|  |  |
| --- | --- |
| **https://lh4.googleusercontent.com/wBuKw10rOYdL8oz5hckyZcuT90n_HKEvK4yyVG6WqBPBh27g71Mt0KWAG_3wGy7YrJXUp2maH1st5U3pRILWjGk_YIpum9osMjUhRoWqFvQX3E5067iw-oqzFo5YNmiH3KtnfhIH**  Diagram animation and visualisation Environment  Software Architecture Document | Group 02  DIT029 Software Architecture for Distributed Systems  University of Gothenburg  AUTHORS  Boyan Dai Erik Laurin Elaine Qvarnström Joacim Eberlén Justinas Stirbys Shaun McMurray  CONTANCT  [davegroup02@gmail.com](mailto:davegroup02@gmail.com) |

Table of Contents

[2 Revision History 3](#_Toc501041520)

[3 Introduction 4](#_Toc501041521)

[4 System Purpose 5](#_Toc501041522)

[4.1 Context 5](#_Toc501041523)

[4.2 External Libraries 6](#_Toc501041524)

[5 Architectural drivers 8](#_Toc501041525)

[5.1 Functional Requirements 8](#_Toc501041526)

[5.2 Quality Requirements 9](#_Toc501041527)

[5.3 Architectural Tactics 10](#_Toc501041528)

[5.3.1 Availability 10](#_Toc501041529)

[5.3.2 Usability 10](#_Toc501041530)

[5.3.3 Security 10](#_Toc501041531)

[5.3.4 Modifiability 10](#_Toc501041532)

[5.3.5 Reliability 11](#_Toc501041533)

[5.4 Quality Attribute Scenarios 11](#_Toc501041534)

[5.5 Constraints 13](#_Toc501041535)

[6 Structure 14](#_Toc501041536)

[6.1 Overview 14](#_Toc501041537)

[6.2 Components 19](#_Toc501041538)

[6.3 Interfaces 23](#_Toc501041539)

[7 Use Case View 26](#_Toc501041540)

[7.1 Use Case Scenarios 26](#_Toc501041541)

[7.2 Use Case Diagram 28](#_Toc501041542)

[8 Logical View 29](#_Toc501041543)

[8.1 Instructor Registration 29](#_Toc501041544)

[8.1.1 Architectural Significance 29](#_Toc501041545)

[8.2 Upload Diagram 30](#_Toc501041546)

[8.2.1 Architectural Significance 31](#_Toc501041547)

[8.3 Join Room 31](#_Toc501041548)

[8.3.1 Architectural Significance 32](#_Toc501041549)

[8.4 Animate Diagram 32](#_Toc501041550)

[8.4.1 Spawn Containers 32](#_Toc501041551)

[8.4.2 Simulate Diagram 33](#_Toc501041552)

[9 Process View 35](#_Toc501041553)

[9.1 Upload Diagram 35](#_Toc501041554)

[9.2 Initiate Diagrams 36](#_Toc501041555)

[9.3 Simulate Diagram 38](#_Toc501041556)

[10 Development View 39](#_Toc501041557)

[10.1 Layered Overview 39](#_Toc501041558)

[10.2 Individual Packages 40](#_Toc501041559)

[11 Physical View 47](#_Toc501041560)

[12 Conceptual Framework 48](#_Toc501041561)

[12.1 Domain Lexicon 48](#_Toc501041562)

[13 Architectural risks 50](#_Toc501041563)

[13.1 Risk Table 50](#_Toc501041564)

[13.2 Risk description 50](#_Toc501041565)

[13.2.1 Amazon Web Service Hours 50](#_Toc501041566)

[13.2.2 Fault Recovery 51](#_Toc501041567)

[13.2.3 Container Overload 51](#_Toc501041568)

[13.2.4 MQTT Broker Failure 51](#_Toc501041569)

[13.2.5 Database Failure 51](#_Toc501041570)

[13.2.6 Message Delivery 51](#_Toc501041571)

[14 Conclusion 52](#_Toc501041572)

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 09-10-2017 | 0.1 | Functional and quality requirements | Group02 |
| 26-10-2017 | 0.2 | Initial physical view for DAVE | Group02 |
| 02-11-2017 | 0.3 | Initial development view for DAVE | Group02 |
| 16-11-2017 | 0.4 | External Libraries | Group02 |
| 08-12-2017 | 1.0 | First complete draft for SAD document | Group02 |
| 10-12-2017 | 1.1 | Version 5 of component view for section 3.1 | Group02 |
| 11-12-2017 | 1.2 | Use case diagram for section 4.2 and Conclusion | Group02 |
| 12-12-2017 | 1.4 | Use Case View and Logic View sections added | Group02 |
| 13-12-2017 | 1.5 | Second draft for SAD document | Group02 |
| 14-12-2017 | 1.6 | Added architectural risks | Group02 |
| 16-12-2017 | 1.8 | Final SAD version | Group02 |

# Introduction

**Purpose**

This document is intended to describe the architectural overview of Diagram Animation and Visualization Environment or DAVE for short. The document contains various architectural views that collectively depict significant architectural decisions of which DAVE is constructed upon. In addition, the document also includes the architectural drivers that the system was designed based on.

**Audience**

Due to the technical level of which this document possesses, the intended readers are people that either is in the field of software engineering or fields that are closely related.

Furthermore, stakeholders include, but not exclusively, the development team, instructors and students of DAVE as well as universities and people working therein.

**Scope**

DAVE was developed in the project course DIT029 Software Architecture for Distributed Systems and this document is set to describe the final and intermediate results of DAVE.

The system has two predefined primary actors which the system is built around, the student and the instructor. The instructor, an authorized user, may create so called rooms where they can add diagrams. The student, an unauthorized user, on the other hand, may only select the rooms they would like to explore.

# System Purpose

## Context

DAVE provides an initiative way of illustrating UML diagrams. The diagrams are visualized in a 3D environment enabling the users of DAVE to easier get a thorough understanding of the diagrams and their intra, as well as inter, relationships. Exploring diagrams in a 3D environment eradicates the abstract and intangible sense that may otherwise arise when examining a regular static 2D UML diagram. This innovative approach to diagrams is aimed towards improve the students’ understanding of UML, by providing physical and tangible objects for the student to interact and learn from.  
 Effectively, DAVE replaces the need to look at aforementioned static 2D UML diagrams and step by step imagining the interactions and flows. Instead, the application offers animated interactions in a chronological fashion for sequence diagrams and deployment diagrams. With class diagrams the user is capable of exploring class relationships by navigating a city build based on the diagram as blueprint.

In order, to experience the 3D renditions an Instructor first needs to upload a data file with relevant information. A data flow diagram was produced to visualize the data flow between DAVE and outside forces.

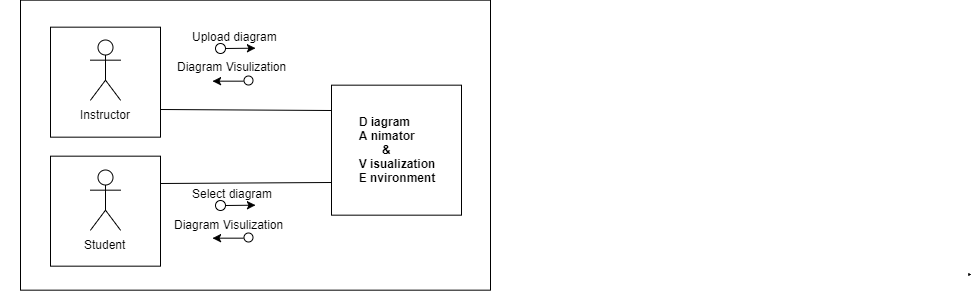


Figure 1 – Data Flow Diagram

A simple domain model was created to show DAVE’s solution for diagram simulation. Furthermore, it improves the understanding of data flow and the interactions between different system parts.

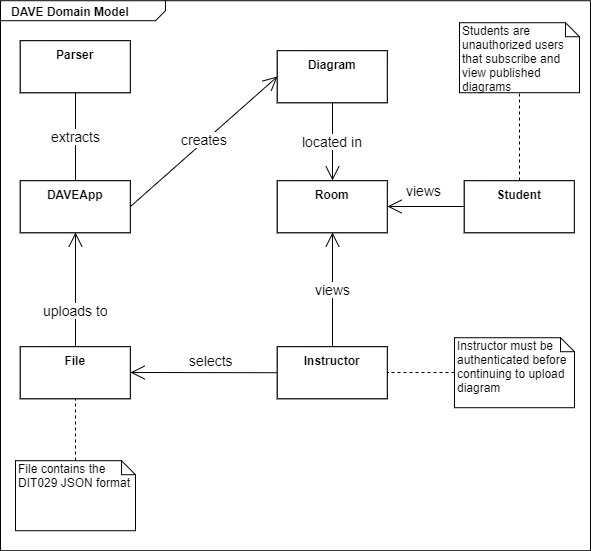


Figure 2 - Dave Domain Model

## External Libraries

DAVE was built using Unity engine, Docker, Erlang and Python. To achieve the functionality that DAVE has, several external libraries were used. This section contains the aforementioned libraries and where to find them.

|  |  |
| --- | --- |
| **Name** | **Where to find** |
| UnityStandaloneFileBrowser: Unity package providing a native file browser for Unity standalone platforms | https://github.com/gkngkc/UnityStandaloneFileBrowser |
| Json.Net: JSON framework for .NET | https://www.newtonsoft.com/json |
| Doors Free: Unity package providing openable doors and a Player object that has the ability to move around. | https://www.assetstore.unity3d.com/en/#!/content/38694 |
| Simple A\*: Unity package providing path finding algorithm. | https://www.assetstore.unity3d.com/en/#!/content/6385 |
| Unity3d\_MQTT: Unity package providing a MQTT client | https://github.com/vovacooper/Unity3d\_MQTT |
| Docker-py: A Python library for the Docker Engine API | https://github.com/docker/docker-py |
| Eclipse Paho MQTT Python: A Python library providing a MQTT client | https://github.com/eclipse/paho.mqtt.python |
| Docker: software providing containers | https://www.docker.com/ |
| Mosquitto: message broker that implements the MQTT protocol | https://mosquitto.org/ |
| Rethink.DB: distributed document-oriented database | https://www.rethinkdb.com/ |
| JSX: erlang application for consuming, producing and manipulating JSONs | https://github.com/talentdeficit/jsx |
| Asynchronous Erlang MQTT Client for subscribing to broker and forward files | https://github.com/emqtt/emqttc |

# Architectural drivers

Both functional and quality requirements were used by the development team for the purpose of narrowing in further the project scope and to act as guidelines for what DAVE must be capable.

## Functional Requirements

Functional requirements were used bring into fruition the previously mentioned domain model (See section 4.1). However, time constraints proved to be too great and not all the functional requirements were managed to be implemented. The team used MoSCoW notation to prioritize the task importance. The priority is distinguished between Musts, Shoulds, Coulds and Won’ts. Must is used to express features that are required to be fully operational for the final demo. Should is used to express significant features, although implementation deadline is not set in stone. Could is used to express features that would provide more value to the application, but are not necessarily implemented, due to budgetary constraints and restrictions, such as time and available developers. Won’t is used to keep track of features that will not be implemented. Causes for won’ts may include task vagueness, product owner refusal or functions were no longer necessary.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Requirement** | **Description** | **Status** | **Priority** |
| FR1 | Upload JSON | DAVE must provide an option for the authorized user to upload a file in JSON format | Implemented | Must |
| FR2 | Animate JSON | The system must animate diagram based on the provided JSON | Implemented | Must |
| FR3 | Simulate Sequence Diagram | The system must simulate uploaded JSON sequence diagrams | Implemented | Must |
| FR4 | Registration | The system should support user account creation | Implemented | Should |
| FR5 | Login | The system should support a sign in functionality, by retrieving relevant information from a database | Implemented | Should |
| FR6 | Make room | The system should allow authorized users to create a room where their diagrams will be displayed | Implemented | Should |
| FR7 | Join room | DAVE should provide an option to join a room and see its diagrams | Implemented | Should |
| FR8 | Data Storage | DAVE should support storage of username and password | Implemented | Shoud |
| FR9 | Multiple Diagrams | The system should support having multiple diagrams animated simultaneously | Implemented | Should |
| FR10 | Alter JSON | The system could provide an option to alter the displayed diagram, the changes being saved to a JSON file | Not implemented | Could |
| FR11 | Export JSON | The system could provide an option to export the altered diagrams to JSON file | Not implemented | Could |
| FR12 | Invite to room | DAVE could allow room creators to invite other people to join their created room | Not implemented | Could |
| FR13 | Send Messages | The system could allow different users in the same room to send messages to each other | Not implemented | Could |
| FR14 | Display Messages | DAVE could support for the messages to be displayed for correct users | Not implemented | Could |
| FR15 | Save Diagram | DAVE could be able to save the diagrams, as JSONs, in the remote database, linked to the account that saves it | Not implemented | Could |
| FR16 | Load Diagram | DAVE could be able to load a diagram saved in the database | Not implemented | Could |

Table 2 - Functional Requirements

## Quality Requirements

Quality requirements were defined, so the development team would not only produce features, but produce them well.

