2QUEUEOOL4U

OCTOBER 31, 2017

ARCHITECTURE DESCRIPTION AND SYSTEM DESIGN

Part 0: Metadata

Project name: NextText

Team name: 2QueueOOL4U

Team members:

James Mulvenna/100965629

Yue Zhang/100980408

Devon Plouffe/100715712

Part 1: Architecture Description

Over the course of our application development, our strategy was frequently modified due to our repeatedly changing conditions. We ran into multiple architectural complications. One being Android mobile application research with regards to our proposed product. As it stands, we have used a collection of architectural styles, as no one style best suited our system. Diverse non-functional/functional properties apply to our product, and for this reason particular imperative styles were used respectively.

During the D0 phase 2QueueOOL4U was most concerned with the practicability of the system, we assessed the inter-dependent components and conducted research to determine how achievable our proposed system was. Fortunately, we found in the investigation period that designing our proposed application on iOS would be near unattainable by cause of Apples restrictions on message automation. Despite running into arguably our most important issue, we made the decision as a team to change platforms, as Androids development guidelines solved this obstacle.

In totality, at this moment our product consists of numerous architectural styles including but not limited to: object-oriented, client-server, mobile code, and finally event-driven. Although at this moment these architectural styles apply to our system, it would be naive of 2QueueOOL4U to declare these as the only styles. Realizing the remaining term of development, perhaps a new style might be used to account for outstanding system requirements.

Object-Oriented

By reason of 2QueueOOL4U's system utilizing many independent/dependent objects, using an object-oriented design gave us the ability to administer proper data abstraction. As it goes, our system employs several models including: Message, Time, Location, Weather, and more in which are all wrapped in an object which serves our backend database. Considering objects such as Location, and Weather service external API processes, encapsulation was used by wrapping the respective objects in an object which suppresses these background methods. All user input is recorded in the objects listed above and compressed into the final wrapped object called MessageWrapper. Proceeding, our database retrieves this information to be called upon when applicable. In light of the authentic definition of the Object-oriented architectural description, it is clear to notice NextText's division of responsibility as well as its' collaborative objects communicating through interfaces leading to the final result.

Client-Server

NextText exploits its client-server composition constantly when operating. Any user information retrieved, is stored on the backend server. When required, the unified trigger imposed by the client retrieves the respective data from the server to be dispatched accordingly. The server utilizes SQL to effectively compose transactions such as, get all, get sms, get email, add, update, delete, delete all, and many more useful methods.

As a result of the server being accessed indefinitely by the client, it serves the most important role in the system. In order to access/store data as per client, the server must ensure effortless transactions, integrity, scalability, synchronization, and more mated traits.

