Stage 2
Building a Service-Oriented Architecture

Search Engine Project

Big Data
Grado en Ciencia e Ingeniería de Datos
Universidad de Las Palmas de Gran Canaria

1 Introduction

The goal of Stage 2 is to extend the data layer developed in Stage 1 into a fully operational *service-oriented architecture (SOA)*. In this phase, students will develop a collection of independent, cooperative *microservices* that communicate through *REST APIs* and exchange structured information using standard data formats such as *JSON*. This stage introduces distributed execution, modular scalability, and realistic system design principles inspired by modern Big Data and cloud-based environments.

While Stage 1 focused on designing and implementing the *data layer*—including the *datalake* and the *datamarts*, Stage 2 focuses on transforming those static components into a dynamic ecosystem of services. Each functional element of the previous stage becomes a standalone unit, capable of running, scaling, and failing independently, while still collaborating as part of a unified architecture.

By the end of this stage, each group will have developed a modular and distributed *search engine prototype* capable of:

- Reacting autonomously to data ingestion and indexing events.
- Handling multiple concurrent users through RESTful APIs.

However, the fundamental goal of this stage is not only to achieve functional correctness, but to identify the *limits of the current architecture* and begin reasoning about how it could evolve into a truly scalable system. In practical terms, students will explore how the system behaves as workloads increase—how many simultaneous requests it can handle, how performance degrades under load, and which components become bottlenecks first.

Through systematic benchmarking and analysis, students will determine which components of their architecture limit performance, which could benefit from replication or parallelization, and what architectural changes would be necessary to achieve scalability in future stages. This exploration marks the transition from building a functioning system to *engineering a system capable of growth.*

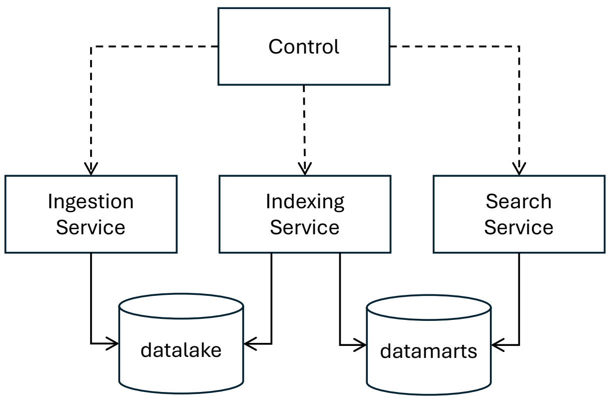

Figure 1: Service Oriented Architecture for Search Engine Project

2 System Architecture Overview

The service-oriented architecture designed for Stage 2 decomposes the monolithic pipeline into a set of autonomous and interoperable components. It will be composed of three core services:

- The *Ingestion Service*, which downloads new data from external sources and stores it in the datalake, preserving the ingestion hierarchy defined in Stage 1.
- The *Indexing Service*, which processes both book bodies and headers to extract metadata, build inverted indexes, and store optimized structures in the datamarts.
- The *Search Service*, which exposes endpoints for keyword queries, metadata filtering, and ranking operations using the previously built indexes.

This separation of concerns allows the system to evolve, deploy, and scale individual components independently. *Figure 1* illustrates the overall architecture. The architecture is composed of theses primary services together with a *control* that coordinates their interactions with centralized supervision. Each service focuses on a specific functional responsibility and communicates with others using lightweight *REST APIs* over HTTP, exchanging information in structured *JSON* format.

In this project, services will be implemented in *Java 17* using the lightweight framework *Javalin*. Javalin provides a minimalist approach to building RESTful services, offering a clean API for defining routes, handling JSON requests and responses, and managing concurrency. Its simplicity and low overhead make it ideal for educational environments and microservice-based systems, where transparency and control over the server logic are essential. Each microservice will be self-contained, exposing its own HTTP endpoints and configuration through environment variables or configuration files.

