UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Team Brooks
Architectural Requirements Specifications and Design of QuantumUP

Seonin David u15063021

Kudzai Muranga u13278012

Francois van der Merwe u18358137

Priscilla Madigoe u13049128

Contents

1 Execution Manager **4**
1.1 Performance Requirements 4
1.2 Design Constraints 4
1.3 Software System Attributes 4
1.4 Class diagram 4
1.5 Activity diagram 5
1.6 Sequence diagram 6
1.7 State diagram 7
1.8 Use Case diagram 7

2 Users Unit **8**
2.1 Performance Requirements 8
2.2 Design Constraints 8
2.3 Software System Attributes 8
2.4 Class diagram 8
2.4.0.1 Design Patterns Used 8
2.5 Activity diagram 9
2.6 Sequence diagram 9
2.7 State diagram 10
2.8 Use Case diagram 11

3 Input Unit **12**
3.1 Performance Requirements 12
3.2 Design Constraints 12
3.3 Software System Attributes 12
3.4 Class diagram 13
3.5 Activity diagrams for Input Unit 14
3.5.1 Create Algorithm 14
3.5.2 Remove Algorithm 15
3.5.3 Get Algorithm 16
3.5.4 Update An Algorithm's Name 17
3.6 Sequence diagrams for Input Unit 18
3.6.1 Create Algorithm 18
3.6.2 Remove Algorithm 19
3.6.3 Get Algorithm 20
3.6.4 Update An Algorithm's Name 21
3.7 State diagrams for Input Unit 22
3.7.1 Create Algorithm 22
3.7.2 Remove Algorithm 23
3.7.3 Get Algorithm 24
3.7.4 Update An Algorithm's Name 25
3.8 Use Case diagram 26

4 Admin UI **27**
4.1 Performance Requirements 27
4.2 Design Constraints 27
4.3 Software System Attributes 27
4.4 Class diagram 27
4.5 Activity diagram 28
4.6 Sequence diagrams 29
4.7 State diagram 34
4.8 Use Case diagram 35

5 Deployment Diagram **36**
5.1 Architectual Design 36
5.2 Technologies used(refer above for description of how theses technologies are used) 37

1 Execution Manager

1.1 Performance Requirements

- system should be able to handle 1000 concurrent requests
- Allocate nodes sequentially so they are executed in the order they arrived

1.2 Design Constraints

1. Ensuring efficient communication to the node manager and input unit
2. The hardware of the system should be able run in an isolated environment and continuously perform to the requirements of the experiment

1.3 Software System Attributes

Availability: This process is only accessible via the execution manager.

Modularity: By using the mediator pattern and encapsulating the modules it makes it easy to add or remove modules.

Coupling: By making each sub-system independent. It is easier to add and remove subsystems without it affecting other components in system thus resulting in low coupling.

1.4 Class diagram

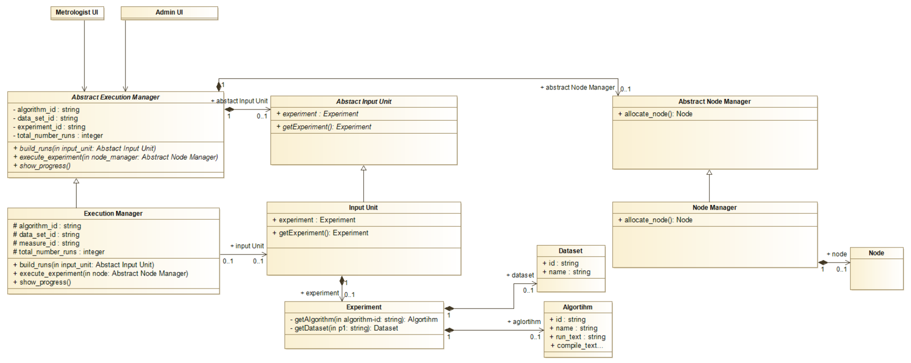

Figure 1: Class diagram for Execution Manager

Design Pattern Used

I used the mediator design pattern because the execution manager uses the input unit as well as the node manager to perform the build runs and execute experiment respectively. The execution manager also persists to the admin UI and the Metrologist UI thus a pattern that can communicate with the reinvent modules and not be extremely dependent is needed. That is why I selected the mediator pattern because it encapsulates the way that objects communicate with each other thus the relevant modules can communicate with each other without knowing the components underlying structure. By using this pattern the modules will be loosely coupled which makes the overall structure of the system much better and easier to manage.In this case the mediator is the **Execution Manager** is the Mediator. The **Input Unit** and **Node Manager** are the colleagues of the mediator. The **Metrologist** and **Admin UI** are the colleagues and as you can see modules are encapsulated.

1.5 Activity diagram

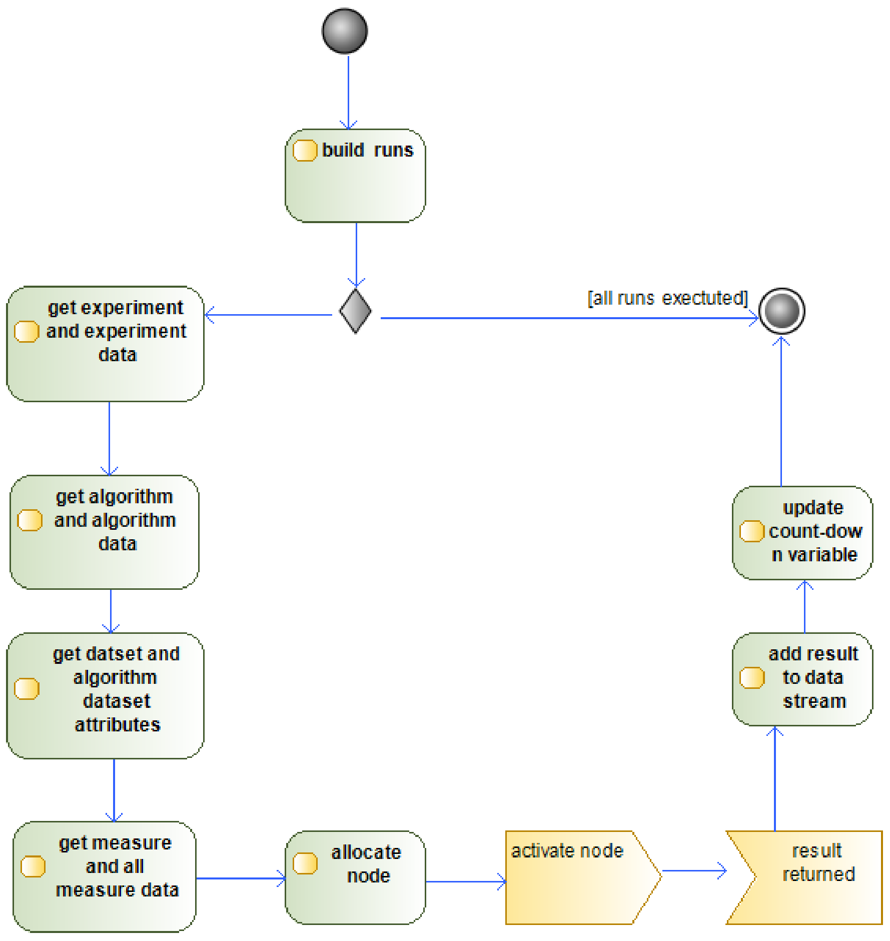
build runs
[all runs exectuted]
get experiment and experiment data
get algorithm and algorithm data
get datset and algorithm dataset attributes
get measure and all measure data
allocate node
activate node
result returned
add result to data stream
update count-down variable

Figure 2: Activity diagram for Execution Manager

1.6 Sequence diagram

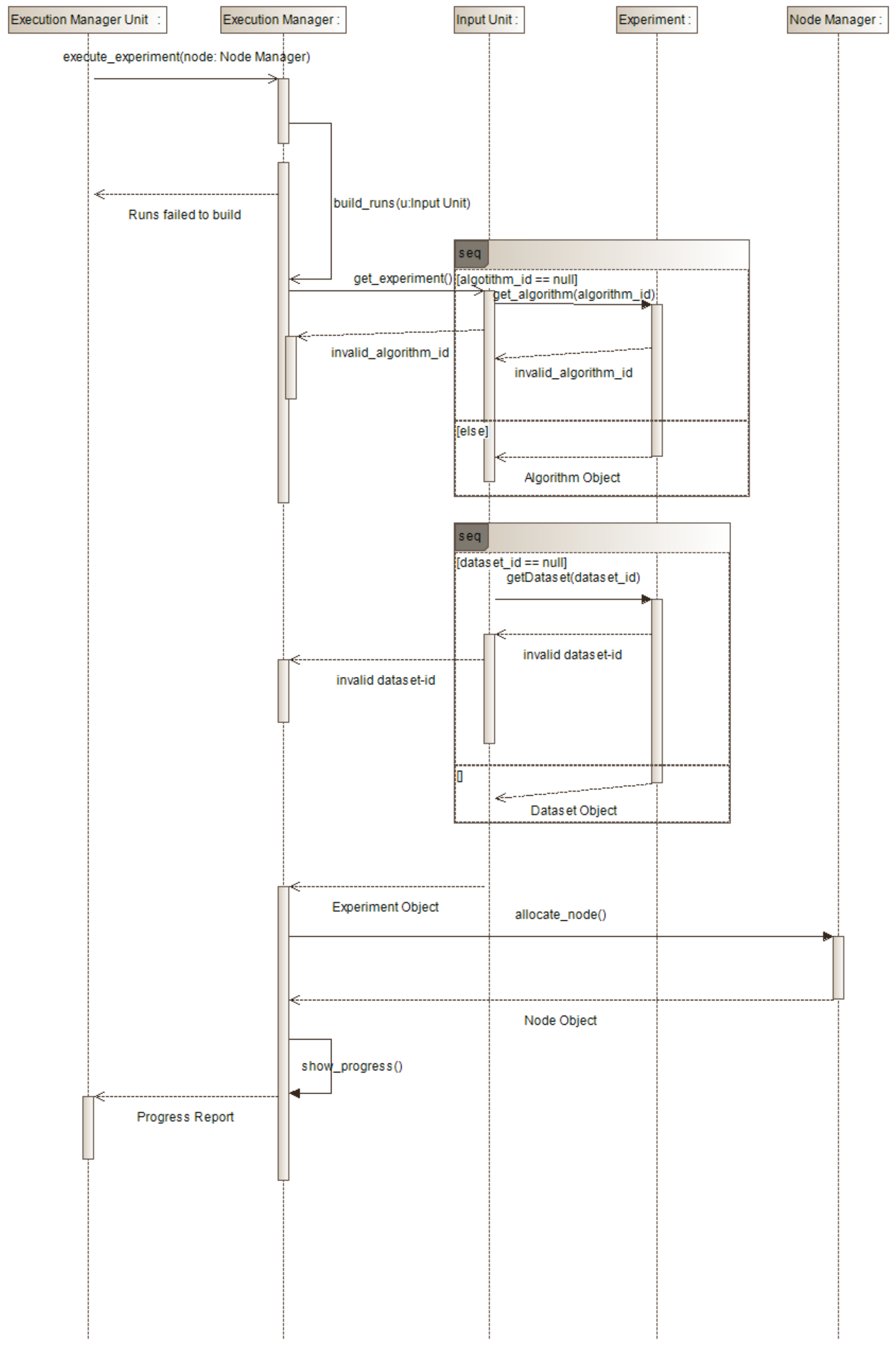

Figure 3: Sequence diagram for Execution Manager

1.7 State diagram

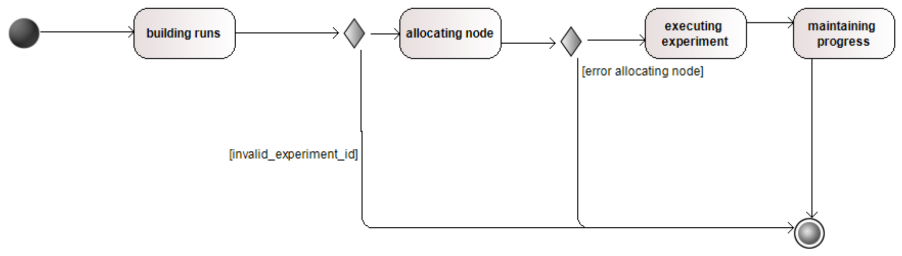

Figure 4: State diagram for Execution Manager

1.8 Use Case diagram

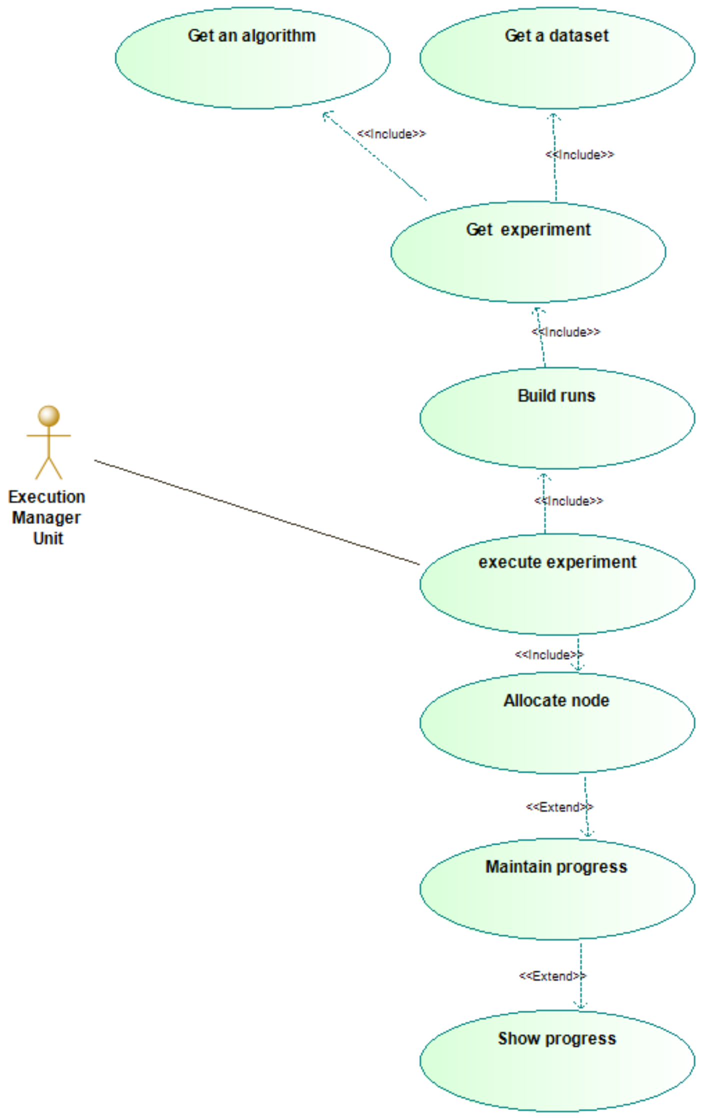

Figure 5: Use case diagram for Execution Manager

2 Users Unit

2.1 Performance Requirements

- **Response Time**: The system must allow users to log in and edit their profiles without minimum wait time. The response time must be the must not vary for the same function executed numerous times.