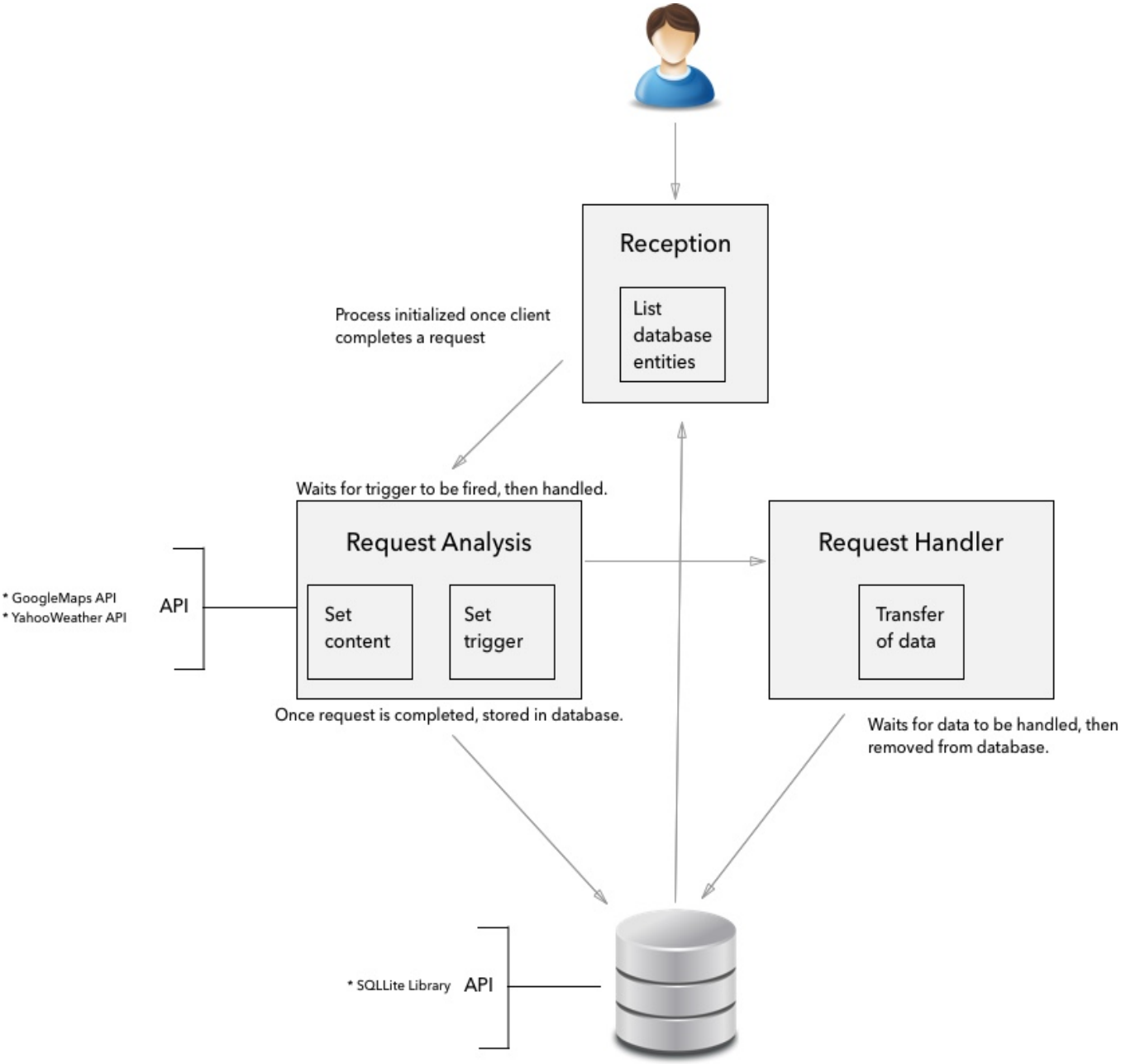
Mobile Code

For the sake of our weather trigger adopting the Yahoo Weather API, we must make use of requests to JSON data on the Yahoo servers by cause of the client assigning a constraint based on their weather expectations. These requests are done in intervals of time, consisting of mandatory information such as the city and country to perform the remote function and obtain the information needed to inquire the clients' constraint.

