications, subscribe to API plans (obtaining API keys or credentials), and view their application's consumption analytics. It also interacts with specific endpoints exposed by the gravitee-apim-rest-api module.

**Q2.** What are the main technologies or frameworks used in the API Management project, both on the back-end and front-end?  
Examine the source code and configuration files to identify the tools/frameworks used.

Answer:

**I. Back-End Technologies**

1. **Java:**
   * **Role:** The primary programming language for all backend components (Gateway, Management API, Repository implementations).
   * **Evidence:** Ubiquitous .java files, Maven build structure targeting JVM.
2. **Spring Framework (specifically Spring Boot):**
   * **Role:** Forms the core application framework for the **Management API** (gravitee-apim-rest-api). Used extensively for:
     + Dependency Injection (@Autowired, @Component, @Service, @Repository).
     + Web framework (likely **Spring MVC** given the @RestController, @RequestMapping annotations seen in resource classes) for building the REST APIs.
     + Configuration Management (@Configuration, @Value, loading application.yml).
     + Security (**Spring Security**) for authentication (JWT handling, integration with Identity Providers) and authorization (role-based access control).
     + Transaction Management (@Transactional) in service layers.
     + Data Access Abstraction (used by JDBC repository).
   * **Evidence:** pom.xml files heavily depend on org.springframework.boot artifacts (spring-boot-starter-web, spring-boot-starter-security, etc.). Presence of typical Spring annotations throughout the rest-api, service, and repository modules.
3. **Vert.x:**
   * **Role:** The core framework for the high-performance **API Gateway** (gravitee-apim-gateway). Chosen for its non-blocking, event-driven architecture suitable for I/O intensive tasks like proxying API requests. Handles:
     + HTTP Server/Client operations.
     + Request routing.
     + Asynchronous execution of the policy chain.
     + Managing concurrency efficiently with an event loop model.
   * **Evidence:** pom.xml in gravitee-apim-gateway modules depends heavily on io.vertx artifacts (vertx-core, vertx-web, vertx-rx-java3). Code within the gateway often uses Vert.x APIs (io.vertx.core.\*, io.vertx.ext.web.\*) and reactive patterns.
4. **RxJava (likely RxJava 3):**
   * **Role:** Used often alongside Vert.x, particularly within the **API Gateway** and its plugins (policies, connectors). Facilitates composing asynchronous operations and handling event streams inherent in the reactive nature of the gateway.
   * **Evidence:** Dependencies on io.reactivex.rxjava3 artifacts in gateway modules. Usage of reactive types like Single, Completable, Maybe, Flowable in method signatures within policy and connector implementations.
5. **Jackson:**
   * **Role:** The standard library used throughout the backend for handling JSON serialization (Java objects to JSON) and deserialization (JSON to Java objects). Crucial for REST APIs (request/response bodies) and parsing configuration (like API definitions).
   * **Evidence:** Dependencies on com.fasterxml.jackson.core artifacts. Ubiquitous use of @JsonProperty, @JsonIgnore, @JsonInclude annotations on model classes (especially within gravitee-apim-definition).
6. **JAX-RS (potentially as API standard):**
   * **Role:** While Spring MVC seems dominant for implementation in the Management API, JAX-RS (javax.ws.rs-api or jakarta.ws.rs-api) might be present as an API dependency, providing standard annotations (@Path, @GET, @POST, @Produces, @Consumes). Spring MVC can often interpret these.
   * **Evidence:** Check pom.xml for explicit JAX-RS API dependencies, although the implementation provider is typically Spring or potentially Jersey/CXF if not using Spring Boot's default web starter fully.

