Gatino - Innovation Gateway for Swinburne Vietnam

Detailed System Design and Implementation Report

GROUP 1

Name	Position	Email	Phone
Ta Quang Tung	Team Leader, Client Liaison	104222196@student.swin.edu.au	0921426803
Nguyen Quang Huy	Development Manager	104169507@student.swin.edu.au	0968751524
Tran Hoang Hai Anh	Support Manager	104177513@student.swin.edu.au	0866899518
Phan Sy Tuan	Quality Manager	104072029@student.swin.edu.au	0888052003
Duong Quang Thanh	Planning Manager	104167828@student.swin.edu.au	0976663084

COS40006, COMPUTING TECHNOLOGY PROJECT B, SPRING 2025

Document Change Control

Version	Date	Authors	Summary of Changes
1.0	6/10/2024	Ta Quang Tung, Nguyen Quang Huy, Tran Hoang Hai Anh, Phan Sy Tuan, Duong Quang Thanh	Initial version of the document.
2.0	6/4/2025	Ta Quang Tung	Updated the document to better reflect the process in CTPB.

Document Sign Off

Name	Position	Signature	Date
Ta Quang Tung	Team Leader, Client Liaison	Quang Tung Ta	6/4/2025
Nguyen Quang Huy	Development Manager	Quang Huy Nguyen	6/4/2025
Tran Hoang Hai Anh	Support Manager	Hoang Hai Anh Tran	6/4/2025
Phan Sy Tuan	Quality Manager	Sy Tuan Phan	6/4/2025
Duong Quang Thanh	Planning Manager	Quang Thanh Duong	6/4/2025

Client Sign Off

Name	Position	Signature	Date
Dr. Le Minh Duc	Head of IT Department, Swinburne Vietnam Head of Innovation Lab, Swinburne Vietnam	Le, Minh Duc	6/4/2025
Organization: Innovation Lab, Department of Computer Science, Swinburne Vietnam			

Table of contents

1 - Introduction

This report provides a comprehensive look into the detailed design and implementation of the Gatino - Innovation Gateway platform, an interactive web application developed for Swinburne University Vietnam. Gatino aims to streamline the management, discovery, and demonstration of student innovation projects, offering a robust and scalable solution for students and faculty. By integrating advanced search and demo capabilities, Gatino supports a seamless user experience, enabling users to search projects with refined criteria and engage with real-time demonstrations across diverse project types, including web, Android and IoT.

This Detailed Design and Implementation Report aims to outline the architectural decisions, design patterns, and implementation strategies used to build Gatino. It details the client-server structure, component interactions, and modular deployment of the system, focusing on scalability, maintainability, and user-friendliness. Additionally, this report serves as a reference for developers, system architects, and stakeholders, providing insights into the design rationale, system verification, and the current state of implementation. It ensures that all design elements align with the functional and non-functional requirements outlined in the System Requirements Specification (SRS) and System Architecture Design documents.

1.1 - Overview

This report outlines the detailed design and implementation plan for the Gatino system, emphasizing the system's architecture, design decisions, and implementation progress. The document serves to bridge the requirements and high-level architecture defined in the System Architecture Design and Research Report with the actual development and operational setup of the Gatino platform. Each component of the system architecture is described with object-oriented (or other relevant) design principles to ensure modularity, reusability, and scalability. Additionally, the report includes justifications for key design choices, verification of the design against the System Requirements Specification (SRS), and an overview of the current state of implementation.

1.2 - Definitions, Acronyms, and Abbreviations

Term	Definition
API	Application Programming Interface, a set of functions allowing applications to access data or features.
API Gateway	A server that acts as an API front door, handling client requests and routing them to backend services.
JSON	JavaScript Object Notation, a lightweight format for data exchange.
RESTful API	Representational State Transfer API, an architectural style for networked applications.
SADRR	System Architecture Design and Research Report

SPA	Single-Page Application

1.3 - Assumptions and Simplifications

This detailed design makes several assumptions and simplifications to ensure the system's efficiency, scalability, and ease of implementation:

1. **Platform Accessibility**: It is assumed that users will access Gatino primarily via desktop or laptop devices with stable internet connections. The platform will also be responsive, but heavy reliance on mobile compatibility is outside the immediate scope.

2. **Concurrent Demonstrations**: Due to hardware limitations, the system assumes limited concurrency for Android and IoT demonstrations, with a queueing mechanism in place for single-user access in high-demand scenarios.

3. **Data Source Consistency**: It is assumed that all project metadata and files are complete and accurate at the time of upload. Simplifications have been made to handle missing data minimally without impacting system performance.

4. **System Modularity**: The service-oriented architecture allows each service to function independently. This approach assumes that API Gateway requests can efficiently manage load distribution across these services.

5. **External Libraries and Tools**: The design assumes continued support and stability of third-party libraries like Elasticsearch, Docker, and Apache Tika, which are crucial for search indexing, containerization, and document processing, respectively.

6. **User Familiarity with Search**: It is assumed that users are familiar with search engines like Google, meaning they can intuitively use Gatino’s search functionality without extensive training.

7. **Single Database Access Model**: Project data is split between a main ProjectDatabase and a dedicated SearchDatabase for efficient search operations. This assumes minimal impact on system performance from the split data structure and does not account for complex synchronization needs.

2 - System Architecture Overview

The system utilizes a client-server architecture, aligning with the client's vision for a web application with a distinct frontend and backend. This architecture enables a natural mapping where the frontend represents the client side and the backend represents the server side, making it compatible with the standards of the core Gatino team and fulfilling the Analyzability requirement.

