# Design

**PSE of**

PSE Group

Fraunhofer Institute for Systems and Innovation Research ISI
Advisor: M.Sc. Ankush Meshram

Version 1.0

# Contents

# 1 Design

## 1.1 Front-End

This subsection describes the front-end of the ADIN INSPECTOR - the UI elements the GUI consists of, and how states are handled. A series of final UI design mockups are presented under UI Design Mockups subsection, whereas an overview of the GUI classes can be seen in Figure 8.

The GUI elements are implemented in React and Material UI, whereas the internal logic and application state management are written with MobX.

### 1.1.1 UI Design Mockups

An early stage interactive demo is available at `https://adin-frontend.netlify.com`.

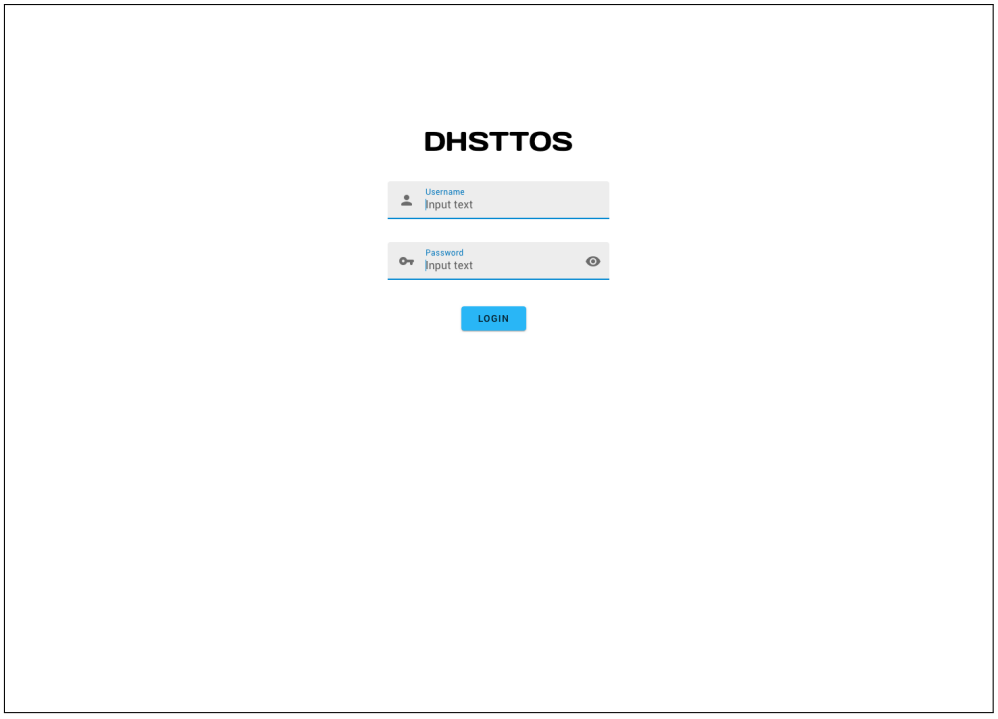



Figure 1: Login screen

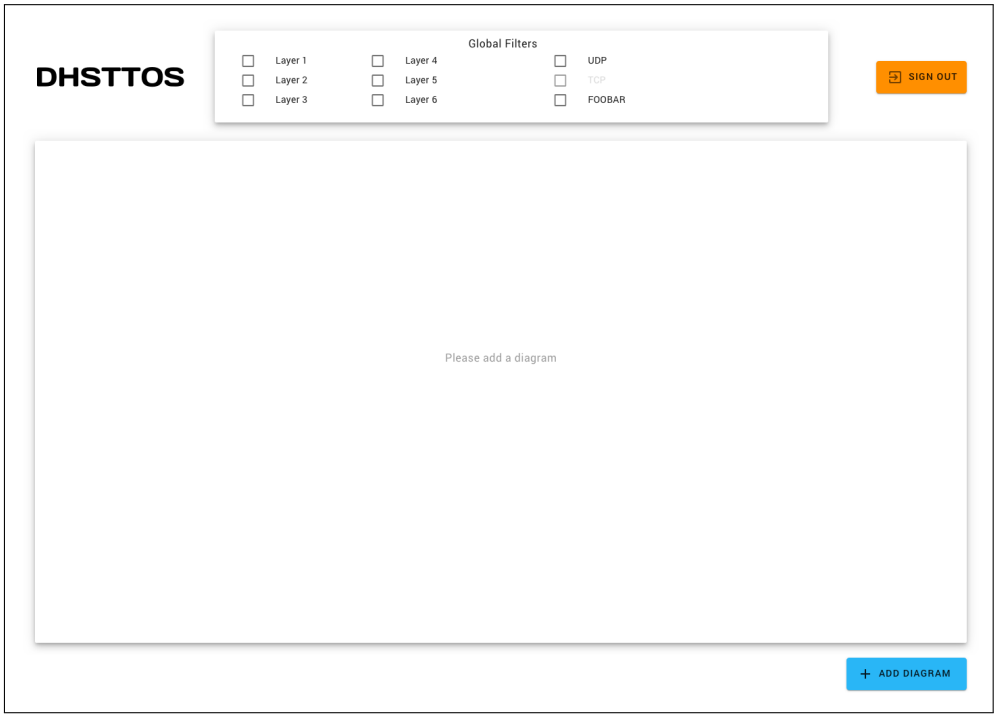