3 Microservice Design and Implementation

The implementation of each service follows the *microservice architectural pattern*, where every component of the system is developed, deployed, and executed as an independent process. Each microservice has its own well-defined responsibility, internal data model, and REST API, and

communicates with other services exclusively through lightweight HTTP interfaces using JSON as the data exchange format.
A microservice should be designed as a *self-contained unit*, meaning that it can be started, stopped, or redeployed without affecting the rest of the system. This design promotes modularity, scalability, and fault isolation: if one component fails or requires maintenance, the others continue functioning normally. Each service is developed as a standalone *Maven project*, producing a single executable `.jar` file that can be run locally
The following subsections provide a detailed example of how to implement a basic microservice. This template should serve as the foundation for all components of the system each one adapting its internal logic while maintaining the same external conventions.

3.1 Project Structure

Each project should follow the standard Maven layout:

```
service/
  pom.xml
  src/
    main/
      java/
        com/example/service/
          App.java
          ServiceController.java
      resources/
        application.properties
    test/
      java/
        com/example/service/
          ServiceTests.java
```

3.2 Maven Configuration

The following `pom.xml` defines the basic dependencies and build configuration for a Javalin-based microservice using *Gson* for JSON serialization:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>service</artifactId>
    <version>1.0.0</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>io.javalin</groupId>
            <artifactId>javalin</artifactId>
            <version>6.1.3</version>
        </dependency>
        <dependency>
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>2.11.0</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>2.0.9</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>3.3.0</version>
                <configuration>
                    <archive>
                        <manifest>
                            <mainClass>com.example.service.App</mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

3.3 Example Code: Basic Service

The following example shows how to implement a simple microservice using Javalin and Gson. This minimal service exposes two REST endpoints: one for health checking and another that returns a list of mock data.

```java
package com.example.service;

import io.javalin.Javalin;
import io.javalin.http.Context;
import com.google.gson.Gson;
import java.util.*;

public class App {
    private static final Gson gson = new Gson();

    public static void main(String[] args) {
        Javalin app = Javalin.create(config -> {
            config.http.defaultContentType = "application/json";
        }).start(7000);

        app.get("/status", ctx -> {
            Map<String, Object> status = Map.of(
                "service", "example-service",
                "status", "running"
            );
            ctx.result(gson.toJson(status));
        });
        app.get("/data", App::handleData);
    }

    private static void handleData(Context ctx) {
        String filter = ctx.queryParam("filter", "none");

        // Example mock dataset
        List<Map<String, Object>> items = List.of(
            Map.of("id", 1, "name", "Item A"),
            Map.of("id", 2, "name", "Item B"),
            Map.of("id", 3, "name", "Item C")
        );

        Map<String, Object> response = Map.of(
            "filter", filter,
            "count", items.size(),
            "items", items
        );

        ctx.result(gson.toJson(response));
    }
}
```

Listing 1: Example of a basic Javalin microservice using Gson

This service can be built and executed with:

```
mvn clean package
java -jar target/service-1.0.0.jar
```

Once running, it responds to HTTP requests such as:

```
GET http://localhost:7000/status
GET http://localhost:7000/data?filter=test
```

4 Service APIs and Orchestration

In this stage, the architecture adopts an *orchestrated control model*, where the global workflow is managed by a centralized *Control Module*. Instead of relying on services that react autonomously to events, the control module explicitly coordinates the execution order — first triggering ingestion, then indexing, and finally exposing the updated data through the search service. This approach simplifies monitoring, debugging, and benchmarking, as the full lifecycle of each dataset can be tracked from a single coordination point.
Each service still operates independently and exposes a *REST API*, but the control module determines when and how those APIs are invoked. The following subsections define the API specification for each microservice and describe the orchestration logic.