The server side follows a service-oriented architecture, where complex backend operations are divided into a series of independent services that collaborate to fulfill functional requirements. These services are individually deployable, enhancing the system's Replaceability and Fault Tolerance. The API gateway pattern bridges the client and backend, hiding backend services from the client while enabling security measures like rate-limiting and request filtering. Many backend services are deployed using Docker, as shown in the Deployment diagram, providing Adaptability, Coexistence, and Replaceability.

Each component within this architecture is designed to support the system’s core features. For project search, users enter queries via the WebInterface, which sends requests to the API Gateway. This gateway forwards the request to the SearchService, which coordinates with the SearchEngine for project metadata and the ProjectDatabase for full project details, before returning the compiled results to the user.

In managing project data, the user requests to add, update, or delete project information via the WebInterface. The API Gateway forwards these requests to the ProjectManager, which updates the ProjectDatabase and publishes the updated information to the ProjectMessageQueue. The SearchDataExtractor then processes this message, extracting relevant data and updating the SearchDatabase.

To demonstrate web projects, users access the project URL through the WebInterface. This URL directs them to the API Gateway, which routes the request to the appropriate WebProjectDemonstrator deployed in a Docker container. Each WebProjectDemonstrator is programmed to handle specific project requests, providing customized demonstrations.

For Android project demonstrations), users initiate demos through the WebInterface, where requests are directed to either the EmulatedMobileDemonstrator or the PhysicalMobileConnector, depending on the type of Android demonstration selected.

IoT and AI project demonstrations are handled differently, as these projects produce data streams rather than standard displays. External sensors feed data into the system’s DataStreamDB. Users can access the demo URL via the WebInterface, which routes to the HeadlessDemonstrator. This component accesses data from the DataStreamDB to create an interactive or visual display for the user.

3 - Detailed System Design

The purpose of this section is to present the detailed system design for the software solution being developed for project Gatino. This design follows a Client-Server Architecture, which focuses on separating the system into two main components: the client which interacts with users, and the server which handles the processing of data handling and search algorithm, and business logic. We decided on this alternative approach since it is ideal for applications requiring distributed resources, and scalability like Gatino.

The **Client-Server Architecture** was chosen for this system for the following reasons:

- **Distributed System**: This architecture is highly effective for systems where different devices need to interact with a centralized backend, such as microservices.
- **Scalability**: The architecture allows for easy scalability as the server can handle many concurrent client requests, and additional server resources can be added to meet increasing demand.
- **Centralized Management**: The server acts as the central point for managing business logic, database access, and user authentication, simplifying maintenance and updates.
- **Security**: Sensitive operations, such as data storage, payment processing, and user authentication, are handled on the server, reducing exposure on the client side.
- **Cross-Platform Support**: The client can be a lightweight application (e.g., web interface, mobile app), while the server can support various types of clients by providing APIs and services.

3.1 - The Detailed Design and Justification

Below is our project class diagram, which is based on the Component Diagram of the SADRR. The bigger version of the diagram can be seen here.