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Quality** | **Description** | **Significance** |
| QR1 | Availability | * The system should automatically recover from outages |  The system will be deployed on different cloud instances which has incorporated fault detection and recovery tactics. In case of failure, a spare instance takes over the failed one automatically |
| QR2  and  QR3 | Usability | * Task completion rate * Time to complete tasks |  The system must provide intuitive controls and be easy to use |
| QR4 | Security | * Encrypted credentials in data storage |  The system will protect user data via the login feature |
| QR5 and QR6 | Modifiability | * Developers must be able to expand the supported diagram types * Developers must be able to easily maintain and refine current components |  Separation of concerns   Low coupling   High cohesion   Modularity |
| QR7 | Reliability | * The system should render diagrams consistently |  All components that are associated with the rendering will be constructed to perform consistently |

Table 3 - Quality Requirements

## Architectural Tactics

### Availability

DAVE being deployed on Amazon Web Services (AWS), requires to be accessible and available over large period. Therefore, it requires to recover from outages (QR1). The team implemented a virtual machine or container pool. The container pool is used to provide additional containers for simulation if necessary, for example if a container in use has crashed. Via this function the team is able to provide fault tolerance. Moreover, to achieve (QR1) fault detection and fault recovery tactics were implemented. These tactics were achieved by fine tuning AWS configuration settings, more specifically by enabling Amazon CloudWatch that automatically detects and recovers from potential failures. After conducting research, it was believed that CloudWatch is quintessential tool for dealing with fault recovery and tolerance, which are properties of distributed systems. This continued to motivate the use of AWS and Amazon CloudWatch within the project. Unfortunately, due to the nature of AWS and Amazon CloudWatch policies, not much control is granted over their recovery and tolerance tactics.

### Usability

For any application ease of use is a vital aspect of the application. For DAVE ease of use was measured using task completion rate (QR2) and time taken to complete tasks (QR3). To achieve these quality requirements the principles of consistency and instant gratification were incorporated. Consistency was implemented in term of displayed user interfaces. Regardless of environment, diagram and user type, meaning student or instructor, similar user interfaces were presented for the purpose of consistency and familiarity. DAVE users are also presented with a set of controls that carry out their purpose instantaneously, thus granting the users instant gratification.

### Security

Security is vital to the developed system; therefore, it was decided to encrypt all instructor credentials(QR4). For this purpose, the development team decided to incorporate user authentication and authorization. To ensure that these tactics are met, instructors are required to complete a simple account creation. Additionally, passwords have a set of constraints that they are required to fulfill. In continuation, maintaining data confidentiality was identified as a desired tactic, which was fulfilled by data encryption.

### Modifiability

DAVE’s financial prospects were also considered during the early stages of the project, resulting in a very nice business plan. To improve financial benefit, it was decided that DAVE must be able to expand to simulate and animate other diagrams besides UML (QR5). To achieve this goal, it is important that DAVE is developed in a way that is easy to maintain and expand system components (QR6). For this reason, the development team identified modifiability as one of the major quality requirements. The development team strived to incorporate several tactics; separation of concerns, increasing cohesion and reducing coupling. However, successfully incorporating these tactics proved to be difficult. It is believed by the team that the team managed to achieve an acceptable level of cohesion, however the separation of concerns and coupling proved to be difficult, due to changing architectural styles mid-way the project (See section 6.1). A devised solution for the future, was to expand the code reviews not only to code produced, but class interactions as well as updating UML diagrams each time a new feature is incorporated.

### Reliability

The intended scenario for DAVE’s use would be a classroom environment, with an instructor uploading and displaying a diagram to all their students. Due to the learning environment it is important that the students are displayed with the same diagram as the instructor, therefore it is vital that DAVE is reliable with its diagram animation (QR7). Another reason for focusing on this quality requirement, was the chosen style, publish-subscribe. The main drawback of which, is message delivery issues, which might present itself as a risk (See section 13). No architectural tactic was identified, instead the team decided to only display a message if the message has been sent and received to the MQTT client containing the same data. Additionally, the use quality of service for further confirmation that the processes and data are executed and delivered correctly.

## Quality Attribute Scenarios

To further refine the quality requirements, quality attribute scenarios were created. In continuation, these scenarios improve the understanding of what part of DAVE is responsible for monitoring the quality attribute realization and DAVE’s capabilities, which are expressed as Artifacts and Responses in accordance

Reasoning for refining Availability to a quality attribute scenario was motivated by fault tolerance seen in distributed systems and the initial project idea, The applied tactics for availability result in a fault tolerant system. Based on the initial project idea, DAVE was to be deployed as a web application. Availability was seen as a main focus point for a web application, as the application would not provide much value if it were unavailable through large periods of time. Due to limitation in the WebGL framework, DAVE could not be deployed through such.

|  |  |
| --- | --- |
| **Scenario ID** | S1 |
| **Quality Attribute** | Availability |
| **Source** | Student |
| **Stimulus** | Student uses DAVE application, DAVE tries to access MQTT broker |
| **Artifact** | AWS ElasticCloud |
| **Environment** | Under heavy load, 100 students are subscribing to 10 different rooms |
| **Response** | MQTT broker should return DAVE’s request |
| **Response Measure** | Available 99.9% of the time through the course of a month |

Table 4 - Availability Quality Scenario

The decision to refine Usability was straight forward. It focused on providing more value to the students and instructors, by implementing an easy to use interface. The quality attribute scenario focused specifically on instructors, for the sake of simplifying their classes. It was thought that the students would have an easier time to follow the lecture, thus learning more, with simplistic controls.

|  |  |
| --- | --- |
| **Scenario ID** | S2 |
| **Quality Attribute** | Usability |
| **Source** | Instructor |
| **Stimulus** | Create a room with two diagrams |
| **Artifact** | DAVE |
| **Environment** | Runtime |
| **Response** | DAVE should respond by:   * Creating the room based on the instructor’s name and chosen room name * Add the chosen diagrams to the room |
| **Response Measure** | 100% tasks completed, and time taken to complete tasks is no longer than 1 minutes |

Table 5 - Usability Quality Scenario

Security was identified as a focus point. With the quality attribute scenario, the developers attempted to protect user integrity. The scenario is used to determine a way to maintain the users’ privacy and confidentiality.

|  |  |
| --- | --- |
| **Scenario ID** | S3 |
| **Quality Attribute** | Security |
| **Source** | Instructors |
| **Stimulus** | Instructors try to create account |
| **Artifact** | DAVE’s data storage |
| **Environment** | Normal operation |
| **Response** | DAVE’s data storage should respond by:   * Verify if credentials meet set requirements * Encrypt new data * Send data to data storage |
| **Response Measure** | 100 % of the passwords in the AccountStorage are hashed according to SHA256 standard. |

Table 6 - Security Quality Scenario

Modifiability importance could be attributed to the financial aspect of the project. During the early project stages the team had discussed on using Software as a Service model, to create a paid monthly subscription. During this discussion, the team raised a couple of points related to making DAVE financially viable and how they intersect with modifiability. The first, increase the type of UML diagrams DAVE can animate/simulate. Doing so would provide more value to existing customers and strengthen DAVE’s notoriety and existing customer base. The second, target customers. By having a modifiable application, the team can expand to animate diagrams outside of UML, by doing so increasing the target customer base and financial gains. Due to these reasons, modifiability was chosen to refine into a Quality Attribute scenario.

|  |  |
| --- | --- |
| **Scenario ID** | S4 |
| **Quality Attribute** | Modifiability |
| **Source** | Developers |
| **Stimulus** | Developers expands DAVE to animate state machine diagrams |
| **Artifact** | DAVE |
| **Environment** | Design time |
| **Response** | Developers should respond by:   * Quickly locating places in architecture to be changed * Make modification without affecting other modules * Test modifications * Deploy changes |
| **Response Measure** | Expanding DAVE to animate state machine diagram, should not take longer to implement than four weeks. |

Table 7 - Modifiability Quality Scenario

## Constraints

In order to experience DAVE to its fullest capabilities it must be run under certain constraints. These conditions are required to be met for the best possible user experience:

* Uploaded JSON diagrams must follow the DIT029 diagram convention format
* Internet connection speed must be of at least 1 Mbit/s down and 0.25 Mbit/s up for the system to work as intended.
* A keyboard is required for player object movement and a mouse is recommended for the best user experience.
* DAVE requires the following system specifications to run
  + OS: Windows XP SP2+ or Mac OS X 10.9+
  + Graphics card: DX9 (shader model 3.0) or DX11 with feature level 9.3 capabilities.
  + CPU: SSE2 instruction set support.
* Only 3 diagram types are able to be animated/simulated; class, sequence and deployment
* Only a single diagram of the same type can be uploaded simultaneously
* It is presumed that diagrams correspond to the same system if they are uploaded simultaneously

# Structure

## Overview

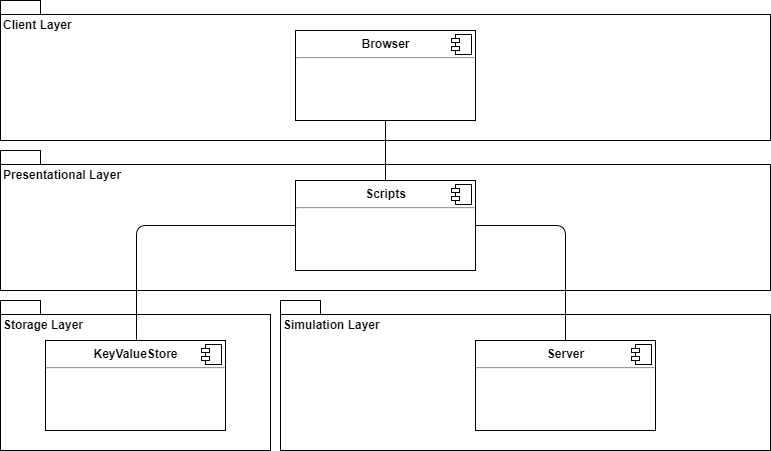


Figure 3 - Component Diagram Version 1

The initial plan was using a tcp server written in erlang with a netstring protocol to animate the diagrams according to the uploaded JSON. The rationale behind this choice was a design decision based on simplicity, and a solution which could give us basic functionality with the ability to expand on it. It contains a layered architecture and a client server style connecting the presentational layer with the storage and simulation layer.

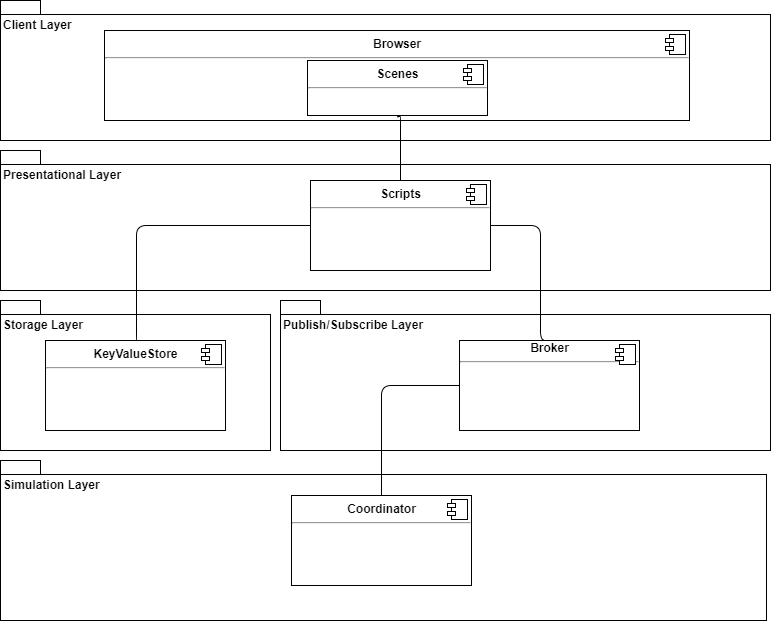


Figure 4 - Component Diagram Version 2

Quite early we noticed some impediments using a client-server pattern to

render a simulation of the data provided. One of these impediments were the unnecessary difficulty of implementation such as message passing using IP addresses. During the second iteration the major change made to the structure was the chosen style. Using a publish-subscribe style for simulating the diagrams combined with a client-server style for the storage layer. This made the end solution easier to implement using a MQTT broker for publishing and subscribing to our planned classrooms for diagram rendition. As a last step in the execution was the coordinator which was supposed to be responsible for simulation, but with this approach we would have lost the distributed part of the system. A solution containing different virtual machines was needed according to current requirements. The style change could be attributed to several benefits gained from the publish-subscribe style. Scalability was brought up when thinking of the main environment DAVE will run in. Said environment was identified as a university classroom, where several dozens of students would subscribe to a specific diagram by an instructor. This led the team to believe that DAVE would perform better with a publish-subscribe style than a regular client-server. In continuation, it was believed that publish-subscribe would be more beneficial for a distributed system. More specifically, the team considered distributed system properties, when choosing a new style. The aforementioned scalability corresponds to one of the properties of distributed systems. Additionally, parallel operations are easier to implement with a pub/sub style, which would help meeting concurrency, another distributed system property.

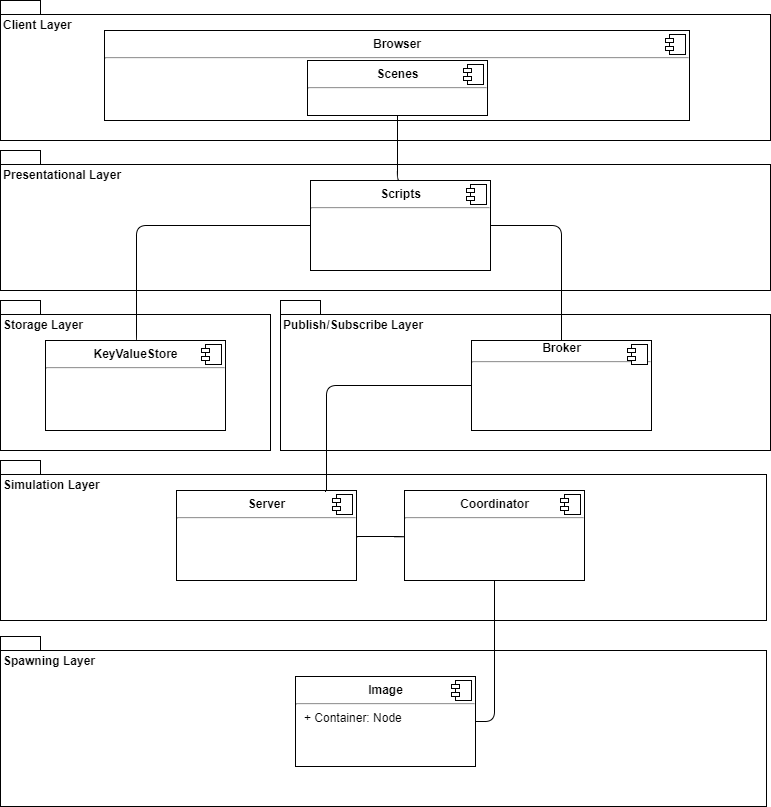


Figure 5 - Component Diagram Version 3

In the third version of the structural diagram the publish/subscribe pattern is taking shape. Looking at the simulation layer: a server handling messages published to the broker and a coordinator passing information to a virtual machine image. Using this style, we can spawn images that executes code responsible for rendering objects in real time. One technical decision remained to be made, which technology to use for the virtualization.

