Memory TSCD

Irene Guerra Déniz and Raúl García Nuez

January 24, 2025

1 Context

Large-scale text analysis generates vast amounts of information that can be represented as graphs. Each word acts as a node, and the connections between them (edges) reflect relationships such as co-occurrence, proximity, or frequency. This model is ideal for representing complex relationships and optimizing queries efficiently.

In this project, the system aims to identify optimal paths in a word graph, where the connections between nodes are weighted by the strength of their relationship. The primary focus is on the efficiency and scalability of the algorithms applied.

2 Problems

1. Shortest Path: Determine the most efficient route between two specific words, minimizing the accumulated weight.

2. Longest Path: Identify the longest possible route between two specific words, considering all nodes and edges within the graph.

3. All Possible Paths: Find all possible paths between two specific words, allowing a comprehensive analysis of their connections.

4. Optimization of Complex Routes: Analyze paths in large subgraphs and identify indirect relationships between sets of words.

5. Dynamic Graph Analysis: Efficiently update the graph as new words or connections are added.

6. Cluster Identification: Detect communities or groups of strongly connected words within the graph to better understand semantic and structural relationships.

3 Problems

1. Shortest Path: Determine the most efficient route between two specific words, minimizing the accumulated weight.

2. Longest Path: Identify the longest possible route between two specific words, considering all nodes and edges within the graph.

3. All Possible Paths: Find all possible paths between two specific words, allowing a comprehensive analysis of their connections.

4. Optimization of Complex Routes: Analyze paths in large subgraphs and identify indirect relationships between sets of words.

5. Dynamic Graph Analysis: Efficiently update the graph as new words or connections are added.

6. Cluster Identification: Detect communities or groups of strongly connected words within the graph to better understand semantic and structural relationships.

4 Process

Book Download from Gutenberg Project

- Obtain books from public sources such as the Gutenberg Project using unique identifiers for each book.
- Verify that the downloaded books are valid and complete.

Storage in AWS S3

- Save the downloaded books in an S3 bucket under the folder `libros_sin_procesar/`.
- Ensure that the data is organized by state and accessible for subsequent processing.

Book Processing

- Read the content of the books stored in S3.
- Split the text into words, remove stopwords, and perform basic cleaning.
- Count the words and assign weights based on their frequency.
- Move the books to the folder `libros_procesados/`.
- Save the processed content in MongoDB.

Reading from MongoDB and Graph Construction

- Connect to and read the words stored in MongoDB.
- Represent the words as nodes and their relationships as edges with weights.

Graph Storage in JSON Format

- Export the constructed graph or relevant subgraphs in JSON format.
- Store the JSON file in an S3 bucket, making it accessible for subsequent retrieval.
- This JSON file will be used by the API to initialize the graph internally.

Graph Path Analysis

- **Shortest Path:** Implement algorithms like Dijkstra to determine the most efficient route between two words.
- **Longest Path:** Explore extensive routes between specific nodes.
- **All Possible Paths:** Generate all possible paths between two words for exhaustive analysis.

Cluster Identification

- Detect communities or groups of related words using algorithms like Louvain.
- Analyze how these communities reflect semantic or structural relationships.

Dynamic Graph Updates

- Incorporate new words and relationships into the graph as more books are processed.
- Recalculate paths, weights, and clusters to keep the graph updated.

API Exposure

- Provide endpoints to:
  - Query the shortest, longest, and all possible paths between words.
  - Retrieve specific subgraphs based on defined criteria.
  - List clusters or communities of related words.
- Load the graph (en formato JSON) desde el bucket S3 y permitir su inicialización dentro de la API.

5 System Objective

To create a distributed system capable of building and analyzing word graphs by optimizing paths and exposing these functionalities through an API. The system must be scalable, efficient, and able to handle dynamic graphs generated from large volumes of text.

6 Application Layer

6.1 What is C4?

C4 is a software architecture modeling approach that divides the system into four abstraction levels:

- **Context**: Shows the system's interactions with users and external systems.
- **Container**: Represents applications, services, and databases within the system.
- **Component**: Breaks down containers into functional building blocks.
- **Code (optional)**: Provides detailed insights into the implementation of specific components.

C4 emphasizes simplicity and clarity, making it ideal for communicating and documenting software designs.

6.2 Why C4 was chosen

1. **Clarity**: It divides the documentation into levels (Context, Container, Component, Code), making it easier to understand for both technical and non-technical audiences.
2. **Standardization**: Provides a uniform and structured format for documenting software systems.
3. **Scalability**: Allows adjusting the level of detail as needed, from a general overview to specific details.
4. **Collaboration**: Facilitates communication between developers, architects, and non-technical stakeholders.
5. **Maintainability**: Simplifies understanding and updating the design over time.

6.3 System Context

System Context Diagram Explanation

This System Context Diagram illustrates the interactions of the **Word Graph System** with external actors and systems:

- **API User**: Queries the system via an API to search the word graph.
- **AWS S3**: Stores books and word graphs in JSON format.
- **AWS EC2 (MongoDB)**: Hosts the database for storing words and their occurrences.
- **GitHub Actions**: Automates tests and deployments for the system.

The diagram highlights how the main system processes data and connects with key services and users.

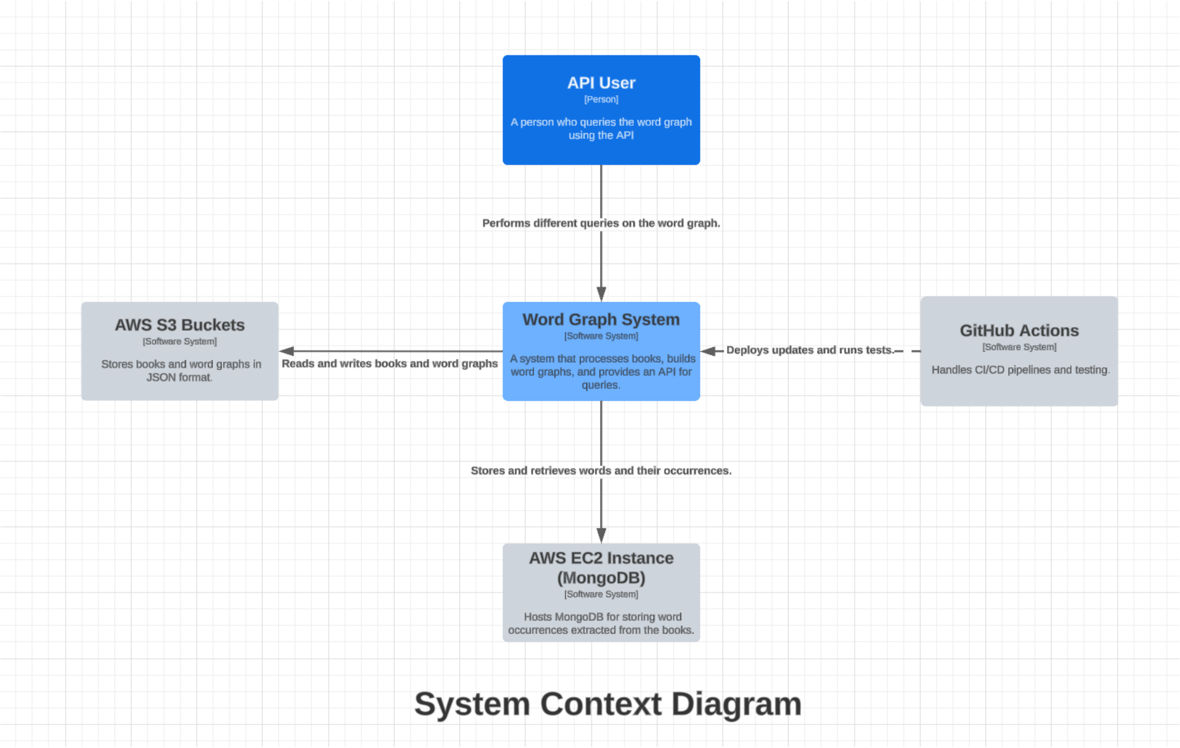
API User
[Person]
A person who queries the word graph using the API
Performs different queries on the word graph.
AWS S3 Buckets
[Software System]
Stores books and word graphs in JSON format.
Reads and writes books and word graphs
Word Graph System
[Software System]
A system that processes books, builds word graphs, and provides an API for queries.
Deploys updates and runs tests.
GitHub Actions
[Software System]
Handles CI/CD pipelines and testing.
Stores and retrieves words and their occurrences.
AWS EC2 Instance (MongoDB)
[Software System]
Hosts MongoDB for storing word occurrences extracted from the books.
System Context Diagram

Figure 1: System Context Diagram.