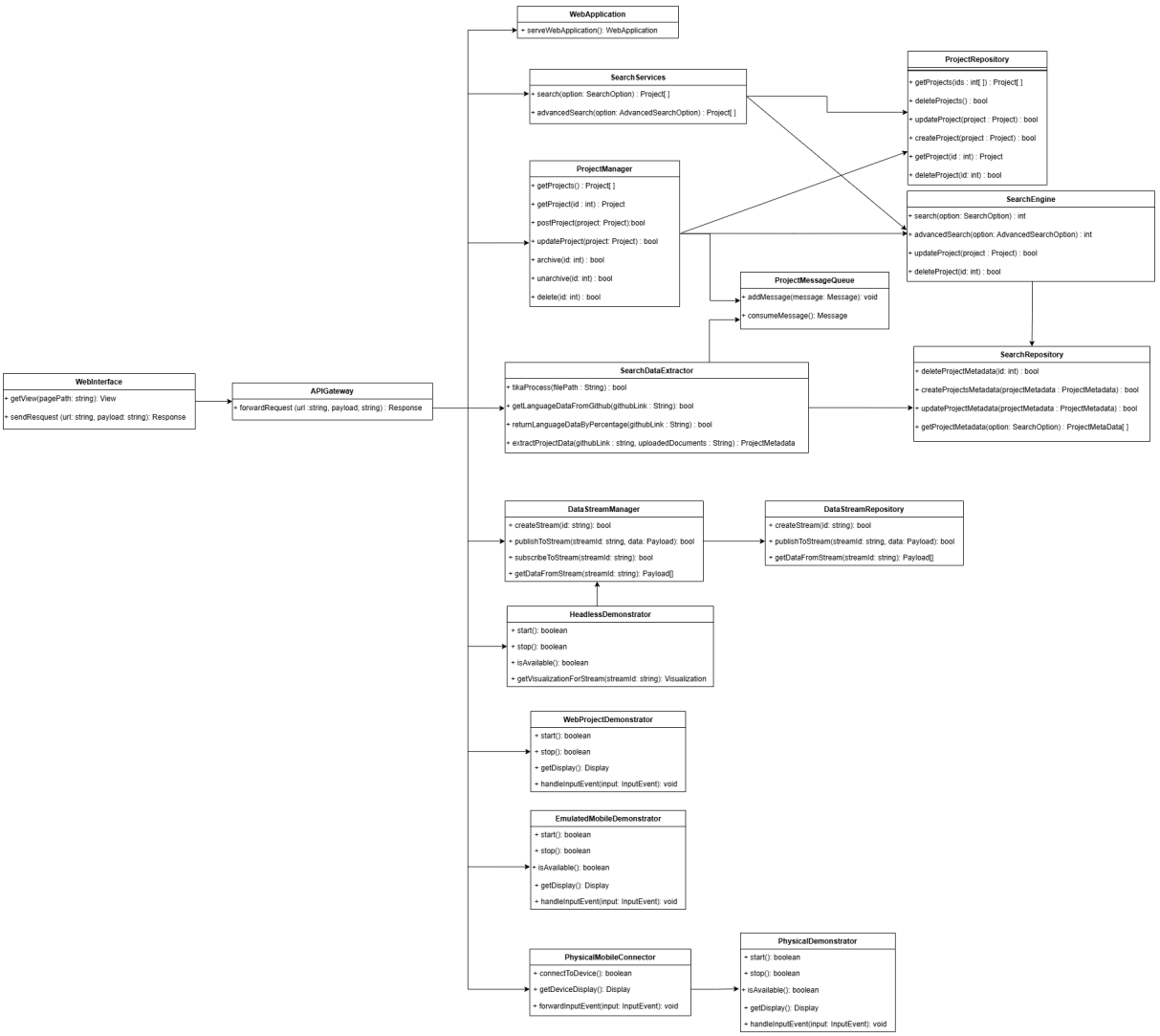


Fig 1: The system class diagram

This UML diagram outlines the structure of the Gatino Innovation Gateway system, designed with object-oriented programming principles in mind. It breaks down the system into various components, each with a clear purpose. At the core is the web application module, which acts as the main interface for users to interact with the system. All incoming requests are funneled through the APIGateway, which directs them to the right services or modules, ensuring everything runs smoothly and securely.

The SearchServices component handles the system's search functionality. It relies on supporting parts like the SearchEngine, SearchRepository, and SearchChatExtractor to process user queries and return relevant results. The ProjectManager module manages innovation projects, using tools like the ProjectMessageQueue to handle tasks in the background and the ProjectRepository to store project data. Similarly, the DataStreamManager oversees real-time data flows, with the DataStreamRepository serving as its storage backbone.

Several Demonstrator modules allow projects to be showcased in different ways. The HeadlessDemonstrator, WebpageDemonstrator, EmulatedMobileDemonstrator, and

PhysicalMobileConnector work together to display projects on web platforms, mobile emulators, and physical devices.

This diagram is just a conceptual design and serves as a starting point. Changes and adjustments will likely be made during implementation to fit the system's needs better.

3.2 - Design Verification

The following scenarios demonstrate how the design supports the core functionalities outlined in the use scenarios (from the SRS). It is important to note that these diagrams' attribute and method names are provisional and only intended to illustrate the scenarios. They will be refined as needed in the future.

Search for projects

The full version of the diagram can be seen here.

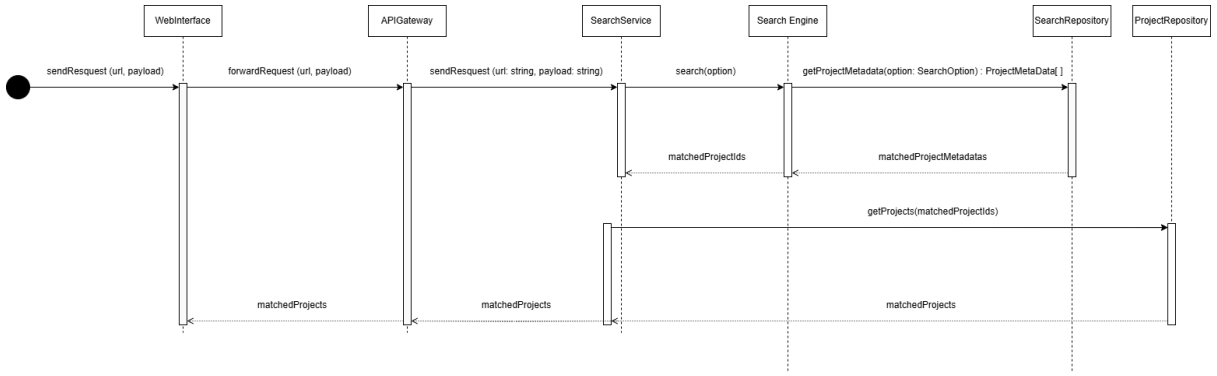


Fig 2: The sequence diagram of searching for projects

The process begins when the WebInterface sends a request to the APIGateway. The APIGateway forwards this request to the SearchService, which initiates the search operation. The SearchService first interacts with the SearchRepository to query the SearchDatabase for metadata that matches the user's request. This step retrieves a list of potential matches, including their associated project IDs.

Using the retrieved project IDs, the SearchService makes a subsequent request to the ProjectRepository. This step fetches detailed information about the projects corresponding to the matched metadata. Once the project details are retrieved, the SearchService sends them back to the APIGateway. The APIGateway then prepares the response and returns it to the WebInterface, allowing the user to view the relevant project details.

Add project search information

The full version of the diagram can be seen here.