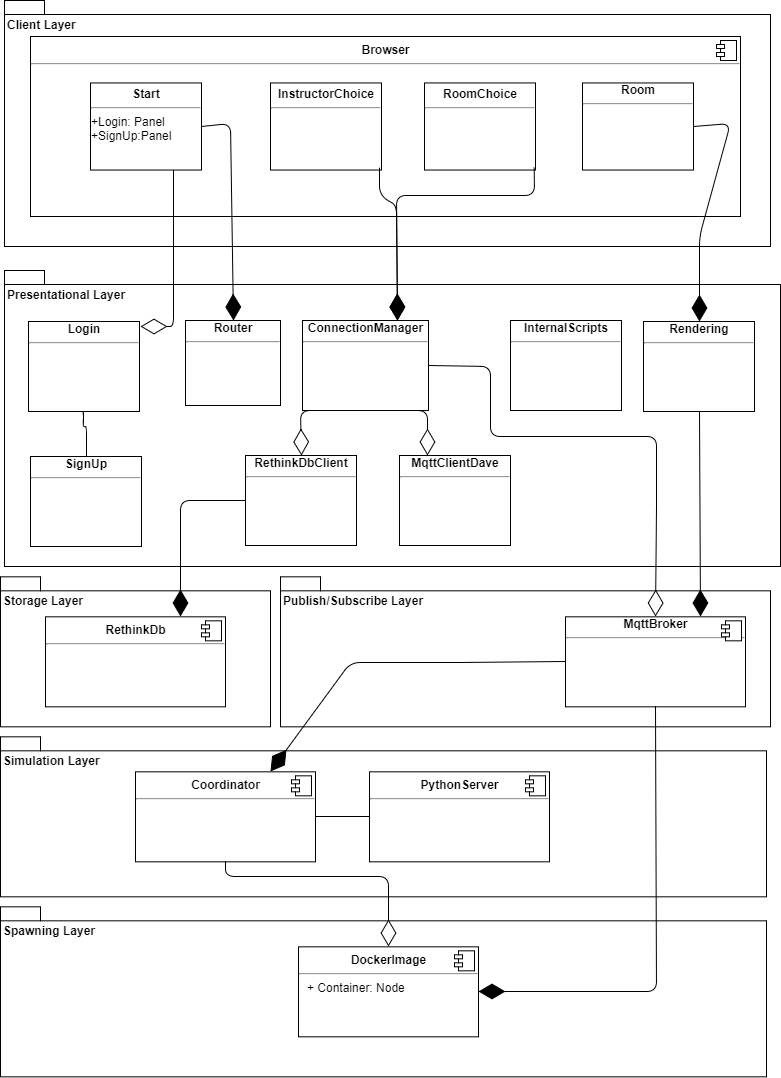


Figure 6 - Component Diagram Version 4

The fourth iteration define Unity scenes as components in the browser view. In the presentational layer the components are composed of scripts controlling the presentational logic. The structure did not change dramatically during this step. Design decisions made this iteration, were rewriting the rendering of objects and using Docker images for virtualization of the data provided.

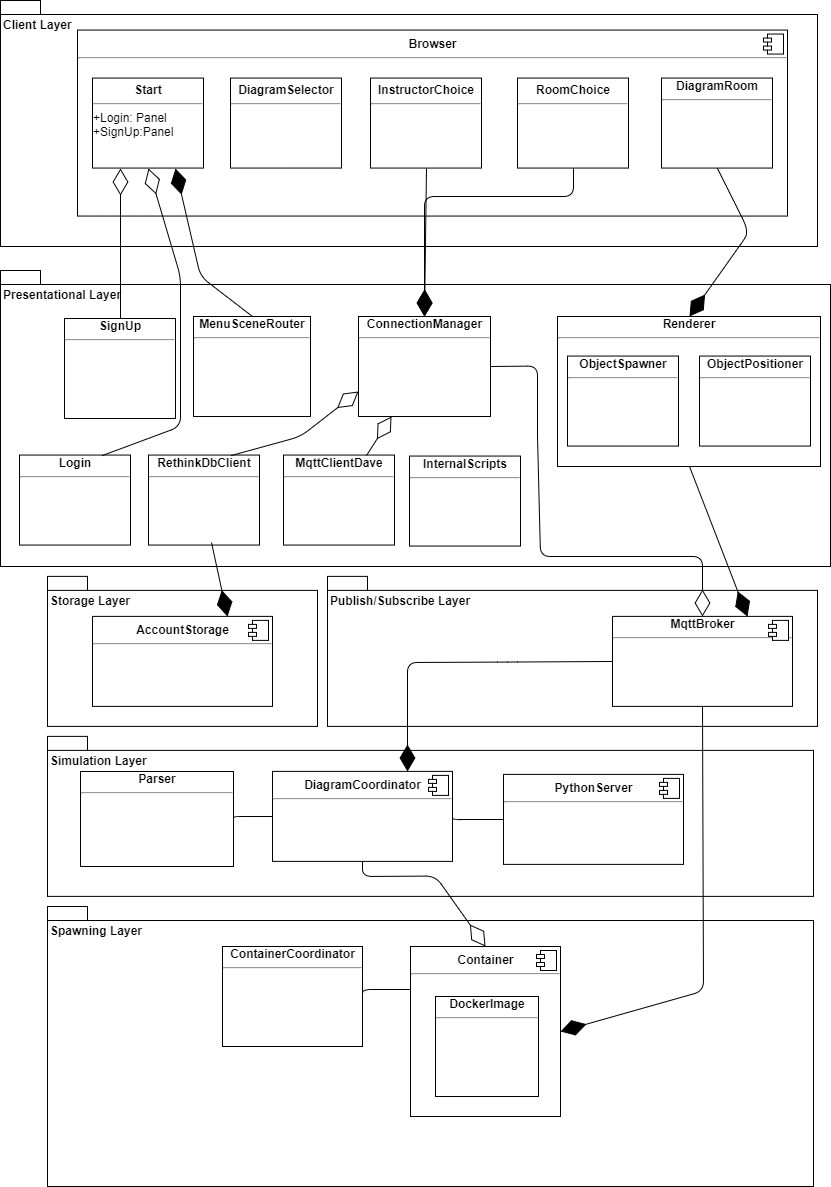


Figure 7 - Component Diagram Final Version

The fifth iteration saw several changes to the component names to hopefully reflect their responsibilities more accurately. The Router became MenuSceneRouter to indicate that it is the main menu whose paths are being alternated. The Coordinator became the DiagramCoordinator. Additionally to the name changes, several important components were added. The Renderer class got expanded with ObjectPositioner and ObjectSpawner. A Parser was added to the Simulation layer and a ContainerCoordinator to the Spawning layer. Last change, was rectifying a mistake in the Spawning layer. The DockerImage was made a child of the Container.

## Components

|  |  |
| --- | --- |
| **Component** | SignUp |
| **Responsibilities** | Create a new account for an instructor. |
| **Collaborators** | Start, Login and RethinkDBClient |
| **Notes** | Using a non-existing username and two matching password fields a new entry in the database can be created. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | Login |
| **Responsibilities** | Scripting for the login of an instructor. |
| **Collaborators** | Start, SignUp, RethinkDbClient |
| **Notes** | Using comparison of username and SHA256 hashed passwords a authenticated session is started on confirmation from the database. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | MenuSceneRouter |
| **Responsibilities** | Presentational logic for start scene menus. |
| **Collaborators** | Start |
| **Notes** | Handle the buttons and activation of objects on the UI interface Start. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | RethinkDbClient |
| **Responsibilities** | A client driver for connecting to our RethinkDb |
| **Collaborators** | RethinkDb |
| **Notes** | Using the IP to our cloud instance a connection is prepared. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | ConnectionManager |
| **Responsibilities** | Creates the connection to our RethinkDb database and the connection to our MQTT broker, both resides in the cloud. |
| **Collaborators** | MqttClientDave and RethinkDbClient |
| **Notes** | Connections are initiated that can be reached from the user interface components. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | MqttClientDave |
| **Responsibilities** | A client for connecting to our MqttBroker |
| **Collaborators** | MqttBroker |
| **Notes** | Using the IP to our cloud instance a connection is prepared. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | Renderer |
| **Responsibilities** | Render the all of the parts of the message from the Broker in our 3D environment. |
| **Collaborators** | MqttBroker |
| **Notes** | This is not one script, it is a compilation of scripts developed to handle the entire rendition of the simulation. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | ObjectSpawner |
| **Responsibilities** | Spawns objects according to simulation. |
| **Collaborators** | Renderer |
| **Notes** | Uses data from the MqttBroker through a subscribe. The uml notation and messages gets rendered from this spawner type class. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | ObjectPositioner |
| **Responsibilities** | Setting the position of multiple diagrams to get rid of any overlap. |
| **Collaborators** | Renderer |
| **Notes** | Depending on number of diagrams, the position of each diagram is set. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | InternalScripts |
| **Responsibilities** | The responsibility of this component is abstract, it contains classes that utilizes supporting roles for the more high level components. |
| **Collaborators** | N/A |
| **Notes** | I.e the rendering of buttons in the InstructorChoice component is one of this collections scripts. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | AccountStorage |
| **Responsibilities** | Key value store containing login information, instructor information and account information. |
| **Collaborators** | RethinkDbClient |
| **Notes** | Using RethinkDb syntax queries can be made, real time updates can be handled with ease. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | MqttBroker |
| **Responsibilities** | Publish/Subscribe broker passing messages to the right parts of application. |
| **Collaborators** | MqttClientDave |
| **Notes** | Handles creation of topics and messages sent. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | DiagramCoordinator |
| **Responsibilities** | Assigns name to diagram components and listens to their state. |
| **Collaborators** | MqttBroker, Parser, PythonServer, Container |
| **Notes** | Using the cloud based Parser names can be assigned to each part of the diagram. Messages are also handled in this class. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | Parser |
| **Responsibilities** | Parses json files from the diagram coordinator - broker association. |
| **Collaborators** | Container |
| **Notes** | An cloud based parser. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | PythonServer |
| **Responsibilities** | Spawns docker images. |
| **Collaborators** | MqttBroker, ContainerCoordinator and DiagramCoordinator |
| **Notes** | Start docker container depending on messages from Broker and DiagramCoordinator. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | Container |
| **Responsibilities** | A Docker container containing the images and scripted behavior. |
| **Collaborators** | Broker, PythonServer, ContainerCoordinator, Rendering |
| **Notes** | Each image is spawned by the PythonServer, the coordinator handles which entities should be rendered. The return of the DockerImage is sent to the MqttBroker, afterwards data is forwarded to the Renderer component. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | DockerImage |
| **Responsibilities** | Acting as the nodes in our system. |
| **Collaborators** | Container |
| **Notes** | Using docker we can create virtual machines in the cloud and use them to simulate our jsons in real-time on the front-end. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | ContainerCoordinator |
| **Responsibilities** | Spawns a container pool and waits for requests. |
| **Collaborators** | Container |
| **Notes** | The ContainerCoordinator is initially a pool of containers, if more containers are requested the pool is extended. |
| **Issues** | No Issues reported |