6.4 Container Diagram

This Container Diagram illustrates the main components of the system and their interactions:

- **Query Consumer**: A user or system querying the API.
- **API Graph Application**: Exposes the processed word graph stored in S3.
- **WordsGraph App**: Generates word graphs based on MongoDB data and saves them in S3.
- **WordCounterDatamart App**: Processes books from S3, analyzes words, and stores occurrences in MongoDB.
- **Book Crawler App**: Downloads books from *Gutenberg* and stores them in S3.
- **AWS EC2 (Job Scheduler)**: Orchestrates the execution of all modules.
- **AWS S3**: Stores books, word graphs, and system data.
- **AWS EC2 (MongoDB)**: Hosts the database for storing words and their occurrences.

The diagram highlights how the modules collaborate to process data, generate word graphs, and expose them via the API.

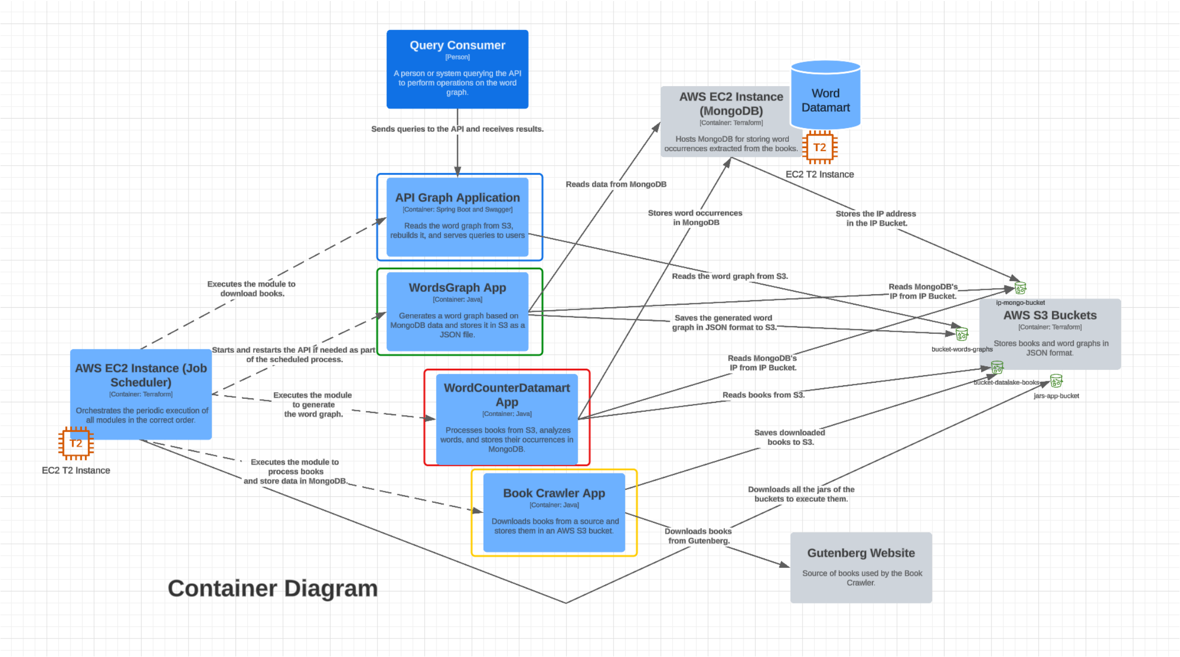
Query Consumer
[Person]
A person or system querying the API to perform operations on the word graph.
Sends queries to the API and receives results.
API Graph Application
[Container: Spring Boot and Swagger]
Reads the word graph from S3, rebuilds it, and serves queries to users
WordsGraph App
[Container: Java]
Generates a word graph based on MongoDB data and stores it in S3 as a JSON file.
AWS EC2 Instance (MongoDB)
[Container: Terraform]
Hosts MongoDB for storing word occurrences extracted from the books.
Word Datamart
T2
EC2 T2 Instance
Reads data from MongoDB
Stores word occurrences in MongoDB
Stores the IP address in the IP Bucket.
Reads the word graph from S3.
Reads MongoDB's IP from IP Bucket.
ip-mongo-bucket
AWS S3 Buckets
[Container: Terraform]
Stores books and word graphs in JSON format.
Saves the generated word graph in JSON format to S3.
bucket-words-graphs
Executes the module to download books.
Starts and restarts the API if needed as part of the scheduled process.
AWS EC2 Instance (Job Scheduler)
[Container: Terraform]
Orchestrates the periodic execution of all modules in the correct order.
T2
EC2 T2 Instance
Executes the module to generate the word graph.
WordCounterDatamart App
[Container: Java]
Processes books from S3, analyzes words, and stores their occurrences in MongoDB.
Reads MongoDB's IP from IP Bucket.
Reads books from S3.
bucket-datalake-books
jars-app-bucket
Saves downloaded books to S3.
Executes the module to process books and store data in MongoDB.
Book Crawler App
[Container: Java]
Downloads books from a source and stores them in an AWS S3 bucket.
Downloads all the jars of the buckets to execute them.
Downloads books from Gutenberg.
Gutenberg Website
Source of books used by the Book Crawler.
Container Diagram

Figure 2: Container Diagram.

6.5 Components Diagram

6.5.1 BookCrawler Component Diagram

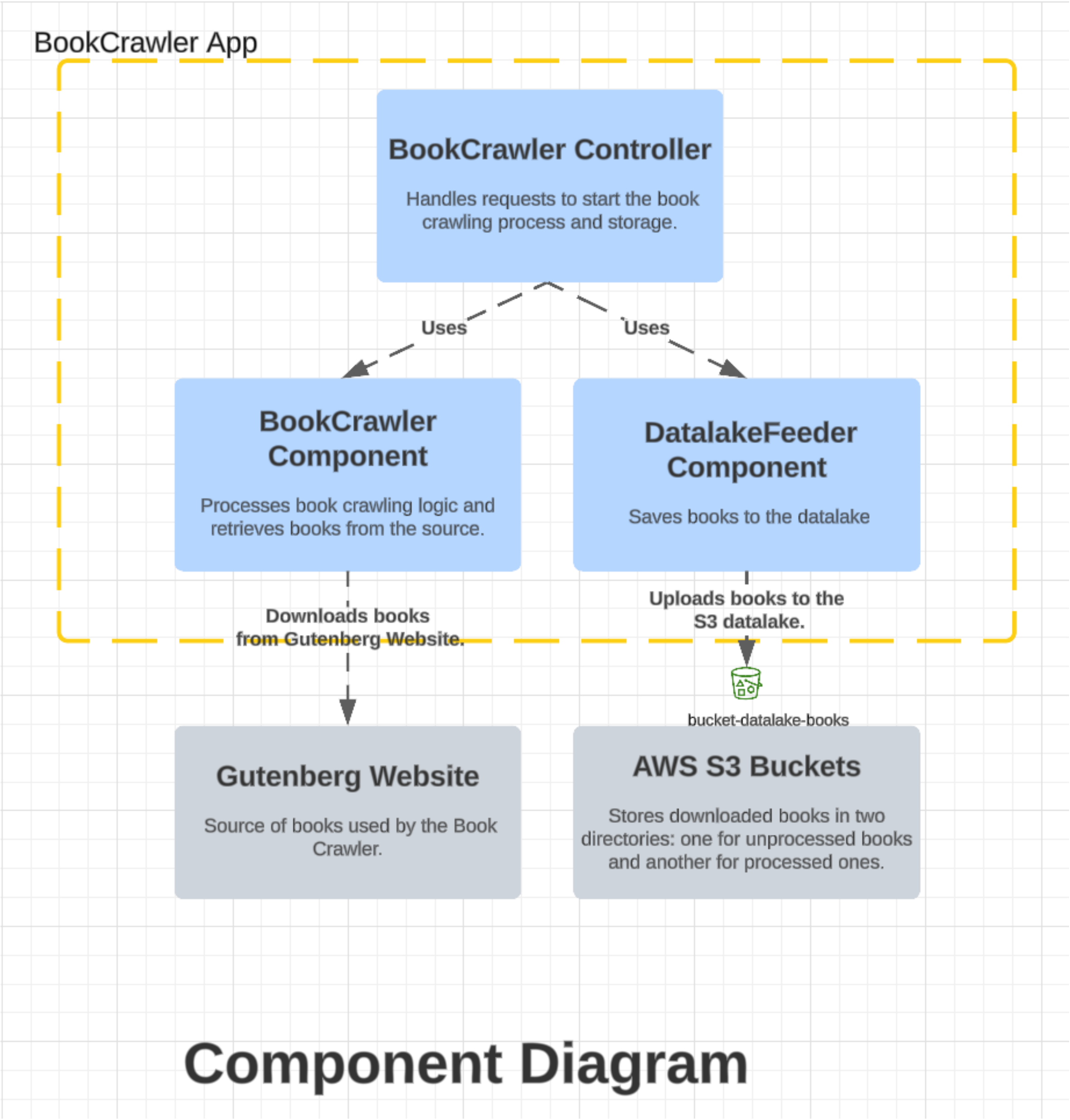
BookCrawler App
BookCrawler Controller
Handles requests to start the book crawling process and storage.
Uses
Uses
BookCrawler Component
Processes book crawling logic and retrieves books from the source.
DatalakeFeeder Component
Saves books to the datalake
Downloads books from Gutenberg Website.
Uploads books to the S3 datalake.
bucket-datalake-books
Gutenberg Website
Source of books used by the Book Crawler.
AWS S3 Buckets
Stores downloaded books in two directories: one for unprocessed books and another for processed ones.
Component Diagram