Figure 2: Initial empty screen

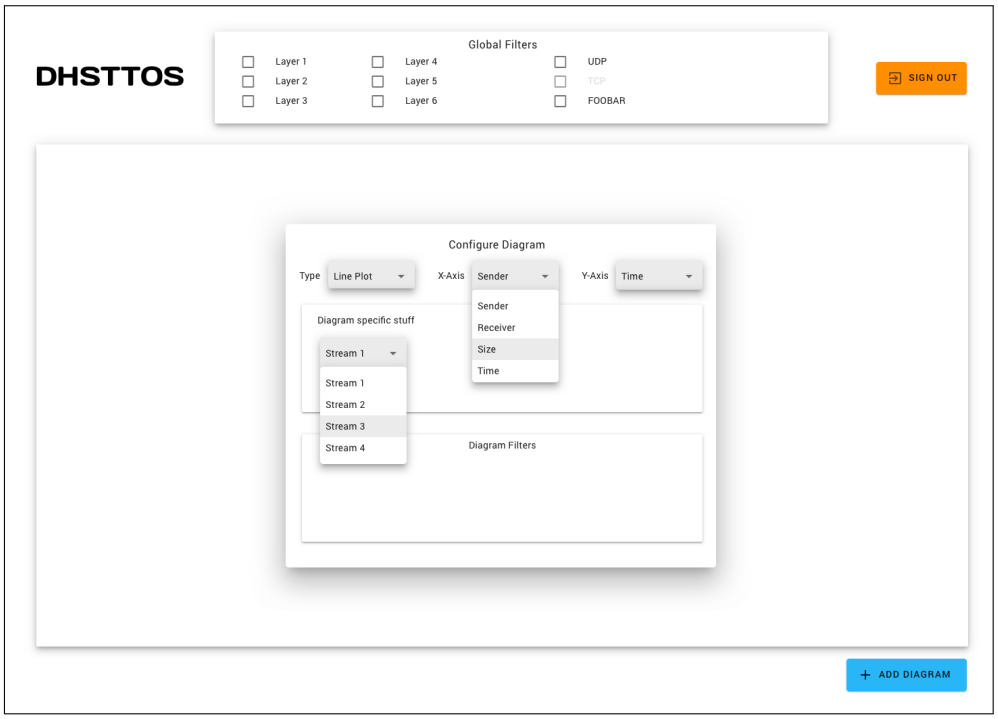



Figure 3: Adding first diagram

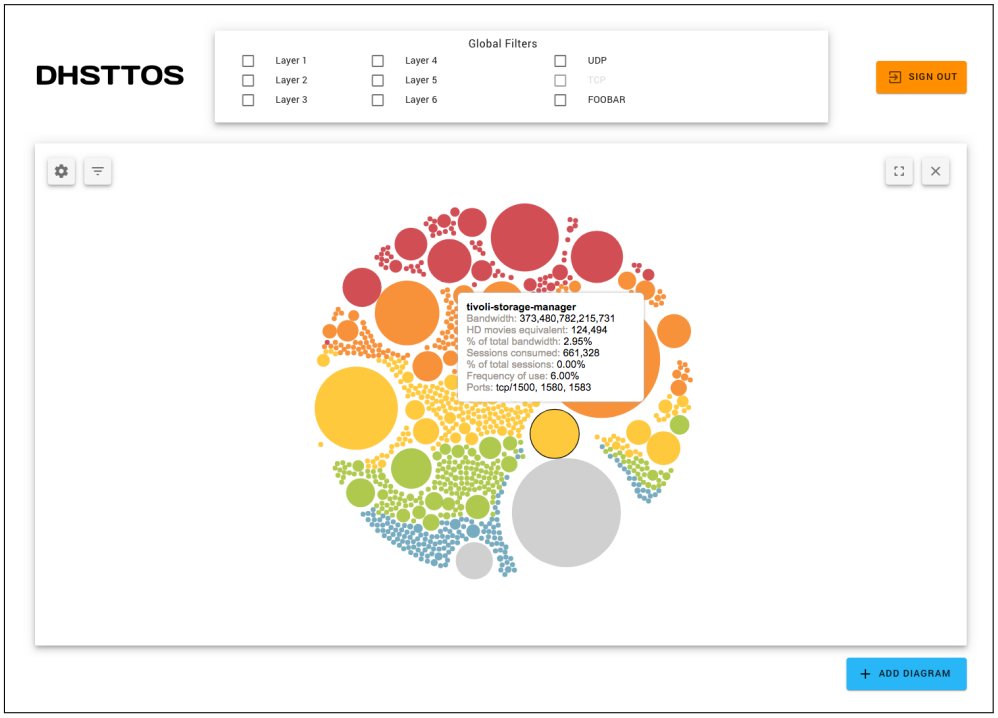



Figure 4: Displaying a single diagram

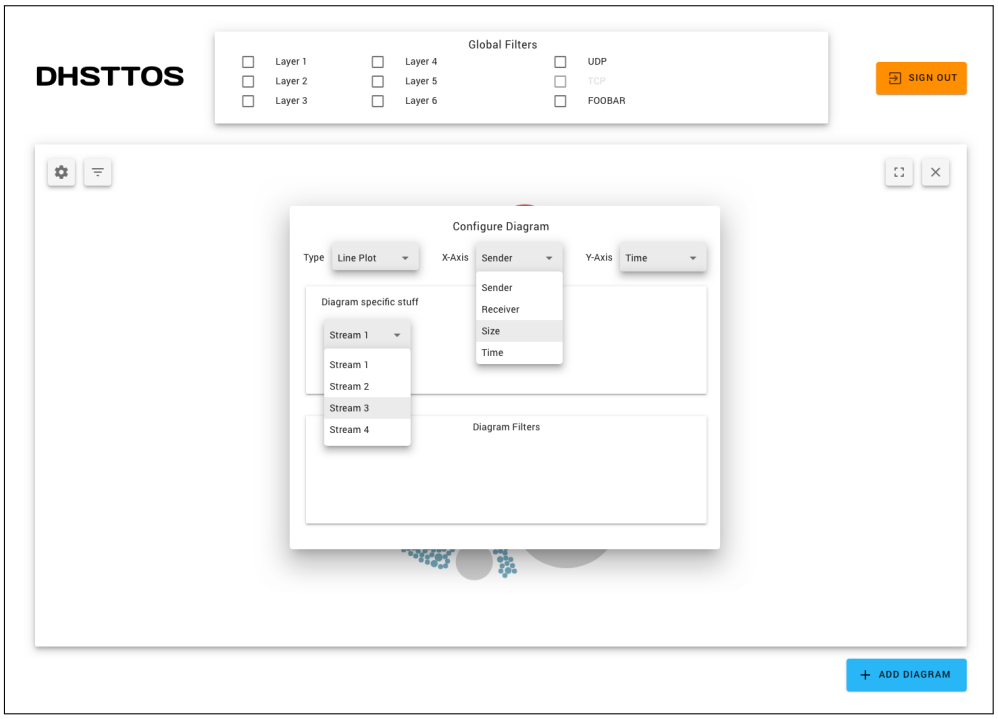



Figure 5: Adding new or configuring existing diagram

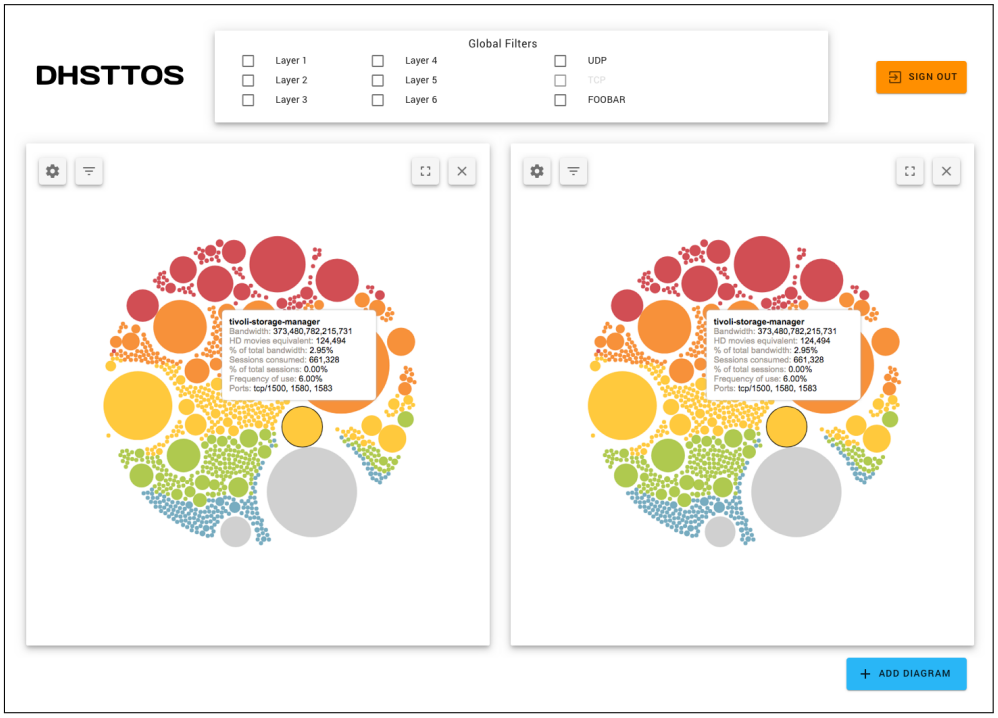



Figure 6: Displaying two diagrams

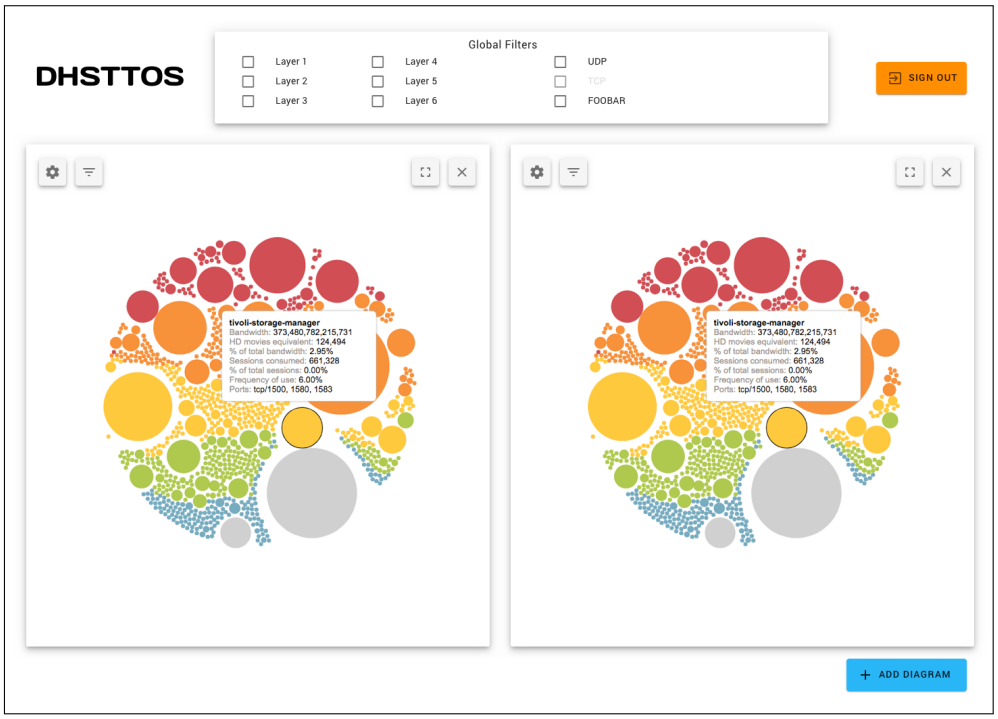



Figure 7: Adding additional or configuring existing diagram

### 1.1.2 Class Diagrams

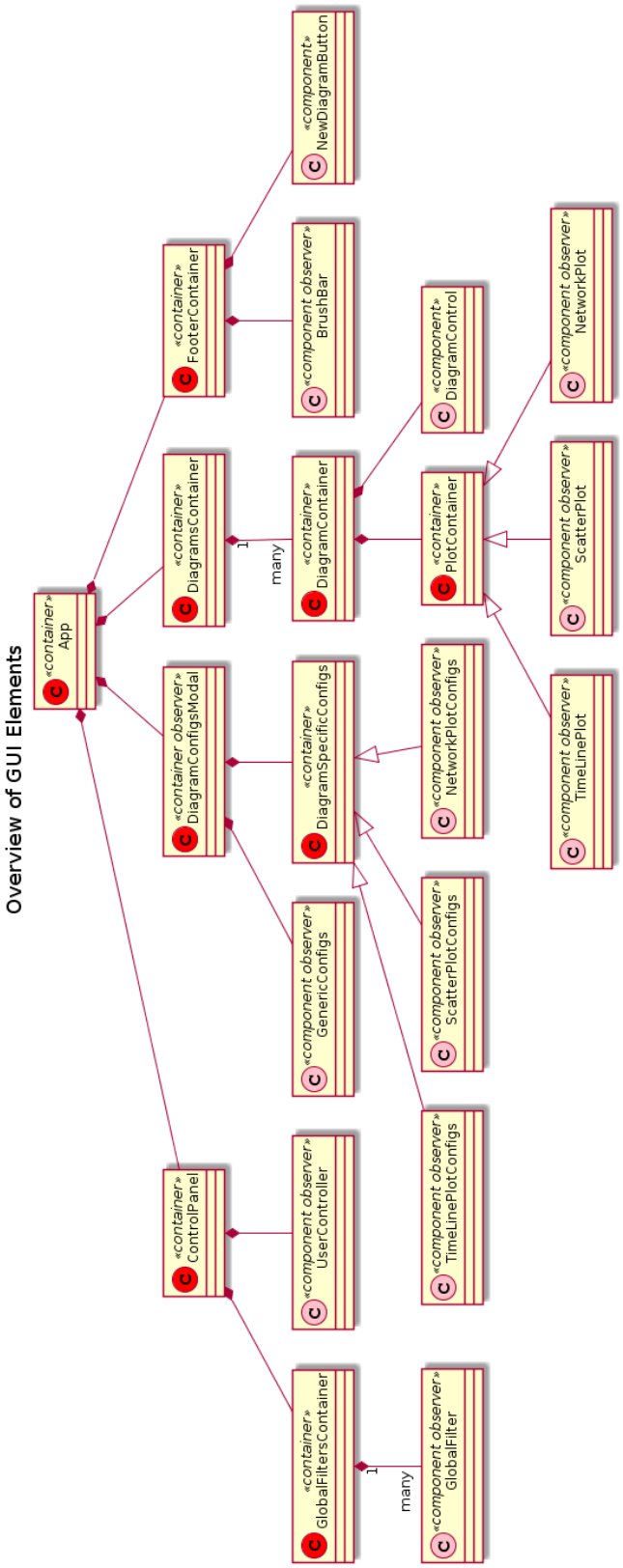



Figure 8: This diagram shows an overview of GUI elements and their relationships inside the main application, when the user has successfully logged in.

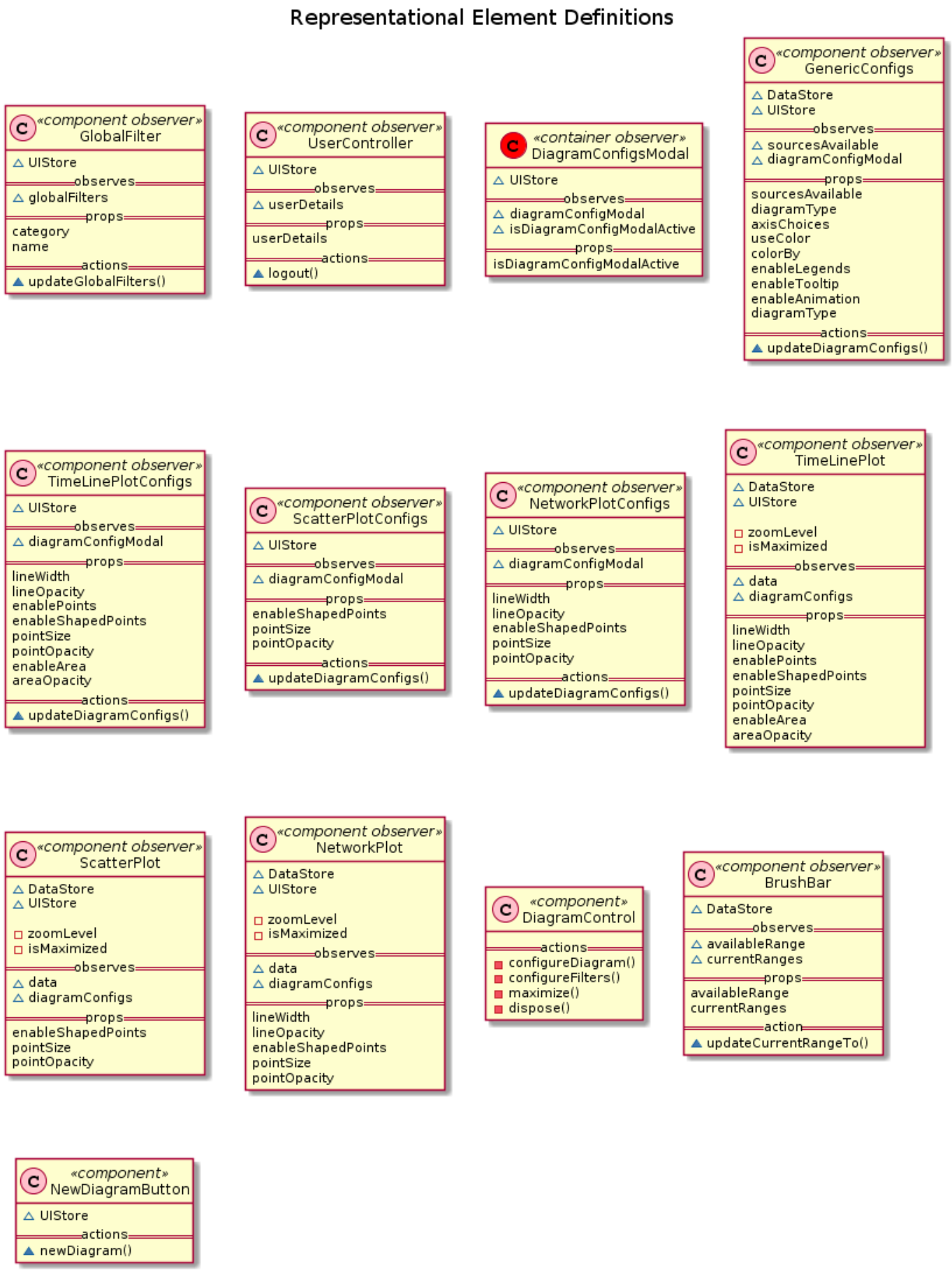



Figure 9: This diagram shows the definitions of all representational elements.