## Interfaces

|  |  |
| --- | --- |
| **Interface** | Start |
| **Responsibilities** | User interface component including login and sign up. Used as a starting point for users. |
| **Operations** | **1. Operation** Login Panel  **Description** Opens a login panel, containing editable input fields for username and password.  Logic is handled by the Login component.  **2. Operation** SignUp Panel  **Description** Using the Sign Up button in the login panel another panel is instantiated.  In this panel there is  editable input fields for username and two passwords.  Logic is handled by the SignUp Component.  **3. Operation** Student Button  **Description** When pressed the button gets deactivated and shows a Student name field and the Student Name Button utilized the Load Scene operation.  **4. Operation** Student Name Button  **Description** Sets a local student name variable and utilizes the Load Scene operation.  **5. Operation** Load Scene  **Description** Loads the next user interface. This is dependant on the user being authenticated or not.  **6. Operation** Instructor Button  **Description** Hides the instructor button and shows the Room Name input field.  **7. Operation** Room Name Button  **Description** Sets the room name variable locally.  **8. Operation** Upload Diagram  **Description** Opens a file explorer where the user can choose json files, inserts diagram name and type into the storage component and utilizes the Load Scene operation. |
| **Protocol** | Not Authenticated: Operation 1, 2, 3, 4. Authenticated: 6, 7, 8. |
| **Notes** | The user can add their name as a student and continue execution of the application or sign up for an instructor account or log in as an instructor. As an instructor the user can start a new room. |
| **Issues** | No issues reported |

|  |  |
| --- | --- |
| **Interface** | DiagramSelector |
| **Responsibilities** | User interface component for the file explorer. |
| **Operations** | **1. Operation** Choose File  **Description** When the file explorer is initiated, json files in the active folder is visible.  In this view they are choosable.  **2. Operation** Cancel upload  **Description** at anytime in the diagram selector view the user can press cancel and discard their current choices. |
| **Protocol** | Authenticated: 1, 2. |
| **Notes** | The files available in the file explorer are of json type. |
| **Issues** | No issues reported |

|  |  |
| --- | --- |
| **Component** | Instructor Choice |
| **Responsibilities** | User Interface component that lets student select their instructor. |
| **Operations** | **1. Operation** Select Instructor  **Description** Pressing Instructor buttons rendered from our RethinkDb storage forwards the student to the next interface.  **2. Operation** Back  **Description** Loads the start scene. |
| **Protocol** | Mqtt connection & RethinkDb connection needed. |
| **Notes** | Instructors buttons are rendered, and from there a student can choose an instructor. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | RoomChoice |
| **Responsibilities** | User Interface component that lets student select their instructor. |
| **Operations** | **1. Operation** Select Room  **Description** Pressing room buttons rendered from our RethinkDb storage forwards the student to the next interface.  **2. Operation** Back  **Description** Loads the Instructor Choice interface. |
| **Protocol** | Mqtt connection & RethinkDb connection needed. |
| **Notes** | Room buttons are rendered, and from there a student can choose an room. |
| **Issues** | No Issues reported |

|  |  |
| --- | --- |
| **Component** | DiagramRoom |
| **Responsibilities** | User interface component responsible for rendering diagrams, showing students present in the room and a log of messages being sent in the simulation. Instructors have full control of the rendition. |
| **Operations** | **1. Operation** Show Student  **Description** Displays the student active in the room.  **2. Operation** Show Log  **Description** Displays a log of messages in the simulation.  **3. Operation** Show Controls  **Description** Displays the control scheme.  **4. Operation** Reset Camera  **Description** Resets the camera to it initial position.  **5. Operation** Play  **Description** Publishes the selected diagram (json file) to the Mqtt broker through the Mqtt connection protocol.  **6. Operation** Back  **Description** Loads the Diagram Choice interface. |
| **Protocol** | Mqtt connection needed: Operation 5. |
| **Notes** | This component gets created for every user of the application, the rendition will be the same but the position of viewpoints of the users will differ. |
| **Issues** | No Issues reported |

# Use Case View

This section provides an overview of some of the most significant scenarios, which are expressed using use case scenarios. Their significance was determined by the value they provide to the end users, the number of components interacting or based on the functional requirements they bring to realization.

## Use Case Scenarios

UC1 was used to implement functional requirement FR4. It is a required step for realizing DAVE’s main goal (see Introduction) of animating and simulating diagrams, therefore it was decided to include the use case. Additionally, this use case has significant architectural importance, due to it being responsible for implementing quality scenario S3.

|  |  |
| --- | --- |
| **ID** | UC1 |
| **Use case** | Instructor Registration |
| **Description** | Users create DAVE account to get Instructor privileges, such as uploading a diagram file |
| **Actors** | Instructor |
| **Preconditions** | Instructor must be authenticated |
| **Steps** | Basic Flow:   1. In the welcome page, user clicks login button 2. Instructor is presented with new screen, press Sign Up button 3. Enter desired username and password twice 4. Click Register 5. Instructor is taken to the Main window, now signed in   .  Alternative Flow:   1. In the welcome page, user clicks instructor button. Continues at Basic Flow 2 2. Password does not match    1. Error message displayed    2. Continues at Basic Flow 3 3. Username already in use    1. Error message displayed    2. Continues at Basic Flow 3 |

Table 8 - Use Case 1

UC2 was used to implement FR1 functional requirement. It’s placement here is attributed to the use case being the initial step towards DAVE’s purpose. Therefore, other use cases and application processes are dependent on UC2 to be completed successfully.

|  |  |
| --- | --- |
| **ID** | UC2 |
| **Use case** | Upload Diagrams |
| **Description** | Instructor selects and uploads diagrams files to room |
| **Actors** | Instructor |
| **Preconditions** | Instructor is already signed in and authenticated |
| **Steps** | Basic Flow:   1. In the welcome page, press Instructor button 2. Input a room name 3. Click continue button 4. Click upload diagrams button 5. In the file browser, select the diagrams you wish to upload and press open 6. Click the play button   Alternative Flow:   1. Room name is empty    1. Error message displayed    2. Continues at Basic Flow 2 2. If upload is pressed from within the scene    1. Continue at Basic Flow 5 3. Cancel is pressed    1. Continue at Basic Flow 4 |

Table 9 - Use Case 2

The following use case relates to FR7 requirement. UC3 was included due to its use of the main architectural style, publish-subscribe.

|  |  |
| --- | --- |
| **ID** | UC3 |
| **Use case** | Join room |
| **Description** | Student joins a room |
| **Actors** | Student |
| **Preconditions** | There is a room to join |
| **Steps** | Basic Flow:   1. In the welcome page, press Student button 2. Student is presented with a list of registered instructors 3. Click desired instructor 4. Student is presented with a list of the instructor’s rooms 5. Click desired room 6. Student is taken to the room |

Table 10 - Use Case 3

Animate Diagram was included, due to it being a scenario representation of DAVE’s main functionality. Moreover, the use case has multiple significant components interacting and arguably the most architecturally significant use case view. UC4 is responsible for envisioning FR2 and FR3.

|  |  |
| --- | --- |
| **ID** | UC4 |
| **Use case** | Animate diagram |
| **Description** | Uploaded diagram is animated |
| **Actors** | Student and Instructor |
| **Preconditions** | Use case scenarios ‘Upload Diagrams’ and ‘Join a room’ must have been completed |
| **Steps** | Basic Flow:   1. The diagram is published to an MQTT room 2. DAVE creates a coordinator for worker processes 3. DAVE creates containers for virtualization 4. The system parses the diagram for sequence diagram processes 5. The containers request for workers 6. Workers are added to the containers 7. Containers are assigned names 8. Each container publishes their state to DAVE front end 9. DAVE creates objects from the state |

Table 11 - Use Case 4

## Use Case Diagram

To depict use case interactions amongst each other, a use case diagram was created. Most noticeably both the Student and the Instructor interact with the Animate Diagram. When students join a room, they are greeted with an environment in which they can walk around and explore, therefore Animate Diagram only extends on the already existing functionality. However, Upload Diagram must have executed for there to be anything to animate, hence their relation is “include”. Moreover, the Upload Diagram use case itself has Instructor Registration included in it, due to registration only providing value to the user through Upload Diagram.

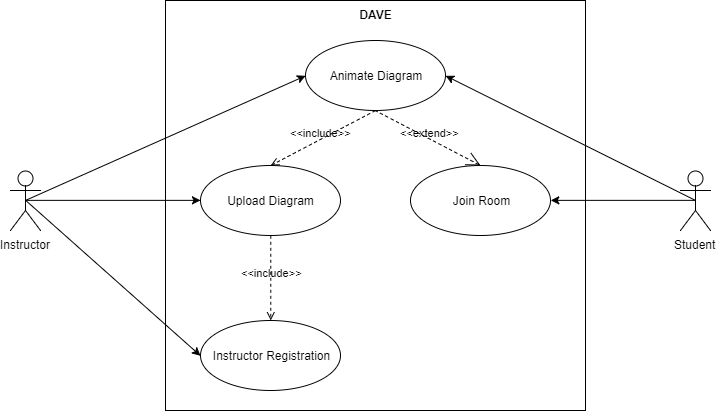


Figure 8 - Use Case Diagram

# Logical View

Logical View was used to graphically express the architecturally significant use cases (See section 7). Sequence diagrams were used for use case realization, while natural language was used to improve the readers’ understanding of the architectural significance the use cases provide.

## Instructor Registration

The following sequence diagram depicts the system components interactions when realizing Instructor Registration (UC1). The unauthenticated instructor must go through a simple registration, resulting in authentication and upload diagram privileges. The instructor uses MenuSceneRouter to traverse DAVE’s main screen and go into the SignUp screen. The username and password are sent to ConnectionManager, once the instructor has entered them. At that point multiple flows of the diagram are created based on the entered values. First flow is dependent on the username and whether it exists in the database, which is checked by ConnectionManager. Second flow of events is dependent on the password and whether it matches the set constraints. If it does then the password is hashed and inserted to the database, the team has used RethinkDB for account information storage. Lastly, the third flow of events is dependent on the on the account information insertion success. If the ConnectionManager failed to insert the username and password to the database, the user is displayed with an error message. On the other hand, if the insertion was successful, the user is able to begin the now authenticated session.

### Architectural Significance

Instructor Registration use case is directly responsible for meeting Security quality requirement (QR4) and for instructor registration (FR4). Moreover, this use case uses resource sharing, one of the properties for distributed systems. This is done by data insertion to remote database, detailed in Quality Attribute Scenario S3. More specifically, the instructor username will be shared with students, when the later search for rooms to join.

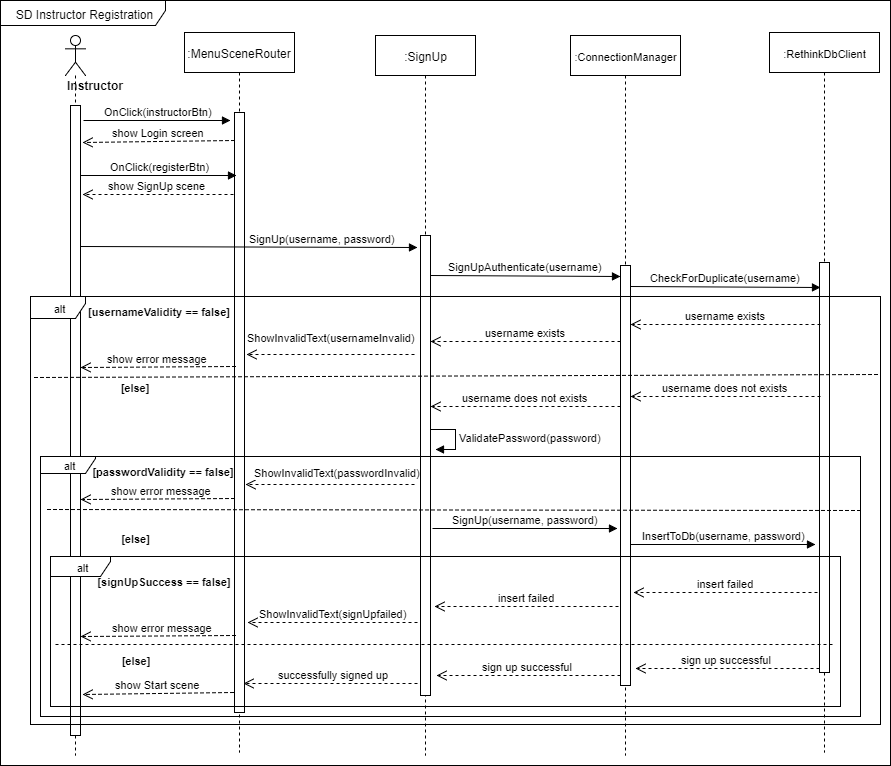


Figure 9 - Register Instructor SD

## Upload Diagram

The following sequence diagram show the components responsible for realizing Upload Diagram (UC2) use case and the interactions between them. The use case begins when an instructor decides to upload diagram file. One or more diagram files can be uploaded simultaneously. The DiagramSelector uses the AddJson(filePath) method inside a loop to create an async coroutine for each diagram file. The number of files correspond to the length of the loop. Once the method is executed the DiagramSelector checks whether the diagram file adheres to the JSON structure. If the structure is valid the name of the instructor’s room or “topic” is inserted into the database, in order for the student to be able to find it later. The AddSelectedJson method enques the diagram file (JSON) strings. This step is done using the ConnectionManager component. Once this is done the instructor has one final step, which is to press a button called “Play”, visible in the bottom of the scene. By clicking the button, the ConnectionManager will initialize two process that run parallel, one of which will publish the diagram file to a room and the other publish the room location. Reason behind this, is so that the MqttBroker component could be used later find the files and simulate them (refer to UC4).

### Architectural Significance

Use case is responsible for implementing FR1. UC2 also incorporates the publish-subscribe style. The ConnectionManager uses the MqttClientDave to establish connection to the MqttBroker, which used for publishing. Additionally, this use case possesses one of the strengths of publish-subscribe style, parallel operations. This strength is utilized when simultaneously publishing room and the files within them.