- **Scalability**: The system must perform adequately no matter how many users are using the system concurrently.

2.2 Design Constraints

2.3 Software System Attributes

Accuracy: Users' information must be correct and must record any changes to their profiles accurately.

Availability: Users' profile must always be available to the users whenever the need to log in or edit their profiles.

2.4 Class diagram

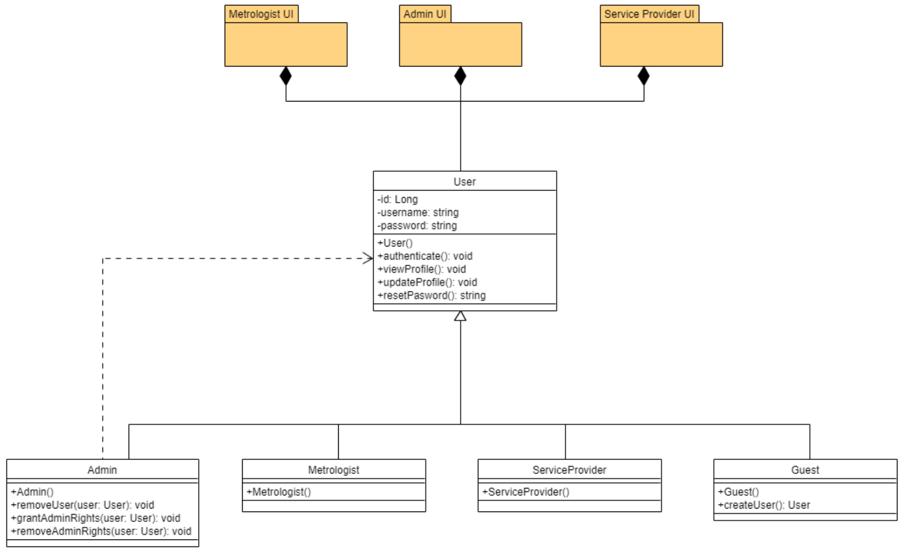

Figure 6: Class diagram for Users

2.4.0.1 Design Patterns Used The Template method design pattern was used for the Users sub-system. The different users of the benchmarking system, Hub admins, metrologists and service providers, have similar functions. They all have functionality to change profile information and to reset passwords. Template method allows code reuse for these shared functions while allowing the other user functionality to differ.

2.5 Activity diagram

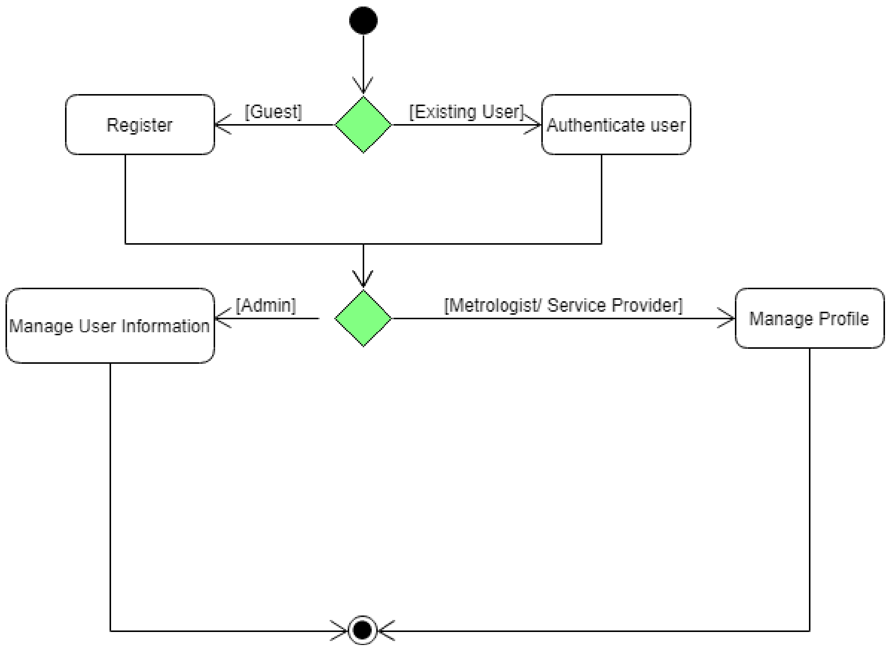

Figure 7: Activity diagram for Users

2.6 Sequence diagram

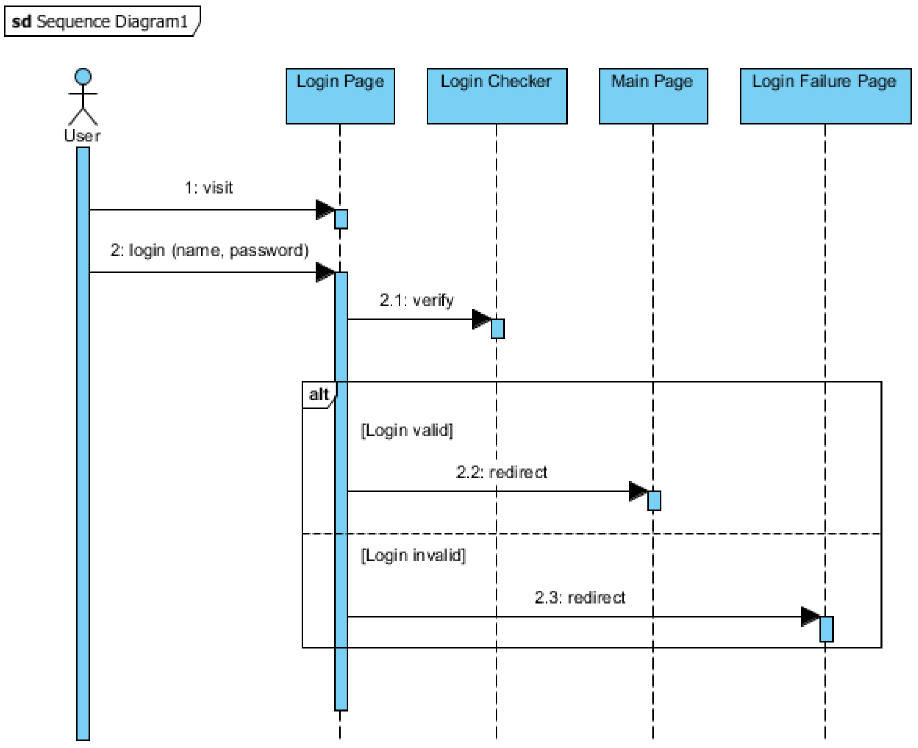

Figure 8: Sequence diagram for Users

2.7 State diagram

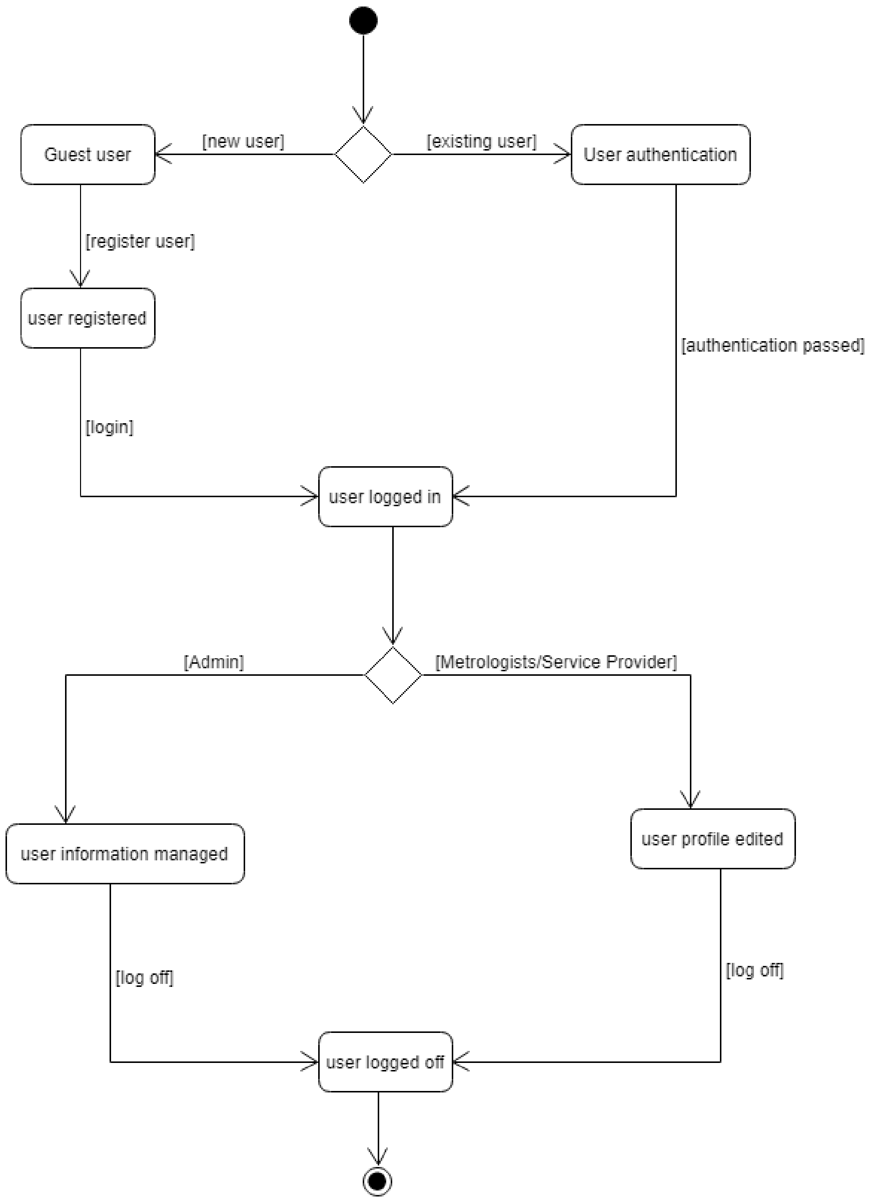

Figure 9: State diagram for Users

2.8 Use Case diagram

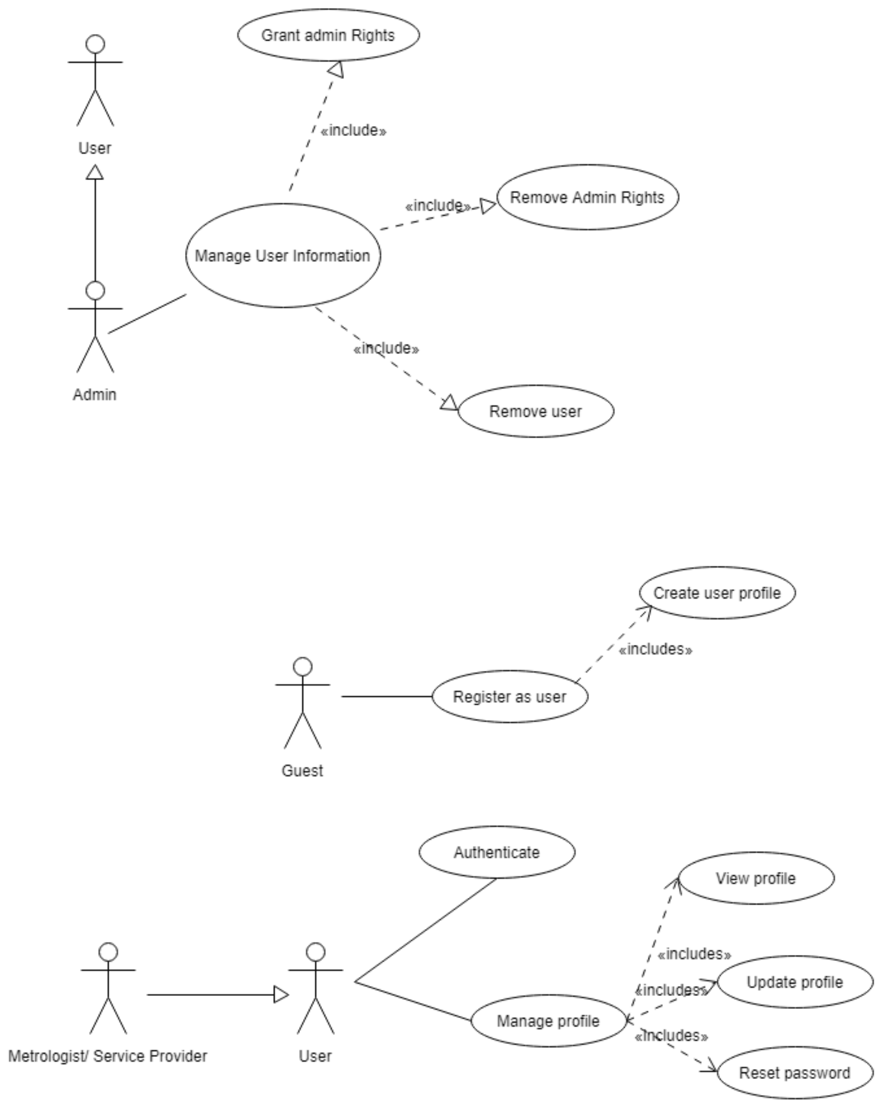

Figure 10: Use case diagram for Users

3 Input Unit

3.1 Performance Requirements

- Reliability: The correct algorithms, datasets and experiments should be returned when requested.