**II. Front-End Technologies**

1. **Angular:**
   * **Role:** The JavaScript framework used for building both the **Management Console UI** (gravitee-apim-console-webui) and the **Developer Portal UI** (gravitee-apim-portal-webui). Provides structure, component model, routing, dependency injection, and tools for building modern Single Page Applications (SPAs).
   * **Evidence:** Presence of angular.json, tsconfig.json, package.json with @angular/\* dependencies (e.g., @angular/core, @angular/common, @angular/router, @angular/common/http). Directory structure typical of Angular CLI projects (src/app, src/environments, .ts files).
2. **TypeScript:**
   * **Role:** The primary language used for developing the Angular frontends. Provides static typing, classes, interfaces, and other features on top of JavaScript.
   * **Evidence:** Predominance of .ts files in the console-webui and portal-webui modules. tsconfig.json files defining TypeScript compiler options.
3. **HTML & CSS/SCSS:**
   * **Role:** Standard web technologies used for structuring the UI content (HTML templates associated with Angular components) and styling the UIs (CSS or more commonly SCSS, a CSS preprocessor allowing variables, nesting, mixins).
   * **Evidence:** .html files for component templates, .scss or .css files for styling within the Angular projects. angular.json often specifies SCSS as the stylesheet format.
4. **Angular Material (Highly Likely):**
   * **Role:** A widely used UI component library providing pre-built, themed, and accessible components (buttons, forms, tables, dialogs, etc.) based on Google's Material Design specification. Accelerates UI development.
   * **Evidence:** Check package.json in the UI modules for dependencies on @angular/material. Imports from @angular/material/\* within TypeScript component/module files.
5. **Node.js / npm / yarn:**
   * **Role:** Used for the frontend development toolchain. Node.js runs the build tools and development server. npm or yarn manage frontend library dependencies (defined in package.json) and execute scripts defined there (e.g., for building, linting, testing, serving).
   * **Evidence:** Presence of package.json and potentially package-lock.json or yarn.lock files. Scripts within package.json invoking ng (Angular CLI commands).

**III. Persistence & Data Stores**

1. **MongoDB Driver:**
   * **Role:** Used by the gravitee-apim-repository-mongodb module to interact with a MongoDB database for storing API configurations, user data, application details, etc.
   * **Evidence:** Dependencies on org.mongodb:mongodb-driver-sync or mongodb-driver-reactivestreams. Code within the MongoDB repository implementation uses MongoDB Java client APIs.
2. **JDBC & JPA/Hibernate:**
   * **Role:** Used by the gravitee-apim-repository-jdbc module to interact with relational databases (like PostgreSQL, MySQL). JDBC is the standard Java API. Spring Data JPA (using Hibernate as the default provider) is typically used for ORM (Object-Relational Mapping).
   * **Evidence:** Dependencies on spring-boot-starter-data-jpa, org.hibernate:hibernate-core, specific JDBC drivers (org.postgresql:postgresql). Java Persistence API (javax.persistence.\* or jakarta.persistence.\*) annotations like @Entity, @Table, @Id on model classes used by the JDBC repository.
3. **Elasticsearch Client:**
   * **Role:** Used for storing and querying analytics, metrics, and logs. Can be accessed via a specific repository module (gravitee-apim-repository-elasticsearch) or reporter plugins sending data.
   * **Evidence:** Dependencies on Elasticsearch client libraries (org.elasticsearch.client:elasticsearch-rest-high-level-client or the newer Java client). Code performing indexing and search operations against Elasticsearch.

**IV. Build & Dependency Management**

1. **Maven:**
   * **Role:** The primary build automation tool and dependency manager for the entire Java-based backend. Defines module structure, dependencies, build lifecycle (compile, test, package), and plugins.
   * **Evidence:** pom.xml files in the root directory and in almost every Java module. The build process is executed using Maven commands (mvn clean install).