4.1 Ingestion Service API

Purpose: Download raw book data and store it in the *datalake* using the hierarchical directory structure defined in Stage 1.

```
POST /ingest/{book_id}
    Description: Downloads a book from Project Gutenberg and stores it in
                 the datalake. Splits the content into header/body files.
    Request: none
    Response: {
        "book_id": 1342,
        "status": "downloaded",
        "path": "datalake/20251008/14/1342"
    }

GET /ingest/status/{book_id}
    Description: Checks whether a specific book has been successfully downloaded.
    Response: {
        "book_id": 1342,
        "status": "available"
    }

GET /ingest/list
    Description: Lists all downloaded books currently stored in the datalake.
    Response: {
        "count": 120,
        "books": [1342, 5, 17, 42, ...]
    }
```

Notes:

- Each successful ingestion should be logged persistently.
- The control module periodically queries this service to identify new books available for indexing.

4.2 Indexing Service API

Purpose: Transform the unstructured text from the datalake into structured data, extract metadata, and update the inverted index stored in the *datamart*.

```
POST /index/update/{book_id}
    Description: Reads the specified book from the datalake, extracts metadata,
                 and updates both metadata tables and the inverted index.
    Request: none
    Response: {
        "book_id": 1342,
        "index": "updated"
    }

POST /index/rebuild
    Description: Rebuilds the entire index and metadata tables from scratch,
                 processing all books available in the datalake.
    Request: none
    Response: {
        "books_processed": 1000,
        "elapsed_time": "35.2s"
    }

GET /index/status
    Description: Returns statistics about the indexing process.
    Response: {
        "books_indexed": 1200,
        "last_update": "2025-10-08T14:05:00Z",
        "index_size_MB": 42.7
    }
```

Notes:

- The service must ensure idempotency — reindexing an already processed book should not create duplicates.
- The control module decides when indexing is required, avoiding unnecessary recomputation.

4.3 Search Service API

Purpose: Provide access to indexed data for search and filtering operations.

```
GET /search?q={term}
    Description: Searches for a keyword across the indexed books.
    Response: {
        "query": "adventure",
        "filters": {},
        "count": 25,
        "results": [
            {"book_id": 5, "title": "Robinson Crusoe", "author": "Daniel Defoe",
             "language": "en", "year": 1719},
            {"book_id": 1342, "title": "Pride and Prejudice", "author": "Jane Austen",
             "language": "en", "year": 1813}
        ]
    }

GET /search?q={term}&author={name}
    Description: Searches for a keyword filtered by author.
    Response: {
        "query": "adventure",
        "filters": { "author": "Jane Austen" },
        "count": 3,
        "results": [
            {"book_id": 1342, "title": "Pride and Prejudice", "author": "Jane Austen",
             "language": "en", "year": 1813},
            {"book_id": 158,  "title": "Emma", "author": "Jane Austen",
             "language": "en", "year": 1815},
            {"book_id": 201,  "title": "Sense and Sensibility", "author": "Jane Austen",
             "language": "en", "year": 1811}
        ]
    }

GET /search?q={term}&language={code}
    Description: Searches for a keyword filtered by language (ISO 639-1 code).
    Response: {
        "query": "adventure",
        "filters": { "language": "fr" },
        "count": 4,
        "results": [
            {"book_id": 6500, "title": "Les Misérables", "author": "Victor Hugo",
             "language": "fr", "year": 1862},
            {"book_id": 4201, "title": "Vingt mille lieues sous les mers",
             "author": "Jules Verne", "language": "fr", "year": 1870}
        ]
    }

GET /search?q={term}&year={YYYY}
    Description: Searches for a keyword filtered by publication year.
    Response: {
        "query": "adventure",
        "filters": { "year": 1865 },
        "count": 2,
        "results": [
            {"book_id": 11, "title": "Alice’s Adventures in Wonderland",
             "author": "Lewis Carroll", "language": "en", "year": 1865},
            {"book_id": 12, "title": "De la Terre à la Lune",
             "author": "Jules Verne", "language": "fr", "year": 1865}
        ]
    }

GET /search?q={term}&author={name}&language={code}&year={YYYY}
    Description: Performs a combined search with multiple filters.
    Response: {
        "query": "adventure",
        "filters": { "author": "Jules Verne", "language": "fr", "year": 1865 },
        "count": 1,
        "results": [
            {"book_id": 12, "title": "De la Terre à la Lune",
             "author": "Jules Verne", "language": "fr", "year": 1865}
        ]
    }
```

4.4 Control Module

Purpose: Coordinate the sequential execution of the ingestion and indexing services, ensuring data consistency and completeness across the pipeline.
The control module can be implemented as a standalone process (in Java) that sends HTTP requests to the corresponding microservice APIs. It maintains the global state of the system through simple tracking files or a lightweight database.