- Data Integrity: Consistent, correct and complete output should always be generated for all supported input to the Execution Manager and the Metrologist UI.

3.2 Design Constraints

1. The algorithms, datasets and experiments stored in the database for a specific user should be available to the input unit.
2. The input unit should be able to communicate with the execution manager and the Metrologist UI efficiently and whenever it needs to.

3.3 Software System Attributes

Extensibility: More future features and objects can be added to the input unit seamlessly without affecting the functionality of the code because the client and the underlying structure of the Experiment object have been decoupled.

Modifiability: The Builder design pattern enables the input unit to be highly customizable without affecting the client and other functionalities that depend on it negatively. As a result, the individual objects such as Algorithm and Datasets can be accessed independently or combined to form a more complex Experiment object without affecting the client's usage.

Robustness: Object modification an side-effect generations are minimized which ensures that few code failures are generated.

Availability: Objects generated will always be available to the Execution manager and the Metrologist UI on request.

3.4 Class diagram

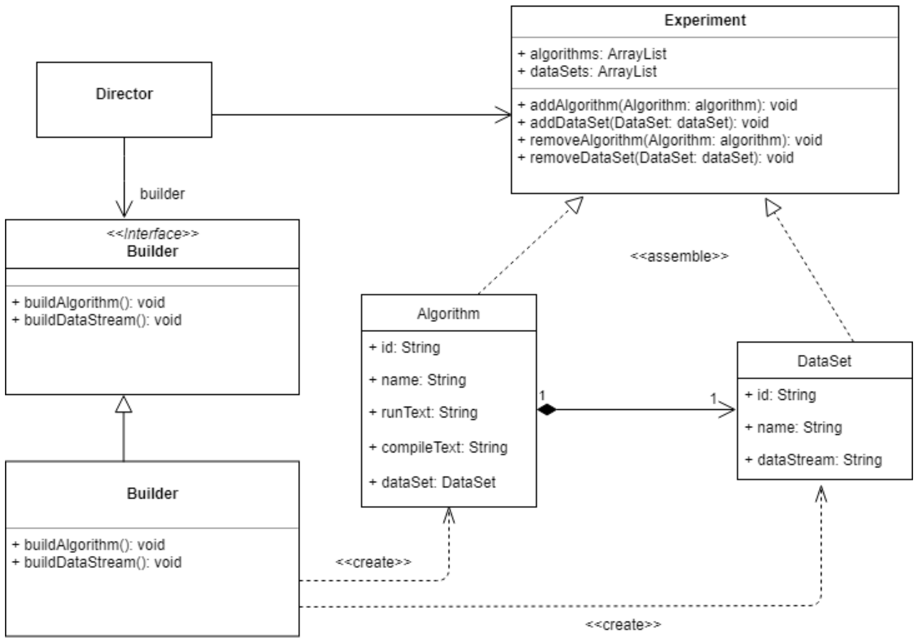

Figure 11: Class diagram for Input Unit

Design Pattern Used: I used the Builder design pattern because it enables the Algorithm and DataSet objects to be configurable so that multiple types of data sets, compilation environments and executables can be considered with minimal effort to the developer. The consumer of the service has no access to the underlying structure of the object requested and so decoupling is highly utilized. This also allows the client to directly instantiate the Algorithm and DataSet classes as self-standing objects that can be used for other purposes.

3.5 Activity diagrams for Input Unit

3.5.1 Create Algorithm

With regards to Experiment and Data sets, *mutatis muntandis* the same as regarding algorithms.

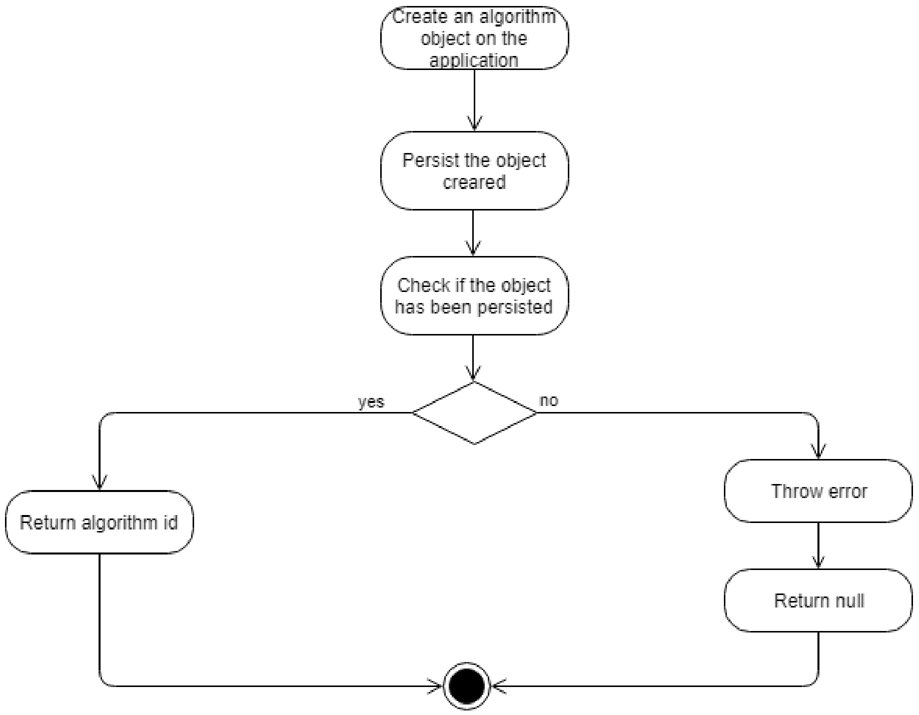
Create an algorithm object on the application
Persist the object creared
Check if the object has been persisted
yes
no
Return algorithm id
Throw error
Return null

Figure 12: Activity diagram for Creating an algorithm

3.5.2 Remove Algorithm

With regards to Experiment and Data sets, *mutatis muntandis* the same as regarding algorithms.

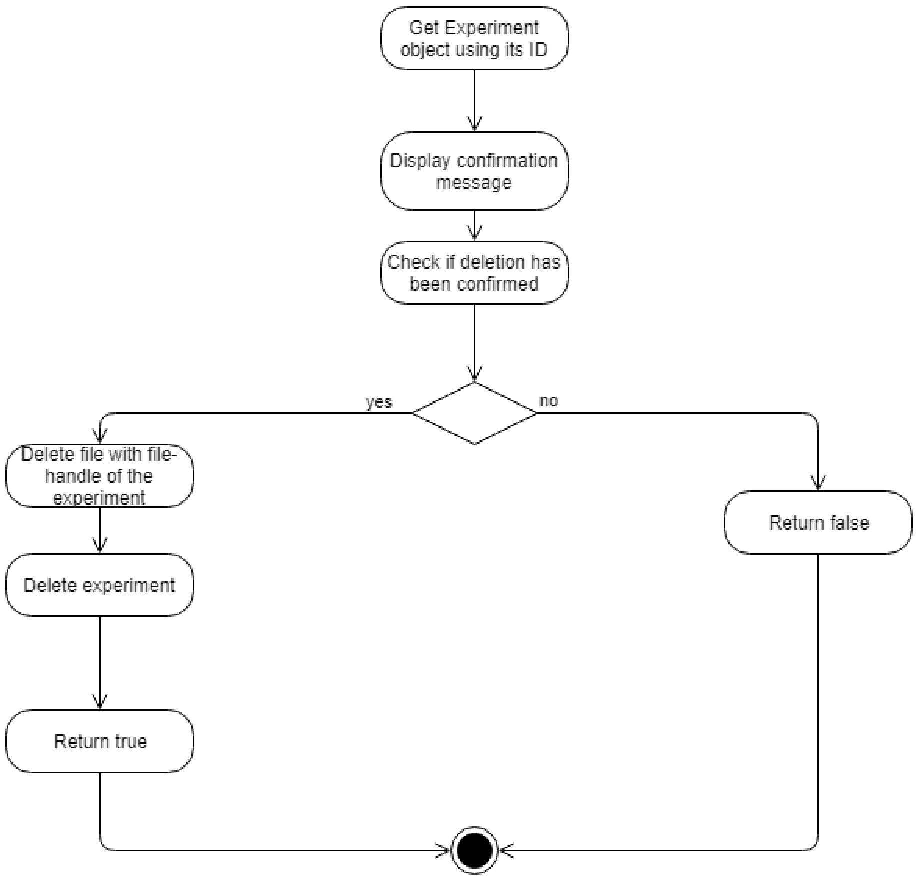
Get Experiment object using its ID
Display confirmation message
Check if deletion has been confirmed
yes
no
Delete file with file-handle of the experiment
Delete experiment
Return true
Return false

Figure 13: Activity diagram for Deleting an algorithm

3.5.3 Get Algorithm

With regards to Experiment and Data sets, *mutatis muntandis* the same as regarding algorithms.

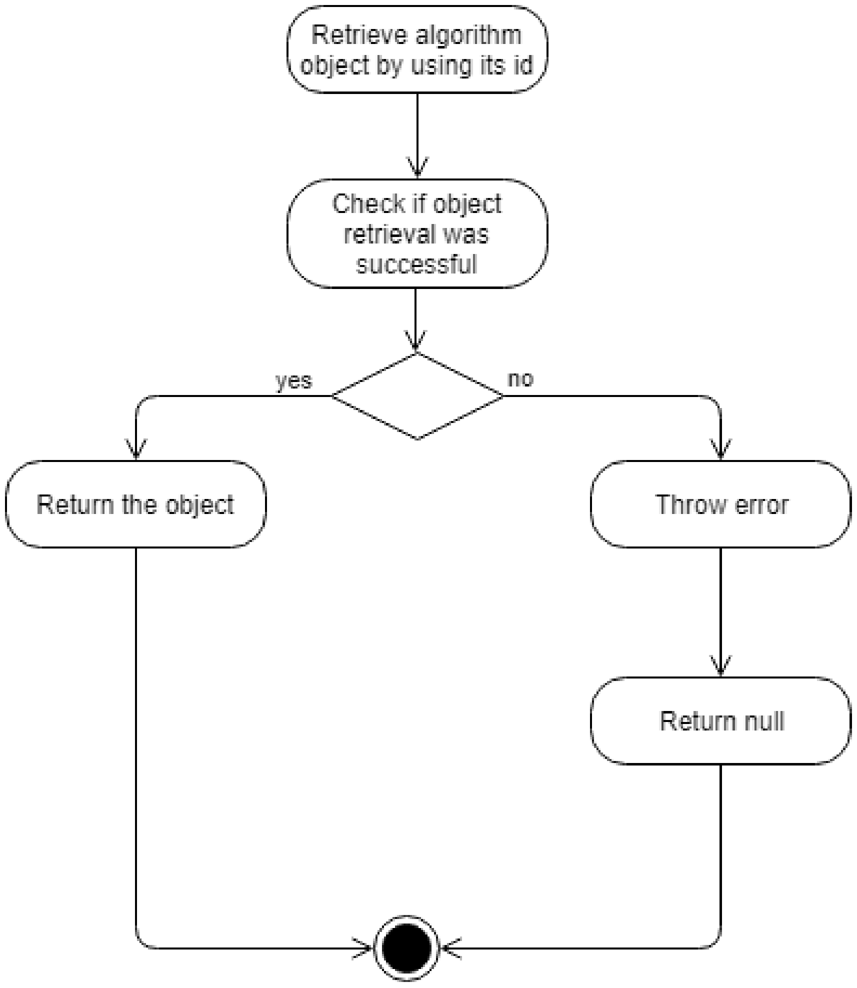
Retrieve algorithm object by using its id
Check if object retrieval was successful
yes
no
Return the object
Throw error
Return null

Figure 14: Activity diagram for getting an algorithm

3.5.4 Update An Algorithm's Name

With regards to Experiment and Data sets attributes, as well as other algorithm's attributes, *mutatis muntandis* the same as regarding an update to an algorithm's name.