**Q3.** What dependency managers are used in this project? How are they configured and managed?  
~~Look for files such as package.json, pom.xml, build.gradle, etc.~~

Answer:

1. **Maven (for the Java Backend)**
   * **What it manages:** Dependencies for all Java-based modules, including the API Gateway (gravitee-apim-gateway), REST API (gravitee-apim-rest-api), Repository implementations (gravitee-apim-repository-\*), plugins (gravitee-apim-plugin), core definitions (gravitee-apim-definition), and backend services. It also manages the build lifecycle (compiling, testing, packaging) for these modules.
   * **Configuration File:** pom.xml. There is a root pom.xml file at the top level of the repository, and each individual Java module (e.g., gravitee-apim-gateway/pom.xml, gravitee-apim-rest-api/management/pom.xml) has its own pom.xml.
   * **How it's Managed and Configured:**
     + **Multi-Module Project:** The root pom.xml defines the overall project structure, listing all the submodules in the <modules> section. This allows building the entire project or specific modules with single Maven commands.
     + **Parent POM:** The project likely uses a parent POM (<parent> tag in module pom.xml files), potentially gravitee-io-parent or inheriting from frameworks like spring-boot-starter-parent. Parents define common configurations, plugin versions, and potentially dependency versions inherited by child modules.
     + **Dependency Declaration:** Specific libraries required by a module are listed within the <dependencies> section of that module's pom.xml.
     + **Centralized Version Management (<dependencyManagement>):** Crucially, the root pom.xml (or a parent POM it references) likely contains a <dependencyManagement> section. This section *declares* dependencies and their *versions* but doesn't add them to modules directly. Child modules then list the dependencies they need in their own <dependencies> section *without* specifying the version, thereby inheriting the centrally managed version. This ensures version consistency across all modules and makes version updates much easier (only need to change it in one place).
     + **Properties (<properties>):** Often, versions for dependencies and plugins are defined in the <properties> section of the root (or parent) POM (e.g., <spring.version>5.3.x</spring.version>) and referenced elsewhere using ${property.name} syntax (e.g., <version>${spring.version}</version>). This further centralizes version control.
     + **Scopes:** Dependencies are assigned scopes (e.g., compile, test, provided, runtime) to control when they are included in the classpath and packaging.
     + **Plugin Management (<pluginManagement>):** Similar to <dependencyManagement>, this section in the root/parent POM centrally defines the versions and configurations for Maven plugins used during the build.
2. **npm (Node Package Manager) or Yarn (for the JavaScript/TypeScript Frontend)**
   * **What it manages:** Dependencies for the Angular-based user interfaces: the Management Console (gravitee-apim-console-webui) and the Developer Portal (gravitee-apim-portal-webui). This includes Angular framework libraries (@angular/\*), UI component libraries (like Angular Material), state management libraries, testing tools, linters, and other JavaScript/TypeScript utilities. It also manages scripts for building, testing, and serving the frontends.
   * **Configuration Files:**
     + package.json: The core configuration file located at the root of each frontend project (console-webui, portal-webui). It lists project metadata, dependencies, development dependencies, and scripts.
     + package-lock.json (if primarily using npm) or yarn.lock (if primarily using Yarn): These are auto-generated lock files.
   * **How it's Managed and Configured:**
     + **package.json Structure:**
       - dependencies: Lists packages required for the application to run in production.
       - devDependencies: Lists packages needed only for development and build processes (e.g., Angular CLI, testing libraries, build tools, linters).
       - scripts: Defines command-line shortcuts executed via npm run <script-name> or yarn <script-name> (e.g., start, build, test, lint).
     + **Semantic Versioning (SemVer):** Versions in package.json often use semantic versioning ranges (e.g., ^14.2.0, ~1.0.0). The caret (^) allows updates to minor and patch versions, while the tilde (~) typically allows only patch version updates. This provides flexibility while aiming for compatibility.
     + **Lock Files (package-lock.json / yarn.lock):** These files are *critical*. When dependencies are installed (npm install or yarn install), the exact versions of *all* installed packages (including transitive dependencies) are recorded in the lock file. Subsequent installs on other machines or in CI/CD environments will use the lock file to install the *exact same versions*, ensuring deterministic and reproducible builds across different environments and times. This prevents the "works on my machine" problem caused by varying dependency versions.
     + **Installation:** Developers run npm install or yarn install within the frontend project directories to download and install the dependencies listed in package.json according to the versions specified (or resolved via the lock file).