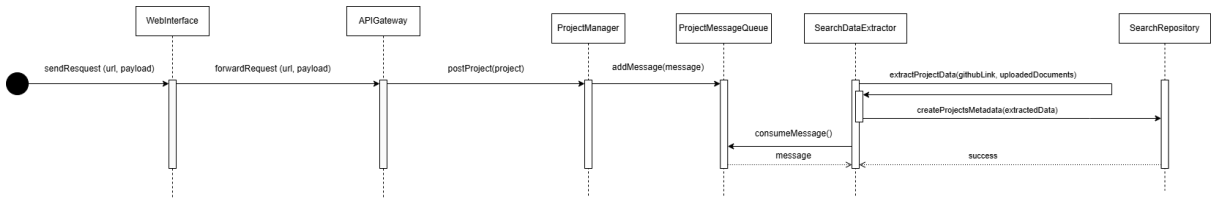


Fig 3: The sequence diagram of adding project search information

The flow begins when the WebInterface sends a request containing the project information to the APIGateway. The APIGateway then forwards this request to the ProjectManager. After that, the ProjectManager processes the received project details and produces a message, which is sent to the ProjectMessageQueue via the addMessage(message) method. This message serves as a trigger for consumers in the system.

After being triggered, SearchDataExtractor consumes the message from the ProjectMessageQueue using the consumeMessage() method. Upon receiving the message, the SearchDataExtractor begins extracting relevant project data. Once the project data is extracted, the SearchDataExtractor sends this extracted data to the SearchRepository, which stores the metadata in the search database.

Update project search information

The full version of the diagram can be seen here.

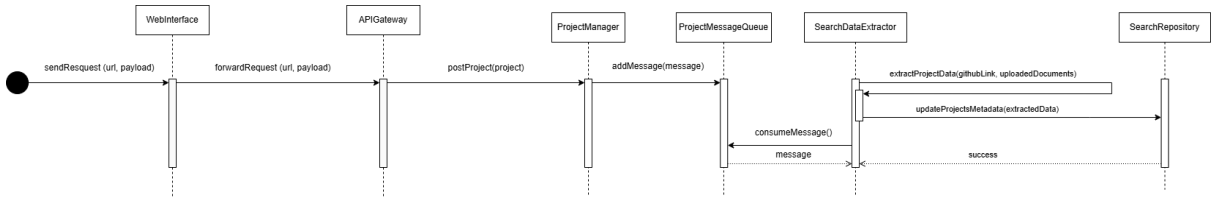


Fig 4: The sequence diagram of updating project search information

After the APIGateway receives a request from WebInterface, it forwards it to ProjectManager. ProjectManger passes it to SearchEngine, where the project metadata will be updated in the search database with searchRepository.

Delete project search information

The full version of the diagram can be seen here.

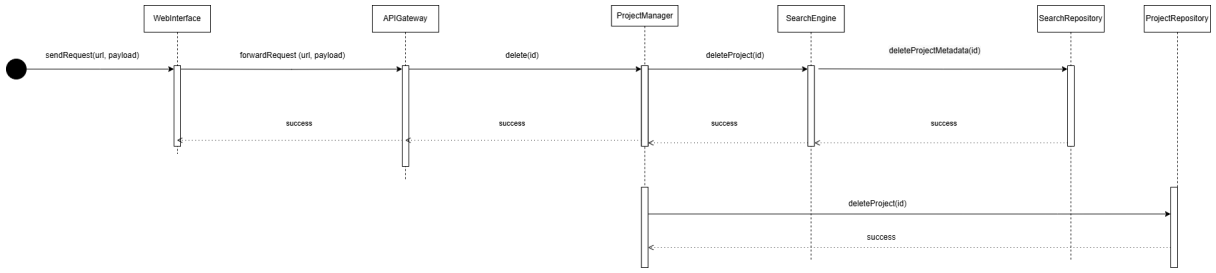


Fig 5: The sequence diagram of deleting project search information

The sequence diagram illustrates the process of deleting a project's metadata across multiple system components. The flow begins with the WebInterface sending a request to the APIGateway, initiating the deletion process. The APIGateway forwards this request to the ProjectManager. The ProjectManager first communicates with the SearchEngine, instructing it to delete the associated project metadata to ensure it no longer appears in search results. The SearchEngine, in turn, interacts with the SearchRepository to perform the actual deletion of the metadata. Simultaneously, the ProjectManager instructs the ProjectRepository to remove the project's data from the system's main repository. Once these deletions are completed, the process concludes, ensuring that all traces of the project are effectively removed from both search and core repositories.

Run a demonstration of a web project

The full version of the diagram can be seen here.