Figure 3: BookCrawler Component Diagram

BookCrawler Component Diagram

This Component Diagram illustrates how the BookCrawler App manages book downloading and storage:

- **BookCrawler Controller**: Coordinates the crawling and storage processes.
- **BookCrawler Component**: Downloads books from the *Gutenberg Website*.
- **DatalakeFeeder Component**: Uploads downloaded books to the S3 bucket.

The diagram highlights the interactions between components and their connections to external sources and storage systems.

6.5.2 WordCounterDatamart Component Diagram

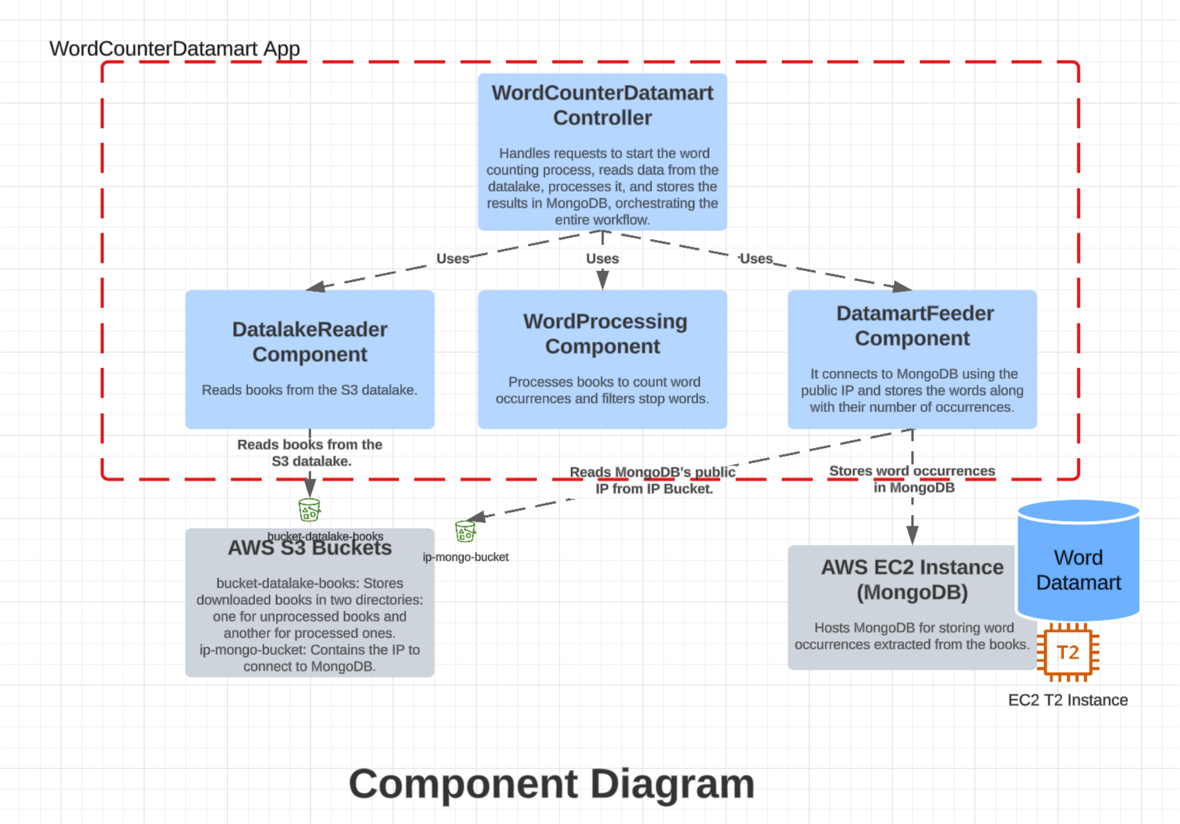
WordCounterDatamart App
WordCounterDatamart Controller
Handles requests to start the word counting process, reads data from the datalake, processes it, and stores the results in MongoDB, orchestrating the entire workflow.
Uses
Uses
Uses
DatalakeReader Component
Reads books from the S3 datalake.
WordProcessing Component
Processes books to count word occurrences and filters stop words.
DatamartFeeder Component
It connects to MongoDB using the public IP and stores the words along with their number of occurrences.
Reads books from the S3 datalake.
Reads MongoDB's public IP from IP Bucket.
Stores word occurrences in MongoDB
bucket-datalake-books
ip-mongo-bucket
AWS S3 Buckets
bucket-datalake-books: Stores downloaded books in two directories: one for unprocessed books and another for processed ones. ip-mongo-bucket: Contains the IP to connect to MongoDB.
AWS EC2 Instance (MongoDB)
Hosts MongoDB for storing word occurrences extracted from the books.
Word Datamart
T2
EC2 T2 Instance
Component Diagram

Figure 4: Container Diagram.

This Component Diagram of WordCounterDatamart App shows:

- **Controller**: Orchestrates word counting and storage.
- **DatalakeReader**: Reads books from AWS S3.
- **WordProcessing**: Counts words and filters irrelevant ones.
- **DatamartFeeder**: Stores words and occurrences in MongoDB.

The diagram highlights interactions with AWS S3 for data and MongoDB on EC2 for storage.