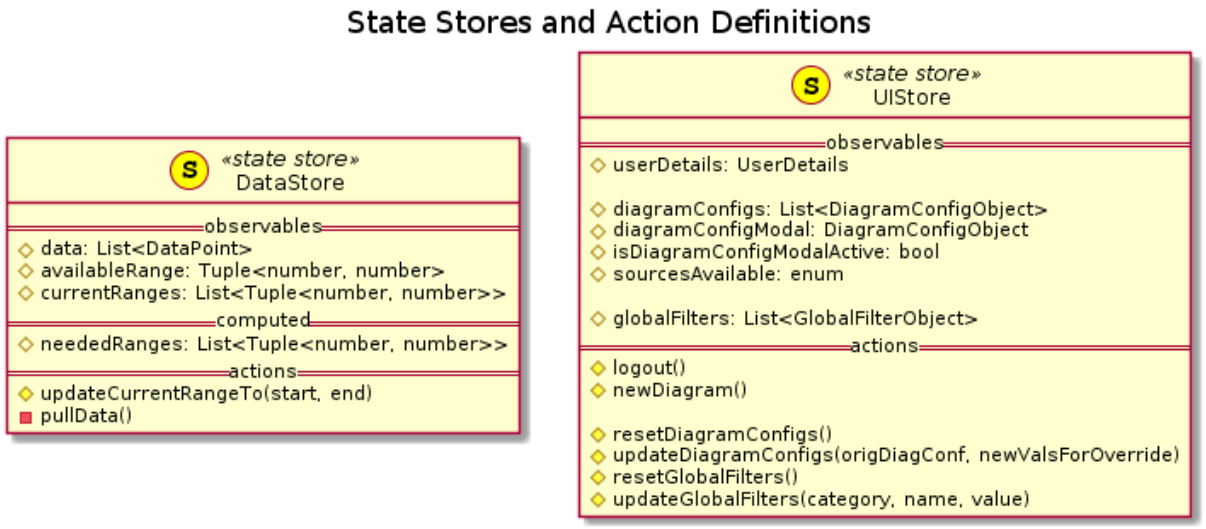



Figure 10: This diagram shows the design of the MobX state store objects and predefined actions to mutate the states.

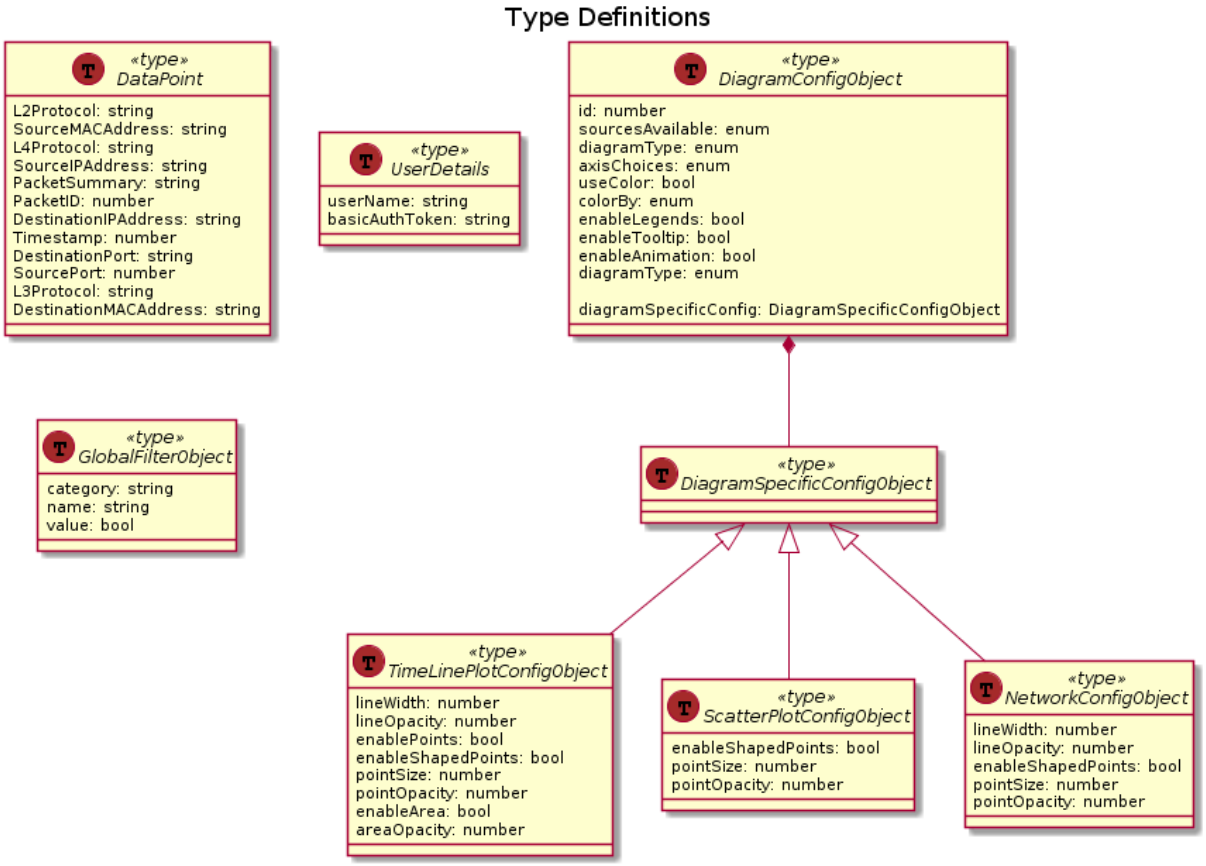
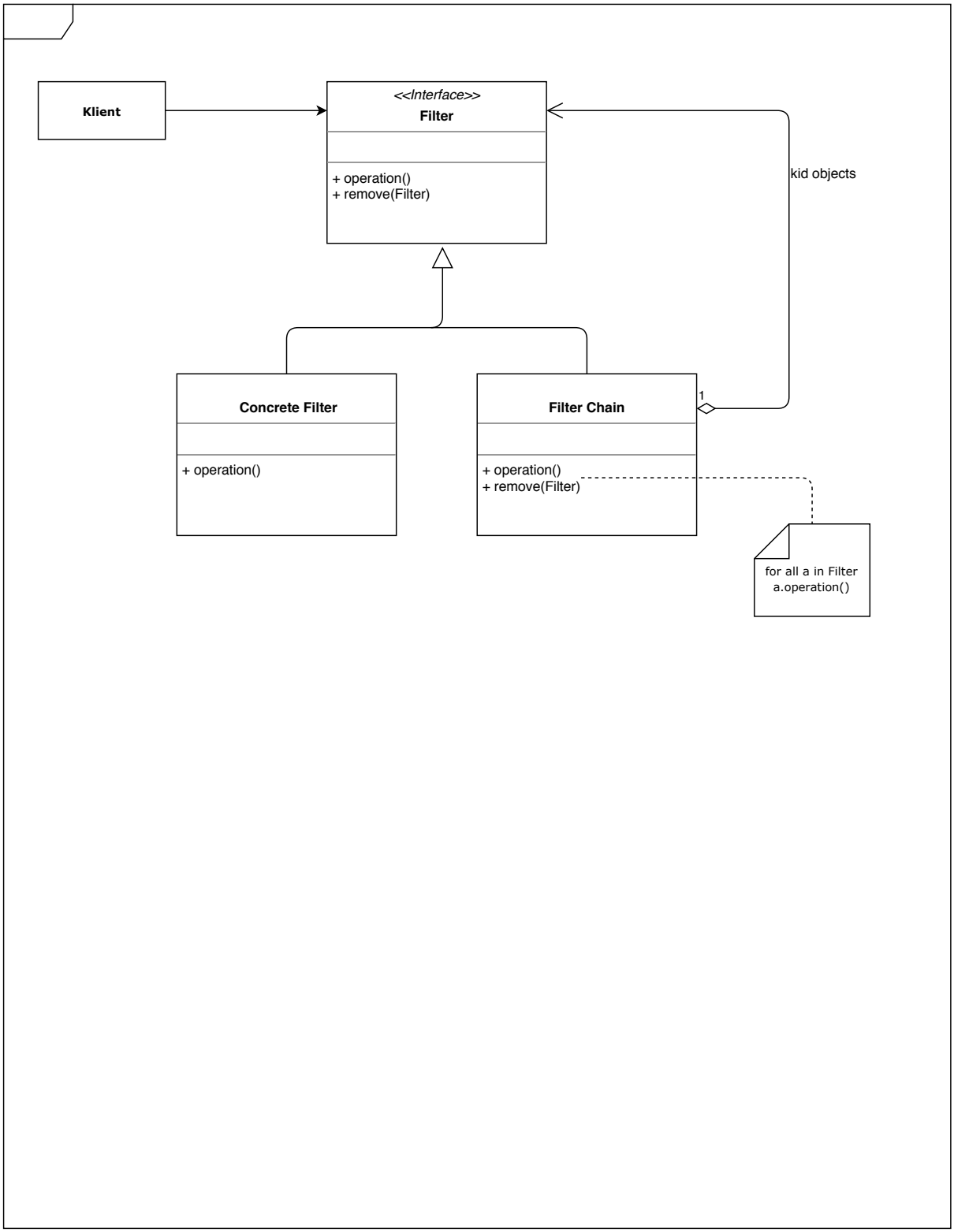
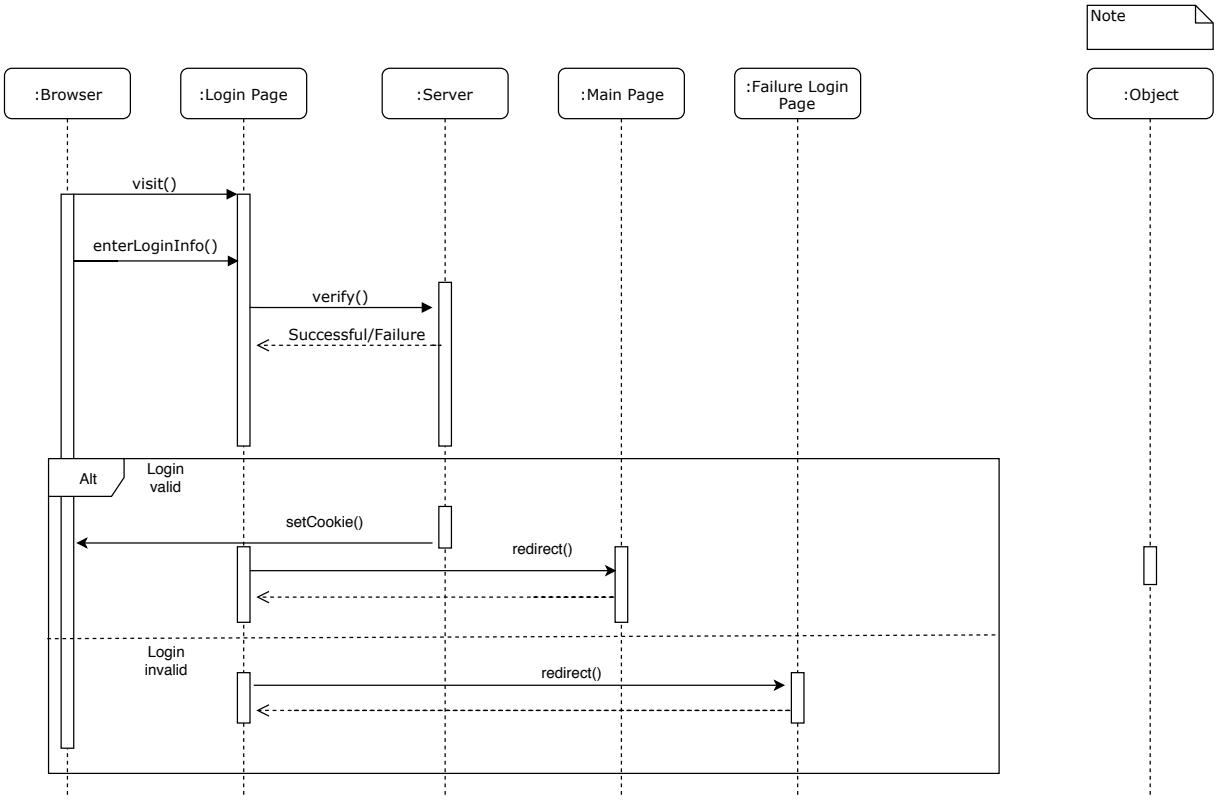