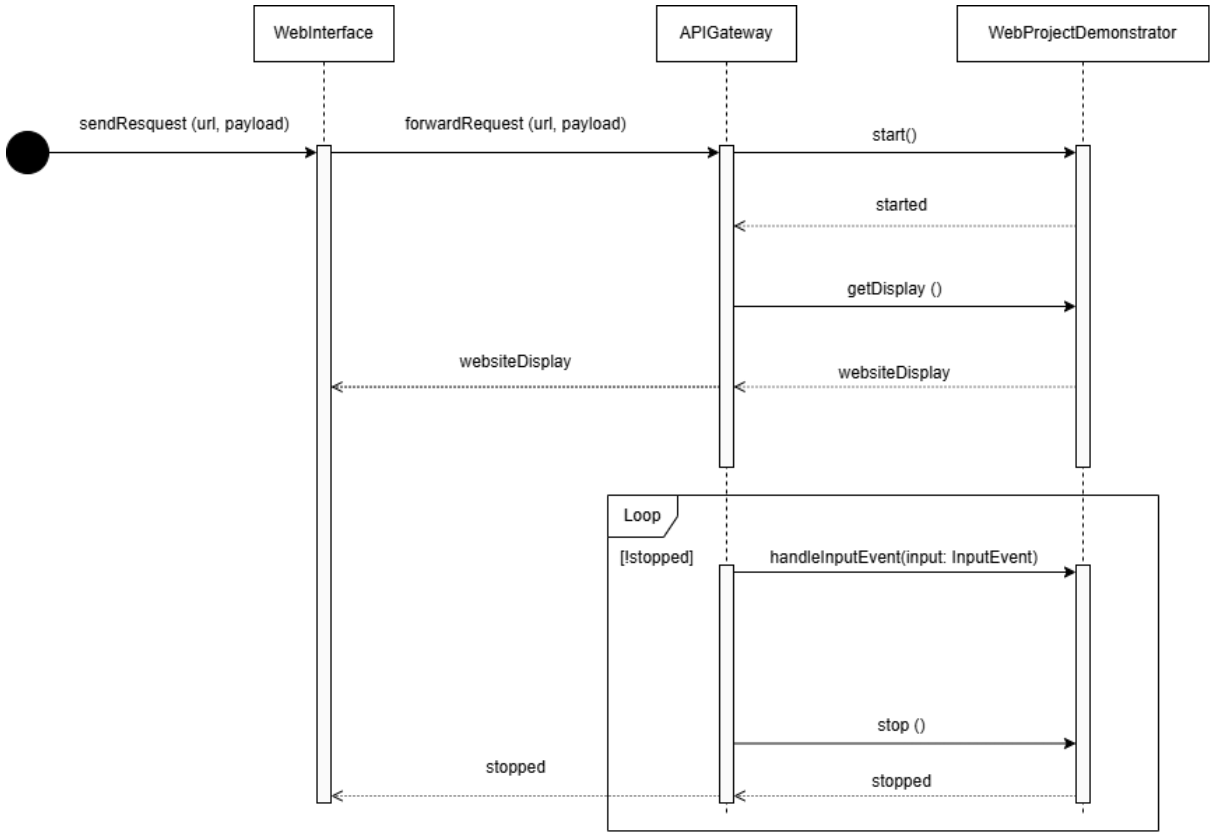


Fig 6: The sequence diagram of running a demonstration of a web project

This sequence diagram illustrates how the system handles a user request to demonstrate a web project. The process begins when the WebInterface sends a request to the APIGateway, where the request will be forwarded to WebProjectDemonstrator. Once the request reaches the WebProjectDemonstrator, it processes the request by retrieving the necessary project data and displaying the website display back to the audiences. After that, it keeps running in a loop to keep getting the input events from users when they interact with the system and should be stopped when receiving a signal. After completing the required

actions, the WebProjectDemonstrator sends the results back to the APIGateway. The APIGateway formats or finalizes the response before forwarding it to the WebInterface, which then presents the results to the user.

Run a demonstration of an Android project

The full version of this diagram can be seen here.

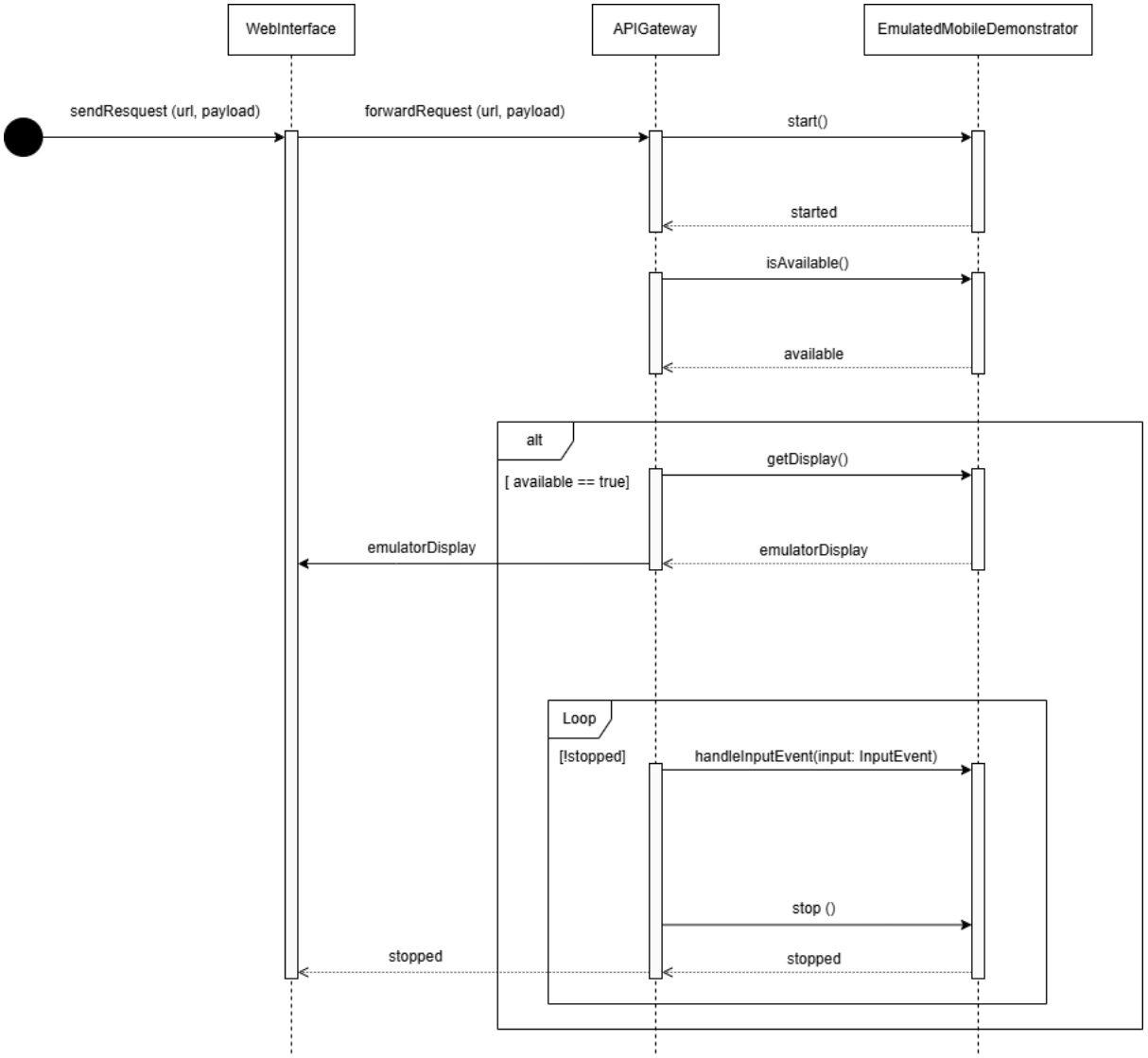
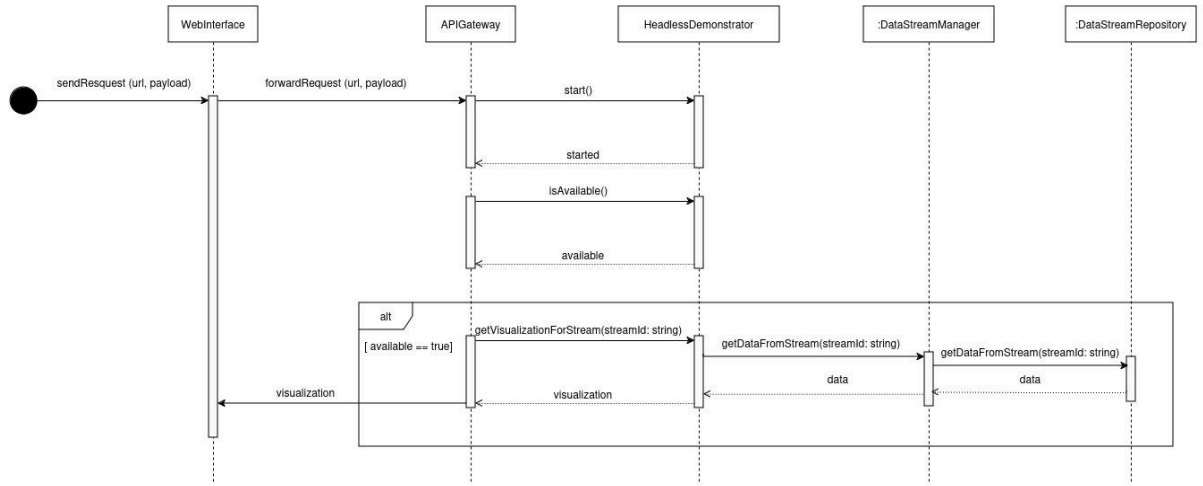


