Alabama Liquid Snake

University of Pretoria

Epi-Use

# Botic - Privacy aware chatbot
# System Architectural Design

Justin Grenfell - u16028440
Peter Msimanga - u13042352
Alicia Mulder - u14283124
Kyle Gaunt - u15330967
Lesego Mabe - u15055214

# Contents

# 1 Architectural Design

## 1.1 Determine System Types

From a top view of the entire system, we have observed that the system is more of an interactive system type. Our system is focused on the customer's interaction with the Chatbot, the customer support's interaction, and the administrator's interaction. This means that the interaction begins and ends with each user. The results in the ordering of all subsystems into a number of layers. This has the consequence of creating a clear and well-documented separation of concerns[1].

The layers that result are the following:

1. The User Interface Layer
2. The Chatbot Layer
3. The Training Layer
4. The Persistance Layer

*This comes from the application of a 4-Tier Architecture.

These four layers we treat as the total system's four main subsystems. Each of these layers, or subsystems, produce exhibit system types of their own. Going into each, we will apply the architectural design process until the we find that the individual leaf node elements are relatively easy to design and implement[2].

Now we have a look at the "User Interface Layer." This layer deals heavily with actor requests and responses; all actors primarily interact with this subsystem in order satisfy business processes. This is clearly an interactive subsystem.

Looking at the Chatbot layer, we observe that the layers in a transformational subsystem. The Support layer, or rather the Training layer we observe a transformational subsystem. The Perisistance Layer, however, is very clearly an object-persistence subsystem.

## 1.2 Architectural Tactics

In order to achieve and adhere to our quality attributes, we use architectural tactics. These are design decisions that influence the achievement of a quality attribute as they directly affect the system's response to some stimulus[1].

Below we list the architectural tactics that will be used satisfy our system's quality attributes (non-functional requirements), by order of the nonfunctional requirements listed above:

### 1.2.1 Availability

In order to fascilitate the application of tactics that improve availability, we would have to include the following into our system:

1. Logging of faults
2. Notifying appropriate entities of the faults
3. Disable the source of events causing the fault
4. Be temporarily available if need be
5. Fix or mask the fault/failure

6. or Operate in a degraded mode

In order for these to be effected, faults need to be detected, the system needs to be able to recover from faults, and it is appropriate to have a measure of preventing faults before they happen.

In order to log faults, the fault detection mechanism or tactic need to be able to store the detected 'faulty' state. The memento design pattern would clearly be useful here. We can even go as far as to have types of faults being associated with the types of mechanisms set out to identify each. For these, we may apply the Abstract Factory Pattern in order to produce specific fault logs, and store them in our fault logging database through the memento pattern. Event's can be stored in this way, so that we may be able to have a trace of operations that led to an error.

Once that is stored in the caretaker, which in and of itself is a state change, we notify the module responsible for notifying the relevant parties of faults in the system. In order to effect these responsibilities, we would have to assign functionality that stores fault logs, and notifies users. These will be done in reference to use cases, because availability is an attribute that affects users - which are the actors in the use cases. The fault detection mechanisms (architectural tactics) depend on the particular operations of the modules, thus each of the modules or subsystems stand to have their own fault detection mechanisms (availability architectural patterns). This logic applies to tactics responsible for recovering from faults, and preventing faults.

The table below provides the subsystems yeilded from functional clusters for high cohesion, and their system types as well as our chosen fault detection tactic, recovery tactic and fault prevention tactic. These are only subsystems that are necessary, and ordered according to priority. The architectural tactics are solution specific, therefore we expect to add many more of them as the development of the project continues; software engineering is a wicked problem afterall:

| Subsystem | Subsystem Type | Fault Detection | Recovery from Faults | Fault Prevention |
|---|---|---|---|---|
| Chatbot Subsystem | Transformational Subsystem | ping/echo, Retry | Exception Handling | Removal from Service |

Please note that the Chatbot Subsystem consists of the Message Scraper, Classifier and the Response Generator subsystems; thus all the mentioned tactics actually apply to each of this subsystem's components.