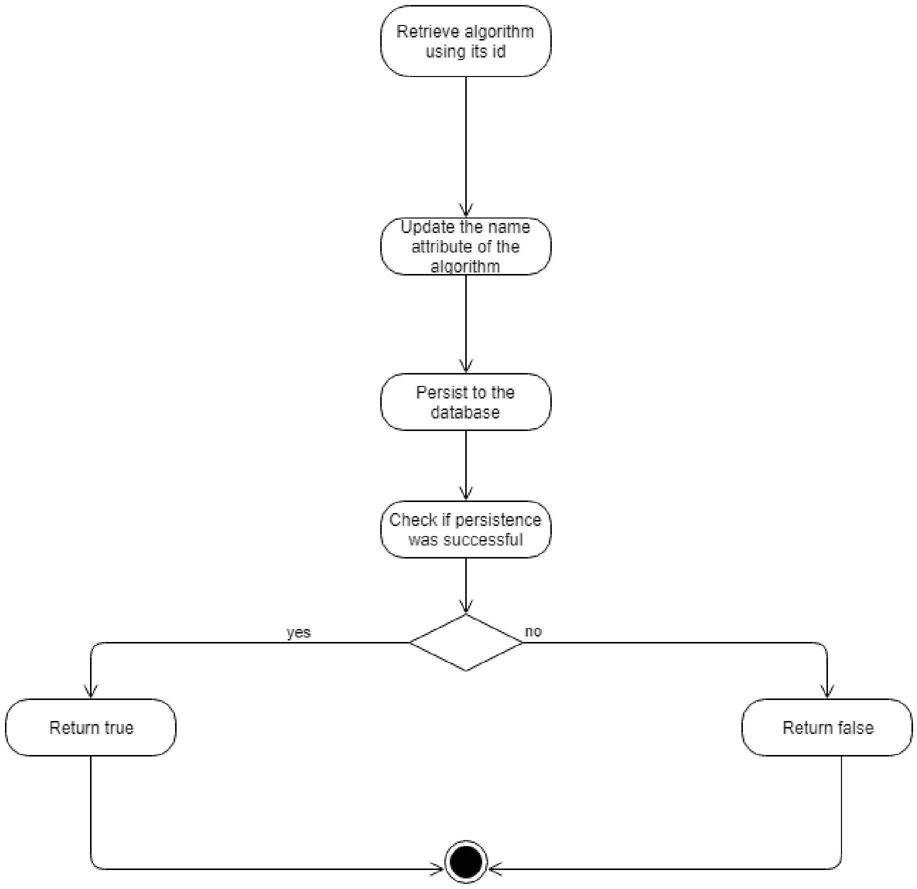
Retrieve algorithm using its id
Update the name attribute of the algorithm
Persist to the database
Check if persistence was successful
yes
no
Return true
Return false

Figure 15: Activity diagram for updating an algorithm's name attribute

3.6 Sequence diagrams for Input Unit

Please note that a "service consumer" is a general term to group the Execution Manager and Metrologist UI consumers that use the Input Unit.

3.6.1 Create Algorithm

With regards to Experiment and Data sets, *mutatis muntandis* the same as regarding algorithms.

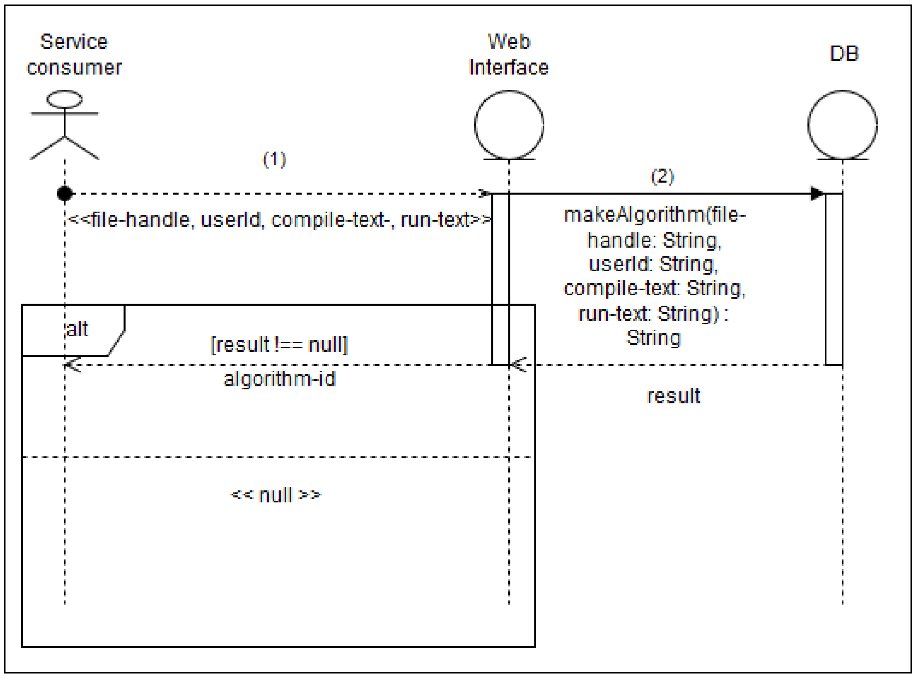
Service consumer
Web Interface
DB
(1)
(2)
<<file-handle, userId, compile-text-, run-text>>
makeAlgorithm(file-handle: String, userId: String, compile-text: String, run-text: String) : String
alt
[result !== null]
algorithm-id
result
<< null >>

Figure 16: Sequence diagram for Creating an algorithm

3.6.2 Remove Algorithm

With regards to Experiment and Data sets, *mutatis muntandis* the same as regarding algorithms.

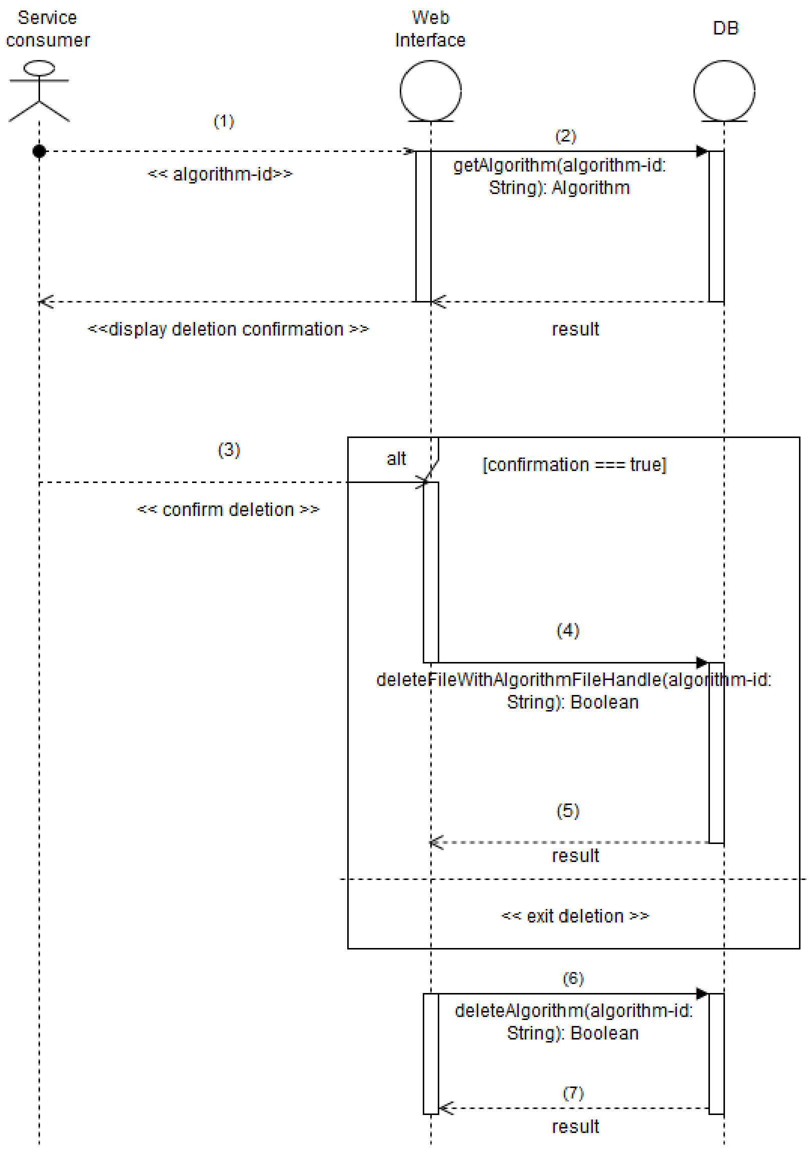
Service consumer
Web Interface
DB
(1)
<< algorithm-id>>
(2)
getAlgorithm(algorithm-id: String): Algorithm
<<display deletion confirmation >>
result
(3)
alt
[confirmation === true]
<< confirm deletion >>
(4)
deleteFileWithAlgorithmFileHandle(algorithm-id: String): Boolean
(5)
result
<< exit deletion >>
(6)
deleteAlgorithm(algorithm-id: String): Boolean
(7)
result

Figure 17: Sequence diagram for Deleting an algorithm

3.6.3 Get Algorithm

With regards to Experiment and Data sets, *mutatis muntandis* the same as regarding algorithms.

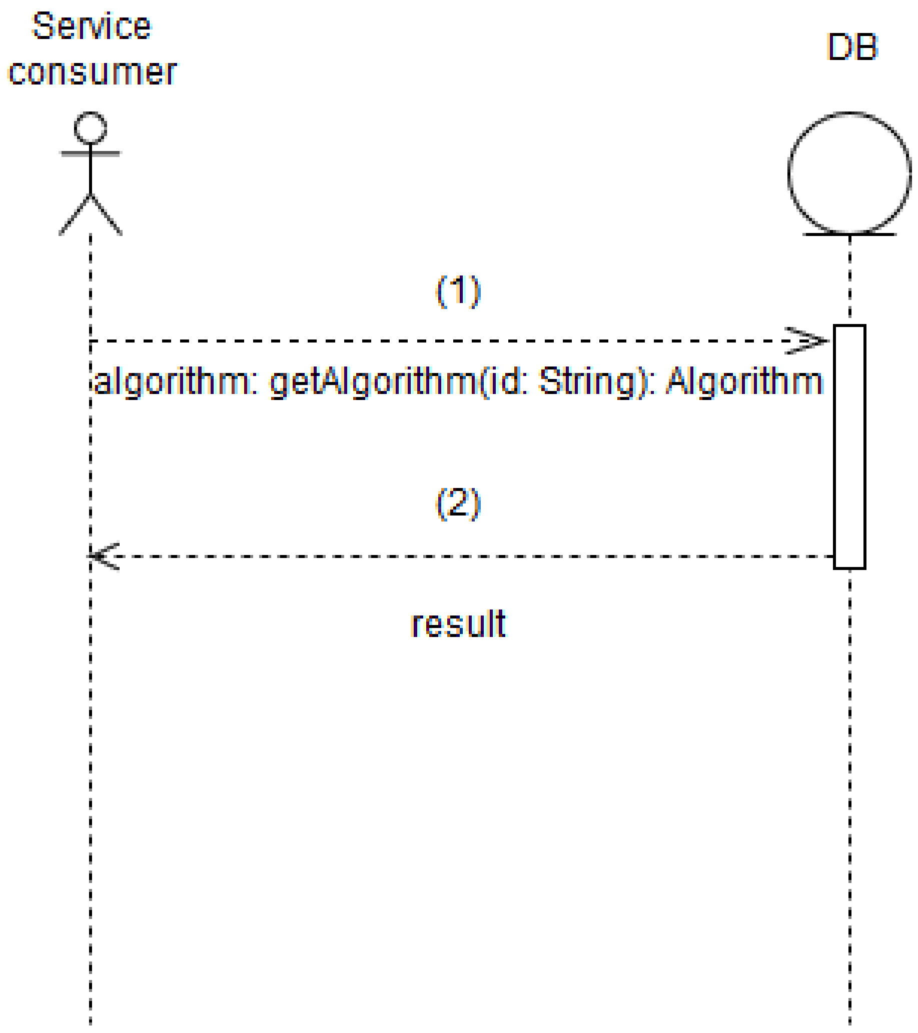

Figure 18: Sequence diagram for getting an algorithm

3.6.4 Update An Algorithm's Name

With regards to Experiment and Data sets attributes, as well as other algorithm's attributes, *mutatis muntandis* the same as regarding an update to an algorithm's name.

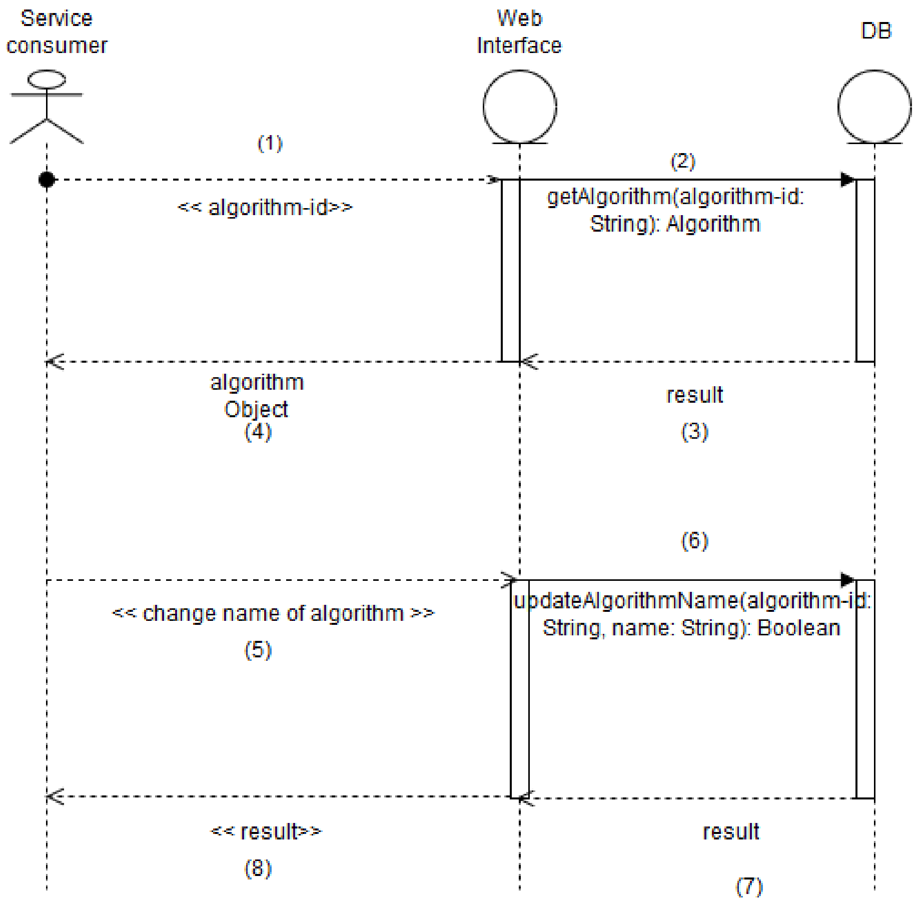
Service consumer
Web Interface
DB
(1)
<< algorithm-id>>
(2)
getAlgorithm(algorithm-id: String): Algorithm
algorithm Object
(4)
result
(3)
(6)
<< change name of algorithm >>
(5)
updateAlgorithmName(algorithm-id: String, name: String): Boolean
<< result>>
(8)
result
(7)

Figure 19: Sequence diagram for updating an algorithm's name attribute

3.7 State diagrams for Input Unit

3.7.1 Create Algorithm

With regards to Experiment and Data sets, *mutatis muntandis* the same as regarding algorithms.