Fig 7: The sequence diagram of running a demonstration of an android project

This sequence diagram illustrates the process of emulating a mobile demonstration based on user interactions via the WebInterface. The flow begins when the WebInterface sends a request to the APIGateway. The APIGateway forwards this request to the EmulatedMobileDemonstrator, which is responsible for executing the emulation process. The diagram incorporates a condition check to check for emulator availability. If the emulation is available, the process enters a loop block, indicating repetitive actions performed by the EmulatedMobileDemonstrator. These actions could include simulating user interactions, or updating the WebInterface with ongoing status updates.

Run a demonstration of an IoT project

The full version of this diagram can be seen here.



The sequence diagram outlines the flow of interactions among WebInterface, APIGateway, HeadlessDemonstrator, DataStreamManager, and DataStreamRepository. The process begins when the WebInterface sends a request to the APIGateway, which then forwards it to the HeadlessDemonstrator for further processing. The HeadlessDemonstrator delegates data-related operations to the DataStreamManager. At this point, the flow enters a conditional (alt) block. If a specific condition is met, the DataStreamManager interacts with the DataStreamRepository to retrieve or update the necessary data. Once the repository operation is complete, the data flows back through the DataStreamManager to the HeadlessDemonstrator. The HeadlessDemonstrator processes the response and sends it to the APIGateway, which then delivers it back to the WebInterface. This flow ensures efficient handling of requests, with flexibility to process conditions dynamically and manage data operations seamlessly.

4 - Implementation

The code for the project is available on https://github.com/pine04/ctp-group-1.

Mapping from Classes to Non-OO Software Elements

The class diagram presented in the previous section is primarily used to validate the software design. However, we will not be following it exactly method-by-method because the technologies we use are not object-oriented (e.g.: React, Express, Elasticsearch, etc.) Instead, the classes and methods specified in the diagram will be mapped into non-OO software elements. The following tables present the mapping for each class.

Class: WebInterface	
Role: Presents the web user interface through which the user can use the app.	
Software element: A single-page application running in the user's browser.	
Method	**Software element**
getView(pagePath: string): View	A route in the single page application that renders the view for a certain page.
sendRequest(url: string, payload: string): Response	An API call made by the single-page application to request data from the server or to perform an action on the server.

Class: APIGateway	
Role: Receives requests from the WebInterface and forwards them to the correct backend service.	
Software element: Apache APISIX, running on port 9080.	
Method	**Software element**
forwardRequest (url :string, payload; string) : Response	A route configured on APISIX that forwards a given request to the correct destination.

Class: WebApplication	
Role: Serves the single-page application.	
Software element: A web server running on port 3000 that serves the React frontend.	
Method	**Software element**
serveWebApplication(): WebApplication	The default route of the web server.