Two modules will be produced here: one for log management, and another for notification. These require that data be persisted.

### 1.2.2 Security

Looking at the quality requirements involving security in following the design checklist specified by [2], we find that it is useful to use physical security as a model for software security as indicated by [1]. This results in the four categories of focus for this quality attribute: detect, resist, react, and recover.

In order to support the design and the analysis of security in our system we will pay attention to the following, and ensure that these responsibilities are assigned and met:

1. Identify the actor
2. Authenticate the actor
3. Authorize actors
4. Grant or deny access to data or services

5. Record attempts to access or modify data or services

6. Encrypt data

7. Recognize reduced availability for resources or services and inform appropriate personnel and restrict access

8. Recover from an attack

9. Verify checksums and hash values

With regards to our data we shall ensure that the following is observed:

1. Seperation of data of different sensitivities

2. Ensure different access to data of different sensitivities

3. Ensure that access to sensitive data is logged and that the log files is suitably protected

4. Ensure that data is suitably encrypted and that keys are separated from the encrypted data

5. Ensure that data can be restored if it is inappropriately modified

This attribute modifies or enhances modules introduced in the availability requirement analysis above, this includes the assignment of new responsibilities in particular, certain logs have to be stored securely– those logging access to sensitive information. It also makes use of the notification module to let the appropriate personnel know that their system is being attacked, when these attacks are detected.

The table below includes the architectural tactics we will employ with respect to each of the four categories (detection, resistance, reaction, and recovery) we defined earlier, the subsystems and their subsystem types. These are in order of priority:

| Subsystem | Subsystem Type | Detection | Resistance | Reaction | Recovery |
|---|---|---|---|---|---|
| User Interface Subsystem | Interactive Subsystem | Detect service denial, Detect message delay | Identify actors, Authenticate actors, Authorize actors, Encrypt data, Separate entities | Revoke access, Lock computer, Inform Actors | Maintain Audit Trail |
| Chabtot Subsystem | Transformational Subsystem | Verify Message Integrity, Detect service denial | Identify actors, Authorize actors, Encrypt data | Revoke access, Inform Actors | Maintain Audit Trail |
| Chatbot Trainer Subsystem | Transformational Subsystem | | Authorize actors, Encrypt data | Inform Actors | Maintain Audit Trail, Data Model checklist above* |
| Database Subsystem | Database subsystem | | Authorize actors, Encrypt data, Separate entities | Inform Actors | Maintain Audit Trail, Data Model checklist above* |

Note: we mention the UI subsystem, but what we really mean are the operations performable by the roles of Administrator and Customer Representative. The UI is how they interact with the system.

These are not meant to be exhaustive; they are meant to be revised according to future needs and adapted with feedback.

### 1.2.3 Reliability

All of these concerns are encapsulated in the availability analysis above as well as the security analysis above. The requirements have nonetheless been associated with the revelant use cases.

### 1.2.4 Performance

Here we will look at design decisions that affect the performance of the system. Here we will consider two contributors to the response time: processing time and blocked time[1]. In these our architectural tactics will be concerned with the control of resource demand as well as the management of resources.

A design checklist proposed by [1] suggests useful design considerations for the sake of ensuring performance; other than just following this checklist as in, on pages 142 to 144, we will note especially that under resource management, it is advised that we manage and monitor important resources under normal and overloaded operation; resources such as process and thread models– a detail that involves the availability quality attribute.

The table below includes the architectural tactics that we will employ with respect to the control of resource demand and the management of resources in our system. This will only be for the performance critical components, and ordered by importance. The list contains the following columns: subsystem, subsystem type, control resource demand, and manage resources. This is not an exhaustive list, but alas software engineering is a wicked problem:

| Subsystem | Subsystem Type | Control Resource Demand | Manage Resources |
|---|---|---|---|
| Chatbot Subsystem | Transformational Subsystem | Prioritize Events, Increase resource efficiency, Reduce overhead | Increase resources*, Introduce concurrency, Load balancer, Schedule resources |
| Message Scraper | Transformation Subsystem | Prioritize Events, Increase resource efficiency | Increase resource*, Introduce concurrency, Load balancer, schedule resources |

Note: With regards to "Increase resources" we mean to say that a recommended deployment environment ought to be a cloud platform such as PaaS (Platform as a Service) which would dynamically increase or reduce resources based on demand and bill the customer accordingly.