Example workflow:

1. Select a new `book_id` not yet processed.
2. Call the Ingestion Service to download the book:

   ```
   POST http://ingestion:7001/ingest/{book_id}
   ```

3. Wait for confirmation of successful ingestion (`status=downloaded`).
4. Call the Indexing Service to process and index the book:

   ```
   POST http://indexing:7002/index/update/{book_id}
   ```

5. Verify indexing completion.
6. Notify the Search Service to refresh caches.

5 Benchmarking Methodology

In this stage, it should be evaluated the *performance, efficiency, and scalability* of the implemented service-oriented architecture under controlled conditions. After achieving a functional microservice-based system, this stage aims to determine how well the architecture performs, where its limits are, and how it can be improved to support larger workloads.

While Stage 1 focused on functionality and data organization, Stage 2 introduces a quantitative analysis of system behavior. The benchmarking process must answer questions such as:

- How many books per second can be ingested or indexed?
- What is the average response time for search queries?
- How do CPU and memory consumption evolve under load?
- Does the architecture scale efficiently?

The goal is not only to measure numbers, but to *interpret trends and identify architectural bottlenecks* that affect overall scalability.

5.1 Benchmarking Scope

Two complementary levels of benchmarking will be applied:

- *Microbenchmarking (component-level):* Conducted directly on the host machine using *Java Microbenchmark Harness (JMH)*. These tests measure the computational efficiency of internal functions, such as text parsing, metadata extraction, or inverted index updates. The goal is to obtain precise and reproducible measurements of pure algorithmic performance, unaffected by network latency.
- *Integration benchmarking (system-level):* Conducted optionally using a full deployment. These experiments measure end-to-end behavior (ingestion + indexing + query) under concurrent user requests, providing insight into the system's operational scalability.

5.2 Host Configuration

All benchmarks must be executed on a machine with known and fixed resources. A reference configuration could be:

- 4 CPU cores
- 32 GB of RAM
- Stable network connection

5.3 JMH Benchmarking Methodology

The *Java Microbenchmark Harness (JMH)* framework will be used to perform precise performance measurements of Java code within each microservice. JMH automatically manages JVM warm-up, JIT compilation, and garbage collection effects, ensuring statistically valid results.
Benchmarks should be integrated in the source tree under a dedicated package such as `com.example.benchmark`. They must target computationally intensive operations like:

- Text tokenization and normalization.
- Metadata parsing and database insertion.
- Index construction and lookup operations.
- Query filtering or ranking functions.

Best practices for valid and reproducible measurements:

1. Include JMH as a Maven dependency and build the benchmark JAR with `mvn package`.
2. Run benchmarks on an idle host to avoid resource interference.
3. Execute at least 5 warm-up iterations and 10 measurement iterations.
4. Fix JVM heap size (e.g., `-Xmx4G`) and thread count to maintain consistency.
5. Export results as CSV or JSON (`-rf csv`) for later analysis.

5.4 Integration Benchmarking

After microbenchmarking, groups may perform complementary system-level tests. These experiments evaluate how independent services interact under concurrent requests, simulating real-world load conditions.
Metrics to be collected:

- Ingestion throughput (books per second).
- Indexing latency (seconds per book).
- Query response time (ms/query, including 95th percentile).
- CPU and memory utilization per service.

5.5 Data Analysis and Visualization

Benchmark data must be summarized in tables and plots showing:

- Throughput vs. concurrency level.
- Latency distribution across iterations.
- CPU/memory usage over time.