**Q4.** How do you think this architecture is managed in the project, particularly regarding the management of different modules and sub-projects?

1. Answer:  
   **Maven Multi-Module Project Structure:**
   * **Central Orchestration:** The entire backend and the overall project structure are managed as a Maven multi-module project. There's a root pom.xml file at the top level.
   * **Module Declaration (<modules>):** This root pom.xml declares all the major sub-projects/modules (like gravitee-apim-gateway, gravitee-apim-rest-api, gravitee-apim-repository, gravitee-apim-plugin-api, etc.) within its <modules> section.
   * **Hierarchical Structure:** Maven understands this hierarchy. Building from the root (mvn clean install) typically builds all declared modules in the correct order based on their inter-dependencies. Sub-modules can also have further sub-modules (e.g., gravitee-apim-rest-api might contain gravitee-apim-rest-api-management, gravitee-apim-rest-api-portal, gravitee-apim-rest-api-service).
2. **Parent POM and Centralized Dependency Management:**
   * **Inheritance (<parent>):** Most modules likely declare a parent POM (often the project's root POM, or a dedicated gravitee-io-parent POM). This parent POM is crucial for management.
   * **Dependency Management (<dependencyManagement>):** The parent POM almost certainly contains a <dependencyManagement> section. This section centralizes the *versions* of all external third-party dependencies (Spring, Vert.x, Jackson, Logback, etc.) used across the entire project. Individual modules then declare their dependencies *without* specifying a version, inheriting the version defined in the parent's <dependencyManagement>. This ensures version consistency and simplifies updates.
   * **Plugin Management (<pluginManagement>):** Similarly, the parent POM typically manages the versions and common configurations of Maven plugins used during the build process (compiler plugin, surefire for tests, assembly plugin, etc.).
   * **Properties (<properties>):** Versions and other common configurations are often defined as properties (e.g., <java.version>11</java.version>, <spring.boot.version>...</spring.boot.version>) in the parent POM for easy reference and updates.
3. **Inter-Module Dependencies:**
   * **Maven Dependencies:** When one module needs code from another module within the same project (e.g., the gravitee-apim-rest-api needs the service interfaces and models from gravitee-apim-rest-api-service or the repository interfaces from gravitee-apim-repository-api), it declares a standard Maven dependency on that module in its pom.xml file's <dependencies> section. Maven resolves these internal dependencies during the build process, ensuring modules are built in the correct topological order.
4. **Separation of Concerns & Clear Boundaries:**
   * **API Modules:** Critical interfaces are defined in dedicated API modules (e.g., gravitee-apim-repository-api, gravitee-apim-plugin-api). Other modules depend only on these *interface* modules, not specific implementations (following the Dependency Inversion Principle). For example, the gravitee-apim-rest-api depends on gravitee-apim-repository-api, but not directly on gravitee-apim-repository-mongodb or gravitee-apim-repository-jdbc. The actual implementation used is determined at runtime via configuration (e.g., Spring profiles).
   * **Frontend/Backend Separation:** The frontend Angular projects (console-webui, portal-webui) are distinct modules managed by their own tooling (npm/yarn and package.json). While Maven *might* trigger their build process (using plugins like frontend-maven-plugin), their dependency management and build lifecycle are largely separate from the Java backend modules.
5. **Configuration Management:**
   * While Maven manages the build-time structure, runtime configuration (e.g., database connection strings, Management API URL for the Gateway, enabled plugins) is managed externally, typically through Spring Boot's mechanisms (application.yml/.properties files, environment variables, system properties). This configuration determines which repository implementation is active or how components discover each other.
6. **Build Profiles (Potentially):**
   * Maven profiles (<profiles> in pom.xml) might be used to customize builds for different environments or configurations (e.g., skipping certain tests, including/excluding specific modules, activating specific repository implementations during integration tests).