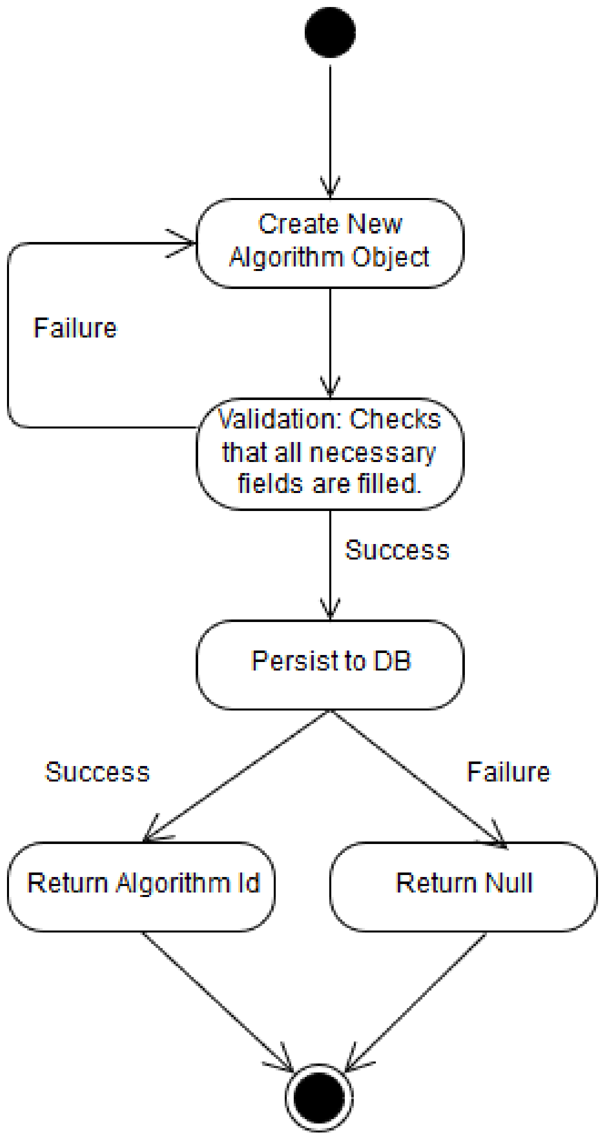
Create New Algorithm Object
Failure
Validation: Checks that all necessary fields are filled.
Success
Persist to DB
Success
Failure
Return Algorithm Id
Return Null

Figure 20: State diagram for Creating an algorithm

3.7.2 Remove Algorithm

With regards to Experiment and Data sets, *mutatis muntandis* the same as regarding algorithms.

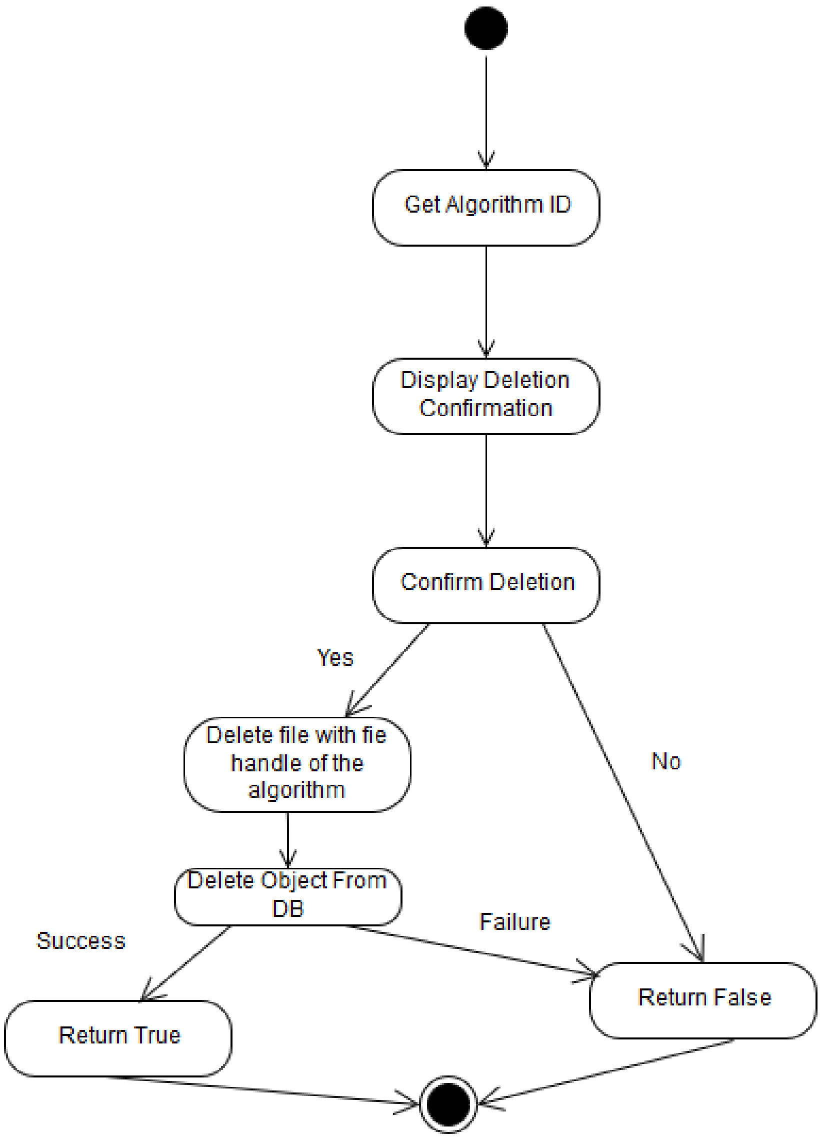

Figure 21: State diagram for Deleting an algorithm

3.7.3 Get Algorithm

With regards to Experiment and Data sets, *mutatis muntandis* the same as regarding algorithms.

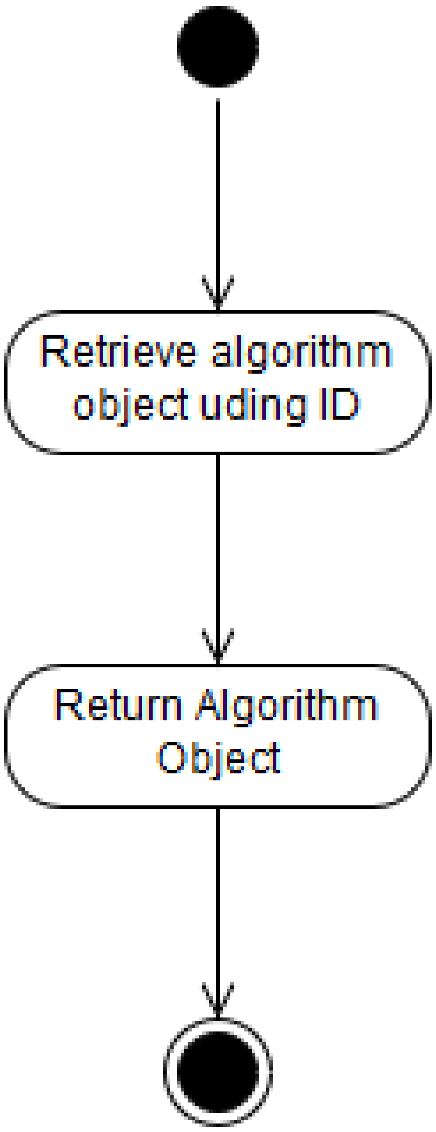

Figure 22: State diagram for getting an algorithm

3.7.4 Update An Algorithm's Name

With regards to Experiment and Data sets attributes, as well as other algorithm's attributes, *mutatis muntandis* the same as regarding an update to an algorithm's name.

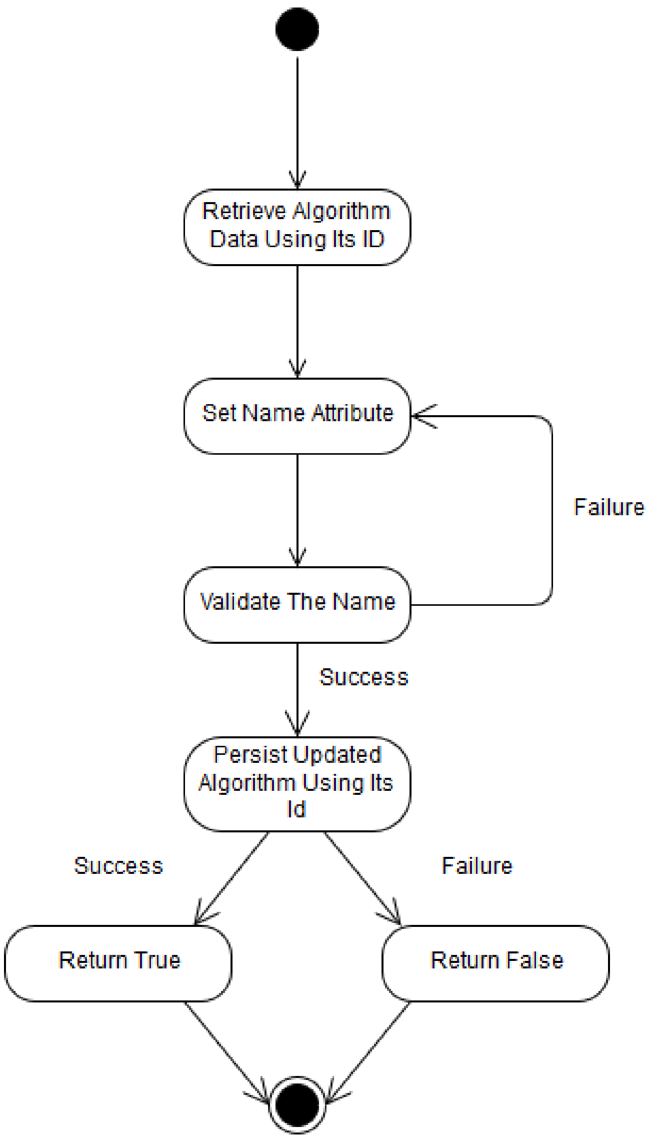

Figure 23: State diagram for updating an algorithm's name attribute

3.8 Use Case diagram

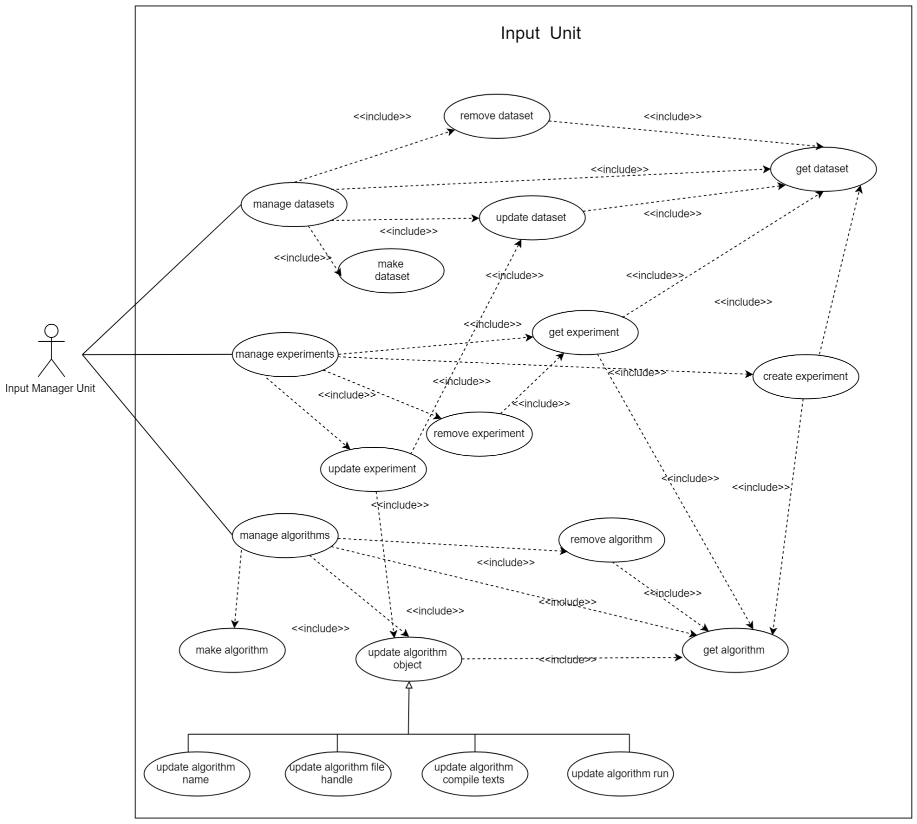

Figure 24: Use case diagram for Input Unit

4 Admin UI

4.1 Performance Requirements

- Since the admin side does not handle the execution side of the project. Little to no performance requirements are needed.
- All that is needed is a decent connection to the system from the admin to the server at times when nodes are being used. If there is an emergency, then the admin needs to be able to kill nodes.