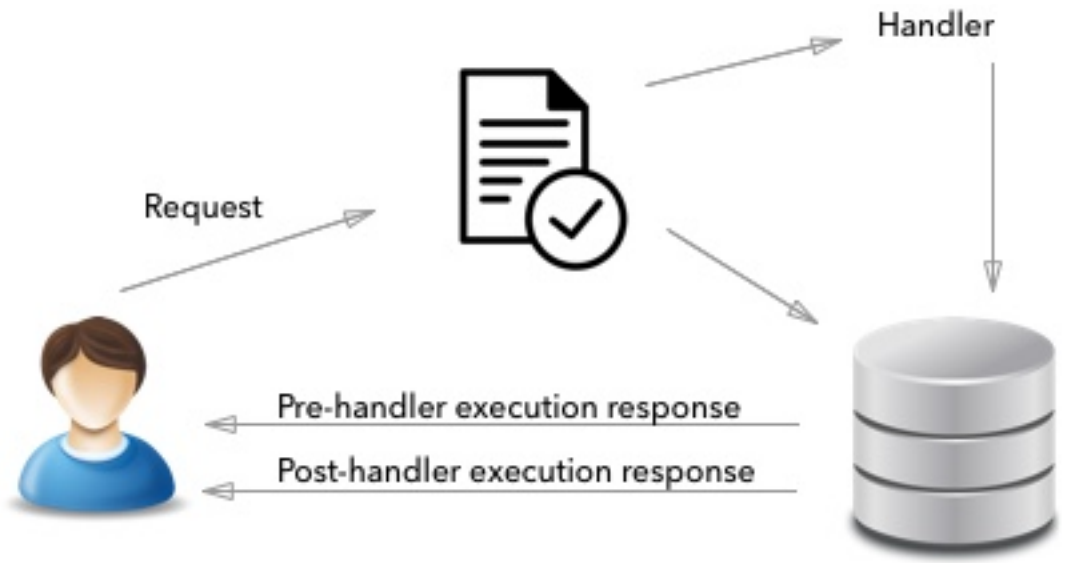
Event Driven

The fundamental objective of NextText is to send scheduled messages such as SMS or Email, based on time, location, or weather. In other words, every message is coupled with an event in which fires an action. Various senders and receivers are in place to handle methodology implicitly. Our system makes use of the Event Driven Architecture at every moment. Concealed methods are called for each user-system interaction in order to accomplish prescribed events. As you can see in the Reference Architecture below, each requested event is driven by some trigger which is handled when applicable. The architecture revolves around this implication.

Part 1.1: Reference Architecture Diagram



Part 1.2: Client-Server Diagram



Part 1.3: Mobile Code Diagram

Part 2: System Design:

To effectively implement our system, we couldn't cease at the architectural properties. Instead, bringing together the various obligations of NextText, an assortment of design patterns were endorsed. Composite, Façade, Iterator, and Chain of Responsibility were all exploited thoroughly. While other designs are comparable, these design patterns were more fitting with regards to our requirements. The condensed reasoning for these patterns, is that our system used abstract templates which exhausted various interfaces, in which were used by composite objects. Object schema is so that they are accessed and traversed with ease, and individually marry a chain of responsibility. In terms of chronology, our design was built from the ground up. Starting from the back-end and the various models that would be required, we depleted time conceptualizing the object entities that would be best suit our system. We next, focused composed the front-end to be as user-friendly and conceptual as possible. By using the method interfaces defined in the backend, we were able to construct a clean interface which handled all user input. By using this development approach, our system is able to inhibit changes relatively smoothly. Changing system components for likewise external components is trivial. For instance, changing our self-made SQLLite database for an external backend API would be uncomplicated. Obviously, we would need to change certain function names or guidelines, however the underlying transactions of a database would still be apparent. This is a good example, considering our current database might not be scalable in the future. However, because this would be a simple transition, this issue would be resolved. Convoluted changes, such as adding a new component would require updating certain objects and accounting for new changes in dependent classes, however would definitely be feasible within our system implementation.

Composite

Considering the application manages numerous objects such as Message, Time, Location, Weather, and more, we had to find respectable way of storing and maintaining these objects without lack of information, however with ease of storing/retrieving. Realizing this, we designed a single model wrapper called MessageWrapper to hold all of the listed objects as well as other attributes such as an id, current time created, and more. MessageWrapper serves both the frontend and backend, which in turn makes the clients interface simple to administer and for lack of a better term; foolproof. Subsequently, using the composite design pattern made it effortless to follow our architectural, and developmental strategies.

Façade

In view of the complexity within each component of the system residing both client-side and server-side, system understandability was absorbed at the client level. Walking through the system starting from the backend, interfaces are arranged such that the overhead module is provided a set of functionalities (templates) it doesn't need to fathom to take advantage of. For instance, the database is serviced with a MessageWrapper object at the highest level of abstraction. The frontend accesses from the database using simple CRUD methods, and more. By utilizing the façade design pattern, we achieved reduced complexity, minimized dependencies between subsystems, and provided a single, simplified interface to the general facilities of the system in its totality.