### 1.2.5 Scalability

Looking into design decisions affecting scalability, we have to consider the impact of adding or of removing resources, and these measures will reflect on associated availability as well as the load that will be assigned to existing and new resources[1]. Here, two kinds of scalability will be dealt with: horizontal scalability (adding more resources to individual units), vertical scalability (adding more resources to individual nodes).

### 1.2.6 Maintainability

Maintainability may encapsulate some aspects of modifiability as well as testability and availability. All three of these will be observed as well as that which is required for maintanence management from [2].

In as far as modifiability is concerned, our architectural tactics in this regard, concern all of our system and its subsystems; they are reducing the size of modules through spliting modules, increasing cohesion

by increasing semantic coherence, reducing coupling by encapsulation, using an intermediary, restrict dependencies, refactoring and abstract common services.

Looking at architectural tactics for testability we are mostly going to rely on language specific testing tools, especially those that can allow us to follow state, pause and playback tests. It is also useful to follow documentation. The high priority subsystems, in terms of availability and functional responsibilities are high priorities for testing.

Availability has already been covered above. As far as preparing the system for maintainability is concern, during development we will pay close attention to the application of software design principles as well as the application of software design patterns.

"An aspect of testing in that arena is logging of operational data produced by the system, so that when failures occur, the logged data can be analyzed in the lab to try to reproduce the faults."[1]

## 1.3 Applying Architectural Styles

### 1.3.1 Entire System

The whole system adheres to the 4-Tier architectural style. It's layers or subsystems include: The User Interface Layer, the Chatbot Layer, the Support/Training Layer, and lastly the Persistence Layer. This architectural style ensure that all the subsystems and modules can be seperate and thus developed seperately with minimal interaction between them (changes), ensuring adherance to the software design principles of high cohesion and low coupling[1]. This makes our entire system more portable and modifiable (and thus maintainable). Here is the solution:

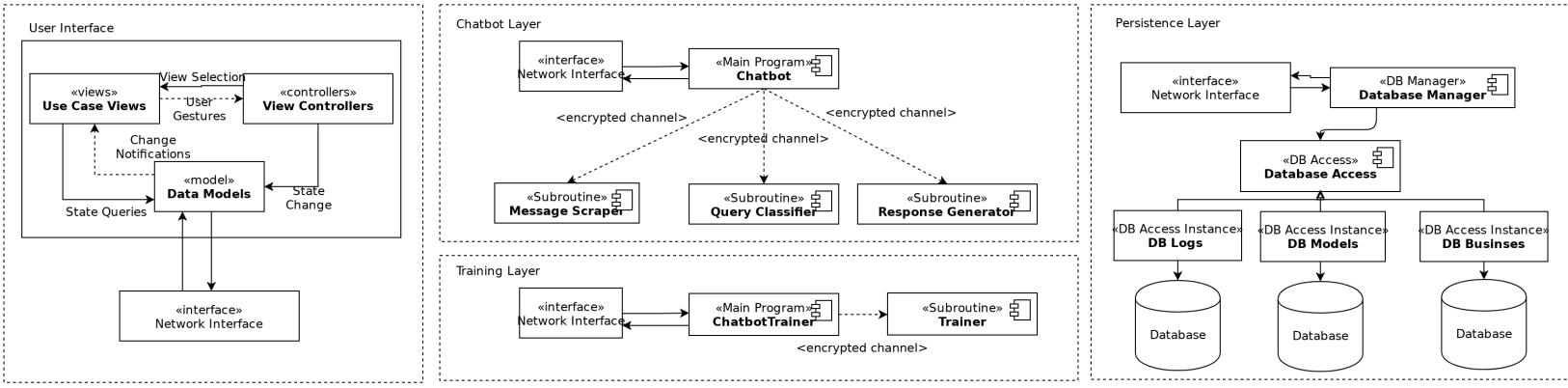


Figure 1: Architectural Stye of the System

The allowed-to-use relation here is denoted by the geometric adjacency of the layers from left to right– layers on the left use layers on the right and the layers on the right are used by the layers on the left. The exception here is the connection between the user interface as well as the persistence layer, as this is required to store information from controllers within the UI's MVC architecture, as well as obtaining information or data for UI level use case i.e. displaying users or obtaining user information. The connections between each layer are made to be through encrypted channels. An appropriate technology will be used to implement this during the implementation phase e.g. SSL.

### 1.3.2 The User Interface Subsystem

The User Interface Layer that is also an interactive subsystem uses an MVC architectural style. Using this architectural style, we ensure that user interface interface functionality, something that tends to change frequently especially in response to knew design trends *Quote IMY theory notes, be kept seperate from the application functionality and yet be ever responsive to user input, the most important thing in an interactive system[1]. This is the solution:

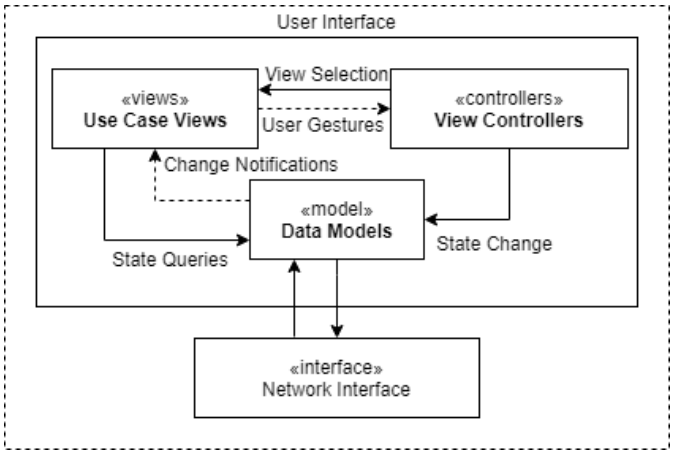


Figure 2: Architectural Style of the User Interface

### 1.3.3 The Chatbot Layer

The Chatbot Layer is a transformational subsystem, and will use a Main Program and Subroutines architectural style.

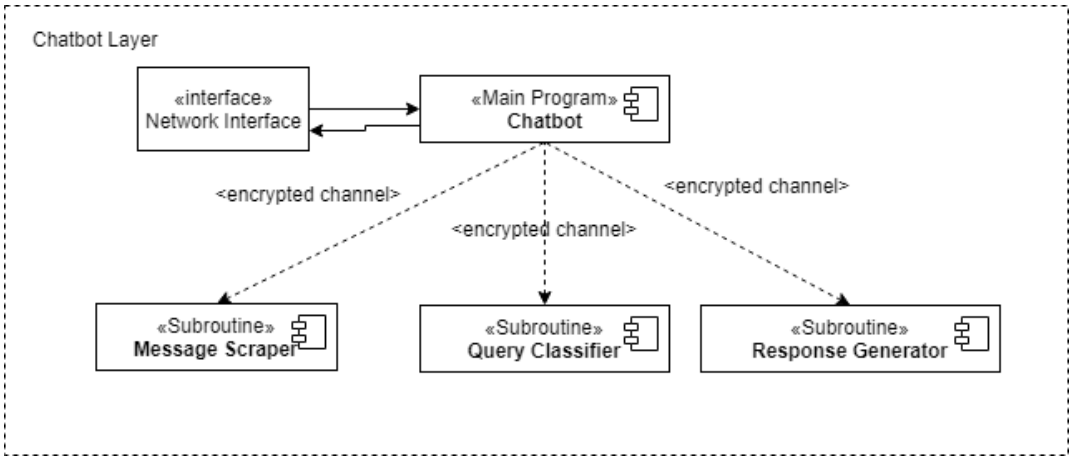


Figure 3: Architectural Style of the Chatbot Layer

### 1.3.4 The Training Layer

This layer is a transformational subsystem and will also use a Main Program and Subroutines architectural style.