4.2 Design Constraints

1. The admin needs connection to the system at all times and this can't be ensured.
2. A connection must be able to be obtained between the system and the database.
3. There also needs to be a connection maintain between the nodes and the system.

4.3 Software System Attributes

Availability: The admin should be able to access the system at any computer since it is a web app. The admin controls would only be allocated to those that were made admin and also only if they were granted certain privileges such as being able to kill a node or create a user.

4.4 Class diagram

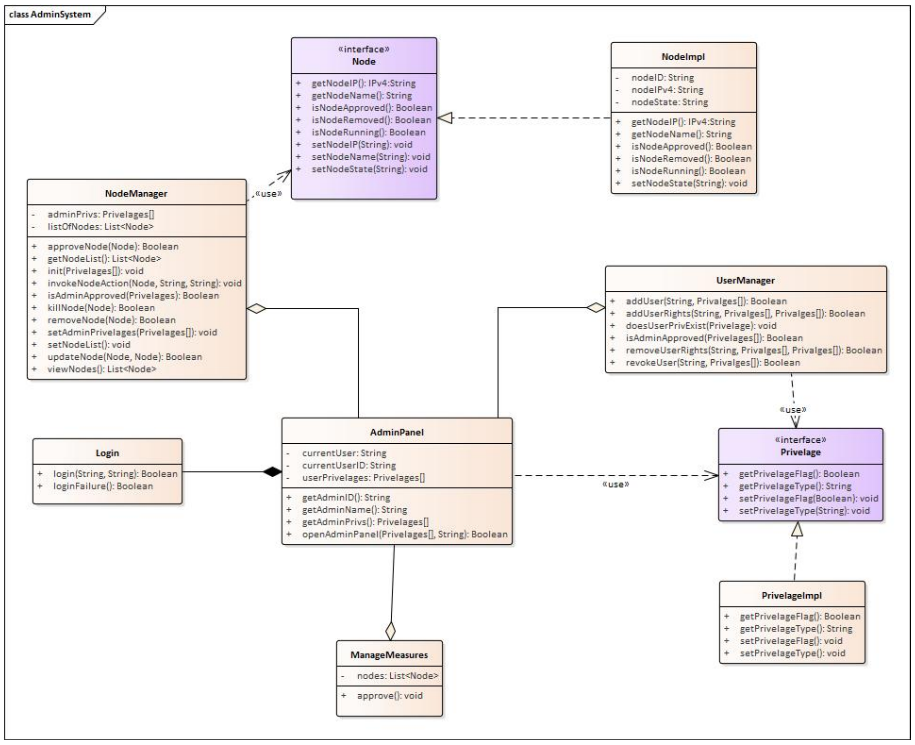
class AdminSystem
«interface»
Node
+ getNodeIP(): IPv4:String
+ getNodeName(): String
+ isNodeApproved(): Boolean
+ isNodeRemoved(): Boolean
+ isNodeRunning(): Boolean
+ setNodeIP(String): void
+ setNodeName(String): void
+ setNodeState(String): void
NodeImpl
- nodeID: String
- nodeIPv4: String
- nodeState: String
+ getNodeIP(): IPv4:String
+ getNodeName(): String
+ isNodeApproved(): Boolean
+ isNodeRemoved(): Boolean
+ isNodeRunning(): Boolean
+ setNodeState(String): void
«use»
NodeManager
- adminPrivs: Privelages[]
- listOfNodes: List<Node>
+ approveNode(Node): Boolean
+ getNodeList(): List<Node>
+ init(Privelages[]): void
+ invokeNodeAction(Node, String, String): void
+ isAdminApproved(Privelages): Boolean
+ killNode(Node): Boolean
+ removeNode(Node): Boolean
+ setAdminPrivelages(Privelages[]): void
+ setNodeList(): void
+ updateNode(Node, Node): Boolean
+ viewNodes(): List<Node>
UserManager
+ addUser(String, Privalges[]): Boolean
+ addUserRights(String, Privalges[], Privalges[]): Boolean
+ doesUserPrivExist(Privelage): void
+ isAdminApproved(Privelages[]): Boolean
+ removeUserRights(String, Privalges[], Privalges[]): Boolean
+ revokeUser(String, Privalges[]): Boolean
«use»
Login
+ login(String, String): Boolean
+ loginFailure(): Boolean
AdminPanel
- currentUser: String
- currentUserID: String
- userPrivelages: Privelages[]
+ getAdminID(): String
+ getAdminName(): String
+ getAdminPrivs(): Privelages[]
+ openAdminPanel(Privelages[], String): Boolean
«use»
«interface»
Privelage
+ getPrivelageFlag(): Boolean
+ getPrivelageType(): String
+ setPrivelageFlag(Boolean): void
+ setPrivelageType(String): void
PrivelageImpl
+ getPrivelageFlag(): Boolean
+ getPrivelageType(): String
+ setPrivelageFlag(): void
+ setPrivelageType(): void
ManageMeasures
- nodes: List<Node>
+ approve(): void

Figure 25: Class diagram for Admin UI

Design Patterns Used I used the Façade strategy as the idea was to simplify the Admin interface and interaction to the web component. As most operations will occur on the back end systems the idea is to keep the user blind from what is happening at the back end. Making their lives much less complicated. With a simple press of a button most of all the work is done.

4.5 Activity diagram

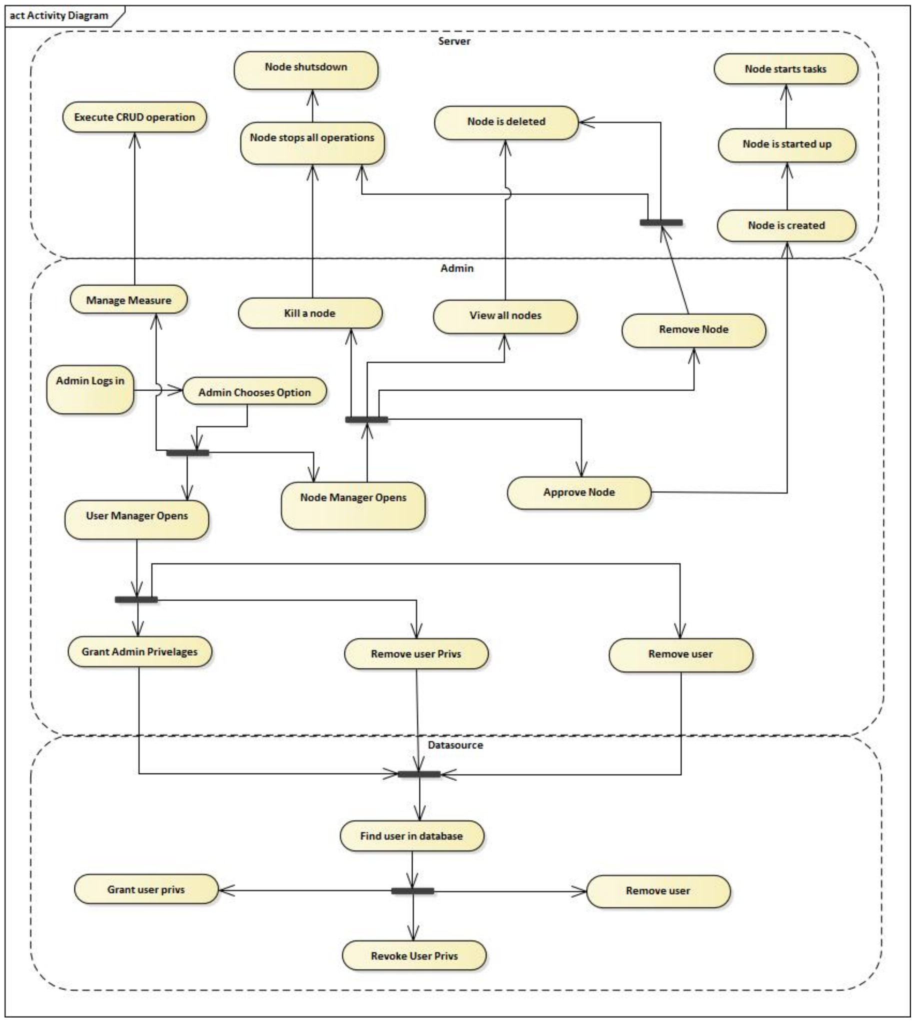

Figure 26: Activity diagram for Admin UI

4.6 Sequence diagrams

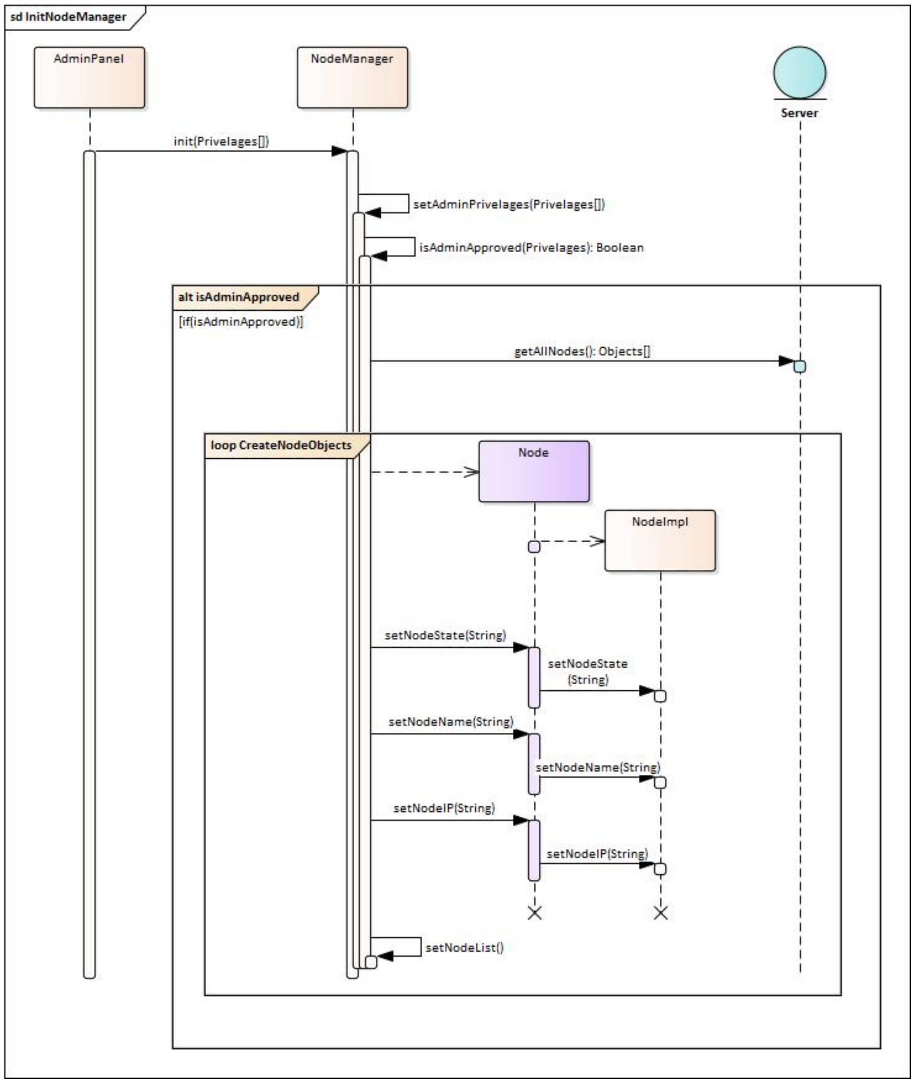

Figure 27: Initialize Node Sequence Diagram

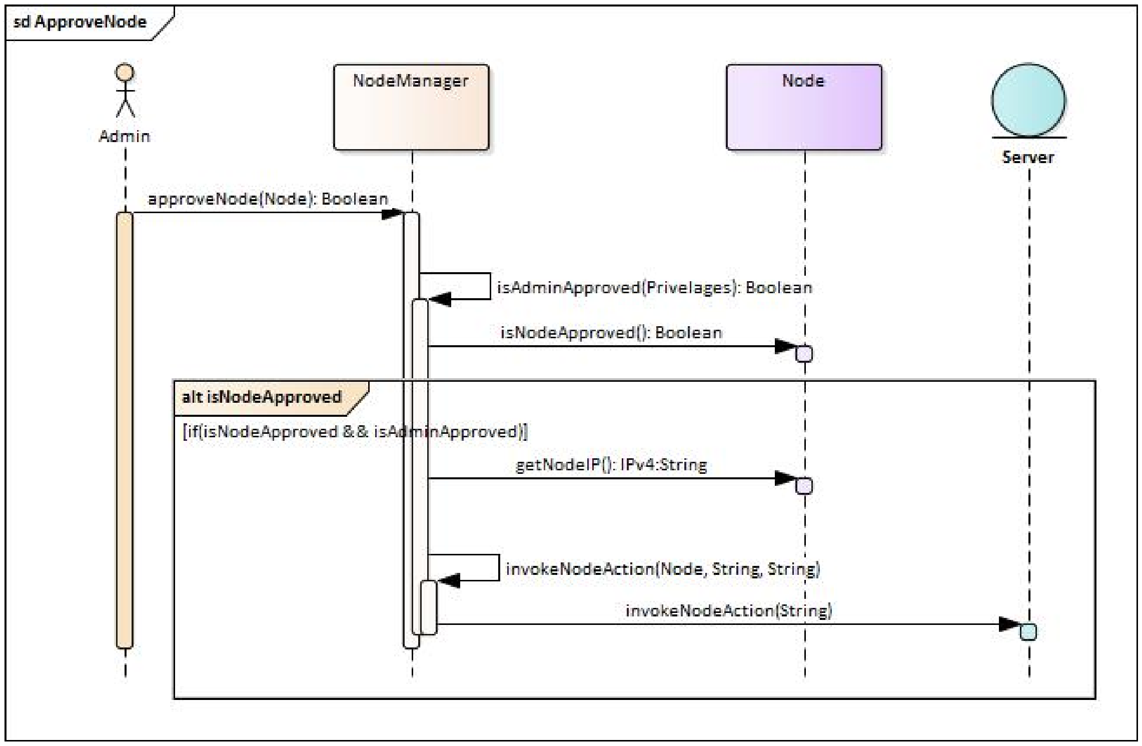
sd ApproveNode
Admin
NodeManager
Node
Server
approveNode(Node): Boolean
isAdminApproved(Privelages): Boolean
isNodeApproved(): Boolean
alt isNodeApproved
[if(isNodeApproved && isAdminApproved)]
getNodeIP(): IPv4:String
invokeNodeAction(Node, String, String)
invokeNodeAction(String)