In essence, Gravitee APIM leverages **Maven's robust multi-module and dependency management capabilities** as the backbone for organizing and building the Java parts of the project. Key management strategies include:

* A **hierarchical module structure** defined in the root POM.
* A **parent POM** enforcing consistency via centralized dependency and plugin management.
* Clear **API modules** defining contracts between components, promoting loose coupling.
* Standard Maven **inter-module dependencies** to link components together at build time.
* Separation of build/dependency management for **backend (Maven) vs. frontend (npm/yarn)**.

### **2. Quality Tools & Work Methodologies**

**Q5.** What tools or processes do you identify as being used to ensure code quality in this project (e.g., tests, continuous integration, linters, etc.)?  
Briefly describe the files or sections of the project that lead you to these conclusions. *Tip: Explore configuration files like pom.xml, build.gradle, or CI/CD pipelines.*

*Ans:* **1. Testing Frameworks**

* **JUnit Tests**: Extensive unit tests throughout the codebase (found numerous \*Test.java files)
* **Integration Tests**: Dedicated module **gravitee-apim-integration-tests** for integration testing
* **E2E Tests**: Comprehensive end-to-end testing module **gravitee-apim-e2e** with Cypress UI tests
* **Performance Tests**: Module **gravitee-apim-perf** for performance testing

**2. Static Code Analysis**

* **SonarCloud**: Configuration in **sonar-project.properties** files across modules for continuous code quality analysis
* **Code Coverage**: JaCoCo configuration for test coverage reporting (**sonar.coverage.jacoco.xmlReportPaths**)

**3. Linting & Formatting**

* **Prettier**: Uses **prettier-maven-plugin** for consistent code formatting
* **ESLint**: Configuration in **.eslintrc.js** for JavaScript/TypeScript linting in web UI components
* **License Checking**: **license-maven-plugin** to ensure proper license headers in source files

**4. Continuous Integration**

* **CircleCI**: Comprehensive CI/CD pipeline defined in **.circleci/config.yml**
* **Multiple Workflows**: Different workflows for PRs, merges to master, and releases as documented in **.circleci/Workflows.md**
* **Automated Testing**: Automated test runs for backend, UI components, and E2E tests

**5. UI Component Testing**

* **Storybook**: For testing UI components in isolation
* **Chromatic**: For visual regression testing of UI components

**6. Architecture Validation**

* **ArchUnit**: Referenced in dependencies (**archunit-junit5.version**) for enforcing architectural rules

**7. Documentation Quality**

* **Swagger/OpenAPI**: Tools for API documentation and validation

**8. Security Scanning**

* **Snyk**: Referenced in README badge for vulnerability scanning

**Q6.** How does the project manage testing? What types of tests or testing frameworks do you identify in the project, both on the back-end and front-end? *You can refer to test files or configuration and share examples.*

*Ans:*

**Back-end Testing**

**1. JUnit 5 Testing Framework**

The back-end uses JUnit 5 (Jupiter) for unit and integration testing, as seen in the test files:

java

CopyInsert

// From HttpRequestKeylessV4IntegrationTest.java

import org.junit.jupiter.api.DisplayName;

import org.junit.jupiter.api.DisplayNameGeneration;

import org.junit.jupiter.api.DisplayNameGenerator;

import org.junit.jupiter.api.Test;

@GatewayTest

@DeployOrganization(organization = "/organizations/organization-add-header.json", apis = { "/apis/v4/http/api.json" })

@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)

class HttpRequestKeylessV4IntegrationTest extends AbstractGatewayTest {

// Test methods

}

**2. Custom Testing SDK**

The project has developed a custom testing SDK for gateway testing:

java

CopyInsert

import io.gravitee.apim.gateway.tests.sdk.AbstractGatewayTest;

import io.gravitee.apim.gateway.tests.sdk.annotations.DeployApi;

import io.gravitee.apim.gateway.tests.sdk.annotations.DeployOrganization;

import io.gravitee.apim.gateway.tests.sdk.annotations.GatewayTest;

This SDK provides annotations like **@GatewayTest**, **@DeployApi**, and **@DeployOrganization** to simplify test setup.