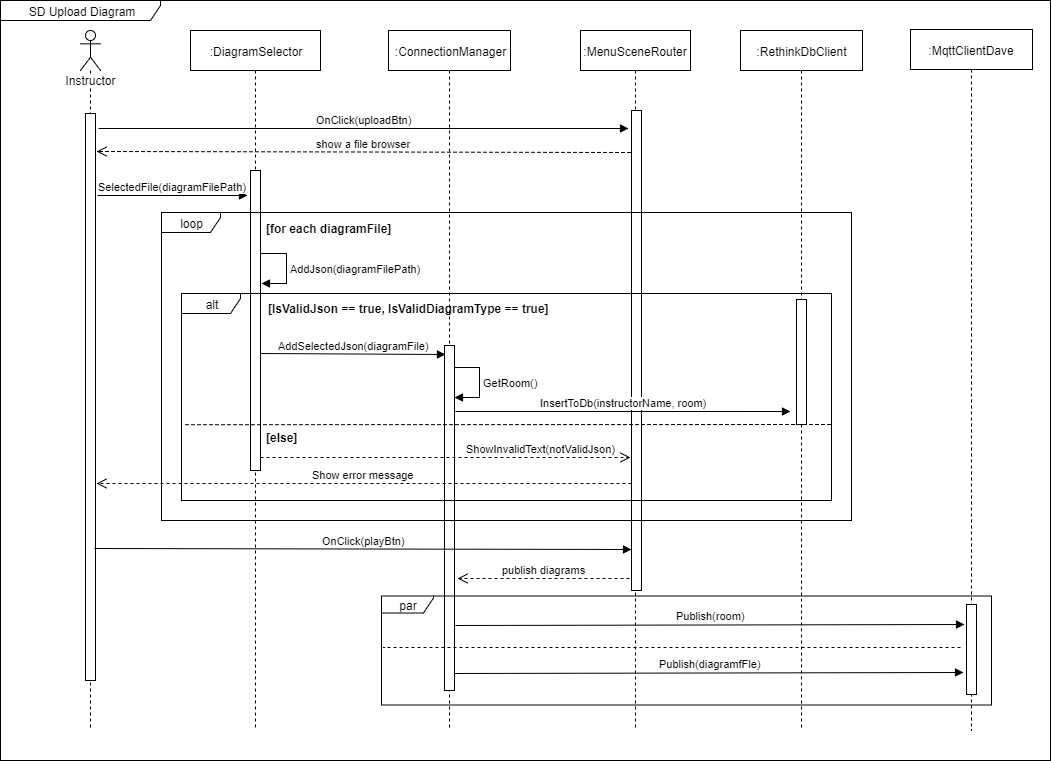


Figure 10 - Upload Diagram SD

## Join Room

The following sequence diagram aims to inform the readers on the components interactions that take place to realize the Join Room use case (UC3). It begins when the Student goes from the main screen to the instructor selection screen, which is done by pressing the “Student” button. The student is then forced to enter their desired usernames so the instructor could identify them, using a show student panel in the instructors’ DAVE applications. Once this is done the ConnectionManager retrieves all the instructor names that have published any diagram rooms from the RethinkDbClient. These names are then rendered as buttons. The students’ progress by pressing their desired instructor name. At this point, the students are presented with the selected instructors published diagram rooms. Much like the instructor names the diagram room names are retrieved from the RethinkDbClient using the ConnectionManager component. Once the student has pressed their selected diagram room the MqttClientDave on the student's’ DAVE application will subscribe to this specific diagram room by the selected instructor. At this point the student will be taken to a new environment where they are given control over a player object.

### Architectural Significance

The importance of this use case is attributed to several distributed system properties that in possess. Firstly, it is believed the use case realization possesses resource sharing properties. The instructor names, their diagram room names and the diagram files are retrieved and shared with an array of students.  Secondly, this use case realization incorporates concurrency, as a number of students are able to retrieve this data simultaneously using the MqttBroker and the MqttClientDave. Thirdly, elements of transparency, more specifically concurrency transparency exhibited when students retrieve the same diagram file without sharing interactions. This is done using the publish-subscribe style, more specifically with the MqttBroker and the MqttClientDave components. The later property serves as motivation for choosing the publish-subscribe style. Moreover, the use case helps bring into fruition the availability quality attribute scenario (S1), by improving the scalability of users via the incorporation of the pub/sub style. The use case also fulfills the Join Room functional requirement (FR7)

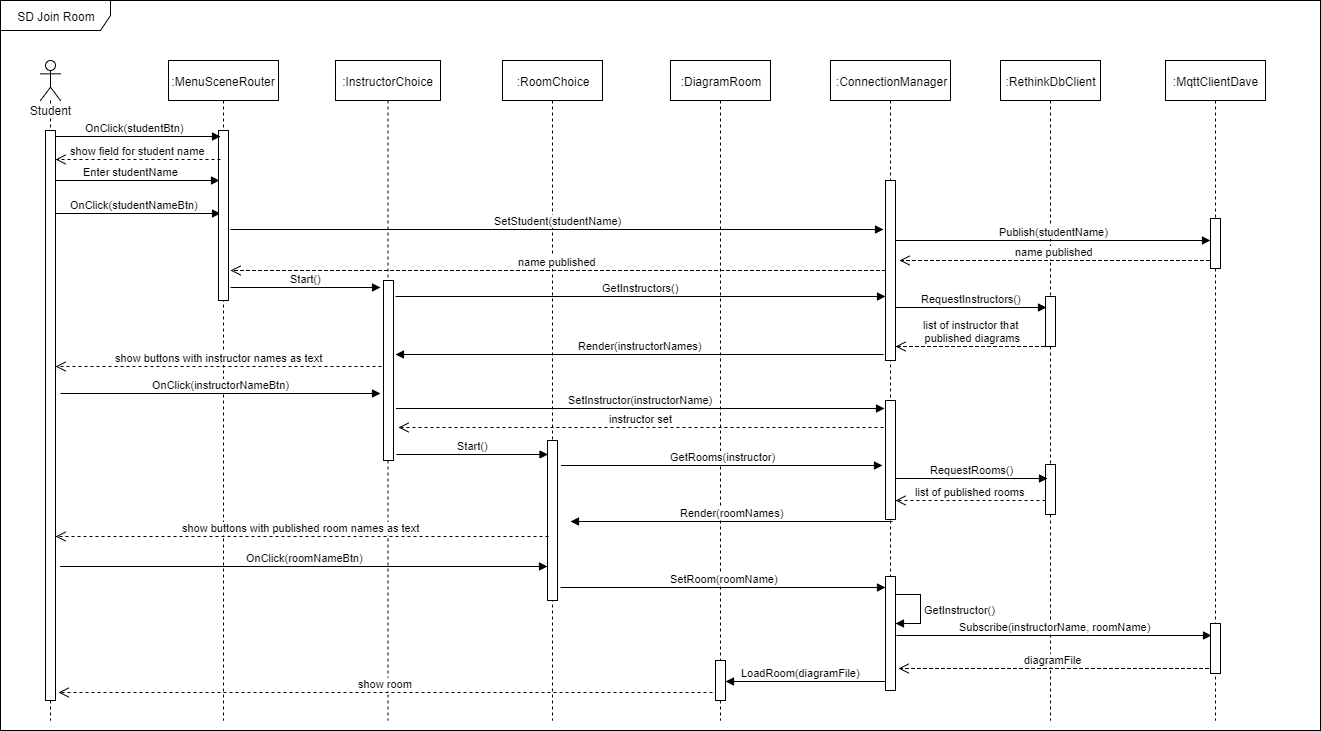


Figure 11 - Join Room SD

## Animate Diagram

The Animate Diagram use case (UC4) was identified as the most significant. Due to its large nature, it was decided to express it by dividing its two main concerns; create virtual machines and animate their interactions. Therefore, the Animate Diagram was split up to two sequence diagrams; Spawn Containers and Simulate Diagram. The first sequence diagram focuses on creation of virtual machines (FR3) and the later sending and animating communication between the virtual machines (FR2).

### Spawn Containers

The use case begins once the published diagram has arrived at the DiagramCoordiantor. At this point parallel processes are created. The first process is used for virtualization. Using the ContainerCoordinator the PythonServer creates containers and publishes their location so other processes could find them. To create the containers the ContainerCoordinator first request several free containers from the PythonServer. Once this is done the ContainerCoordinator switches to await incoming requests. The second process is responsible for extracting and passing around the diagram file data. This is done by firstly initiating the DiagramCoordinator and then passing the diagram file from said coordinator to the Parser component. Once the parsed data is returned to DiagramCoordinator, it requests the ContainerCoordinator to provide a number of containers based on the number of processes in the diagram file, if more than the currently free containers are needed. The ContainerCoordinator fur fills this request by returning the location of free containers to the DiagramCoordinator. From this point onwards, the use case diagram continues with SD Simulate Diagram.

#### Architectural Significance:

This part of the application was implemented using Erlang, apart the PythonServer. The reasoning behind this, was to use Erlang’s distributed processes to ease the distribution for DAVE. Additionally, this part of the application has transparency, seen in distributed systems. To begin with, using the ContainerCoordinator DAVE is able to possess scaling transparency properties, as any number of containers can be added without changing the current structure or operations of the system. In continuation, location transparency is incorporated here as well. The system, DAVE, is able to create containers and share information amongst them, without ever disclosing their location to its users. Properties of concurrency can be seen as well, specifically the parallel operations that run once the diagram file is received. Moreover, openness is achieved in this part of the system, by having containers. Said containers are identical and shared by multiple clients, although at separate points in time. Once the containers have completed their task there are returned to the container pool with a rest state, to await new request from new clients.

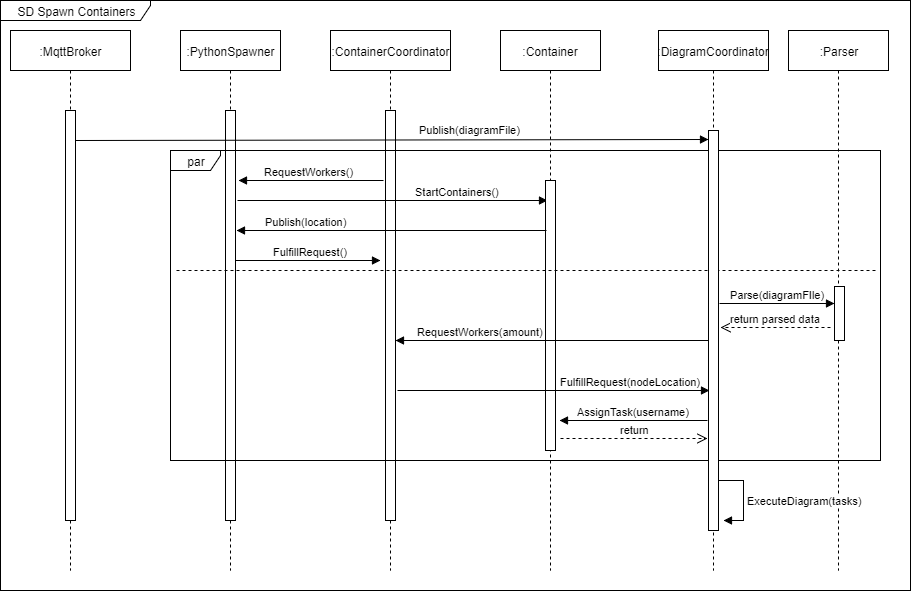


Figure 12 - Spawn Containers SD

### Simulate Diagram

This sequence diagram continues the implementation for Animate Diagram use case

(UC4). The following sequence diagram continues after the Spawn Containers and depicts the simulation of the processes. The flow of events is rather simple to follow in this sequence diagram. The DiagramCoordinator orders the container states. The states of the containers are then publishing from MqttBroker to MqttClientDave. Once the MqttClientDave receives the state it animates the simulation for the subscribed student to see. The ObjectPositioner coordinates the location of where to spawned objects while the ObjectSpawner creates them for the subscribed students.

#### Architectural Significance

The way the development team incorporated the publish-subscribe style with distributed system can be seen in this example. The MqttBroker would listen for state changes and publish them to MqttClientDave, that are listening to the subscribed room, which is done through Join Room (UC3). By adding the MqttClientDave and MqttBroker components to the system, the team managed to create a scalable and open distributed system. Thus, supporting the change from client-server to publish-subscribe styles. This part of the use case also incorporates location transparency, by publishing the container states and animating them without disclosing the container locations via the MqttClientDave. Furthermore, resource sharing property is achieved by sharing containers states with the subscribed students.

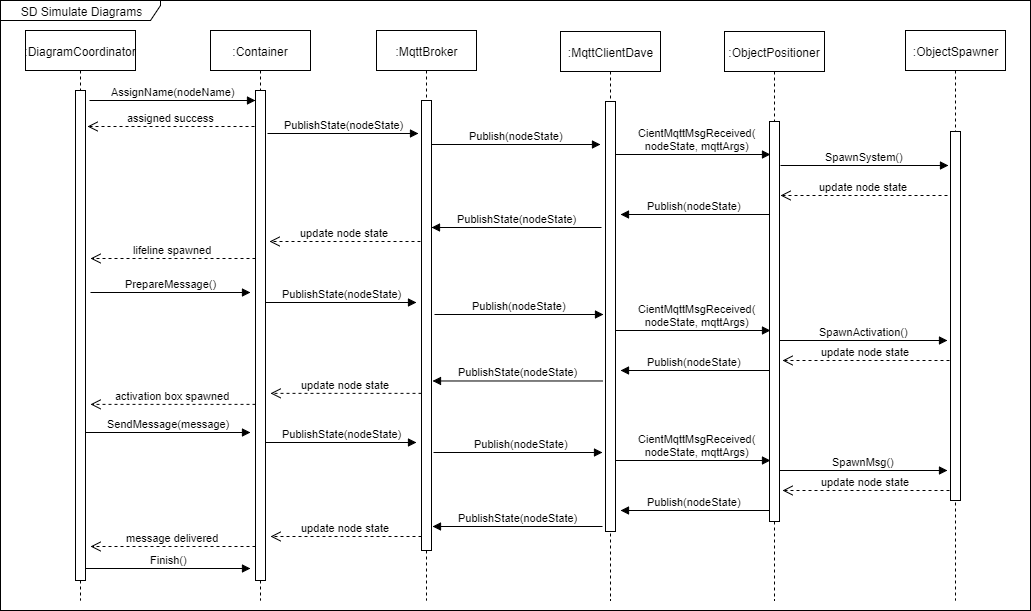
****

Figure 13 - Simulate Diagram SD

# Process View

Three main steps were identified while attempting to achieve diagram simulation and animation. To simulate a diagram an instructor would first be required to upload a diagram file, second DAVE would be required to create virtual containers based on the file and lastly DAVE would send messages between the containers and animate to provide visual representation for the instructor. These steps span several processes, therefore to aid understanding of the work flow within the application, a process view was created. This process view is used to provide an overview of the significant processes.

## Upload Diagram

DAVE only allows registered users, meaning instructors, to upload diagrams. Users are required to complete a simple registration before continuing to the next step. The account information is then hashed and stored within the database.