Figure 28: Approve Node Sequence Diagram

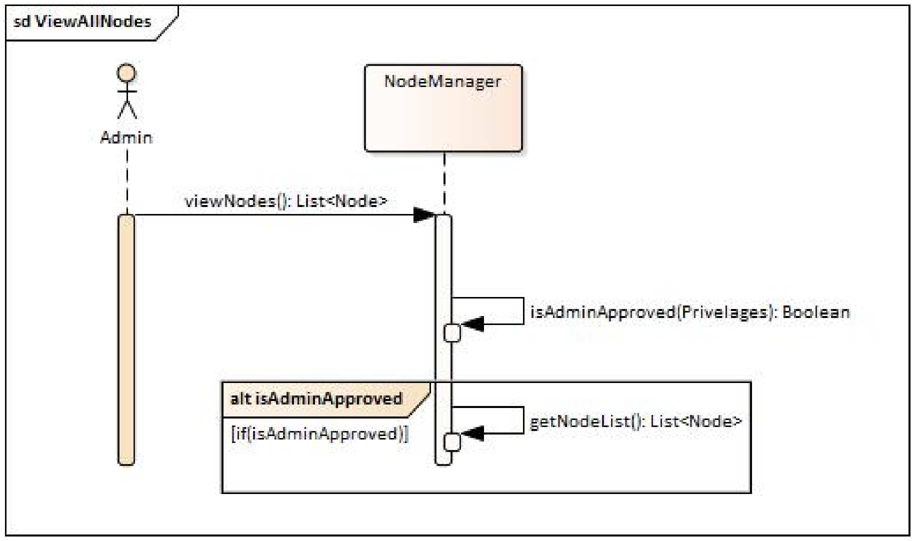

Figure 29: View All Nodes Sequence Diagram

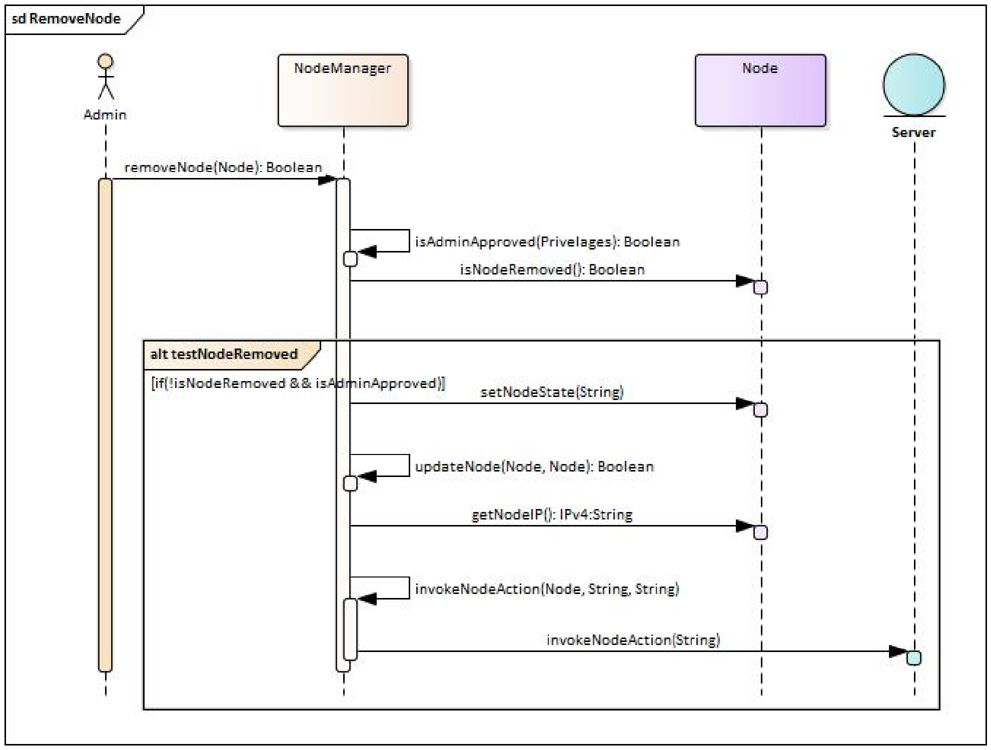
sd RemoveNode
Admin
NodeManager
Node
Server
removeNode(Node): Boolean
isAdminApproved(Privelages): Boolean
isNodeRemoved(): Boolean
alt testNodeRemoved
[if(!isNodeRemoved && isAdminApproved)]
setNodeState(String)
updateNode(Node, Node): Boolean
getNodeIP(): IPv4:String
invokeNodeAction(Node, String, String)
invokeNodeAction(String)