**3. AssertJ for Assertions**

The project uses AssertJ for fluent assertions:

import static org.assertj.core.api.Assertions.assertThat;

// In test methods

assertThat(response.statusCode()).isEqualTo(200);

assertThat(body).hasToString("response from backend");

**4. WireMock for HTTP Mocking**

WireMock is used to mock HTTP endpoints:

import static com.github.tomakehurst.wiremock.client.WireMock.\*;

// In test methods

wiremock.stubFor(get("/endpoint").willReturn(ok("response from backend")));

wiremock.verify(

1,

getRequestedFor(urlPathEqualTo("/endpoint")).withHeader(AddHeaderPolicy.HEADER\_NAME, equalTo(AddHeaderPolicy.REQUEST\_HEADER))

);

**5. RxJava Testing**

The project uses RxJava's testing capabilities for asynchronous code:

httpClient

.rxRequest(HttpMethod.GET, "/test")

.flatMap(HttpClientRequest::rxSend)

.flatMapPublisher(response -> {

// assertions

return response.toFlowable();

})

.test()

.await()

.assertComplete()

.assertValue(body -> {

// assertions

return true;

})

.assertNoErrors();

**6. Integration Testing Structure**

The project has a dedicated **gravitee-apim-integration-tests** module with tests organized by feature:

* HTTP tests (cors, failover, logging, etc.)
* gRPC tests
* WebSocket tests
* TCP tests
* and more

**Front-end Testing**

**1. Cypress for E2E Testing**

The front-end uses Cypress for end-to-end testing:

// From cypress.config.ts

import { defineConfig } from 'cypress';

export default defineConfig({

env: {

managementApi: 'http://localhost:8083',

wiremockUrl: 'http://wiremock:8080',

gatewayServer: 'http://localhost:8082',

},

e2e: {

baseUrl: 'http://localhost:4000',

watchForFileChanges: false,

projectId: 'ui-test',

specPattern: 'ui-test/integration/\*\*/\*.spec.ts',

// other configuration

},

});

**2. TypeScript for Test Writing**

Tests are written in TypeScript, providing type safety:

// From ui-api-creation-workflow.spec.ts

describe('API creation workflow', () => {

const apiVersion = `${faker.number.int({ min: 1, max: 9 })}.${faker.number.int({ min: 1, max: 9 })}`;

const apiPath = `${faker.commerce.productName().replace(/ /g, '\_')}`;

const apiName = faker.commerce.productName();

let apiId: string;

describe('Create a V4 Proxy REST API', function () {

// Test methods

});

});

**3. Faker.js for Test Data Generation**

The project uses Faker.js to generate test data:

import { faker } from '@faker-js/faker';

const apiName = faker.commerce.productName();

**4. Custom Cypress Commands**

The project extends Cypress with custom commands:

// Usage in tests

cy.loginInAPIM(ADMIN\_USER.username, ADMIN\_USER.password);

cy.getByDataTestId('api\_create\_v4\_button');

cy.teardownV4Api(apiId);

**5. UI Testing Structure**

UI tests are organized by feature:

* API management (creation, events, info, etc.)
* Gateway management
* Settings
* Login functionality

**Other Testing Tools and Approaches**

1. **Jest** - Used for JavaScript/TypeScript unit testing (referenced in **jest.config.js** files)
2. **JaCoCo** - For Java code coverage reporting:

CopyInsert

sonar.coverage.jacoco.xmlReportPaths=gravitee-apim-gateway-coverage/target/site/jacoco-aggregate/jacoco.xml