Figure 11: This diagram shows the definitions of custom types that are used in the MobX state stores.

### 1.1.3 Sequence Diagram






This diagram shows the control flow for handling a movement of the slider by the user.

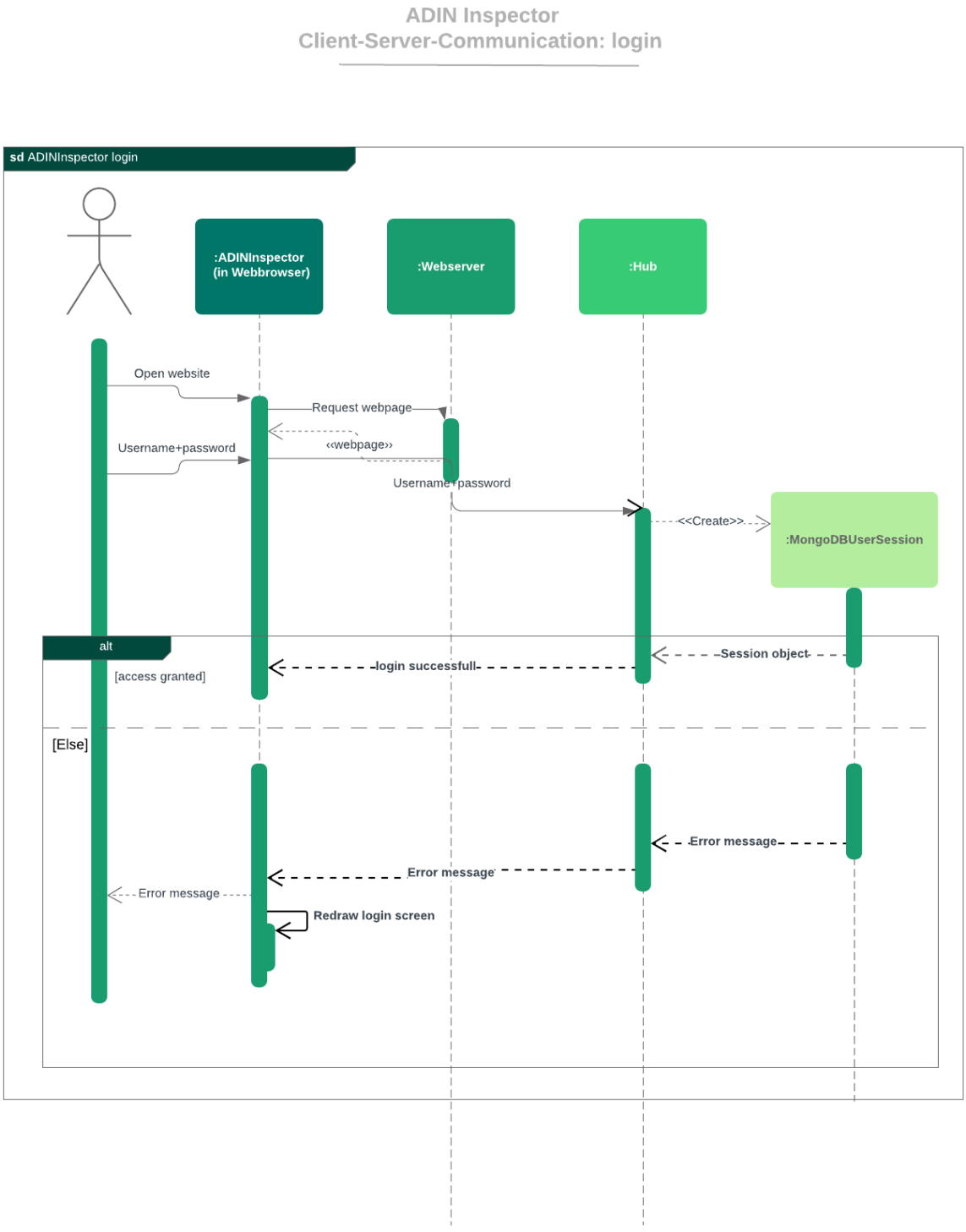
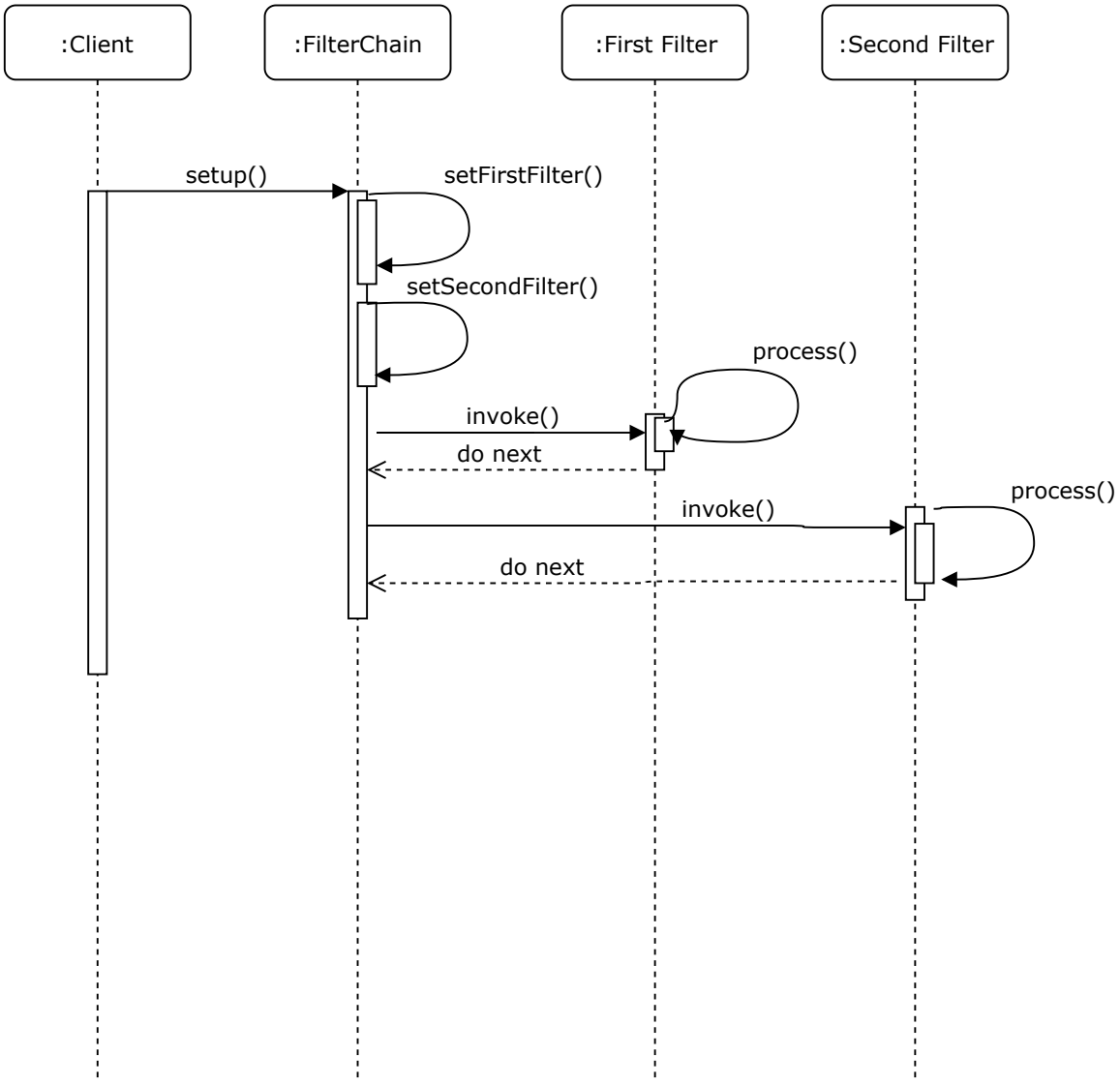
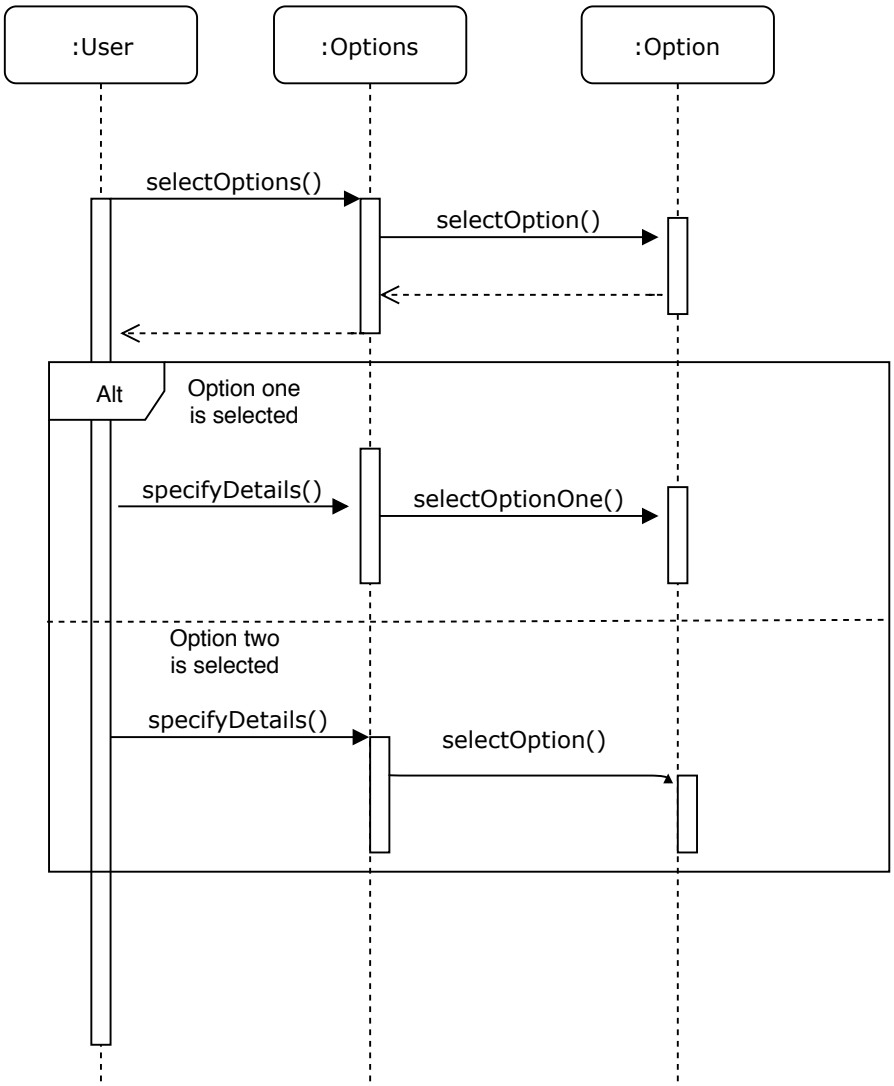