	Requesting this route will return the whole SPA.

Class: SearchService	
Role: Receives search requests sent by the user and forwards them to the search engine.	
Software element: An Express router that contains the search routes.	
Method	**Software element**
search(option: SearchOption) : Project[ ]	The `GET /search` route of the router.
advancedSearch(option: AdvancedSearchOption) : Project[ ]	The `GET /advanced-search` route of the router.

Class: ProjectManager	
Role: Handles project CRUD requests sent by the user.	
Software element: An Express router that contains the project management routes.	
Method	**Software element**
getProjects() : Project[ ]	The `GET /projects` route of the router.
getProject(id : int) : Project	The `GET /projects/:id` route of the router.
postProject(project: Project):bool	The `POST /projects` route of the router.
updateProject(project: Project) : bool	The `PUT /projects/:id` route of the router.
archive(id: int) : bool	The `POST /projects/:id/archive` route of the router.
unarchive(id: int) : bool	The `POST /projects/:id/unarchive` route of the router.
delete(id: int) : bool	The `DELETE /projects/:id` route of the router.

Class: ProjectRepository
Role: Stores the primary project data.

Software element: A MySQL database.	
Method	**Software element**
getProjects(ids : int[ ]) : Project[ ]	SQL queries to the database.
deleteProjects() : bool	
updateProject(project : Project) : bool	
createProject(project : Project) : bool	
getProject(id : int) : Project	
deleteProject(id: int) : bool	

Class: SearchEngine	
Role: Performs the searching and indexing.	
Software element: Elasticsearch, running on port 9200.	
Method	**Software element**
search(option: SearchOption) : int	The `GET /<target>/_search` route of Elasticsearch.
advancedSearch(option: AdvancedSearchOption) : int	The `GET /<target>/_search` route of Elasticsearch.
updateProject(project : Project) : bool	The `POST /<index>/_update/<_id>` route of Elasticsearch.
deleteProject(id: int) : bool	The `DELETE /<index>/_doc/<_id>` route of Elasticsearch.

Class: SearchDataExtractor	
Role: Extracts search data for the uploaded project.	
Software element: A continuously-running process on the server.	
Method	**Software element**
tikaProcess(filePath : String) : bool	Apache Tika, running on port 9998.
getLanguageDataFromGithub(githubLink : String): bool	A request to the GitHub API.

returnLanguageDataByPercentage(githubLink : String) : bool	A method that computes the percentage based on the data returned by getLanguageDataFromGithub.
extractProjectData(githubLink : string, uploadedDocuments : String) : ProjectMetadata	Calls the above 3 methods.

Class: SearchRepository	
Role: Store the projects' metadata for searching	
Software element: ElasticSearch	
Method	**Software element**
deleteProjectMetadata(id: int) : bool	Internal to Elasticsearch
createProjectsMetadata(projectMetadata : ProjectMetadata) : bool	
updateProjectMetadata(projectMetadata : ProjectMetadata) : bool	
getProjectMetadata(option: SearchOption) : ProjectMetaData[ ]	

Class: ProjectMessageQueue	
Role: Decouples the project upload process from the search data extraction process.	
Software element: Apache Kafka, running on port 9092.	
Method	**Software element**
addMessage(message: Message): void	The `send` method of Kafka's Producer API.
consumeMessage(): Message	The `eachMessage` method of Kafka's Consumer API.

Class: DataStreamManager
Role: Allows other systems to create, produce, and subscribe to data streams.
Software element: Mosquitto MQTT broker, running on port 1883, working in conjunction with an Express router.

Method	**Software element**
createStream(id: string): bool	Publishing or subscribing to a topic is enough to create a stream.
publishToStream(streamId: string, data: Payload): bool	Publishing data to an MQTT topic through the MQTT client library.
subscribeToStream(streamId: string): bool	Subscribing to an MQTT topic through the MQTT client library.
getDataFromStream(streamId: string): Payload[]	The `GET /streams/:id` route of the router.

Class: DataStreamRepository	
Role: Stores the data for each data stream.	
Software element: A MySQL database.	
Method	**Software element**
createStream(id: string): bool	SQL queries to the database.
publishToStream(streamId: string, data: Payload): bool	
getDataFromStream(streamId: string): Payload[]	

Class: HeadlessDemonstrator	
Role: Displays a given data stream in the form of interactive visualizations.	
Software element: A web page that renders interactive visualizations.	
Method	**Software element**
start(): boolean	A script that starts the web server to handle demo requests.
stop(): boolean	A script that stops the web server and does any necessary cleanup.
isAvailable(): boolean	The server maintains a queue of waiting requests. The demo is not available to requests that have to wait.
getVisualizationForStream(streamId: string): Visualization	A page that renders the interactive visualization.

Class: WebProjectDemonstrator	
Role: Enables a user to view and interact with a web project in operation.	
Software element: Docker containers running the web projects.	
Method	**Software element**
start(): boolean	Issue a command to start the project's Docker container
stop(): boolean	Issue a command to stop the project's Docker container
getDisplay(): Display	Making a request to the Docker container will return a web page that can be rendered in the user's browser.
handleInputEvent(input: InputEvent): void	API requests that the web projects make to its backend when the user interacts with the page.

Class: EmulatedMobileDemonstrator	
Role: Enables a user to view and interact with a mobile application via an emulated mobile device.	
Software element: Docker Android, running on port 6080.	
Method	**Software element**
start(): boolean	Issue a command to start the Docker Android running on port 6080
stop(): boolean	Issue a command to stop the Docker Android container
isAvailable(): boolean	The server maintains a queue of waiting requests. The demo is not available to requests that have to wait.
getDisplay(): Display	Requesting the Docker Android container will return the Android emulator's screen, which can be rendered in the user's browser.
handleInputEvent(input: InputEvent): void	API requests that the emulator makes to its backend when the user interacts with the screen.