1. **Storybook** - For UI component testing in isolation (referenced in CircleCI workflows)
2. **Test Fixtures** - Extensive use of test fixtures and test data in both back-end and front-end tests
3. **CI Integration** - Tests are integrated into the CI/CD pipeline with CircleCI

**Q7.** How are contributions to the project managed? What process is in place to validate and integrate pull requests? Are there specific steps, such as code reviews or automated tests, before code is merged? Is a specific Git workflow used? *Refer to pull requests and discussions to answer this question.*

**Fork & Branch**

Contributors fork the repository and create a feature branch named according to the issue, e.g.:

git checkout -b issue/<issue-id>-my-feature master

where <issue-id> is the GitHub or JIRA issue number.

**Pull Request Template & JIRA Integration**

When opening a PR, authors fill out the minimal .github/pull\_request\_template.md, linking to one or more JIRA issues (e.g., <https://gravitee.atlassian.net/browse/APIM-1234>).

**Conventional Commits**

Commit messages follow the Conventional Commit Message format (feat:, fix:, chore:, etc.), ensuring semantic versioning and automated changelog generation.

**Code Owners & Reviews**

A CODEOWNERS file designates the @gravitee-io/apim team as mandatory reviewers for all code, with additional owners for the /helm/ folder. GitHub automatically requests reviews from those teams on every PR.

The PR UI enforces at least one approving review before merge:

“At least 1 approving review is required to merge this pull request.”

**Automated Validation**

**Build & Unit Tests:** A CircleCI pipeline (triggered automatically on push to a PR branch) runs mvn clean install, executes unit and integration tests, and publishes artifacts.

**Static Analysis:** SonarCloud analyzes both backend and UI code, enforcing a quality gate (no new bugs, security hotspots, and minimum coverage thresholds).

**End‑to‑End Tests:** Labeling a PR with run-e2e-tests triggers a GitHub Actions workflow which POSTs to CircleCI to run a full end‑to‑end test suite in Docker.

**Mergify‑Driven Backports & Merges**

A .mergify.yml configures automated backporting: applying an apply-on-4.7.x (or apply-on-master, 4.6.x, etc.) label will cherry‑pick the PR into the corresponding maintenance branch and assign the original author.

**Rebasing & Keeping Up‑to‑Date**

If a PR falls behind master, maintainers or contributors may be asked to:

git fetch upstream

git rebase upstream/master

git push -f

### **3. API Management**

**Q8.** What is an API management platform, and what do you think are the main challenges of such a platform?  
Explain how an API Management solution like Gravitee.io is important for businesses. What needs or problems does an API Management platform aim to address? Are you familiar with standards such as OAuth2, JWT, and API Key, and their usefulness for APIs?

Ans:

Think of an API Management platform as the air traffic control tower of your digital services. It stands between your backend systems—those microservices, databases, legacy apps—and the outside world of mobile apps, partner portals, B2B integrations, or even your own frontend teams. Its job is to make sure every request is routed correctly, secured appropriately, and monitored so you can spot trouble before it grounds your entire operation.

When you build one or a hundred APIs, you suddenly face questions you didn’t sign up for when you wrote your first “Hello, World” endpoint: “Who’s calling this?” “Am I getting too much traffic?” “Is someone trying to sneak in malicious data?” and “How do I roll out a new version without breaking everything?” That’s where an API Management solution like Gravitee.io comes in: it gives you a single place to define policies (rate limits, quotas, transformations), to enforce authentication and authorization, and to view real‑time dashboards of usage, errors, and performance.