6.5.3 WordsGraph Component Diagram

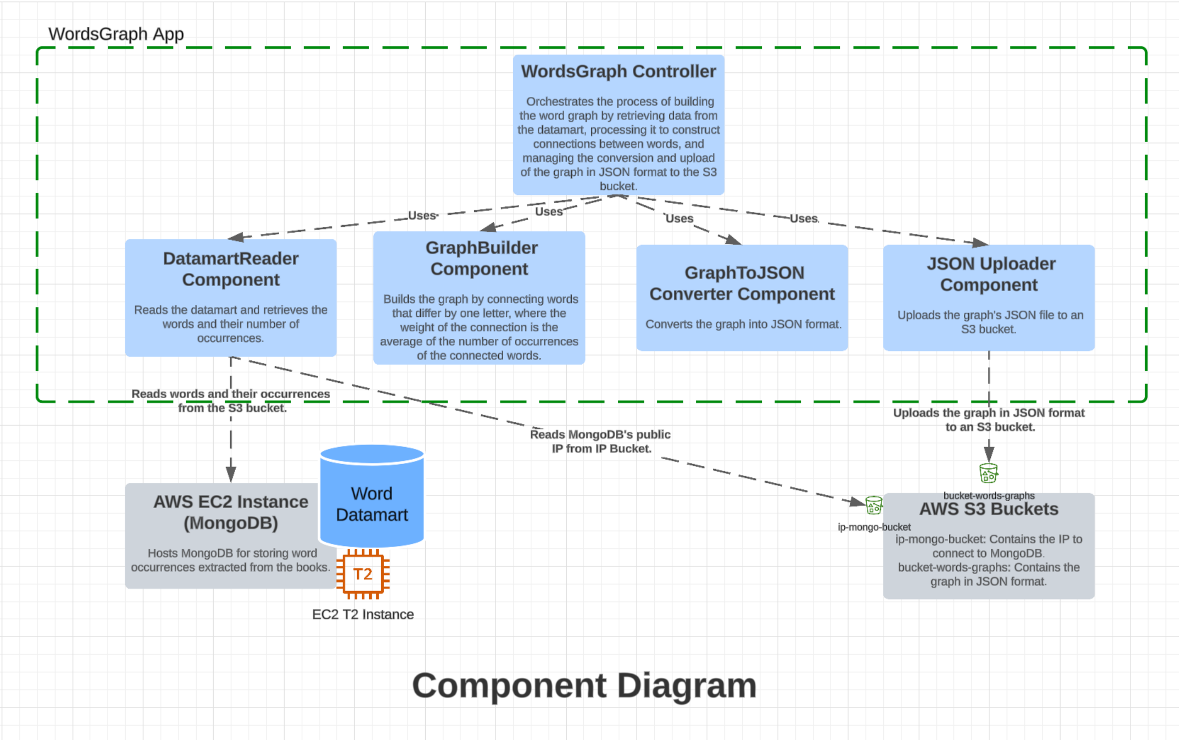
WordsGraph App
WordsGraph Controller
Orchestrates the process of building the word graph by retrieving data from the datamart, processing it to construct connections between words, and managing the conversion and upload of the graph in JSON format to the S3 bucket.
Uses
Uses
Uses
Uses
DatamartReader Component
Reads the datamart and retrieves the words and their number of occurrences.
GraphBuilder Component
Builds the graph by connecting words that differ by one letter, where the weight of the connection is the average of the number of occurrences of the connected words.
GraphToJSON Converter Component
Converts the graph into JSON format.
JSON Uploader Component
Uploads the graph's JSON file to an S3 bucket.
Reads words and their occurrences from the S3 bucket.
Reads MongoDB's public IP from IP Bucket.
Uploads the graph in JSON format to an S3 bucket.
AWS EC2 Instance (MongoDB)
Hosts MongoDB for storing word occurrences extracted from the books.
Word Datamart
T2
EC2 T2 Instance
ip-mongo-bucket
bucket-words-graphs
AWS S3 Buckets
ip-mongo-bucket: Contains the IP to connect to MongoDB.
bucket-words-graphs: Contains the graph in JSON format.
Component Diagram

Figure 5: WordsGraph Component Diagram

This Component Diagram of WordsGraph App shows:

- **Controller**: Orchestrates graph building, conversion, and upload to S3.
- **DatamartReader**: Reads words and occurrences from MongoDB via S3.
- **GraphBuilder**: Constructs the graph by connecting related words.
- **GraphToJSON Converter**: Converts the graph into JSON format.
- **JSON Uploader**: Uploads the JSON graph to AWS S3.

The diagram highlights interactions with AWS S3 for data exchange and MongoDB on EC2 for word occurrences.

6.5.4 APIGraph Component Diagram

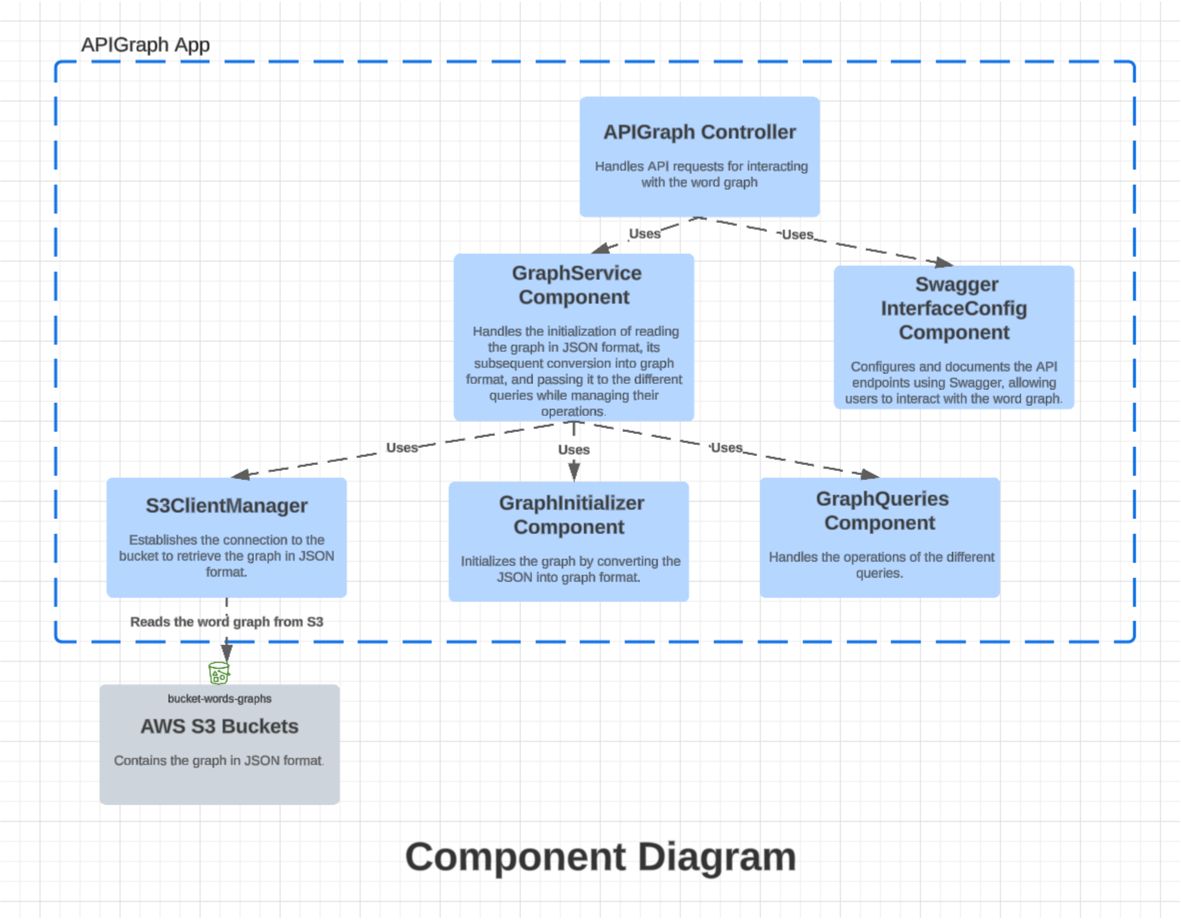
APIGraph App
APIGraph Controller
Handles API requests for interacting with the word graph
Uses
Uses
GraphService Component
Handles the initialization of reading the graph in JSON format, its subsequent conversion into graph format, and passing it to the different queries while managing their operations.
Swagger InterfaceConfig Component
Configures and documents the API endpoints using Swagger, allowing users to interact with the word graph.
Uses
Uses
Uses
S3ClientManager
Establishes the connection to the bucket to retrieve the graph in JSON format.
GraphInitializer Component
Initializes the graph by converting the JSON into graph format.
GraphQueries Component
Handles the operations of the different queries.
Reads the word graph from S3
bucket-words-graphs
AWS S3 Buckets
Contains the graph in JSON format.
Component Diagram

Figure 6: APIGraph Component Diagram

This Component Diagram of APIGraph App shows:

- **Controller**: Handles API requests for interacting with the word graph.
- **GraphService**: Manages graph initialization, conversion, and query operations.
- **GraphInitializer**: Converts the JSON graph into a usable graph format.
- **GraphQueries**: Handles various graph query operations.
- **Swagger InterfaceConfig**: Configures and documents the API endpoints using Swagger.
- **S3ClientManager**: Connects to AWS S3 to retrieve the graph in JSON format.