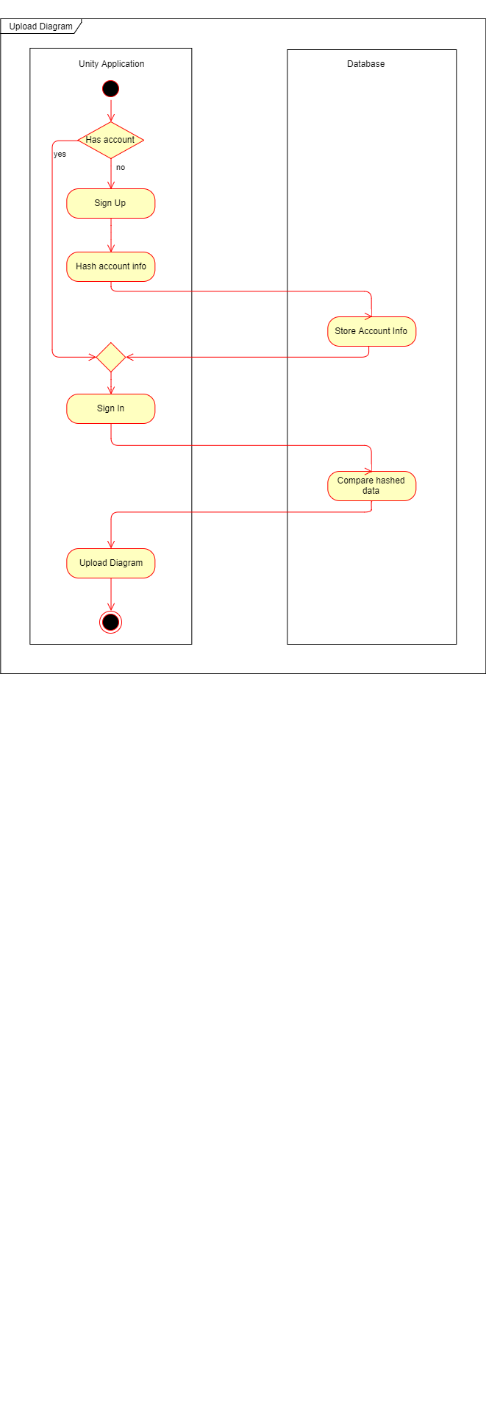


Figure 14 - SignUp Activity Diagram

## Initiate Diagrams

Proceeding to the second step, the second step includes interaction with the CloudServer, a representation of DAVE components located on AWS. The CloudServer is always running. After diagram is uploaded the diagram file is concurrently added to an instructor “room” and said room is begun to simulate by the diagram initiator. During the development process, the development team opted out for using the Publish-Subscribe pattern, which was created using a MQTT protocol, therefore the “room” corresponds to a published topic. Once the diagram initiator has begun, DAVE will create a worker coordinator, which spawns worker processes, and a number of containers, meaning virtual machines, to run on these processes. However, the number of containers will only be clear after the diagram file is finished being parser. Once the parsing is done the correct number of virtual machines are created and assigned names.

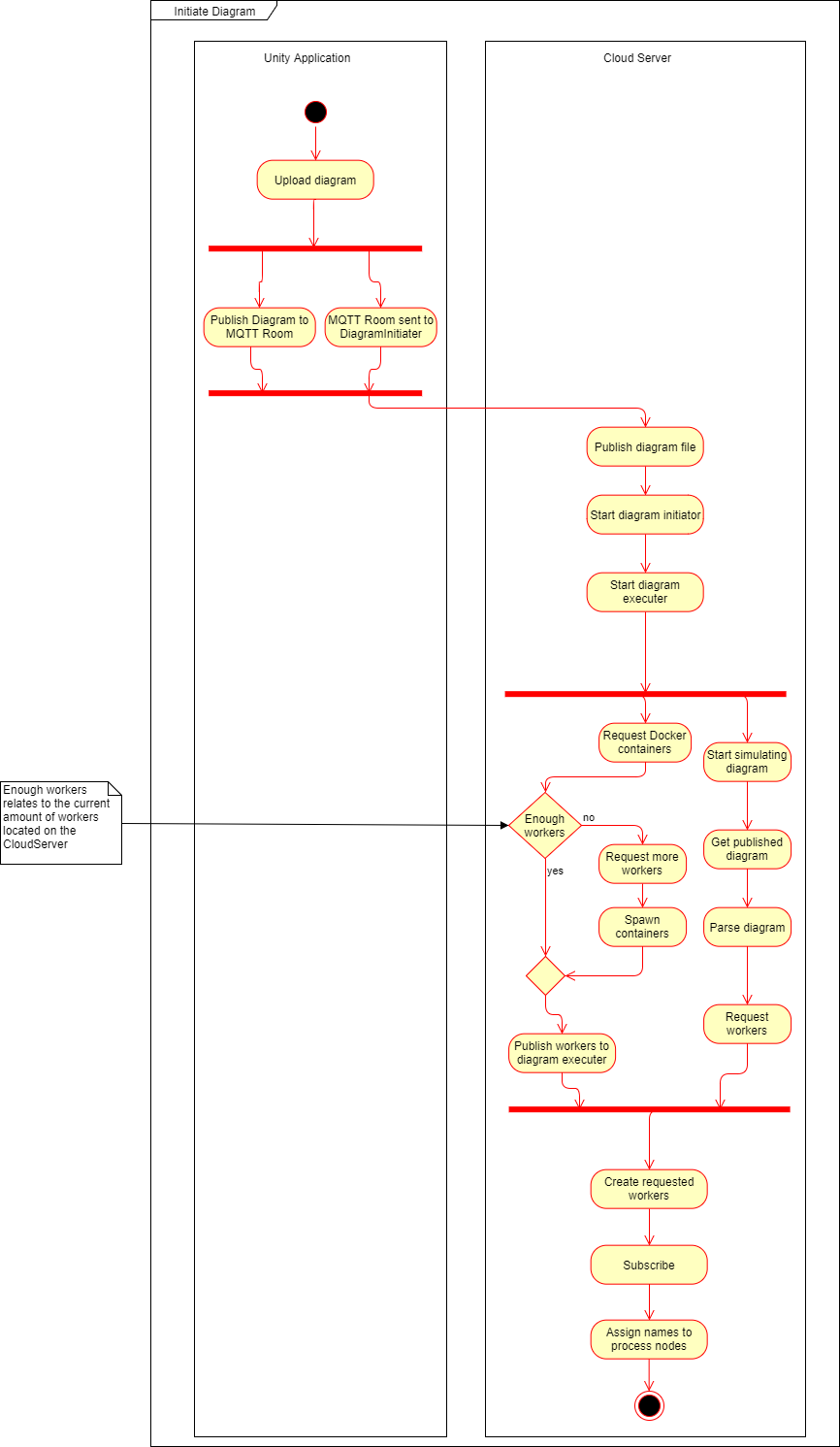


Figure 15 - Initiate Diagram Simulation Activity Diagram

## Simulate Diagram

Once the virtual machines are created their initial states are published, after which the lifelines accompanied by system boxes are rendered for the subscribed people to see. At this point DAVE publishes the process node states once again before preparing to send messages. The Unity application side waits for the new states, so it could begin with animating the messages. After the animations are done, DAVE one final time publishes the node states to indicate that all processes have completed their tasks and are no longer necessary.

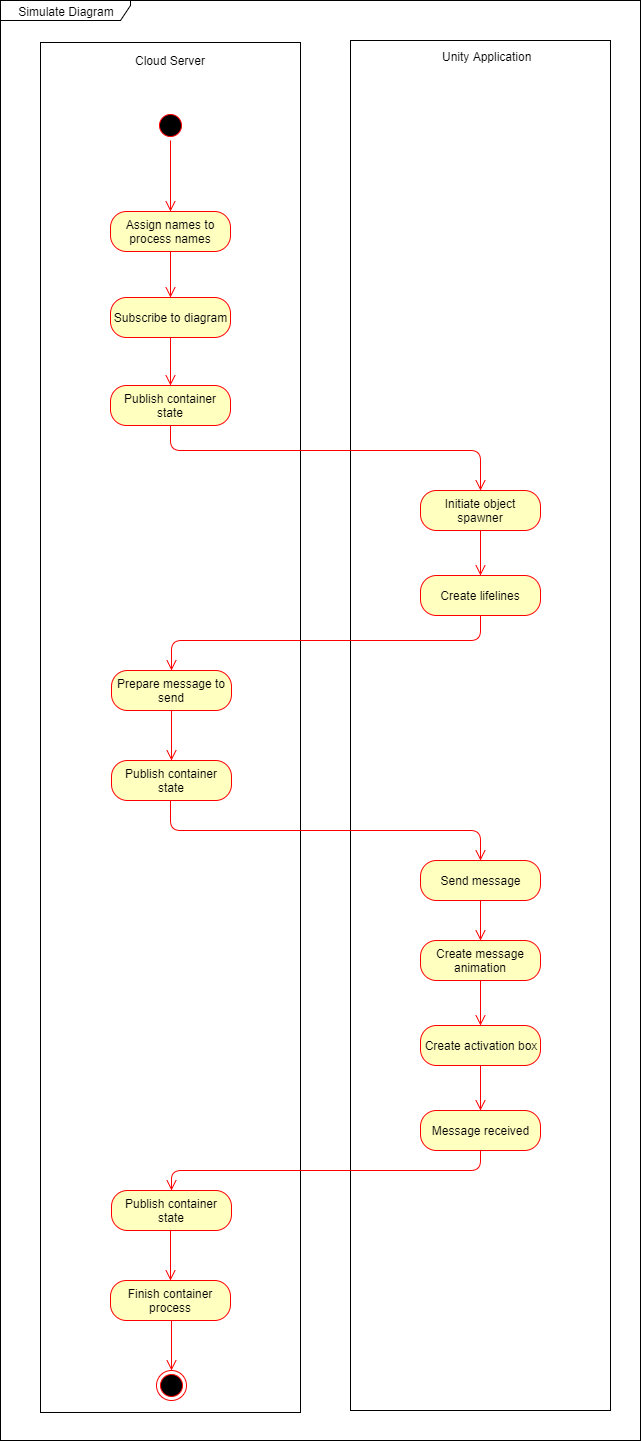


Figure 16 - Initiate Diagram Activity Diagram

# Development View

A development view was created for the purpose of introducing the readers with DAVE’s implementation details. This section aims to familiarize the readers with DAVE from a developer's side. By doing so, hopefully, easing any future software maintenance and answering raised questions regarding implementation. The Development View was expressed using a package diagram.

## Layered Overview

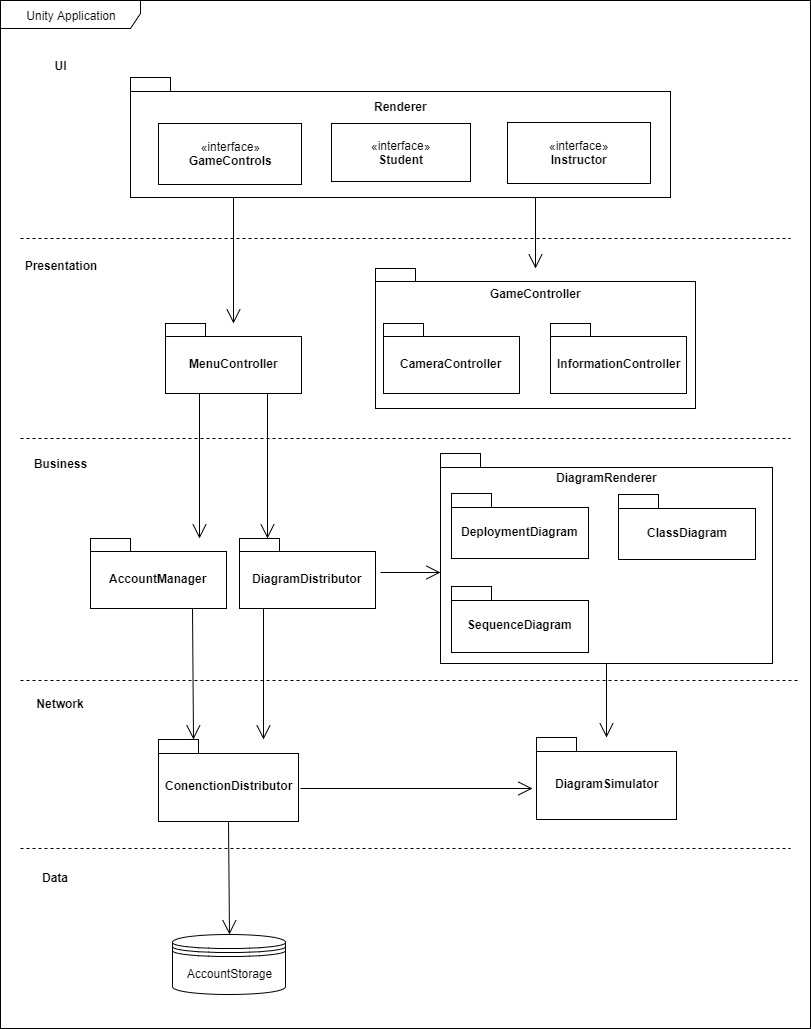


Figure 17 - Layered Architecture

The Layered Overview provides a high-level depiction of the packages throughout the whole system. For the purpose of animating and simulating UML diagrams, DAVE has incorporated 3 different distinct styles; Client-Server, Layered Architecture and Publish Subscribe.

Out of the aforementioned styles, client-server was utilized the least. The only example of this style can be seen between the ConnectionManager and the AccountStorage. Client-Server was used to solve the Instructor registration and login functional requirements (FR4 and FR5). During the development process, a discussion surrounding the use of publish-subscribe to implement these functional requirements took place. However, at the time, it was believed that making registration work through the MqttBroker would create an unnecessary complication, requiring time and human resources to be attributed. Therefore, the use of the client-server style is attributed to its simplicity and saved resources. It is also worth noting that the initial style planned for DAVE was client-server, however it was replaced by a more suitable style for distributed system, publish-subscribe (See section 6.1)

The development team incorporated the use of layered architecture. The use of the style could be attributed to the development process. During the course of development, tasks would be divided amongst individual team members. To increase the project velocity and value granted to the users, the team attempted to separate said tasks as much as possible. Layered architecture was used for this purpose. It enabled the developers to work on separate parts of DAVE with few dependencies. Additionally, this architecture choice was motivated by testing. During the inception phase, the development team had planned to use Jenkins for automated testing. Unfortunately, the plans fell through during the initial iterations of the project. This resulted in a change on how the team approached testing. Instead of the automated processes the developers carried out Unit testing, System testing and Requirement testing. These tests would require more time to be allocated, therefore the team decided to combat this by incorporating layered architecture, which would allow component test to be carried out independently. Thus, saving valuable time in the process. An additional, reason for using the style, would be the team’s previous experiences. Developers had knowledge and experience of designing layered architecture from past courses, therefore less time could be allocated to incorporating said style than others.