Figure 12: This diagram shows an alternative view of the login sequence


:Client
:FilterChain
:First Filter
:Second Filter
setup()
setFirstFilter()
setSecondFilter()
process()
invoke()
do next
process()
invoke()
do next


:User
:Options
:Option
selectOptions()
selectOption()
Alt
Option one
is selected
specifyDetails()
selectOptionOne()
Option two
is selected
specifyDetails()
selectOption()

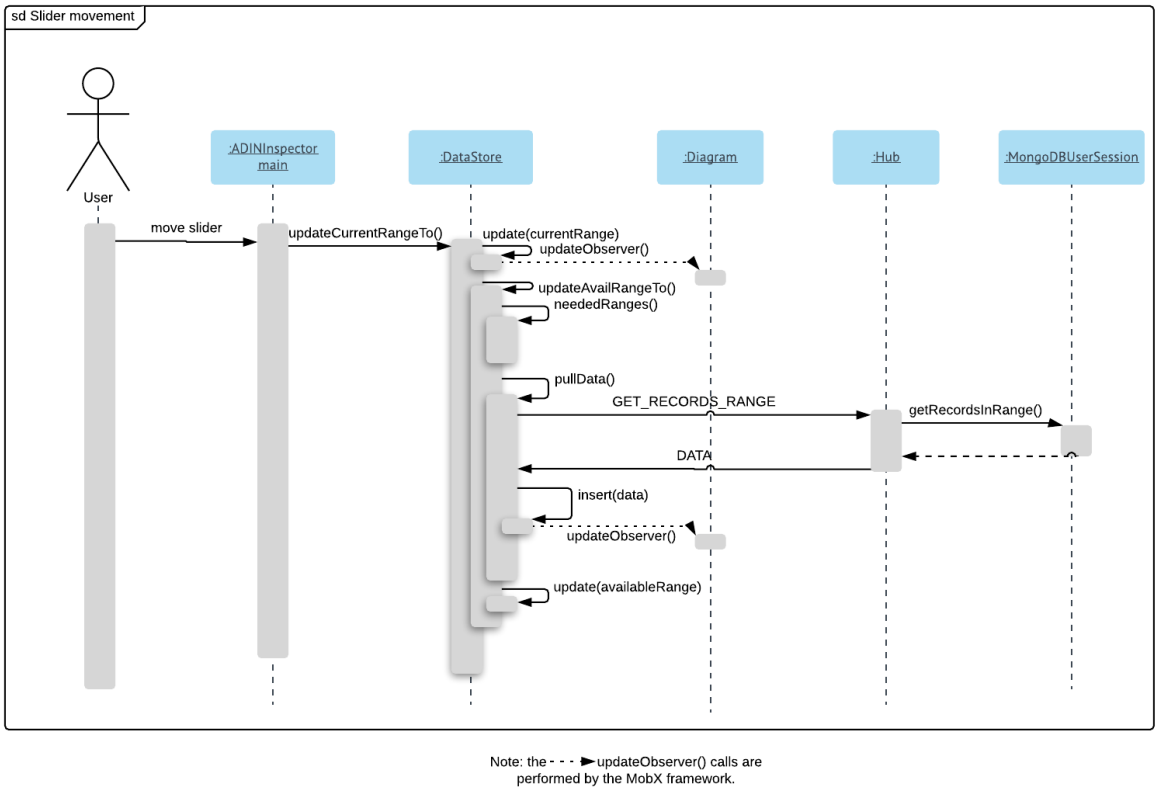



Figure 13: Sequence diagram for slider movement

## 1.2 Client-server protocol

Messages between client and server are exchanged as strings in JSON format.
In the following list words in angle brackets ("<>") are placeholders.

### 1.2.1 Requests from client to server:

- getAvailableCollections
  syntax: {"cmd": "GET_AV_COLL"}
  expected response: list of collections

- getCollectionSize(collection)
  syntax: {"cmd": "GET_COLL_SIZE", "par": "<collection>"}
  where <collection> is the name of a collection
  expected response: collection size

- getCollection(collection)
  syntax: {"cmd": "GET_COLL", "par": "<collection>"}
  expected response: data set

- getRecordsInRange(collection, key, start, end)
  syntax: {"cmd": "GET_RECORDS_RANGE", "par": "<collection>", "key": "<keyvalue>", "start": "<startvalue>", "end:", "<endvalue>"}
  where <key> is the name of a key in the given collection and <startvalue> and <endvalue> are valid values for this key
  expected response: data set

- getRecordsInRangeSize(collection, key, start, end)
  syntax: {"cmd": "GET_RECORDS_RANGE_SIZE", "par": "<collection>", "key": "<keyvalue>", "start": "<startvalue>", "end:", "<endvalue>"}
  expected response: collection size

### 1.2.2 Messages from server to client:

- list of collections
  syntax: {"cmd": "LIST_COL", "par": ["<collection>"]}
  where <collection> is the name of a collection

- collection size
  syntax: {"cmd": "COLL_SIZE", "par": "<size>"}
  where <size> is the number of records in this collection

- data set
  syntax: {"cmd": "DATA", "par": [<record>]}
  where each record is a JSON object

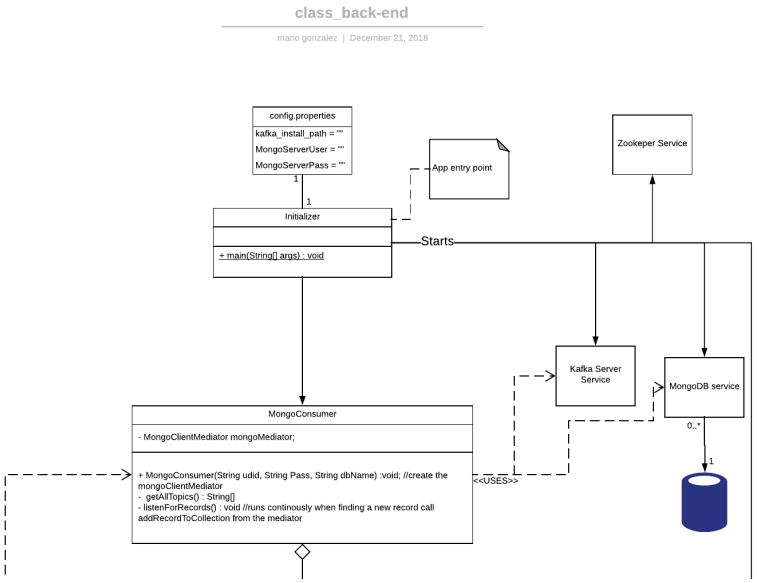



Figure 14: The classes involed in the initialization setup

## 1.3 Back-End

This subsection deals with the back-end of the ADIN INSPECTOR. How the system deals with client http calls, and how kafka interacts with the system. An overview of the system can be seen in Figure 17. Smaller subsections have been expanded in Figure 14, Figure 15, Figure 16.

The connection to the client is handled in the class Hub which contains handlers for the network interface. This class uses a separate class (ClientProtocolHandler) to parse and handle requests from the handler. This setup is according to the strategy design pattern and allows easily modifying or even replacing the client server-protocol. The Hub class and the ClientProtocolHandler access the database via an object that implements the IUserSession interface and encapsulates the database session. Currently there is only an implementation for MongoDB access (MongoDBUserSession), but the abstraction via the IUserSession interface allows to add classes that offer access to Kafka or other databases. Classes that implement IUserSession are instantiated with a factory Method (UserSession()) which guarantees that the returned object represents a successfully logged in database session.

### 1.3.1 Class Diagram

The overview in Figure 17 shows a number of classes and it's interaction with eachother. What follows is a more in-depth view of what each component of this diagram does, what data it's stored and how it fits into the overarching architecture.

- Config properties file
  The config file is stored alongside the built application .jar file and contains the path to the Kafka installation folder, the user name and password of a mongoDB account with the highest level of access and the name of the database.

- Initializer
  Methods:

- main
  parameters: String of arguments from the console
  returns: void
  App entry point.
  We load the config.properties life and use the path provided to start the zookeper, kafka and mongodb services

- MongoConsumer
  The Mongo Consumer, as the name implies, consumes all messages from all topics in the Kafka messaging system. Once a message is found it is passed along to the Mongo Client for further processing.
  Variables
  - clientMediator
    Type : MongoClientMediator
    An instance of the Mongo Client Mediator, created with the credentials from the config file.

  Methods
  - MongoConsumer constructor
    parameters: user name and password of a mongoDB account with the highest level of access.
    Initializes the MongoClient variable and calls listenForRecords();
  - getAllTopics
    parameters: none
    returns: an array of strings containing all the available kafka Topics.
    Asks the kafka server service which topics exists.
  - listenForRecords
    parameters: none
    returns: void
    This Method first calls getAllTopics and uses the array of topics to poll the kafka server for new messages.
    If new messages are found then the messages are passed to the Mongo Mediator for adding them to the Database.
    If no new messages are found for a topic notify the Mongo Mediator that the collection tied to the topic is ready for pre-processing.

- MongoClientMediator This object serves as a nexus between the users who want to get data out of the database and the consumer and dataProcessor who want to add data into the database. This class encapsulates the mongo client from the mongo API.
  Variables
  - client
    type: MongoClient
    An instance of the Mongo Client from the official java API.