The diagram highlights the interactions with AWS S3 for graph retrieval and the components managing graph operations and API requests.

6.6 Code Diagrams

6.6.1 BookCrawler Class Diagram

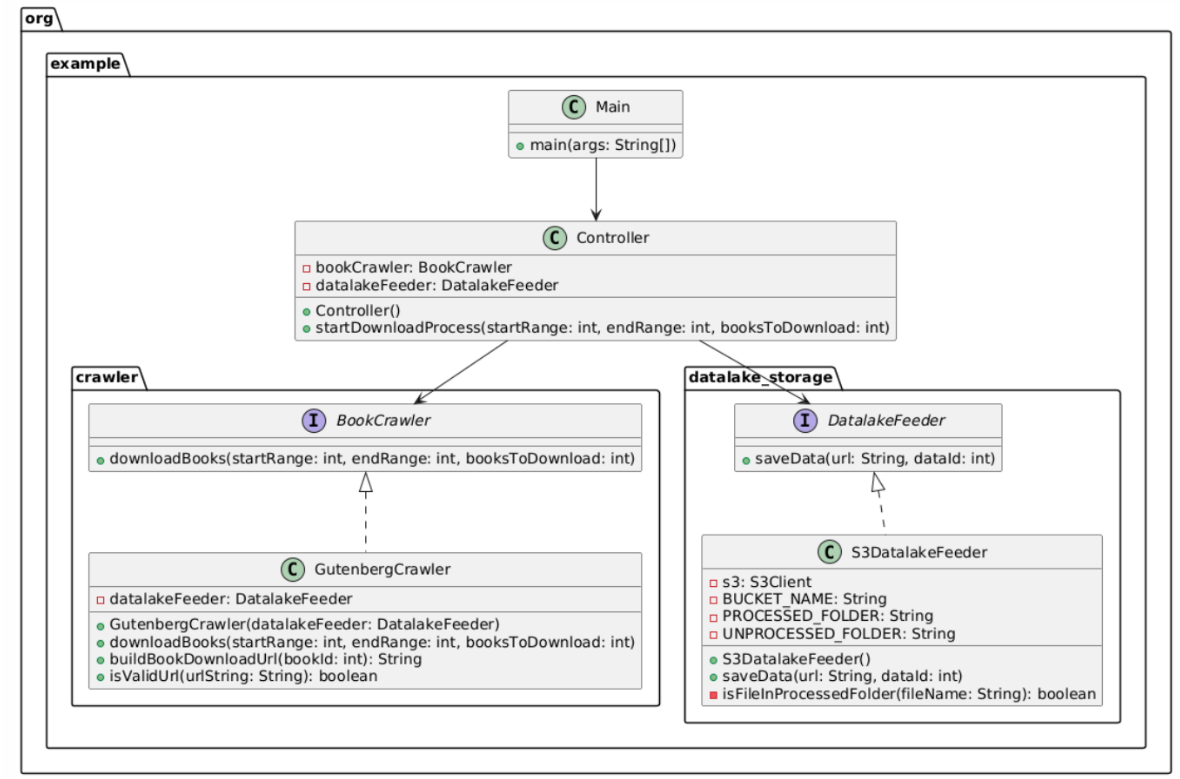
org
example
C Main
main(args: String[])
C Controller
bookCrawler: BookCrawler
datalakeFeeder: DatalakeFeeder
Controller()
startDownloadProcess(startRange: int, endRange: int, booksToDownload: int)
crawler
I BookCrawler
downloadBooks(startRange: int, endRange: int, booksToDownload: int)
C GutenbergCrawler
datalakeFeeder: DatalakeFeeder
GutenbergCrawler(datalakeFeeder: DatalakeFeeder)
downloadBooks(startRange: int, endRange: int, booksToDownload: int)
buildBookDownloadUrl(bookId: int): String
isValidUrl(urlString: String): boolean
datalake_storage
I DatalakeFeeder
saveData(url: String, dataId: int)
C S3DatalakeFeeder
s3: S3Client
BUCKET_NAME: String
PROCESSED_FOLDER: String
UNPROCESSED_FOLDER: String
S3DatalakeFeeder()
saveData(url: String, dataId: int)
isFileInProcessedFolder(fileName: String): boolean

Figure 7: BookCrawler Class Diagram

This Class Diagram illustrates the main classes and their relationships in the system:

- **Main Class**: Entry point of the application, initializes the ‘Controller‘.
- **Controller Class**: Manages the download process by coordinating the ‘BookCrawler‘ and ‘DatalakeFeeder‘.
- **BookCrawler Class**: Handles the logic for downloading books using the ‘GutenbergCrawler‘.
- **GutenbergCrawler Class**: Implements the book crawling functionality and validates URLs.
- **DatalakeFeeder Interface**: Defines the method for saving data to the datalake.
- **S3DatalakeFeeder Class**: Implements the ‘DatalakeFeeder‘ interface, saving data to S3 and managing folder structure.

The diagram shows how components like the ‘Controller‘ interact with crawlers and data storage classes to handle book downloading and storage.

6.6.2 WordCounterDatamart Class Diagram

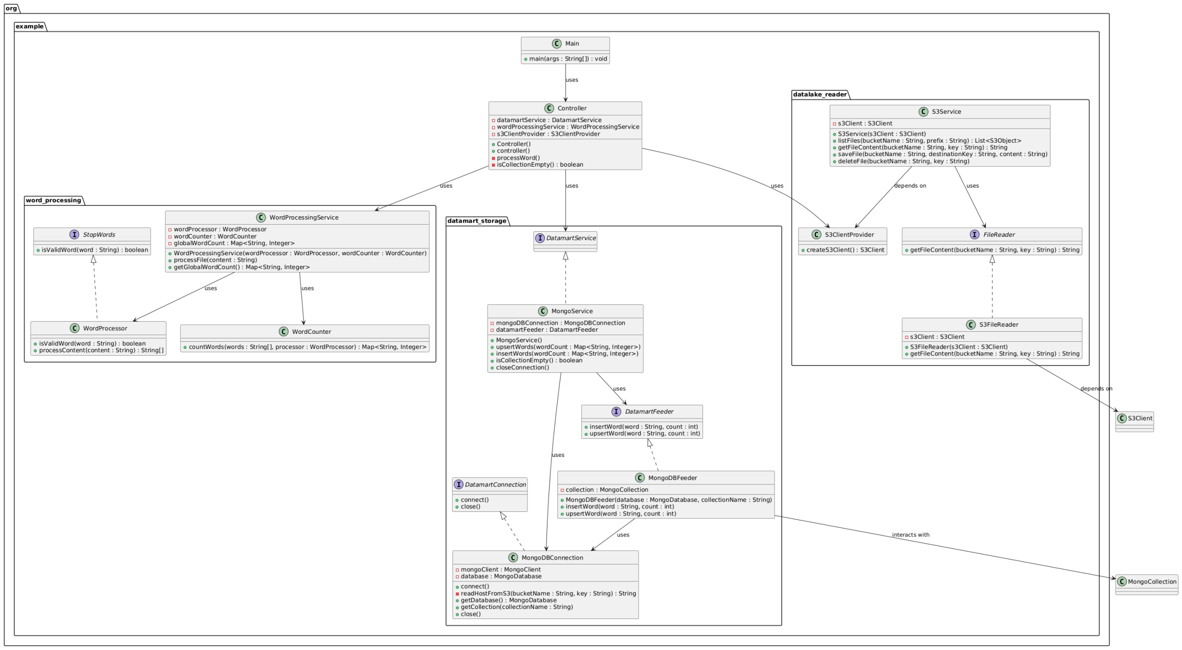
org
example
Main
main(args : String[]) : void
uses
Controller
datamartService : DatamartService
wordProcessingService : WordProcessingService
s3ClientProvider : S3ClientProvider
Controller()
controller()
processWord()
isCollectionEmpty() : boolean
uses
uses
uses
word_processing
StopWords
isValidWord(word : String) : boolean
WordProcessingService
wordProcessor : WordProcessor
wordCounter : WordCounter
globalWordCount : Map<String, Integer>
WordProcessingService(wordProcessor : WordProcessor, wordCounter : WordCounter)
processFile(content : String)
getGlobalWordCount() : Map<String, Integer>
uses
uses
WordProcessor
isValidWord(word : String) : boolean
processContent(content : String) : String[]
WordCounter
countWords(words : String[], processor : WordProcessor) : Map<String, Integer>
datamart_storage
DatamartService
MongoService
mongoDBConnection : MongoDBConnection
datamartFeeder : DatamartFeeder
MongoService()
upsertWords(wordCount : Map<String, Integer>)
insertWords(wordCount : Map<String, Integer>)
isCollectionEmpty() : boolean
closeConnection()
uses
uses
DatamartFeeder
insertWord(word : String, count : int)
upsertWord(word : String, count : int)
DatamartConnection
connect()
close()
MongoDBFeeder
collection : MongoCollection
MongoDBFeeder(database : MongoDatabase, collectionName : String)
insertWord(word : String, count : int)
upsertWord(word : String, count : int)
uses
MongoDBConnection
mongoClient : MongoClient
database : MongoDatabase
connect()
readHostFromS3(bucketName : String, key : String) : String
getDatabase() : MongoDatabase
getCollection(collectionName : String)
close()
interacts with
MongoCollection
datalake_reader
S3Service
s3Client : S3Client
S3Service(s3Client : S3Client)
listFiles(bucketName : String, prefix : String) : List<S3Object>
getFileContent(bucketName : String, key : String) : String
saveFile(bucketName : String, destinationKey : String, content : String)
deleteFile(bucketName : String, key : String)
depends on
uses
S3ClientProvider
createS3Client() : S3Client
FileReader
getFileContent(bucketName : String, key : String) : String
S3FileReader
s3Client : S3Client
S3FileReader(s3Client : S3Client)
getFileContent(bucketName : String, key : String) : String
depends on
S3Client