Publish-Subscribe style was kept in mind throughout the course of the project. Examples of this style implementation can be seen between the ConnectionDistributor and the DiagramSimulator packages. The ConnectionDistributor packages contains an MqttClientDave that listens for changes in virtual machines, while the DiagramSimulator has an MqttBroker publishing the virtual machine states. To see the detailed interactions of the two components, (See section 8.4.2). The decision for incorporating the publish-subscribe style was based on distributed system and their properties. Firstly, the team had reasoned that DAVE’s scalability would increase by using the publish-subscribe style, allowing more students and instructors to subscribe and/or publish diagrams. Thus, improving DAVE’s value. Secondly, the team planned to combine Erlang and publish-subscribe to easier implement parallel operations, thus improving DAVE’s concurrency. Thirdly, it was thought that transparency would be easier to achieve with the publish-subscribe style. This proved to be true, in the case of animating and simulating diagrams (see Simulate Diagram and Upload Diagram, section 6 Logical View).

## Individual Packages

#### Renderer

The package is responsible for creating DAVE´s interfaces. The interfaces were created using Unity scenes and game objects, therefore they do not possess scripts responsible for creating views.

#### AccountStorage

AccountStorage is a RethinkDB data store. The purpose of which is to store Instructor account information, such as username and password.

#### MenuController

This package is responsible for the logic used when interacting with the menu by the users.

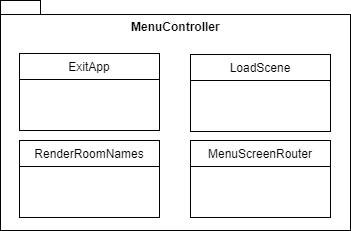


Figure 18 - MenuController Package

#### CameraController

The package is in charge of providing the users with camera controls as well as swapping between camera modes. The package enables camera movement, rotation and zoom.

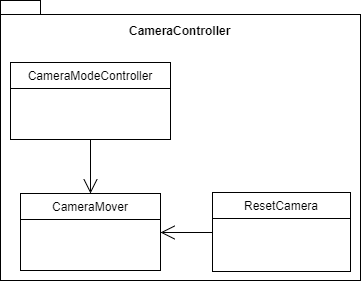


Figure 19 - CameraController Package

#### InformationController

The package provides an overview of extra information, such as number of students in the room, game controls and a log for sequence diagram messages, for the users.

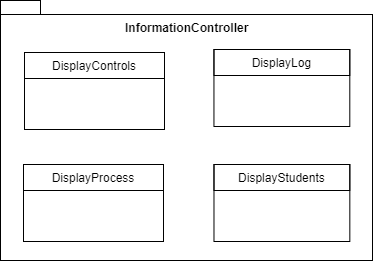


Figure 20 - InformationController Package

#### AccountManager

AccountManager provides the Instructors the possibility to create and access their accounts.

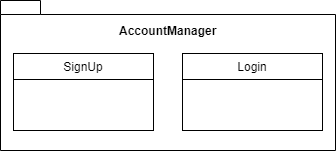


Figure 21 - AccountManager Package

#### DiagramDistributor

This package contains classes that interact with the unparsed diagram file. The package is responsible for local parsing and distribution of the diagram file to the DiagramRenderer and the ConnectionDistributor.

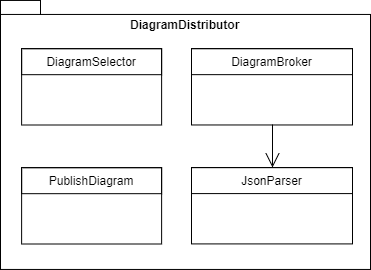


Figure 22 - DiagramDistributor Package

#### DeployementDiagram

Contains the logic responsible for creating and animating the deployment diagram.

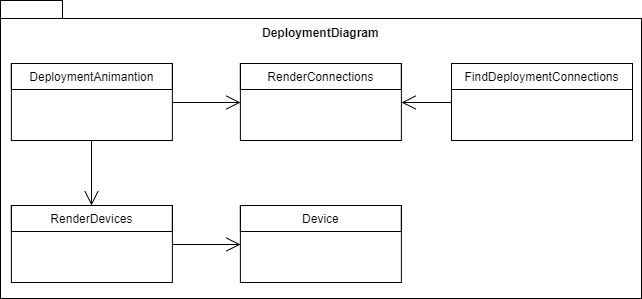


Figure 23 - DeploymentDiagram Package

#### ClassDiagram

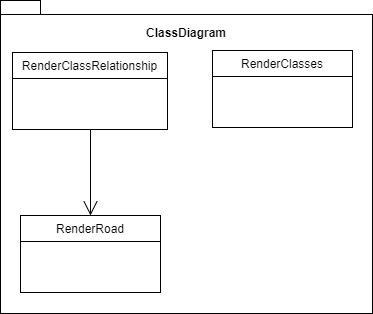
Contains the logic responsible for creating and animating the class diagram.

Figure 24 - ClassDiagram Package

#### SequenceDiagram

Contains the logic responsible for creating and animating the sequence diagram. The package uses a Master-Slave pattern as seen with the SSDSpawner.

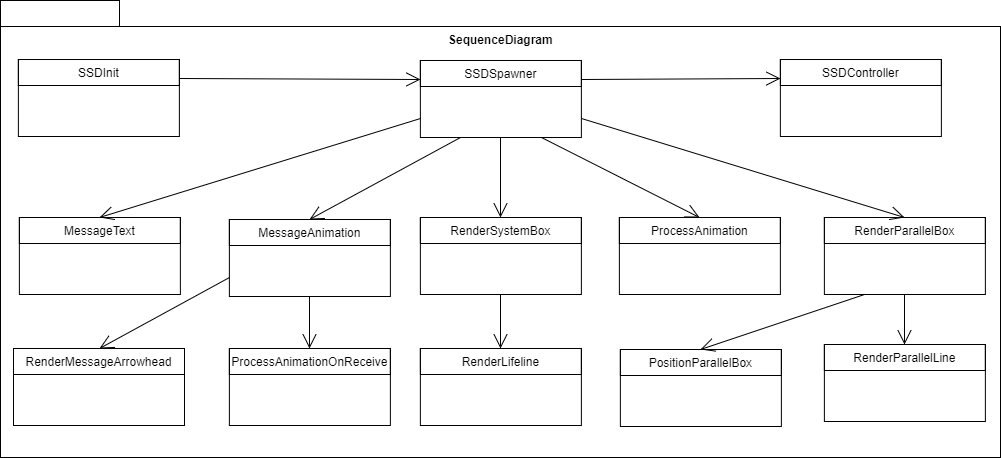


Figure 25 - SequenceDiagram Package

#### ConnectionDistributor

The main responsibility is to provide other scripts connections to the MQTT broker by utilizing publish and subscribe. Additionally, the package provides access to the RethinkDB database.

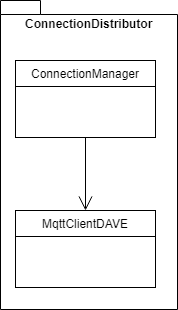


Figure 26 - ConnectionDistributor Package

#### DiagramSimulator

As the name suggests, DiagramSimulator is responsible for simulating the provided diagram files. The package consists of various Erlang scripts and a Python script for creating Docker containers. An MQTT client is used to receive the diagram file information, while docker\_container and worker\_coordinator create and coordinate interactions between Docker Containers. A distributed Erlang parser is used to obtain the necessary sequence diagram file data while pubsub\_ssdchat sends the messages.

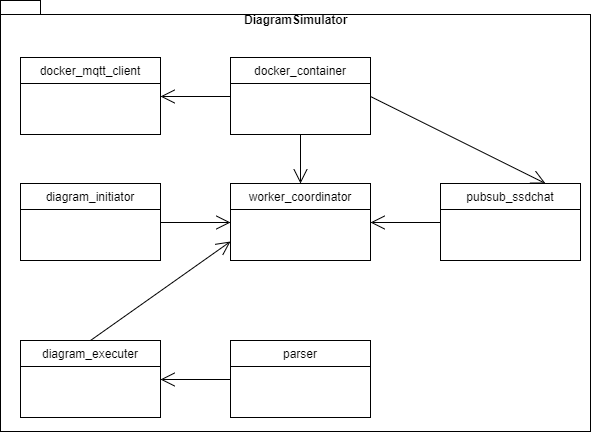


Figure 27 - DiagramSimulator Package

#### Deprecated

Lastly, a deprecated folder, not shown in the layered architecture, is used to store some of the significant obsolete scripts. The NetworkController and MyNetworkManager were used in the earlier iterations of the project. Before swapping architectural style from Client-Server to Publish-Subscribe (See section 6.1). The TextOutline script was an incomplete script used to reinforce the visibility of texts throughout DAVE. The script was put on the backburner, while more important features were prioritized. The remaining scripts were used for the initial sequence diagram, when it was lacking distribution and simulation, meaning parsed and animated locally. The scripts were chosen to be retained, due to the time spent working on them.

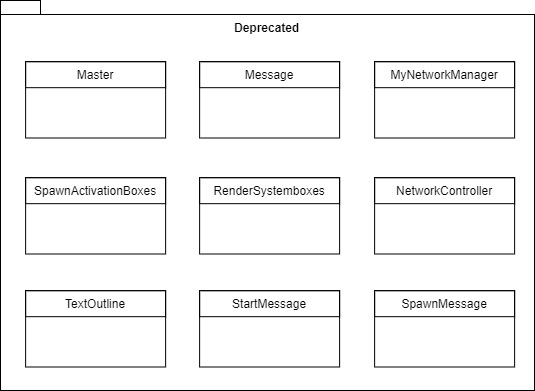


Figure 28 - Deprecated Package

# Physical View

A physical view was developed mid-way the project. It took form as deployment diagram and was used to envision the distribution of the DAVE system. The physical view consists on 3 main parts; the cloud hardware, the InstructorClient and the StudentClient. The cloud hardware consists of; a database used to store the Instructor account information, meaning username and password, a Broker used to publish and subscribe diagrams, and a Simulator used to animate and simulate diagrams. The Simulator intern consists of a Parser used to parse JSON data, a diagram coordinator that creates Docker Images with the parsed information and lastly a MQTT client coordinator used to pass the information between parser and diagram coordinator.

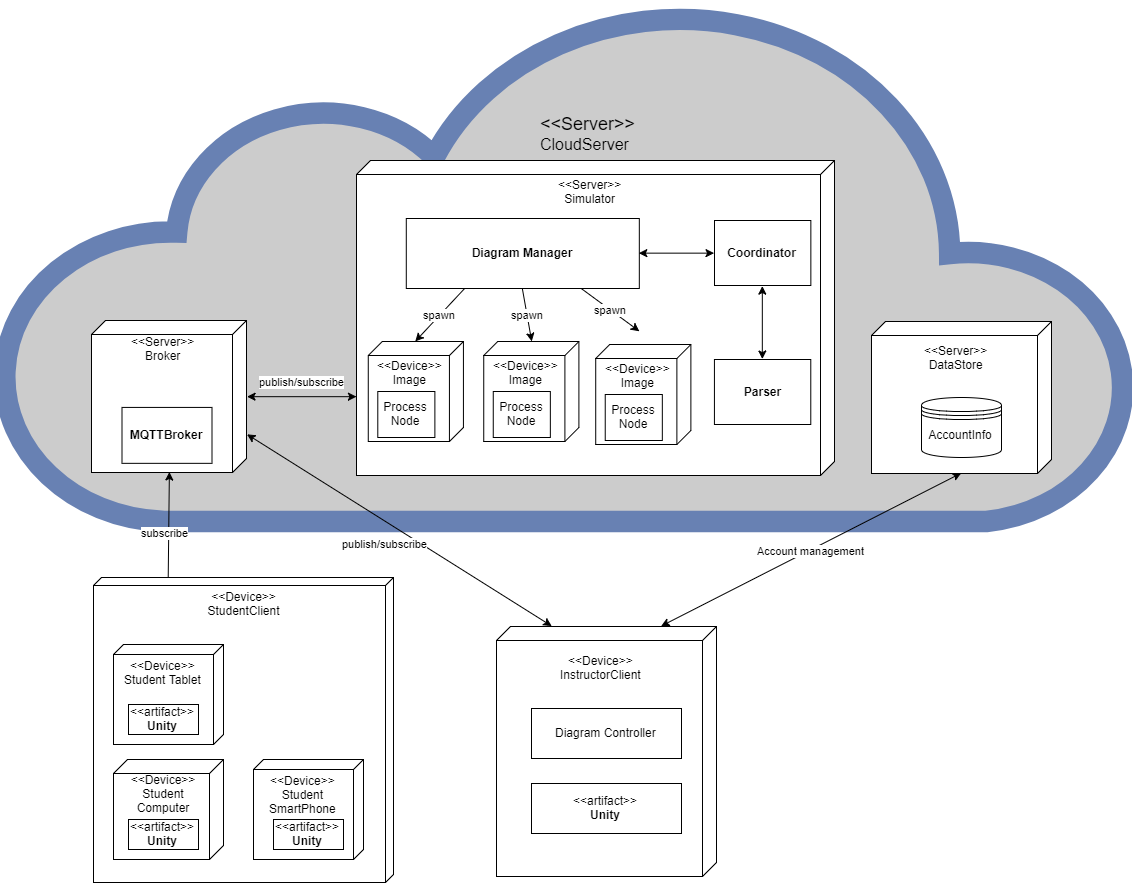


Figure 29 - DAVE Deployment Diagram

# Conceptual Framework

To improve the visualization of DAVE’s solution for UML diagram animation and simulation a conceptual framework has been established. A Domain Lexicon was created to identify the important and relevant areas of the problem domain. A Lexicon Diagram was then created to further identify the interactions between these terms.