All figures should include average values and standard deviations. Interpret the results by discussing:

- When and why performance saturates.
- Which service becomes the bottleneck under load.
- Whether scaling increasing resources yields better results.

5.6 Expected Outcomes

By completing this benchmarking phase, each group should be able to:

- Quantify the performance limits of their architecture.
- Identify critical sections of code or data flow that impact scalability.
- Compare configurations and justify architectural optimizations.
- Present reproducible evidence supporting their conclusions.

Benchmarking thus provides a bridge between implementation and analysis: it transforms a functional distributed system into a measurable, optimizable architecture grounded on empirical data.

6 Project Delivery Guidelines

Each group must select a unique *group name* that will be used to identify their project. The final deliverable for this stage consists of a single *written report* in PDF format. This document serves as the official record of your work and must clearly reference the external repository where all source code and configuration files are hosted.
As in Stage 1, this is the only file that must be submitted to the *virtual campus platform*, and *only one member of the group* should upload it on behalf of the entire team. This ensures consistent version control and avoids duplicate submissions during evaluation.

6.1 Required Structure of the Report

The written report (`.pdf`) must include the following sections:

1. *Cover page:*
   - Course name and academic year.
   - Project title.
   - Full names and student IDs of all group members.
   - The chosen group name.
   - The URL of the GitHub repository.
2. *Introduction and objectives:* Explain the purpose of Stage 2 and the main goals achieved when extending the data layer into a distributed service-oriented architecture.
3. *System architecture:* Describe the design and interaction of the microservices (Ingestion, Indexing, and Search). Include diagrams showing service boundaries, API communication, and Docker Compose integration.
4. *Microservice implementation:* Summarize the internal structure of each service, the chosen technologies (Java, Javalin, Gson), and any inter-service communication mechanisms. Provide example endpoints and describe how orchestration is handled by the Control Module.

5. *Design decisions:* Justify the architectural choices made regarding deployment strategy, API design, database selection, and scalability mechanisms. Discuss the reasoning behind configuration parameters (e.g., ports, threading model...).

6. *Benchmarks and results:* Present experimental data on system performance, including:

   - Indexing throughput (books processed per second).
   - Query latency and concurrency behavior.
   - CPU and memory utilization per container.
   - Scalability limits on a reference server (4 cores, 32 GB RAM).

   Provide visualizations (tables or graphs) and discuss observed bottlenecks and scaling trends.

7. *Conclusions and future improvements:* Reflect on the strengths and weaknesses of your current implementation. Suggest potential optimizations for scaling horizontally or integrating load balancing in future stages.

6.2 Group Name and Repository Structure

The GitHub repository must follow the exact naming format:

```
https://github.com/<group_name>/stage_2
```

The repository must include:

- A clear and detailed `README.md` with instructions for:
  - Building each microservice using Maven.
  - Running all services.
  - Executing test queries and benchmark scripts.
- Source code for all microservices (Ingestion, Indexing, Search, and Control Module).
- Any additional configuration files or datasets required for testing.
- A meaningful Git history showing incremental progress and contributions from all members.

7 Evaluation Criteria

The evaluation will consider both the technical implementation and the written report. Grading will follow these criteria:

- *Report quality* (25 %) – clarity, completeness, organization, and proper documentation of experiments and results.
- *Correctness of the architecture* (30 %) – correct functioning and integration of all microservices through REST APIs and Docker Compose.

- *Code quality* (20 %) – structure, modularity, readability, and documentation of the source code.
- *Benchmarking and scalability analysis* (25 %) – rigor and depth of experiments; ability to identify and discuss performance limits of the system.

Students will also have the opportunity to deliver an *optional oral presentation* of their work. This presentation allows each member to demonstrate their individual contributions, implementation insights, or performance experiments.

- Each student will have a maximum of *4 minutes* to present.
- Presentations should focus on design rationale, implementation challenges, scalability findings, and lessons learned.
- Choosing not to present will not affect the grade; evaluation will rely solely on the report and implementation.

The presentation schedule will be communicated in advance, ensuring that each group knows its assigned time slot.