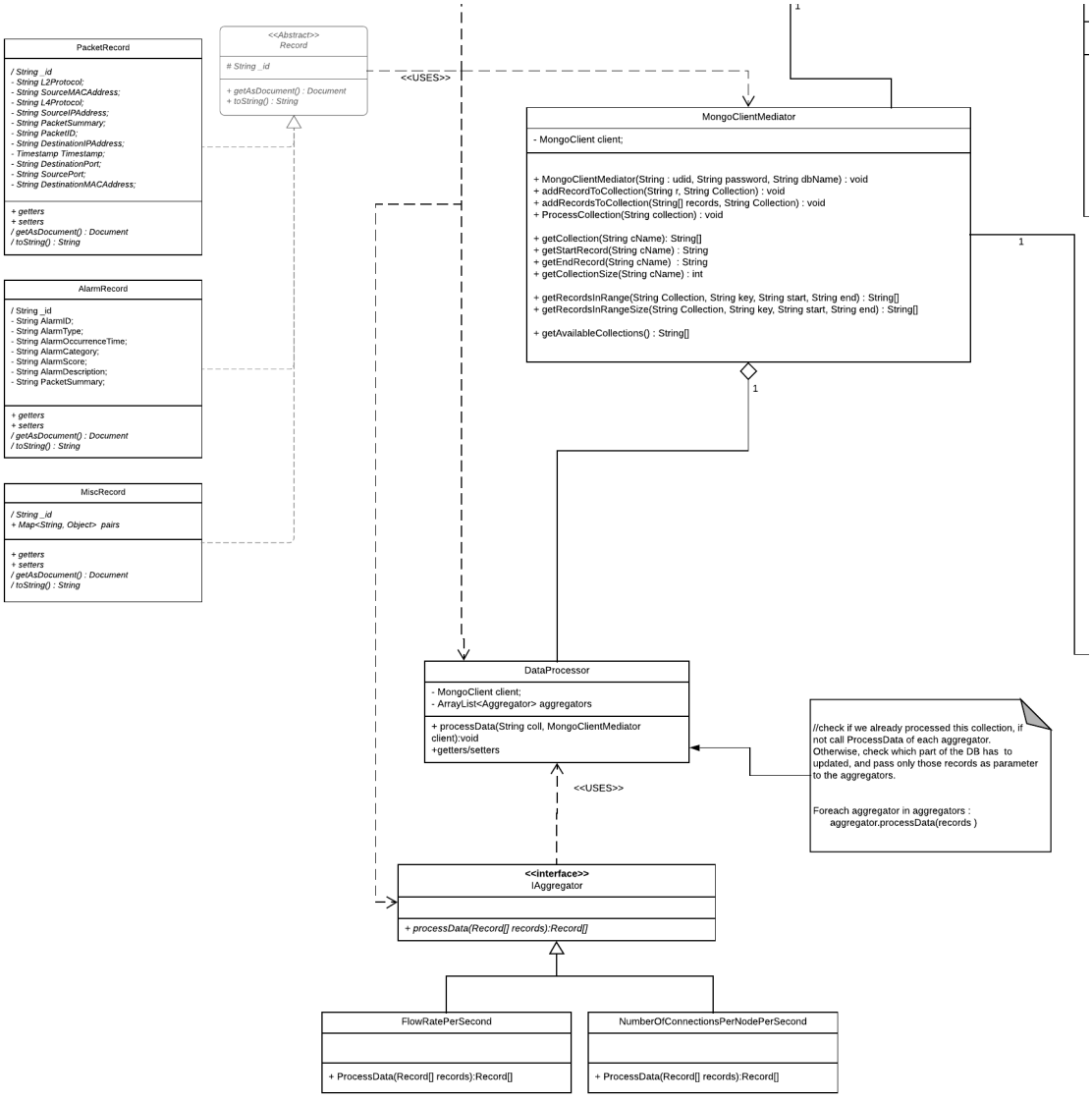



Figure 15: The classes involed in reading and writing data into the database

- dataProc
  A reference to the data processor class for this client.

Methods

- MongoClientMediator constructor
  parameters: Username and password
  Initializes the client variable, throws an error if the user is not found.
- addRecordToCollection
  parameters: String representation of a record in json format
  String name of the collection it should be added to.
  returns: void
  Converts the json string into a java object, then to a bson document and uses the mongoAPI to insert it into the database.
- addRecordsToCollection
  parameters: String Array of records to be added to a collection
  String name of the collection it should be added to.
  returns: void
  for each oneof the members of the array call addRecordToCollection
- ProcessCollection
  parameters: String, name of a collection
  returns: void
  signal the data processor to start the processing of a collection
- getCollection
  parameters: String, name of a collection
  returns: String array containing all entries of the collection
- getStartRecord
  parameters: String, name of a collection
  returns: the first entry of the collection as a String.
- getEndRecord
  parameters: String, name of a collection
  returns: the last entry of the collection as a String.
- getCollectionSize
  parameters: String, name of a collection
  returns: the number of entries in the collectoin as int
- getRecordsInRange
  parameters: String, name of the collection to query
  String, key of the parameter used for filtering
  String start and end ranges for the filtering
  returns: String array containing all entries of the collection within that range
  this Method is very general to allow for flexibility.For example by letting the key be, SourceIPaddresses, or a timeStamp.

  - getRecordsInRangeSize
    parameters: String, name of the collection to query
    String, key of the parameter used for filtering
    String start and end ranges for the filtering
    returns: number of elements matching the range as int

- Record
  Every message that comes from kafka and needs to be added to the database has it's own Record class that inherit from this one.
  Every single class that inherits needs to be able to, using reflection, convert itself into a Bson Document where every variable is a key Value pair of the name of the variable and it's associated value.
  Variables
  - id
    type: String

  Methods
  - getAsDocument()
    parameters: none
    returns: A Document, containing every variable of any class inheriting from this one.
    This function checks for every variable, gets it's name and value as a string and adds it to the document that it eventually returns.

- PacketRecord
  Inheriting from Record, this class contains the variables that match the json string obtained from kafka.
  Variables
  - id
    type: String
    this id is used for determining the ordering when saving to mongoDB, it's the offset of the message in the kafka messaging queue. inherited from Record
  - client
    type: String
  - L2Protocol
    type: String
  - SourceMACAddress
    type: String
  - L4Protocol
    type: String
  - SourceIPAddress
    type: String

- PacketSummary
  type: String
- DestinationIPAddress
  type: String
- Timestamp
  type: String
- DestinationPort
  type: String
- SourcePort
  type: String
- DestinationMACAddress
  type: String

Methods

- getters / setters
  parameters: variable
  returns: variable type
  Each variable has it's getters and setter methods.

- AlarmRecord
  Inheriting from Record, this class contains the variables that match the json string obtained from kafka.
  Variables
  - id
    type: String
  - AlarmID
    type: String
  - AlarmType
    type: String
  - AlarmOccurrenceTime
    type: String
  - AlarmCategory
    type: String
  - AlarmScore
    type: String
  - AlarmDescription
    type: String
  - PacketSummary
    type: String

  Methods

  - getters / setters
    parameters: variable
    returns: variable type
    Each variable has it's getters and setter methods.

- MiscRecord
  Inheriting from Record, this class is used by the data processor as an 'in-between' state before saving to the database. As well as an extension point for adding more types of records into the database programatically in the future.
  Refer to the data processor class for further data on the key value pairs.
  Variables

  - pairs
    A Map of strings to Objects to store any 1 to many relationships

  Methods

  - getters / setters
    parameters: none
    returns: variable type
    Each variable has it's getters and setter methods.

- DataProcessor
  This class is a mediator for each one of our data aggregators used for extraciton of features from the raw data stored in mongoDB.
  We might want to hve multiple data processors for chaining different aggregators together or to split up the work into mutliple threads. This is dependant on further performance testing.
  Variables

  - client
    an instance of the associated mongoClient that requested the data aggregation
  - aggregators
    A Arraylist containing all the aggregators to be applied on a collection.

  Methods

  - getters / setters
  - processData
    parameters: variable
    returns: variable type

- IAggregator
  This interface is the building block for every aggregator to be applied to data
  Variables Methods

  - processData
    parameters: Records array of the records to be processed

- FlowRatePerSecond
  Implements IAggregator. This calculates, per port, the outgoing and ingoing connections. A record processed by this aggregator is stored in a collection as follows:

```
  Name of collection: collectionName\_FlowratePerSec
  structure of record as json:
   {
  "date" : \{" date" " Unix_Timestamp  }
  rounded down to the second this record points to.
  Connections : [
  { Port: "portNumer", "InOut" : " In/Out ", count : "Number" }
  { Port: "portNumer", "InOut" : " In/Out ", count : "Number" }
  ...

  ] This array has an entry per port if the port communicated that second.
  Precomputing this allows us to stream whenever the client needs the information
  for a specific node.
}
```

  Methods

  - processData
    parameters: Records array of the records to be processed
    specific imlpementation left to the classes implementing this interface

- NumberOfConnectionsPerNodePerSecond
  Implements IAggregator. This calculates the outgoing and ingoing connections. A record processed by this aggregator is stored in a collection as follows:

```
  Name of collection: collectionName\_FlowratePerSec
  structure of record as json:
   {
  "date" : \{" date" " Unix_Timestamp  }
  rounded down to the second this record points to.
  Connections : [
  { Port: "portNumer", count : "Number" }
  { Port: "portNumer", count : "Number" }
  ...

  ] This array has an entry per port if the port communicated that second.
  Precomputing this allows us to stream whenever the client needs the information
  for a specific node.
}
```

  Methods

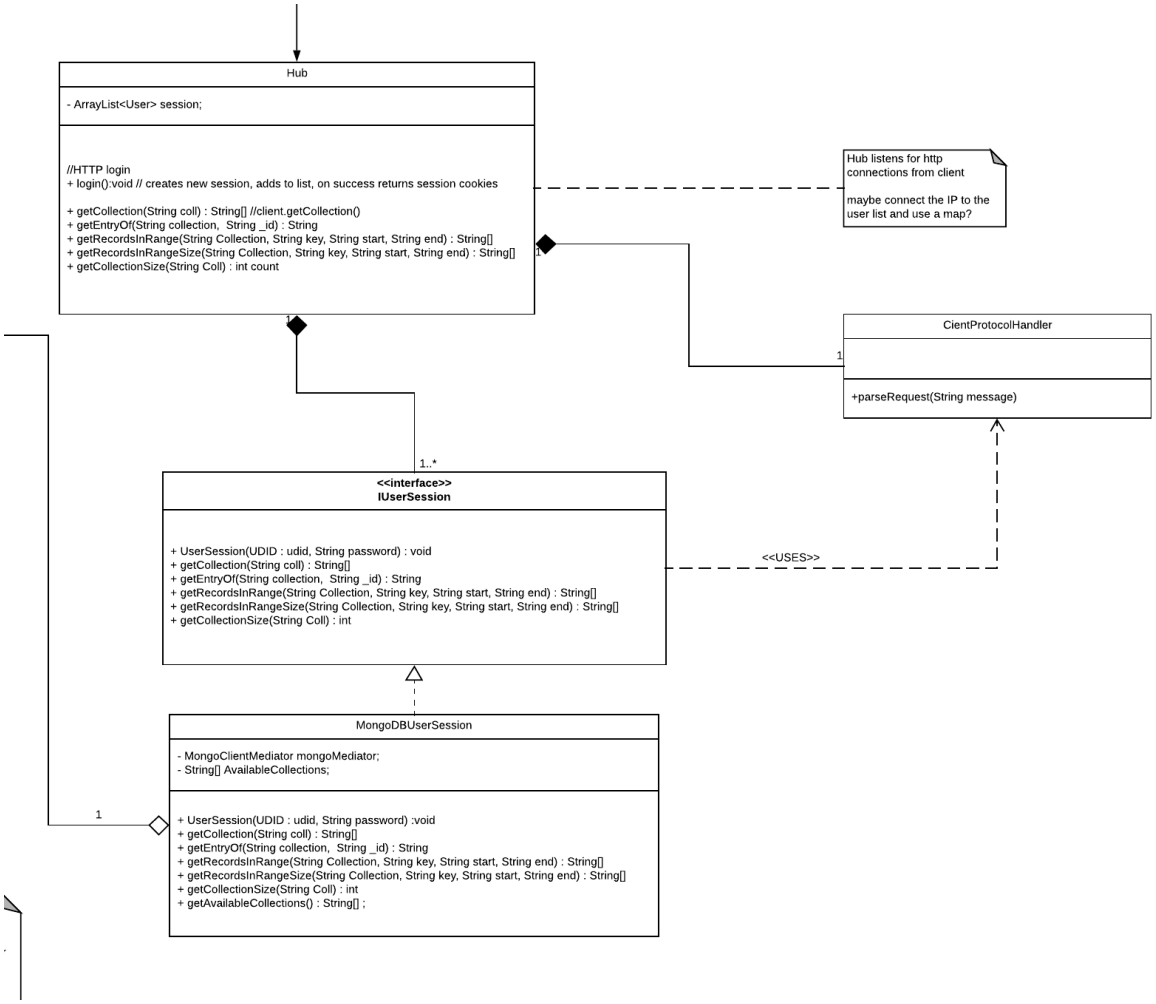



Figure 16: The classes involed in the communication between the server and the client

- processData
  parameters: Records array of the records to be processed

- Hub
  This class implements the network handlers for the websocket connection to the client and access methods for a database connection.
  Variables

  - requestHandler
    Type : ClientProtocolHandler
    The strategy object we call for the actual parsing of the client requests.