## Domain Lexicon

**Student** – simple DAVE users, that only have diagram viewing capabilities. To view a diagram the students must select a diagram and instructors via their names. Before viewing the Diagram, the student must enter their names, to help Instructors identify them.

**Instructor** – the main user of DAVE’s application. These users require a simple account creation, which consists of username and password. Instructors have Diagram publishing capabilities, which is done by uploading a DiagramFile.

**DiagramFile** – JSON file containing information of UML diagrams. Class, Sequence and Deployment diagrams are supported. All DiagramFiles must contain a “type” key to identify between different types of UML diagrams and have standardized JSON structure of the types.

**FileBrowser** – an interface used by Instructors to select which DiagramFiles to upload. The DiagramFiles can be uploaded one by one or multiple at a time.

**Diagram** – graphical representation of the DiagramFile. The Diagram, has a variable diagram\_name to help distinguish between the different Diagrams. The Diagram is published by an Instructor, therefore the instructor\_name is used to identify the individual who has uploaded the DiagramFile.

**Class** – UML Class diagram. The standardized structure in file is divided into 2 main categories; classes and relationships. The classes are structured as lists; each UML class within the file must contain a “name” key and is accompanied by a JSON string value, which represents the class’s name. Moreover, the class must contain a “field” key and must be accompanied by a list of values, which represent the class’s attributes. Additionally, the Class DiagramFile, must contain a list of “relationships”. Each value in the list must contain 3 fields; “type”, “subclass” and “superclass”. The type denotes the relationships between the subclass and superclass. The superclass denotes the parent class and the subclass, the child of the parent.

**Sequence** – UML System Sequence diagram. The standardized structure in file is divided into 2 main categories; processes and diagram. The processes are representation of the lifelines’ systems. The processes are structured as lists; each value must have a “class” key and “name” key. The “class” represents the system components. Additionally, the Class DiagramFile, must contain a list of “diagram”. The “diagram” is divided into subcategories. The sub components are divided to “node” and “content”. The node components represent the “content” role in the system. The “content” which is divided into 4 other parts. The “content” possesses a “node” with accompanied value “send”, indicating system sequence diagrams’ messages. If the node is “send” then the key “from” and “to” indicates the message direction and position. The final key “message” indicates the messages contents.

**Deployment** – UML Deployment diagram. The standardized structure in file must contain a “mapping” key. The value of the key is a list, with each value containing “process” and “device” keys. The “device” represents the physical hardware of the deployment diagram, whereas the “process” represent the software on the hardware devices.

**WebApp** – part of networking layer of DAVE’s system; located on AWS. Contains the DAVE WebClient and Parser.

**WebClient** – the WebClient is responsible for receiving the uploaded DiagramFile and converting it into a Diagram. Once the DiagramFile is uploaded it is passed to the Parser, to retrieve the relevant information. The information is then passed to DiagramCoordinator, which creates the Diagram by simulating it. The WebClient is located on the WebApp.

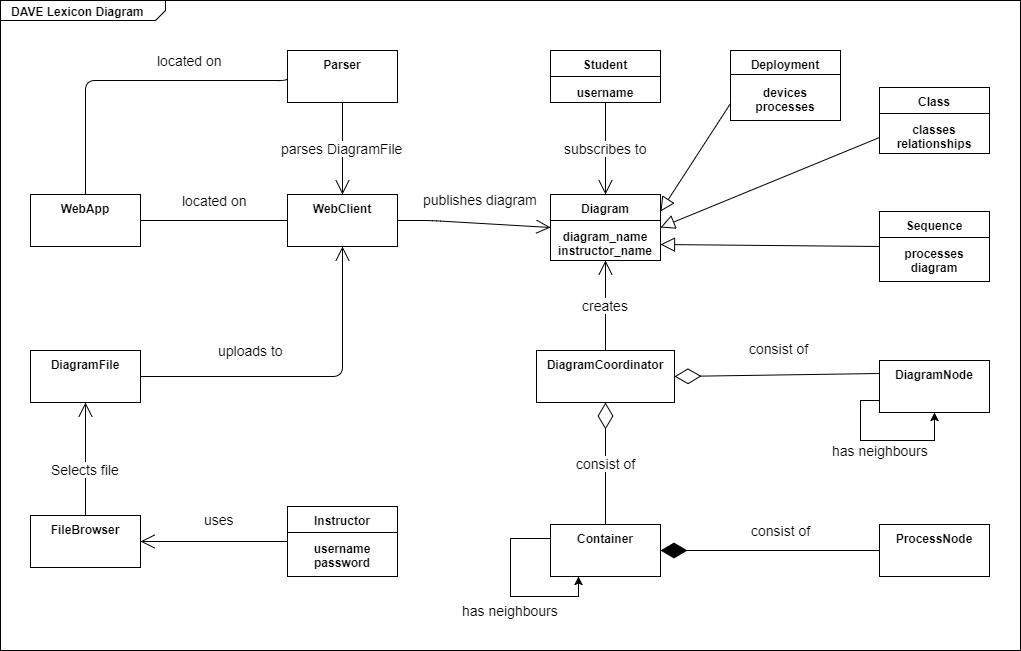
**Parser** – takes in the DiagramFile and parses the information from it. The Parser requires a correctly structured DiagramFile to function. The parser can distinguish and return information for Class, Sequence and Deployment diagrams. The Parser is located on the WebApp.

**DiagramCoordinator** – a coordinator that is used to distribute and simulate the parsed information. The DiagramCoordinator takes in the Sequence Diagram’s “processes” information and creates a Container for each individual process, which is accompanied with the “content” data from the processes “diagram”. The number of Containers depends on the number of process within the DiagramFile.

**Container** – a Container is a software device created by the DiagramCoordinator to store ProcessNodes. The Containers are created in case the Diagram’s type is Sequence. Each Container store 1 ProcessNode. Multiple Containers can exist simultaneously, depending on DiagramFile, and the communication between them is based on the ProcessNode data. The Container name depends on the “process” value.

**ProcessNode** – stores the parsed data retrieved from the “content” key that is found under “diagram” within the Sequence Diagram. The ProcessNode’s send the “message” information from one Container to another.

**DiagramNode** – are created when the Diagram is either Class or Deployment. They store the parsed information found under the standardized JSON key’s. The DiagramNodes number depends on the number of values found under the keys’, therefore it can have neighbours.

****

# Architectural risks

Several architectural risks emerged from the team’s solution. This section is aimed to identify and explain architectural risks. Additionally, the team’s mitigation strategy is provided. The risks were expressed in table form. They are accompanied by impact, a value showing the severity from 1 to 5, and probability, a value showcasing the likeliness of the risk occurring. The magnitude is calculated by multiplying the two values. For risk management the team has assigned conventions for expressing probability. The conventions are aimed at establishing risk thresholds, mainly 25%, 50%, 75% and 100%. If a risk has <25% of occurrence, it is said to be low. The 50% percent threshold is used to express medium probability, whereas 75% is used to express a high probability. The 100% probability is used to express risk that will occur with certainty. These thresholds are used, due to the team’s lack of experience with risk management and risk estimation.

## Risk Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Risk | Impact | Probability | Magnitude | Mitigation Strategy |
| Run out of Amazon Web Service Hours | 5 | 25% | 1 | * Use only 1 instance of AWS * Maintenance downtime |
| Fault recovery is untestable | 1 | 100% | 1 | * Enable scripts to autoboot * Enable monitoring. * Let AWS handle fault recovery |
| Containers overload AWS instance | 5 | 25% | 1 | * Limit the number of containers used on 1 instance |
| MQTT Broker overloaded | 5 | 25% | 1.25 | * Have a separate AWS instance for a broker |
| Database failure | 5 | 25% | 1.25 | * Explain the situation to the user through an error message |
| Message not delivered | 5 | 25% | 1.25 | * Used MQTT Quality of Service 2 (four step handshake) * Additional data confirmation |

Table 12 - Risk Table

## Risk description

### Amazon Web Service Hours

The team has incorporating AWS into the project, which provides a limited number of hours per month that the AWS instance can run. The number of hours is 720. By using a single the development team can maximize DAVE deployment hours, which correlate to 30 days for a single instance. The maintenance downtime would be used in case the month had 31 days in it. By doing so the team would save cost and avoid payment. The impact was assigned a value of 5, due to the team not having any financial backing, thus forcing the team to shut off operations of DAVE. However, the is highly improbable that the risk will occur, due to the short amount of time that DAVE will be deployed for. Therefore, the risk has a low magnitude.

### Fault Recovery

Due to the nature of AWS it is impossible to make the instance crash or have an outage. The team is not capable of replicating things that Amazon deals as failures. The probability is set to high, due to the lack of control over configuration settings. However, Amazon capabilities for fault toleration far exceed those of the team, therefore the impact is set to 1. In case of crash the scripts are autobooted.

### Container Overload

The team is dependent on the performance capabilities of a single instance, provided by Amazon. Free tier is used to obtain the instance, however said instance is of low performance. Test were carried to examine the its capabilities. The team was not able to make the instance crash. It is hypothesized that 100 instructors are required to upload diagram files that create 10 containers each to crash the instance. This sort of use will most likely be not seen during DAVE’s lifespan. In case the risk does occur, it is easy to scale the system to allow it to handle such extensive number of users, this would however would require paying AWS. Due to these factors, the risk was assigned a low probability of occurrence. On the other hand, the impact would be devastating, since any process relying on the MQTT broker would fail.

### MQTT Broker Failure

Entire system is constructed based around the MQTT Broker, in case it breaks or has a significant delay, DAVE will become unreliable. Therefore, the impact was set to high. Stress testing was used to determine the brokers capabilities. The stress test was composed of 10 MQTT clients simultaneously publishing to the broker. The clients published a 100 sequence and 100 class diagrams within 10 seconds with Quality of Service 2 each. All of which were delivered as intended. Therefore, it is believed that MQTT broker is more than fit for DAVE and unlikely to fail. However, if a risk does occur the system will become unreliable with its operations, which would go against the team’s non-functional requirements (QR7). Therefore, the impact was assigned a high value.

### Database Failure

A major design failure was identified via this risk. The students are required instructor names and room names to progress in the program. If the database is unresponsive the users will not be able to progress to diagram animation and simulation, even though the MQTT broker and other parts of the application are working. Thus, making the application seemingly worthless. This risk was assigned a high impact value, and this part of the system was identified as an area in need of design change. The risk, was identified too late to make any major changes to DAVE, thus the team focused more on improving fault prevention. However, the team has incorporated a trustworthy external package, RethinkDB for this function together with applied tactics for availability (See section 5.3.1), consequently reducing the probability of risk occurrence and applying every measure possible to reduce risk occurrence likeliness.

### Message Delivery

By incorporating the Publish/Subscribe style, there is always a risk that the messages sent might be dropped, as this is one of the style’s drawbacks. However, the team chose to use Quality of Service 2 and an additional data conformation as mitigation strategies to combat this risk (See section 5.3.5). Due to the DAVE’s reliance on message delivery, the impact has been assigned the maximum value. However, due to the effectiveness of the mitigation strategies the likeliness was deemed as low.

# Conclusion

In conclusion, the desired non-functional requirements (See section 5.2) of DAVE were met in varying degree. The solutions to developing the system were intended to advance the quality of DAVE, yet they did in some cases entail limitations.

By using Amazon CloudWatch to monitor our server instance we enable the use of recovery tactics to achieve availability. However, AWS does neither provide much control nor insight into how the fault detection and recovery tactics are implemented.

The publish-subscribe style supports scalability which allows for DAVE to be distributed amongst a dynamic number of users. When an instructor publishes a room, students can easily find the room if they know the instructor’s username and the name of the room. The downside of this chosen style is that it cannot guarantee that subscribers receive the messages. This can influence the reliability of the system as the animation of the simulation are not only highly dependent on that the messages are being received by the subscribers. A mean for ensuring message delivery was to incorporate a conformation mechanism into the system.

A layered style was incorporated as the general structure of DAVE. It was chosen because of the ease of performing unit, system and requirements testing tests it provided. The layered style supports adding layers both throughout development and maintenance. The style proved to be a good choice when developing our system, as layers were added during the evolution of DAVE (See section 6.1).

Even though the team estimates the response measure in quality attribute scenario (S4) to be feasible, we can foresee the great amount of effort necessary to implement some of the other planned extensions. This is because of the level of coupling and separation of concerns not being done to their satisfaction. The cohesion within the system is found to be acceptable and would support future extension and the modifiability of DAVE.

For optimal prospect of the different types of diagrams, DAVE requires control over the view. As the controls were not necessarily intuitive, a control key panel indicating which view is currently activated, was added and can be displayed throughout the entire visit inside a room. This, together with consistency and instant gratification, furthers the usability of DAVE.

To avoid publishing sensitive data to the wrong subscriber, the client-server model is used instead of the publish-subscribe to communicate with the database. The password data in the database is encrypted according to SHA256 standard. This sums up the security of the system.

To achieve DAVE several different styles were incorporated; client-server, layered and publish-subscribe (See section 10). The stylistic choices provided low coupling between components helping realize the modifiability quality requirements. However, due to the use of IP addresses when implementing client-server and publish-subscribe styles, an ongoing connection was not able to be establish through WebGL.