Figure 8: WordCounterDatamart Class Diagram

This Class Diagram represents the main components and their interactions within the system:

- **Main Class**: Entry point that initializes the ‘Controller‘.
- **Controller Class**: Coordinates the flow between the ‘DatamartService‘, ‘WordProcessingService‘, and ‘S3ClientProvider‘.
- **WordProcessing Classes**:
  - **WordProcessor**: Validates and processes words from the content.
  - **WordCounter**: Counts words and maintains their occurrences.
  - **StopWords**: Filters irrelevant words.
- **DatamartStorage Classes**:
  - **MongoService**: Manages word storage and retrieval in MongoDB.
  - **DatamartFeeder**: Handles insert and update operations in the database.
  - **MongoDBConnection**: Establishes and closes connections to MongoDB.
- **DatalakeReader Classes**:
  - **S3Service**: Manages interactions with AWS S3 for reading and saving files.
  - **S3ClientProvider**: Creates and provides S3 clients.
  - **S3FileReader**: Reads files from S3 buckets.

The diagram highlights how the system processes data, stores it in MongoDB, and interacts with AWS S3 for reading and writing files.

6.6.3 WordsGraph Class Diagram

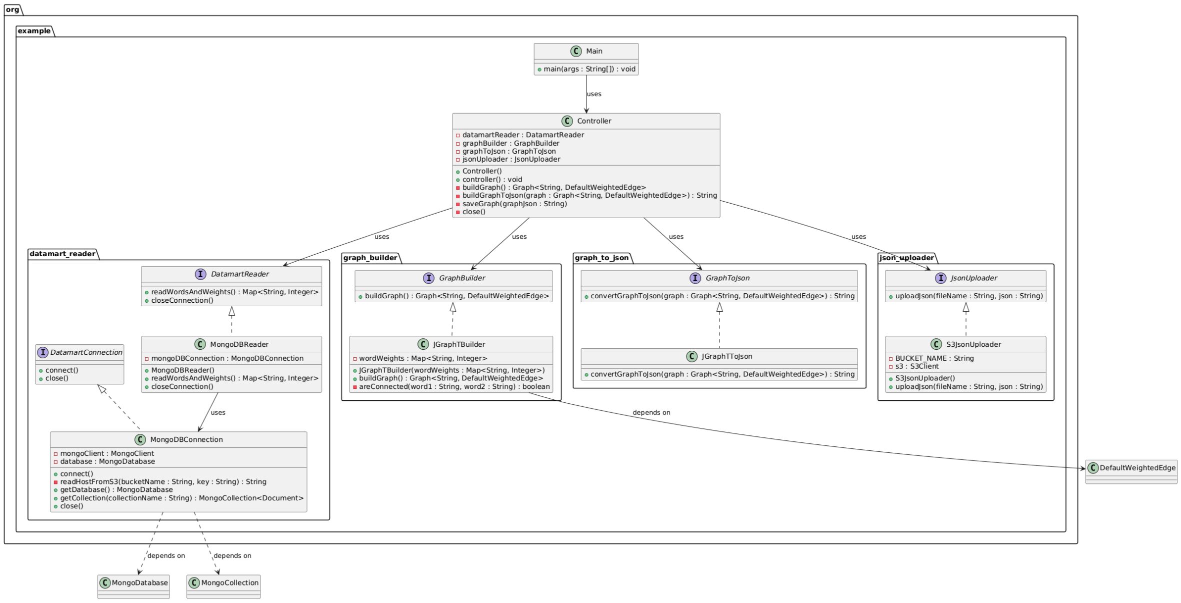
org
example
Main
main(args : String[]) : void
uses
Controller
datamartReader : DatamartReader
graphBuilder : GraphBuilder
graphToJson : GraphToJson
jsonUploader : JsonUploader
Controller()
controller() : void
buildGraph() : Graph<String, DefaultWeightedEdge>
buildGraphToJson(graph : Graph<String, DefaultWeightedEdge>) : String
saveGraph(graphJson : String)
close()
uses
uses
uses
uses
datamart_reader
DatamartReader
readWordsAndWeights() : Map<String, Integer>
closeConnection()
DatamartConnection
connect()
close()
MongoDBReader
mongoDBConnection : MongoDBConnection
MongoDBReader()
readWordsAndWeights() : Map<String, Integer>
closeConnection()
uses
MongoDBConnection
mongoClient : MongoClient
database : MongoDatabase
connect()
readHostFromS3(bucketName : String, key : String) : String
getDatabase() : MongoDatabase
getCollection(collectionName : String) : MongoCollection<Document>
close()
graph_builder
GraphBuilder
buildGraph() : Graph<String, DefaultWeightedEdge>
JGraphTBuilder
wordWeights : Map<String, Integer>
JGraphTBuilder(wordWeights : Map<String, Integer>)
buildGraph() : Graph<String, DefaultWeightedEdge>
areConnected(word1 : String, word2 : String) : boolean
graph_to_json
GraphToJson
convertGraphToJson(graph : Graph<String, DefaultWeightedEdge>) : String
JGraphTToJson
convertGraphToJson(graph : Graph<String, DefaultWeightedEdge>) : String
json_uploader
JsonUploader
uploadJson(fileName : String, json : String)
S3JsonUploader
BUCKET_NAME : String
s3 : S3Client
S3JsonUploader()
uploadJson(fileName : String, json : String)
depends on
DefaultWeightedEdge
depends on
depends on
MongoDatabase
MongoCollection

Figure 9: WordsGraph Class Diagram

This Class Diagram represents the system's key components and their relationships:

- **Main Class**: Entry point of the application, initializing the ‘Controller‘.
- **Controller Class**: Manages interactions between the ‘DatamartService‘, ‘WordProcessingService‘, and ‘S3ClientProvider‘.
- **WordProcessing Classes**:
  - **WordProcessor**: Processes and validates word content.
  - **WordCounter**: Counts and tracks word occurrences.
  - **StopWords**: Filters out irrelevant words.
- **DatamartStorage Classes**:
  - **MongoService**: Handles word storage and retrieval in MongoDB.
  - **DatamartFeeder**: Performs insert and update operations in the database.
  - **MongoDBConnection**: Manages database connections.
- **DatalakeReader Classes**:
  - **S3Service**: Reads and writes files from AWS S3 buckets.
  - **S3ClientProvider**: Creates and manages S3 clients.
  - **S3FileReader**: Reads specific files from S3.