  - database
    Type : IUserSession
    The database we use during a user session.

  Methods

  - handleOpen
    parameters: Session session - the current session

returns: void
Event handler for the start of websocket connection.

- handleClose
  parameters: Session session - the current session
  returns: void
  Event handler for closing a connection.
- handleMessage
  parameters: String message - the message that we received from the client
  Session session - the current session
  returns: String - the response to be sent to the client
  Event handler for receiving a message. The message is passed to the ClientProtocol-Handler.
- handleError
  parameters: Session session - the current session
  Throwable t - the exception that occurred
  returns: void
  Event handler for errors/exceptions during communication.

- IUserSession
  An IUserSession object encapsulates a data base session. On instantiation an IUserSession connects to a database using the given user id and password and uses this connection for all following data base access.
  Methods
  - UserSession
    parameters: String username - the user id to login with
    String password - the password
    returns: IUserSession
    Factory method to instantiate a new UserSession and log in into the database using the given credentials.
  - getAvailableCollections
    parameters: -
    returns: String array with collection names
    Returns an array with the names of the collections available to the current user.
  - getCollectionSize
    parameters: String collection - the collection to query
    returns: long - the number of records
    Returns the number of records in the specified collection.
  - getCollection
    parameters: String - name of a collection
    returns: String array containing all entries of the collection
  - getRecordsInRange
    parameters: String - name of the collection to query

String key - the parameter used for filtering
String start and end - range for the filtering
returns: String array containing all entries of the collection within the filter range
Returns an array containing all records of this collection for which the value of the specified key is in the range [start, end). The records will be in the same order as they are in the collection.

- getRecordsInRangeSize
  parameters: String - name of the collection to query
  String key - the parameter used for filtering
  String start and end - range for the filtering
  returns: number of elements matching the range as int
  Returns the number of records in the specified collection for which the value of the specified key is within the range [start, end).

- MongoDBUserSession
  Encapsulates a user session for a connection to a MongoDB database.
  Methods
  - MongoDBUserSession constructor
    parameters: -
    Private constructor to create a new MongoDB session.
  - UserSession
    parameters: String username - the user id to login with
    String password - the password
    returns: a new MongoDBUserSession object
    Factory method to instantiate a new MongoDBUserSession and log in into the database using the given credentials.
  - getAvailableCollections
    parameters: -
    returns: String array with collection names
    Returns an array with the names of the collections available to the current user.
  - getCollectionSize
    parameters: String collection - the collection to query
    returns: long - the number of records
    Returns the number of records in the specified collection.
  - getCollection
    parameters: String - name of a collection
    returns: String array containing all entries of the collection
  - getRecordsInRange
    parameters: String - name of the collection to query
    String key - the parameter used for filtering
    String start and end - range for the filtering
    returns: String array containing all entries of the collection within the filter range
    Returns an array containing all records of this collection for which the value of the

specified key is in the range [start, end). The records will be in the same order as they are in the collection.

- getRecordsInRangeSize
  parameters: String - name of the collection to query
  String key - the parameter used for filtering
  String start and end - range for the filtering
  returns: number of elements matching the range as int
  Returns the number of records in the specified collection for which the value of the specified key is within the range [start, end).

- ClientRequestHandler
  This class handles client requests by parsing them, executing the requested action and producing responses. The requested actions are typically executed by calls to the database session object.
  Methods
  - handleRequest
    parameters:
    IUserSession dbSession - the current database session
    Session session - the current client session
    String message - the client request to process
    returns: String - the response to be sent to the client
    Parse the message from the client, execute the requested action (typically a database query) and construct the response message.

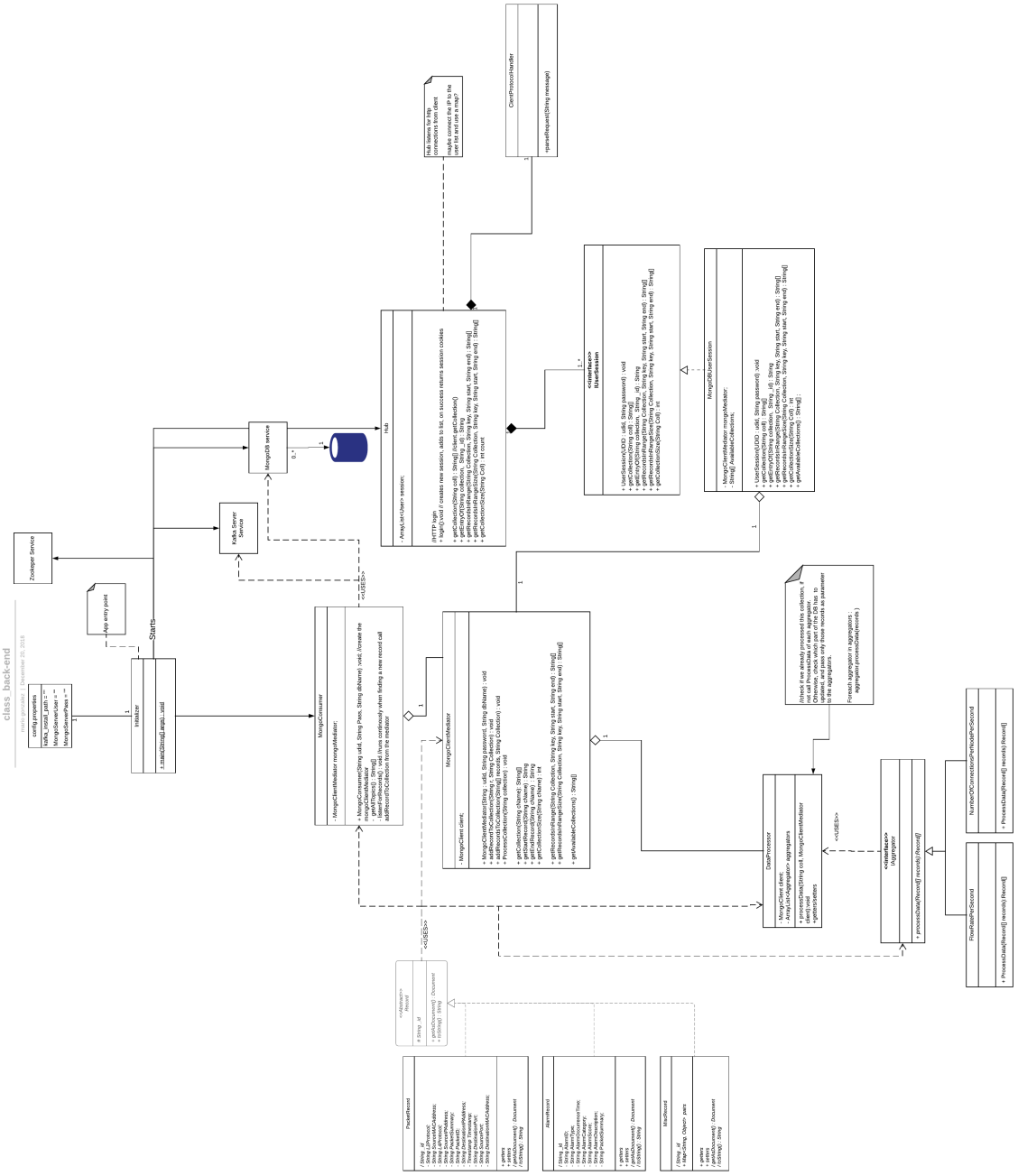



Figure 17: This is the class diagram for the whole back-end system

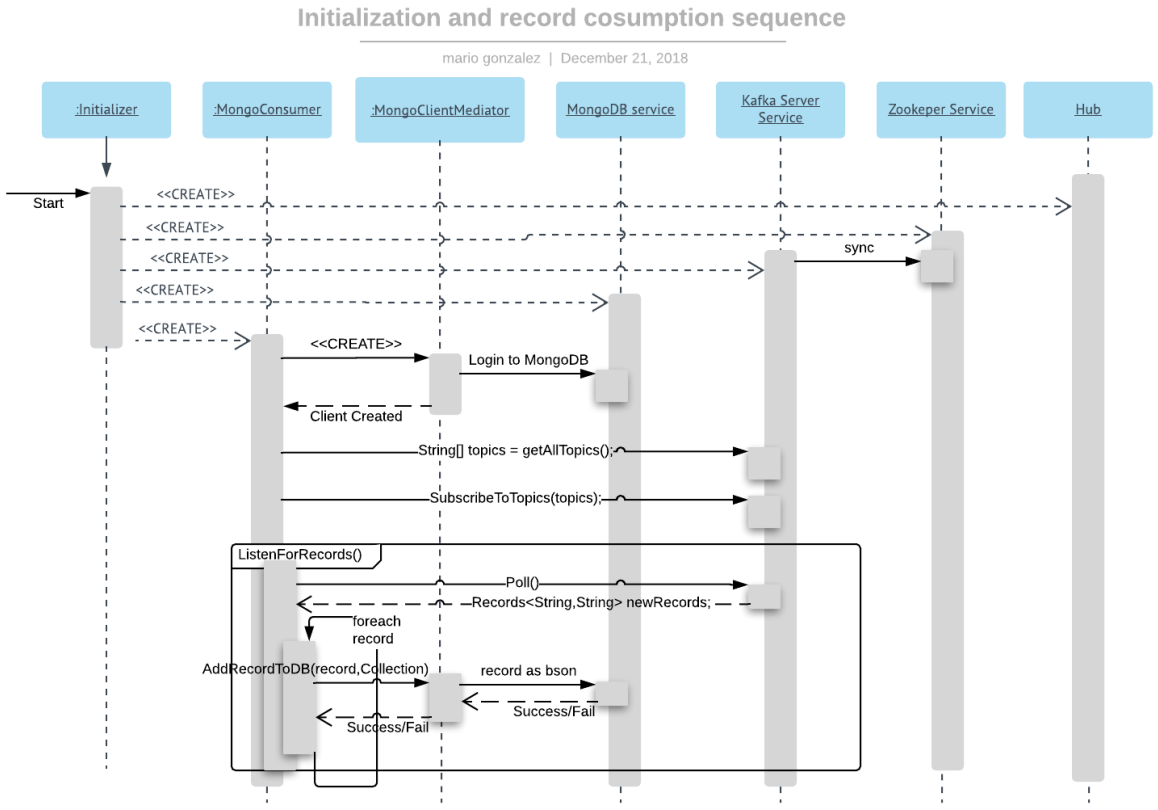



Figure 18: Initialization sequence and message consumption

### 1.3.2 Sequence Diagram

The following diagram show the initalization sequence order, the correspoding class diagram is Figure 14, the program dependes on a couple of services namely (in order),the zookeper serivce, the kafka server service and the mongoDB service. Once all services are up and running the MongoConsumer is created and can start consuming messages and the Hub can start listening to client logins, requests, etc.