Class: PhysicalMobileConnector	
Role: Enables a user to connect to the physical Android demonstrator.	
Software element: Apache Guacamole (which consists of guacd on port 4822 and guac-web on port 8081).	
Method	**Software element**
connectToDevice(): boolean	Guacamole internal method to perform an RDP connection to a remote virtual machine
getDeviceDisplay(): Display	Requesting the Apache Guacamole container will return the remote virtual machine's screen, which can be rendered in the user's browser.
forwardInputEvent(input: InputEvent): void	API requests that the Apache Guacamole forwarded to PhysicalDemonstrator when the user interacts with the screen.

Class: PhysicalDemonstrator	
Role: Enables a user to view and interact with an Android project running on a physical device.	
Software element: A virtual machine running a XRDP server and Scrcpy that is also connected to a physical Android device (phone or tablet).	
Method	**Software element**
start(): boolean	Script to turn on the VM and possibly XRDP and Scrcpy.
stop(): boolean	Script to turn off the VM and possibly XRDP and Scrcpy.
isAvailable(): boolean	The server maintains a queue of waiting requests. The demo is not available to requests that have to wait.
getDisplay(): Display	Scrcpy will return the display of the Android device, which is then returned by XRDP to the Apache Guacamole web interface.
handleInputEvent(input: InputEvent): void	Apache Guacamole sends the request to the XRDP server, which forwards it through Scrcpy to the Android device. The device will respond to this input.

Current Implementation Progress

The development of Gatino has progressed steadily, with key components at varying stages of completion. The API Gateway is fully operational, enabling secure and efficient routing of client requests to backend services. The search module, powered by Elasticsearch, is functional for basic search queries, and work is ongoing to refine advanced filters. Project management features, such as uploading, updating, and deleting projects, are partially implemented, with an interactive user interface to support these functionalities. Demonstration services for web projects have been Dockerized and are fully operational, while Android project demonstrations are in progress with a queueing mechanism under development. The frontend interface has a basic structure, supporting project uploads and search capabilities, with additional interactive features planned for future iterations.

Chosen Technologies and Tools

Gatino leverages a modern tech stack to meet the demands of scalability, maintainability, and performance. The frontend is developed with React.js, providing a responsive user interface for uploading projects, searching, and running demonstrations. The backend is built with Node.js and Express.js, enabling robust API development. Elasticsearch is integrated for full-text search and advanced query functionalities, while Apache Tika extracts metadata from uploaded documents to enrich search results. Demonstration services for web and mobile projects are deployed using Docker, offering isolated and scalable environments. MySQL serves as the primary database for project metadata. Version control is handled through GitHub, and task management is streamlined with Jira to support Agile development processes.

Implementation Strategies

Search Module

The search module is powered by Elasticsearch, providing fast and accurate search capabilities. Upon project upload, metadata is indexed in real-time, with Apache Tika extracting relevant information from documents. Users can perform basic text-based searches, and work is ongoing to integrate advanced filters, such as searching by programming language or technology used. To optimize search performance, future steps include implementing synonym and stemming support and enhancing Elasticsearch configurations for handling large-scale datasets.

Demonstration Module

The demonstration module enables users to interact with projects in real-time. Web project demonstrations are deployed in Dockerized containers, accessible via URL-based interactions. Android project demonstrations use Scrcpy and Apache Guacamole, with a queueing mechanism to manage concurrent access. For IoT and AI projects, the system supports data-streaming visualizations using placeholder APIs, with additional visualization tools under development. Next steps include refining the queueing mechanism and integrating advanced visualization capabilities for IoT/AI data streams.

API Gateway

The API Gateway is fully implemented and ensures secure communication between the frontend and backend services. It provides role-based access control for students, lecturers, and administrators while implementing rate-limiting to prevent system overload. Future enhancements include request caching to reduce response times for frequently accessed data.

Frontend

The frontend, built with React.js, provides an initial interface for uploading projects, performing basic searches, and viewing results. While responsive design is partially implemented, future updates will include advanced search filters, interactive controls for project demonstrations, and user-friendly error messages. Plans also involve adding onboarding tutorials to improve the user experience for first-time users.

Backend Infrastructure

The backend infrastructure employs Dockerized services for search, demonstration, and project management. RabbitMQ facilitates real-time communication between services, ensuring efficient synchronization. To improve fault tolerance, plans include implementing health checks and auto-restart policies for Docker containers.

Challenges and Mitigation

The implementation phase has encountered several challenges, such as network latency between services, limited concurrency for Android demonstrations, and potential data synchronization issues between databases. To address these challenges, caching mechanisms and asynchronous message handling are being integrated to mitigate network latency. For Android demonstrations, the team is exploring scaling options for Dockerized emulators and implementing inactivity timeouts to free resources. Data synchronization between the ProjectDatabase and SearchDatabase will be ensured through periodic reconciliation and robust error-handling.

Deployment Strategy

The deployment of Gatino utilizes Dockerized environments for consistency across development, testing, and production stages. The codebase is version-controlled via GitHub, with a CI/CD pipeline managed using GitHub Actions to automate testing and deployment. Multi-container deployment is orchestrated with Docker Compose, and the production server is hosted on Swinburne Vietnam’s cloud infrastructure. These strategies ensure a streamlined deployment process and facilitate the smooth integration of future features.