The diagram shows how the system processes data, interacts with MongoDB for storage, and communicates with AWS S3 for file management.

6.6.4 APIGraph Class Diagram

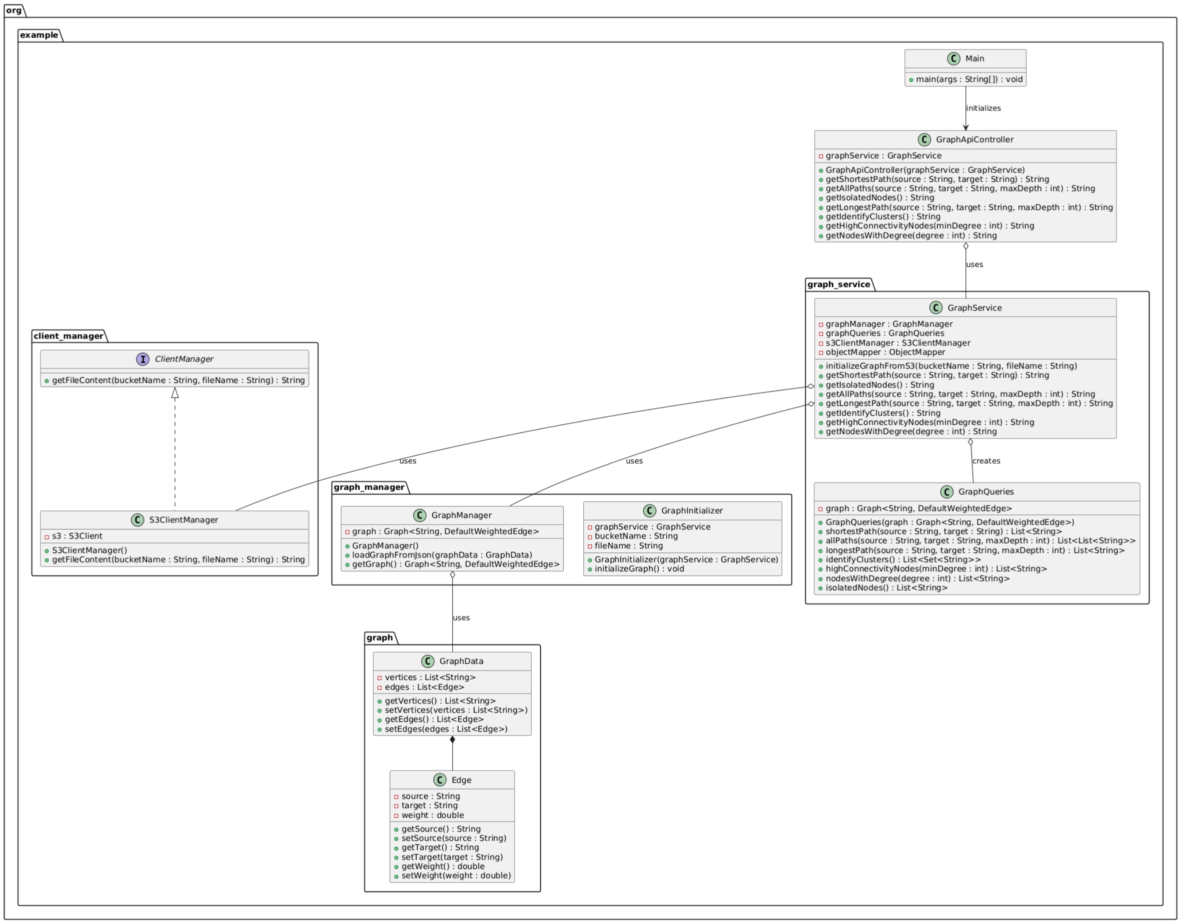
org
example
Main
main(args : String[]) : void
initializes
GraphApiController
graphService : GraphService
GraphApiController(graphService : GraphService)
getShortestPath(source : String, target : String) : String
getAllPaths(source : String, target : String, maxDepth : int) : String
getIsolatedNodes() : String
getLongestPath(source : String, target : String, maxDepth : int) : String
getIdentifyClusters() : String
getHighConnectivityNodes(minDegree : int) : String
getNodesWithDegree(degree : int) : String
uses
graph_service
GraphService
graphManager : GraphManager
graphQueries : GraphQueries
s3ClientManager : S3ClientManager
objectMapper : ObjectMapper
initializeGraphFromS3(bucketName : String, fileName : String)
getShortestPath(source : String, target : String) : String
getIsolatedNodes() : String
getAllPaths(source : String, target : String, maxDepth : int) : String
getLongestPath(source : String, target : String, maxDepth : int) : String
getIdentifyClusters() : String
getHighConnectivityNodes(minDegree : int) : String
getNodesWithDegree(degree : int) : String
creates
GraphQueries
graph : Graph<String, DefaultWeightedEdge>
GraphQueries(graph : Graph<String, DefaultWeightedEdge>)
shortestPath(source : String, target : String) : List<String>
allPaths(source : String, target : String, maxDepth : int) : List<List<String>>
longestPath(source : String, target : String, maxDepth : int) : List<String>
identifyClusters() : List<Set<String>>
highConnectivityNodes(minDegree : int) : List<String>
nodesWithDegree(degree : int) : List<String>
isolatedNodes() : List<String>
client_manager
ClientManager
getFileContent(bucketName : String, fileName : String) : String
S3ClientManager
s3 : S3Client
S3ClientManager()
getFileContent(bucketName : String, fileName : String) : String
uses
uses
graph_manager
GraphManager
graph : Graph<String, DefaultWeightedEdge>
GraphManager()
loadGraphFromJson(graphData : GraphData)
getGraph() : Graph<String, DefaultWeightedEdge>
GraphInitializer
graphService : GraphService
bucketName : String
fileName : String
GraphInitializer(graphService : GraphService)
initializeGraph() : void
uses
graph
GraphData
vertices : List<String>
edges : List<Edge>
getVertices() : List<String>
setVertices(vertices : List<String>)
getEdges() : List<Edge>
setEdges(edges : List<Edge>)
Edge
source : String
target : String
weight : double
getSource() : String
setSource(source : String)
getTarget() : String
setTarget(target : String)
getWeight() : double
setWeight(weight : double)

Figure 10: APIGraph Class Diagram

This Class Diagram represents the key classes and their interactions within the graph management system:

- **Main Class**: Entry point that initializes the ‘GraphApiController‘.
- **GraphApiController**: Handles API requests and interacts with the ‘GraphService‘ to provide graph-related functionalities.
- **GraphService**:
  - Manages the graph operations using the ‘GraphManager‘ and ‘GraphQueries‘.
  - Reads and initializes graphs from S3 using ‘ClientManager‘.
- **GraphManager**:
  - Handles graph data, including loading and retrieving nodes and edges.
  - Interacts with ‘GraphData‘ to manage graph structures.
- **GraphData**:
  - Stores the vertices and edges of the graph.
  - Provides methods to access and update graph components.
- **GraphQueries**: Implements various graph algorithms, such as shortest paths, node isolation, and clustering.

- **ClientManager**: Manages interactions with AWS S3 to retrieve graph data in JSON format.

The diagram highlights the system's flow from API requests to graph data retrieval and processing, showcasing the use of AWS S3 and internal graph structures.