Figure 30: Remove Node Sequence Diagram

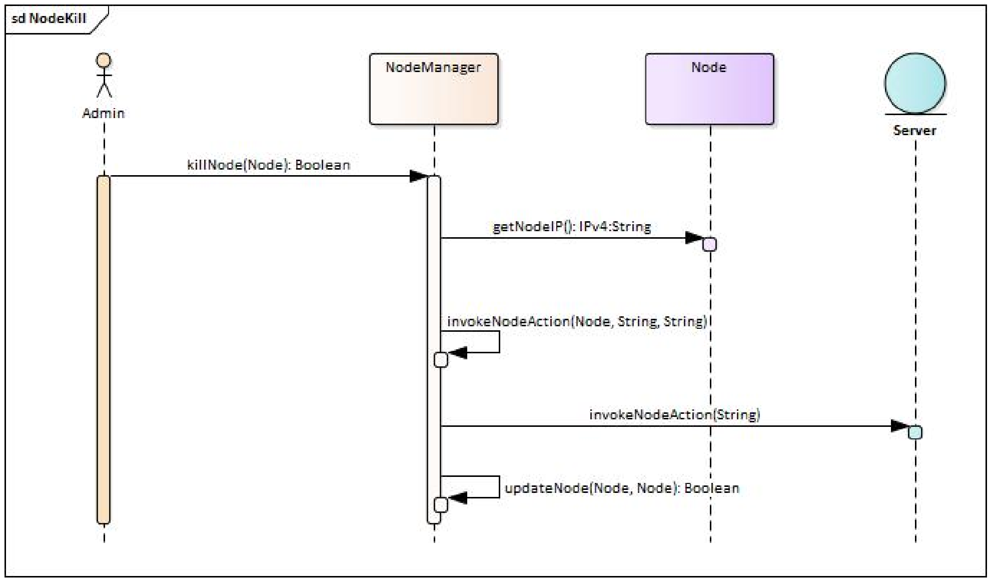

Figure 31: Kill Node Sequence Diagram

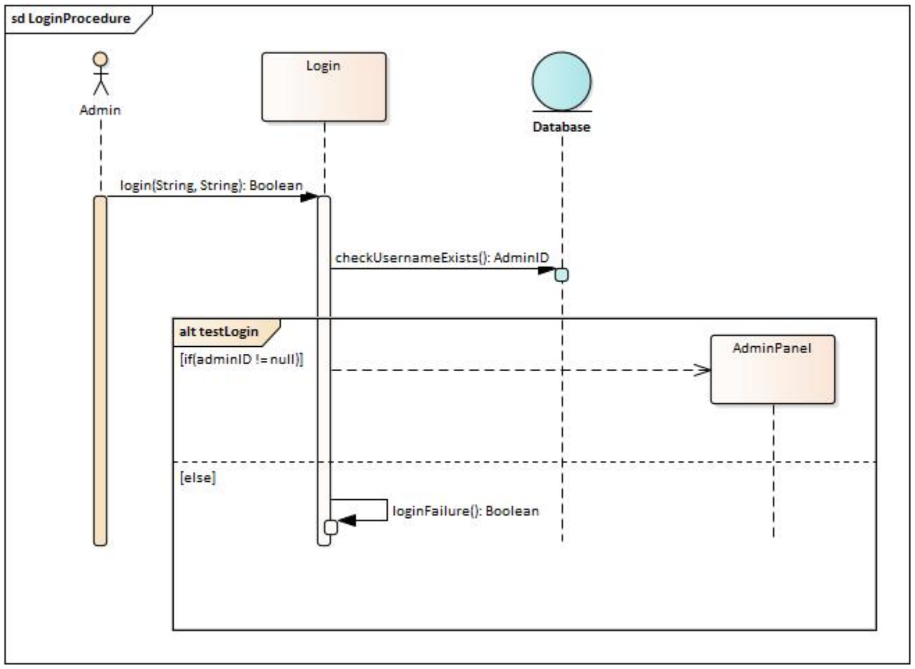

Figure 32: Login Sequence Diagram

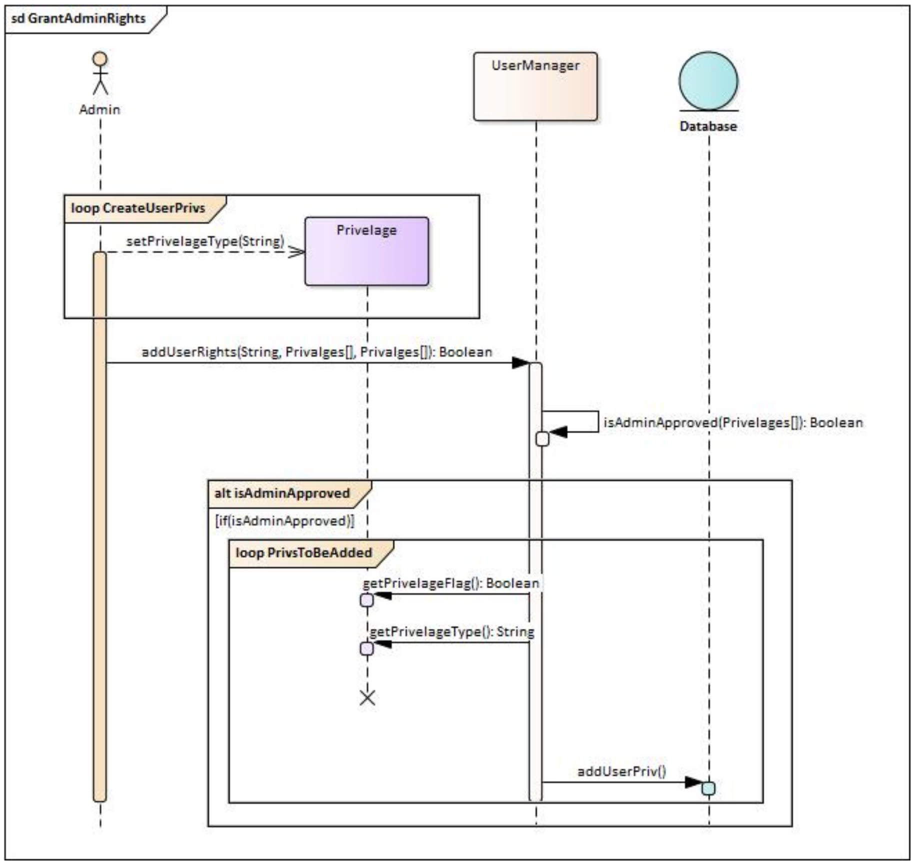

Figure 33: Grant Admin Rights Sequence Diagram

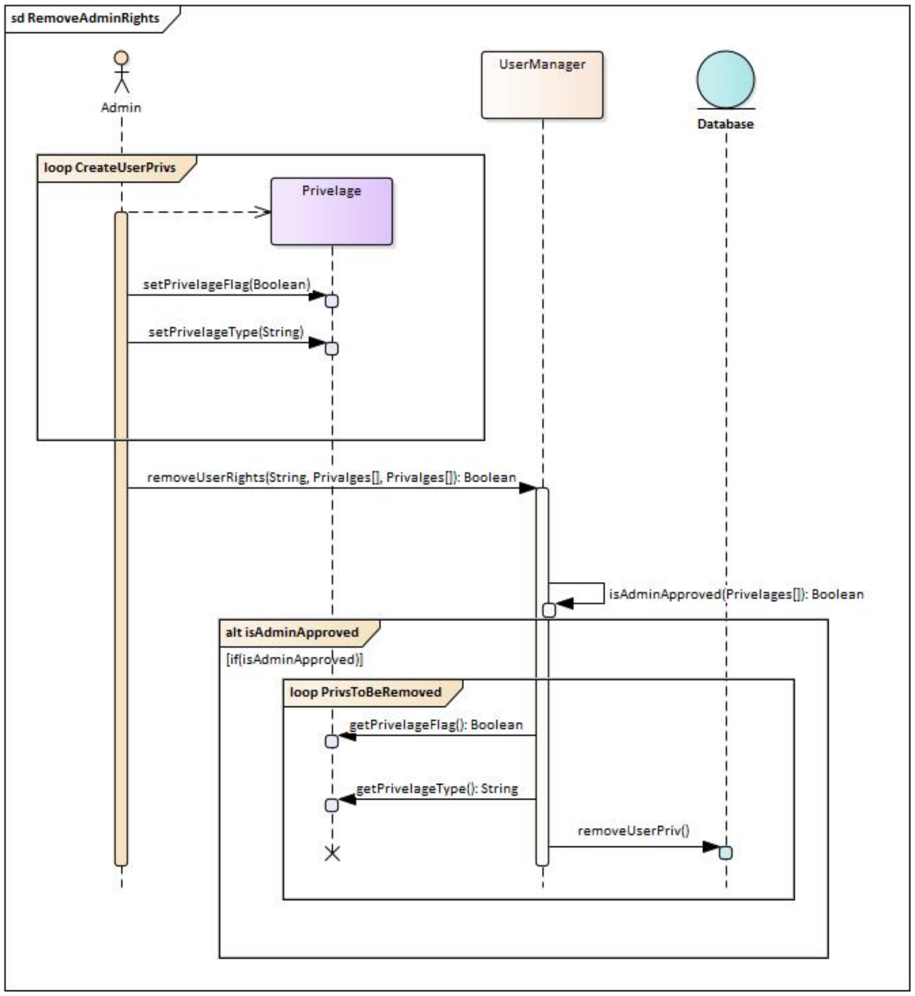
sd RemoveAdminRights
Admin
UserManager
Database
loop CreateUserPrivs
Privelage
setPrivelageFlag(Boolean)
setPrivelageType(String)
removeUserRights(String, Privalges[], Privalges[]): Boolean
isAdminApproved(Privelages[]): Boolean
alt isAdminApproved
[if(isAdminApproved)]
loop PrivsToBeRemoved
getPrivelageFlag(): Boolean
getPrivelageType(): String
removeUserPriv()

Figure 34: Remove Admin Rights Sequence Diagram

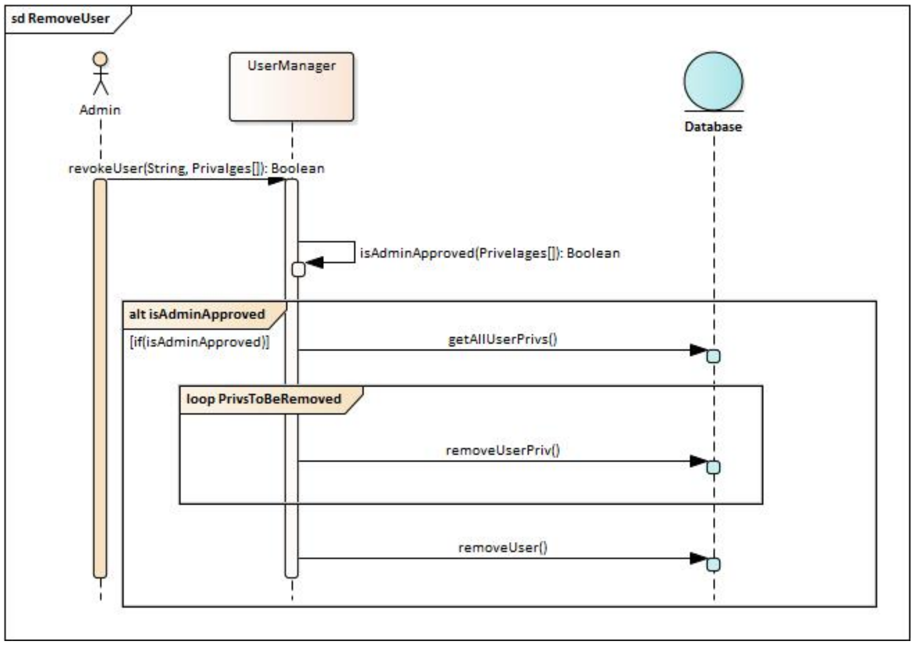

Figure 35: Remove User Sequence Diagram

4.7 State diagram

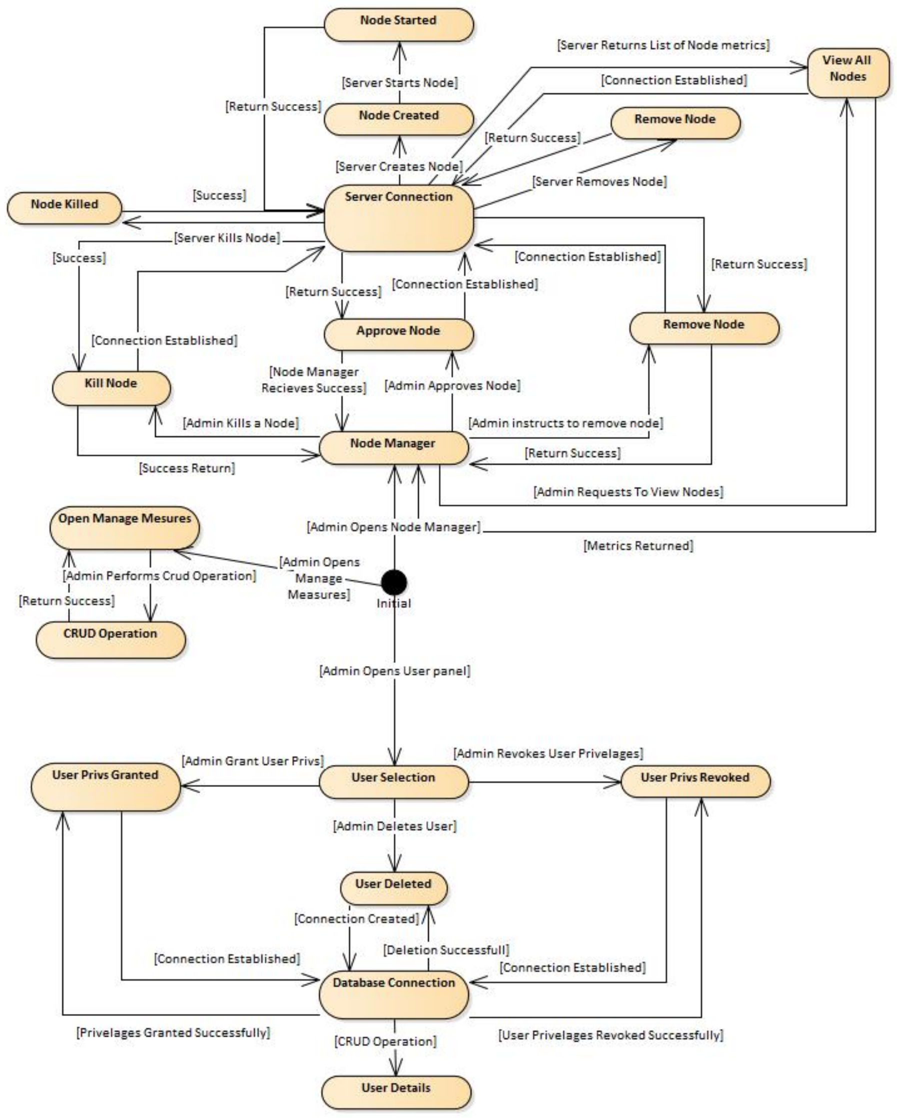

Figure 36: State Diagram for Admin UI

4.8 Use Case diagram

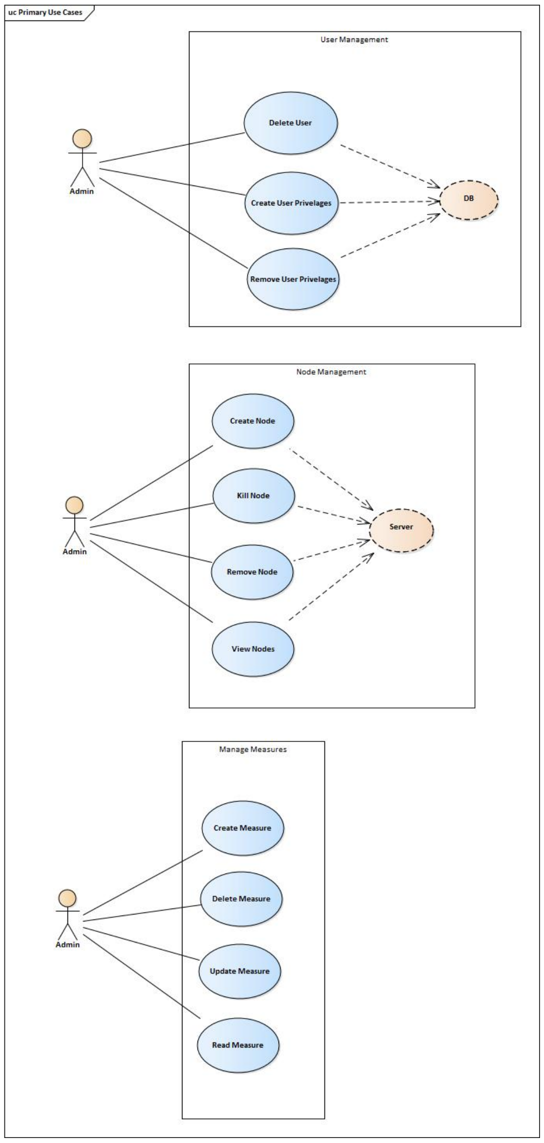

Figure 37: Use case diagram for Admin UI

5 Deployment Diagram

5.1 Architectual Design

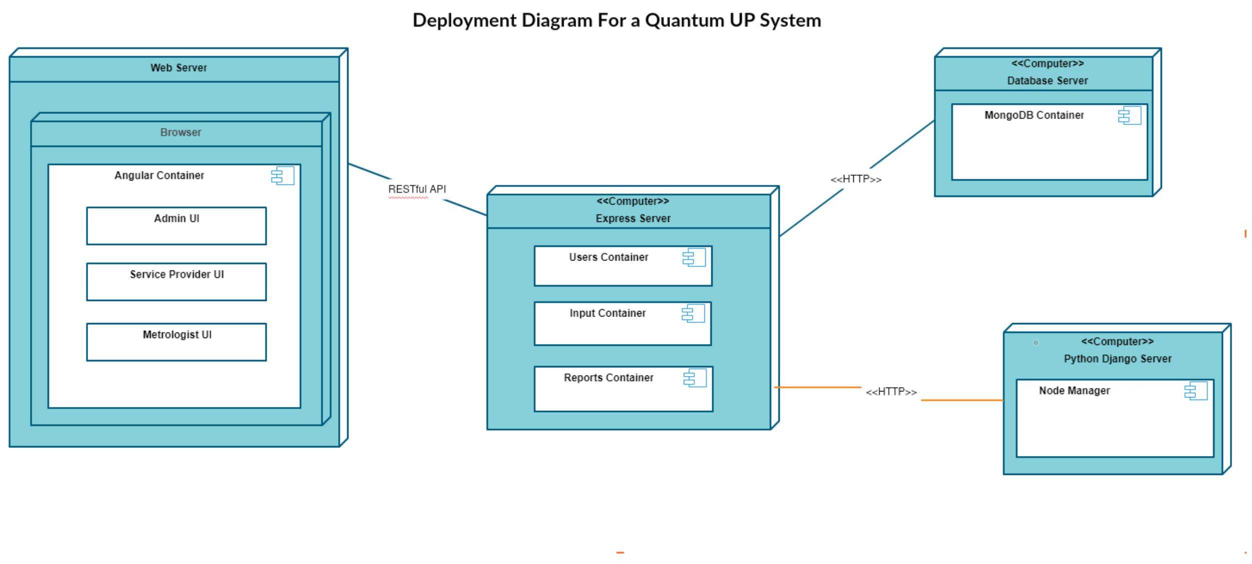

Figure 32: Deployment Diagram for Quantum UP System

For the architectural design we went with micro services because micro services promotes **low coupling** by making each component run only their services that is required by it. The main way that this will be done is to use docker where each **docker** container will communicate with each other via HTTP and ports. By using docker it makes each container very secure and also promotes **portability** where on any server the relevant docker containers can be run and linked. This makes it very easy to move your system without having to spend time installing dependencies and make sure everything is installed before running the system. The GUI services will be provided by an **Angular** app because angular make binding and usage very simple which aids in user experience. The Angular app will make API calls to a dockerized express server which will hold the user,input and reporting services. When a request is made to the relevant service.

The service will either make a request to store data on the dockerized **MongoDB** database server or it will make a request to the node manager container that will run the nodes services for benchmarking. The reason to use MongoDB as the server is because it is lightweight and allows for rapid scaling on large amounts of data, which might be the case in this situation. Another good thing about micro services is that each service does not have to know what the other service is doing and does not have to rely on other services thus making it very **modular**. If you would not want to depend on a local Db or just one DB, the ease of micro services allows you to switch out database with minimal changes. For example switching out MongoDb for a cloud database such as Cosmos Db with a Mongo API so there is no need to change database write commands.

Finally the node manager that will run the benchmarking services will run in a python Django server. The reason for using python is that it works well with getting low level data like CPU usage which is useful in this case and since micro services are used it allows for language neutral communication because data is communicated via HTTP and API calls.Theses reasons above are why a micro services architecture would be good for this system.

5.2 Technologies used(refer above for description of how theses technologies are used)

- Docker - To containerize services
- Express JS - Server
- Nodejs - Server language
- Angular 5 - Live Binding, Very responsive UI
- MongoDB - Database
- Django - Python Server