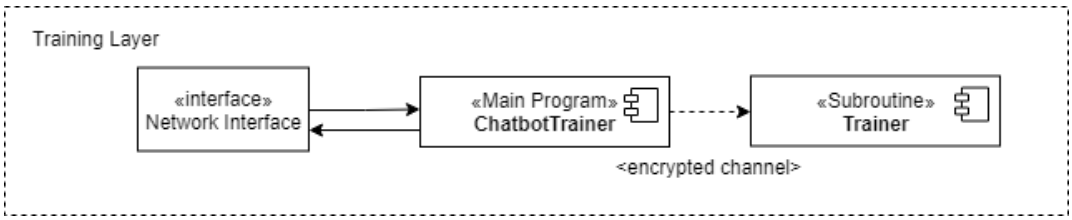


Figure 4: Architectural Style of the Training Layer

The ChatbotTrainer is a single container that has 4 instances: these are to service requests from the user interface (requests which are sent by the Administrator user), one to train the Message Scraper, one to train the Classifier, and one to train the Response Generator. It uses the Strategy Desgin Pattern to switch different training strategies/methods to suit each subsystem it services. This results in multiple instances of the same subsystem, but each using a different training method and data on a different chatbot subsystem.

### 1.3.5 The Persisitance Layer

This layer is a clear candidate for the Persistence Framework architectural style, and it's architectural style is just that. Here is our solution:

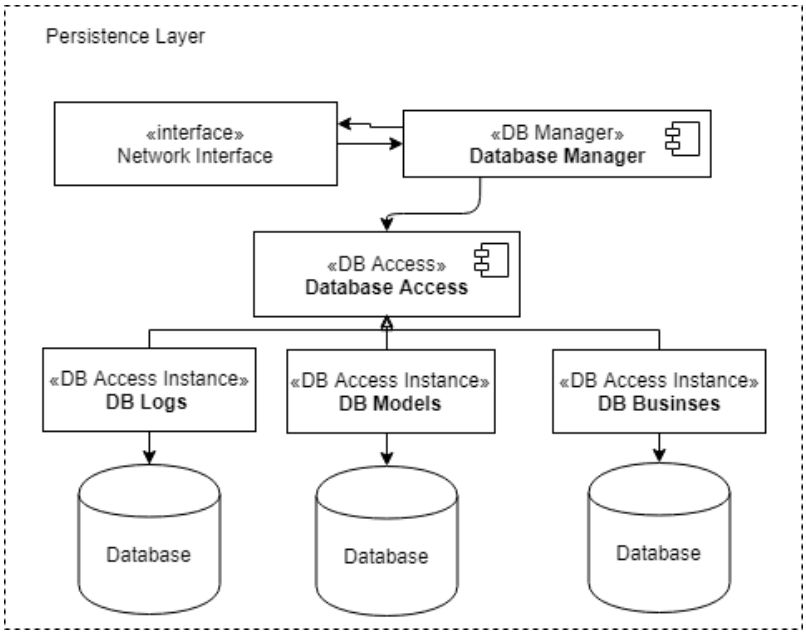


Figure 5: Architectural Style of the Persistence Layer

In accordance to the architectural tactics we have chosen to observe to achieve availability and security quality attributes and requirements, in particular the security tactic to seperate access and data of different sensitivities, we have made the decision to have three databases. The first is for logs and audit trails, the second for the data models what would be used for the chatbot subsystems, and the third for recording queries and responses– data that is concerned with the functional business processes.

## 2 References

[1] R. K. Len Bass, Paul Clements, Software Architecture in Practice. Addison-Wesley, 2013.

[2] D. C. Kung, Object-Oriented Software Engineering An Agile Unified Methodology. McGraw-Hill, 2014.