In practice, the biggest headaches usually boil down to four things:

1. **Scale and Performance** – APIs often live or die by latency. When you hit a sudden surge—say, your mobile app goes viral—you need an architecture that can elastically scale without adding noticeable lag.
2. **Security and Governance** – You want consistent, company‑wide rules: no more “each team does their own thing.” That means gated access (OAuth2, API keys), encrypted traffic, threat detection, audit trails, and compliance checks.
3. **Developer Experience** – You want external and internal developers to feel empowered, not confused. A slick developer portal with interactive docs, easy sign‑up, self‑service key issuance—and clear error messages—makes integration quick and painless.
4. **Observability and Troubleshooting** – Distributed systems are noisy. Being able to trace a request from edge to backend, set up alerts for anomaly detection, and dive into logs and metrics in one place is a game‑changer.

From a business perspective, investing in API Management means accelerating time‑to‑market (spinning up new APIs in hours, not weeks), protecting your infrastructure from abuse, opening new revenue streams through API monetization, and gaining data‑driven insights into how partners and customers are actually using your services.

Under the hood, most platforms support industry‑standard security protocols:

* **OAuth2** handles delegated authorization (so you don’t have to share passwords) and offers flows for web apps, machine‑to‑machine scripts, mobile clients, and more.
* **JWTs** (JSON Web Tokens) carry signed identity claims—user IDs, scopes, expiry—so the API gateway can validate requests without a database lookup.
* **API Keys** are simple static tokens, great for quick onboarding or low‑risk use cases, often paired with IP restrictions or rate limits.

### **4. Reflection and Personal Feedback**

**Q9.** *Do you think you could run the project locally with the different front-end and back-end parts?*

*Ans:*Yeah, I think running Gravitee locally is totally doable. I've actually tried both approaches - the full local dev setup and the Docker way.

For the full dev setup, you'd need Java 17+, Maven, and the usual suspects. The CONTRIBUTING guide lays it out pretty well - build with Maven, spin up MongoDB and Elasticsearch in Docker, then run the Gateway and Management API from their distribution folders. The UI parts are standard Angular apps you can run with Yarn.

But honestly, the Docker approach is way simpler if you just want to try it out. I pulled their official images with:

docker pull graviteeio/apim-gateway:latest

docker pull graviteeio/apim-management-api:latest

docker pull graviteeio/apim-portal-ui:latest

They've got a nice docker-compose file in their docs that wires everything together - gateway, management API, portal, and the required MongoDB and Elasticsearch. Just a quick **docker-compose up** and you're good to go.

The Docker setup is perfect for testing or demoing, but for actual development work, I'd go with the local setup so you can make changes and see them immediately. The IntelliJ run configurations they mention are pretty handy too.

One thing to watch out for - this is a pretty hefty system with multiple components, so make sure your machine has enough resources. I had to bump up Docker's memory allocation when running the full stack

**Q10.** If you had to make improvements to this project, what would they be? Why do you think these changes would be beneficial for the team or the project?  
Be constructive and suggest improvement ideas based on your analysis of the project.

Ans:  
Looking at Gravitee, I think there are a few things they could definitely improve on:

First off, the setup process is way too complicated. I spent hours trying to get everything running locally. They should just create a simple dev script that handles all the dependencies and startup processes. Something like **./dev-start.sh** that fires up MongoDB, Elasticsearch, and all the services in the right order.

The UI feels pretty dated too. Angular works, but it's getting heavy compared to newer options. I've worked on a few API management tools and most are moving to React or Vue for the frontend. The build times would be way faster, and they'd probably attract more frontend contributors.

Documentation is another pain point. When I was digging through the code, I kept having to jump between files to understand how things connect. Some high-level docs explaining how the gateway talks to the management API would save everyone a ton of time.

Also - testing is all over the place. Some modules have great coverage, others barely any. I'd standardize on a testing approach across the project.

Oh, and they really need to figure out their versioning strategy. Maintaining both v3 and v4 API versions must be a nightmare. I'd rather see them focus on a clean migration path and sunset the older versions quicker.

The plugin system has potential but it's not super developer-friendly. A better SDK with examples would make it easier for people to extend the platform.

Don't get me wrong - it's a solid project overall, and definitely better than some commercial options I've used. These changes would just make it easier to work with for both users and developers