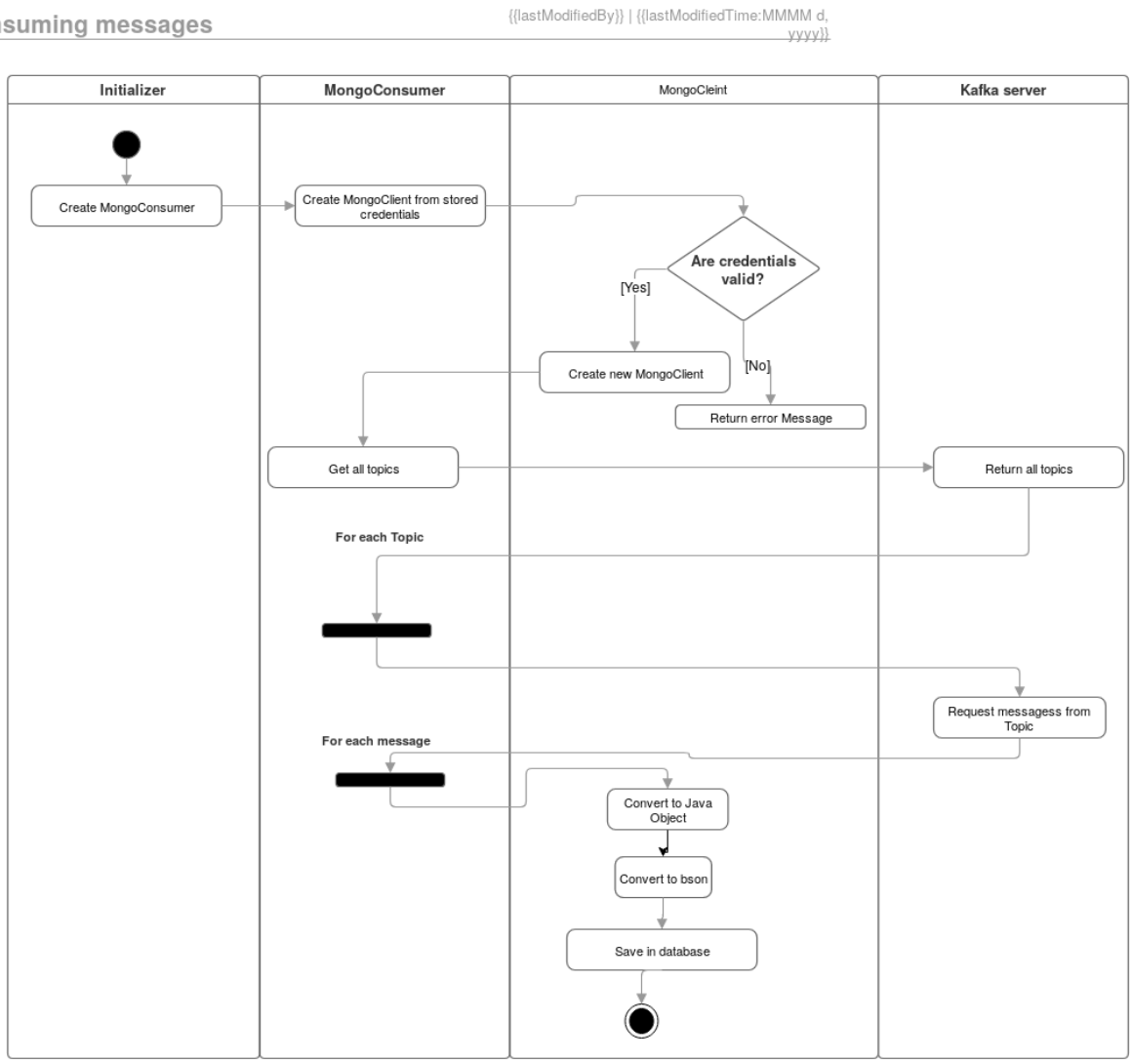



Figure 19: Message consumption workflow

### 1.3.3 Activity Diagram

For user access control the built-in UAC system in MongoDB is used, whereas every user can have roles assign to them.
A Role determines what can be done and seen within a database. For the purposes of the ADIN INSPECTOR there are three basic roles, Admin, Operator and Analyst. The admin role can create and destroy users as well as assign specific roles to them. An analyst can see all collectoins on the database and an Operator can only see part of them. The following Diagram workflow shows the user/role creation workflow an Administrator can use.

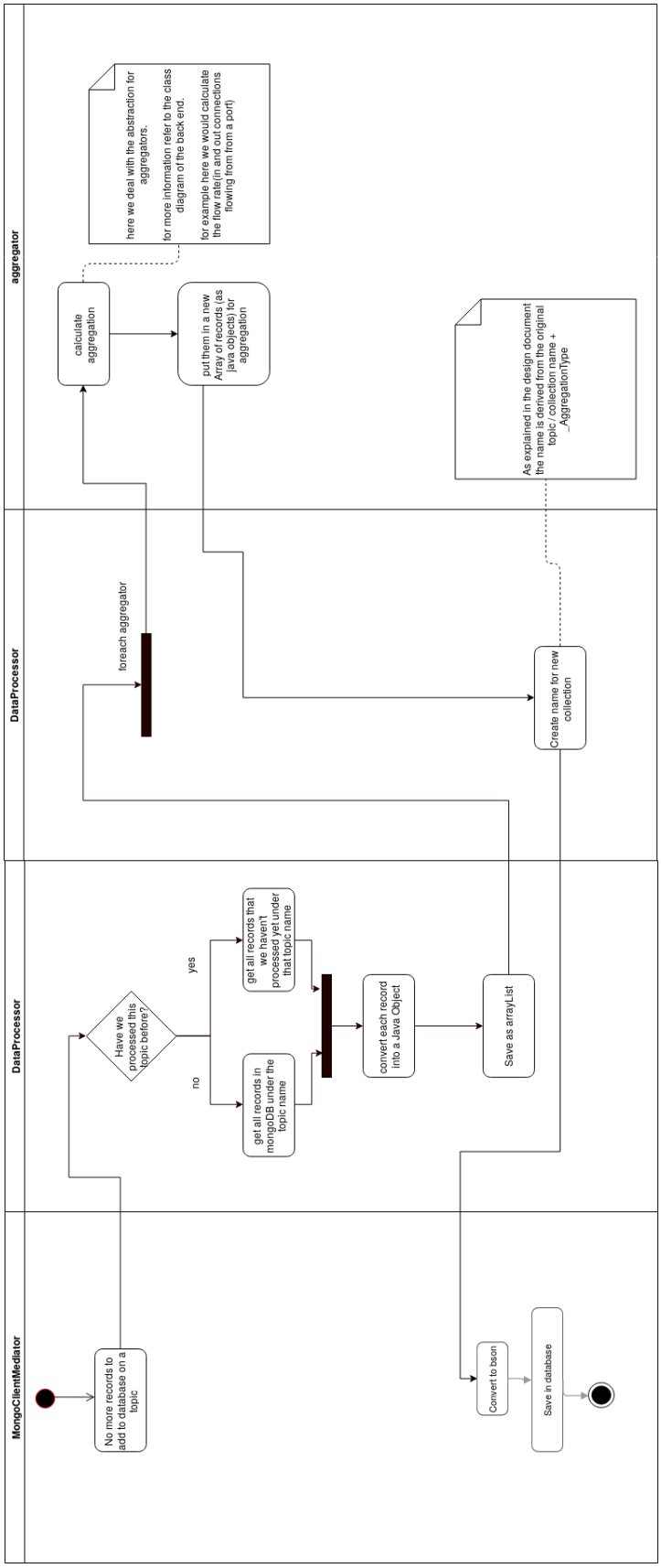



Figure 20: Processing Collection/Records workflow

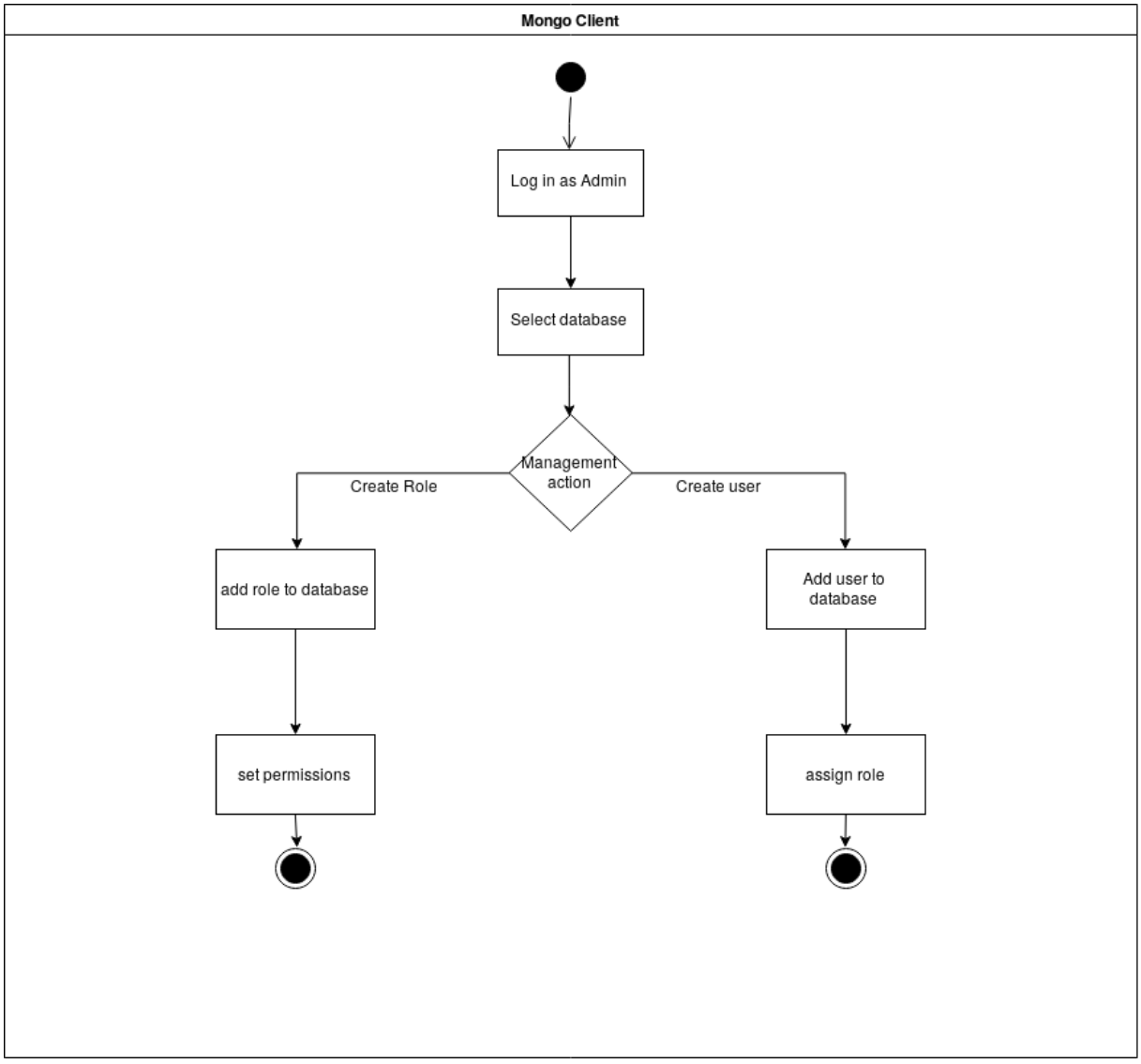



Figure 21: User Management workflow