7 Infrastructure Layer

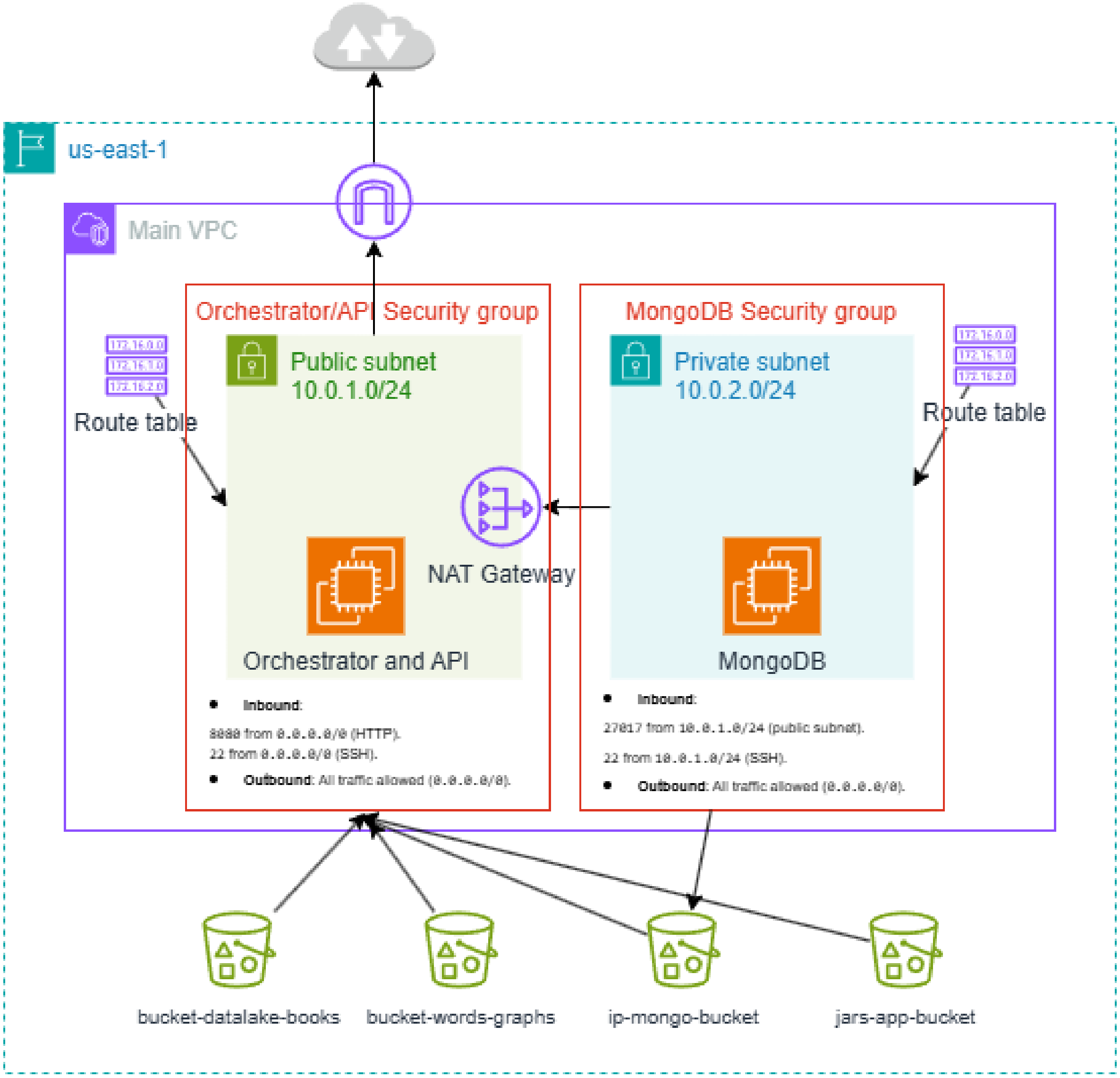

Figure 11: AWS Infraestructure

This diagram represents the AWS infrastructure setup for the application:

- **Main VPC**: The Virtual Private Cloud in the ‘us-east-1‘ region hosting the infrastructure.
- **MongoDB Security Group (Private Subnet)**:
  - Hosts MongoDB in a private subnet (‘10.0.2.0/24‘) with restricted inbound (port 27017) and outbound access via a NAT Gateway.
  - Ensures the database is accessible only from the orchestrator via a private connection.
- **Orchestrator/API Security Group (Public Subnet)**:
  - Hosts the orchestrator and API in a public subnet (‘10.0.1.0/24‘), allowing inbound traffic on ports 8080 and 22.
  - Communicates with MongoDB and S3 buckets to manage data flow.
  - In this architecture, an orchestrator has been implemented to run the project every day at 2 a.m.

- **Internet Gateway**: Provides internet access for the public subnet and NAT gateway.
- **S3 Buckets**:
  - ‘bucket-datalake-books‘: Stores raw and processed book data.
  - ‘bucket-words-graphs‘: Stores the word graph in JSON format.
  - ‘ip-mongo-bucket‘: Contains MongoDB’s private IP for connectivity.
  - ‘jars-app-bucket‘: Stores application JAR files for deployment.

The diagram highlights the secure interaction between the MongoDB instance in the private subnet, the orchestrator in the public subnet, and the S3 buckets for data storage and retrieval.

8 Justification of Selected Technologies

The following technologies were chosen for this project based on their suitability to address specific problems and meet the system’s objectives:

8.1 CI/CD

- **Terraform**: Used for Infrastructure as Code (IaC). Terraform was chosen for:
  - Automated and repeatable provisioning of cloud resources (e.g., EC2, S3, VPC).
  - Simplifies infrastructure management and versioning.
  - Integration with multiple cloud providers, including AWS.
- **GitHub**: Used for source code management and workflow. Its benefits include:
  - Efficient collaboration through version control.
  - Centralized issue tracking and documentation.
- **GitHub Actions**: Used to automate CI/CD pipelines directly from the GitHub repository.
  - Automation of testing, deployments, and other tasks.
  - Seamless integration with the GitHub ecosystem for a smooth developer experience.
- **Jacoco**: A tool for code coverage analysis. Its benefits include:
  - Automates the generation of coverage reports within CI/CD pipelines.
  - Ensures the code is well-tested before deployment.
  - Identifies areas of improvement in test coverage to enhance code quality.

8.2 Development

- **Java**: The main programming language used for developing the application logic and API. Java was selected due to:
  - Broad support community and libraries.
  - Strong performance for enterprise applications.
  - Compatibility with frameworks like Spring Boot and tools like JGraphT.
- **Spring Boot**: Used to build the API and application logic. Its benefits include:
  - Simplified configuration and dependency management.
  - Rapid development with production-ready features.
  - Compatibility with RESTful services and enterprise-grade applications.
- **MongoDB**: Selected as the database for storing word occurrences and their relationships due to:

  - Flexible schema to handle diverse data structures.
  - High performance in read/write operations.
  - Easy integration with Java applications.

- **JGraphT**: Used to create and manage the word graph in memory. It was chosen because:
  - The dataset is not excessively large (English words), making an in-memory approach more efficient.
  - Provides simplicity and speed for building and manipulating the graph without external storage overhead.
  - This option was preferred over using Neo4j, as the limited amount of data made it neither fast nor cost-effective.

- **JUnit**: Used for unit testing within the application. Its benefits include:
  - Standard tool for Java testing.
  - Facilitates the creation of repeatable and automated tests.
  - Improves code quality by ensuring proper functionality.

- **Swagger**: Provides a graphical interface for the API. Its benefits include:
  - Easy visualization and exploration of API endpoints.
  - Testing of API functionalities directly from the browser.
  - Enhances the developer experience by making the API easier to understand and use.

8.3 Other Technologies

- **AWS**: Cloud service provider used to host and manage the project's infrastructure. AWS services include:
  - **Amazon EC2**: Used to host the MongoDB database and application components.
    - Full control over the operating environment for dynamic services.
    - Scalability to handle variable workloads.
    - Security through Virtual Private Cloud (VPC) and security groups to control access.
  - **Amazon S3**: Used for storing books, word graphs, and other data.
    - High scalability and reliability for handling large datasets.
    - Cost-effective storage for static files.

These technologies collectively support the system's requirements for scalability, efficiency, security, and maintainability.