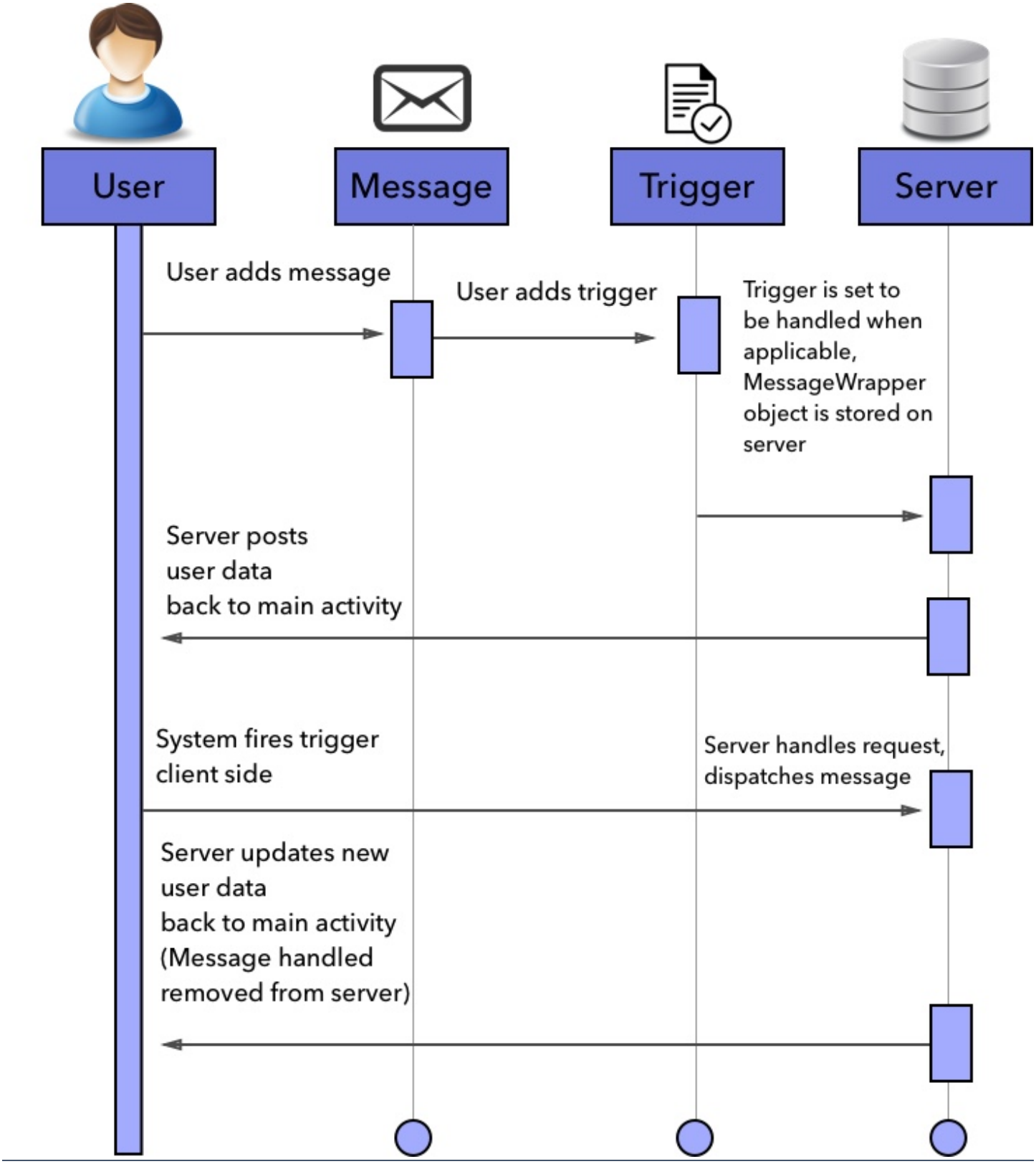
Iterator

In order to provide a way to access the messages in the database sequentially without exposing its underlying structure, an iterator was initialized in the client-side main list activity to expose the useful message information to the end-user at their request. This iterator displays messages such as SMS, Email, or all queued at the click of a button. While the user sees this as a simple product of communication, the underlying methods of retrieving this data with relation to the sorting algorithms based on time inputted into the database, type of message, and more are obscurely functioning. It was essential to use an iterator design pattern to iron out this obstacle, as it was necessary to disguise the client transactions utilizing traversal and accessibility.

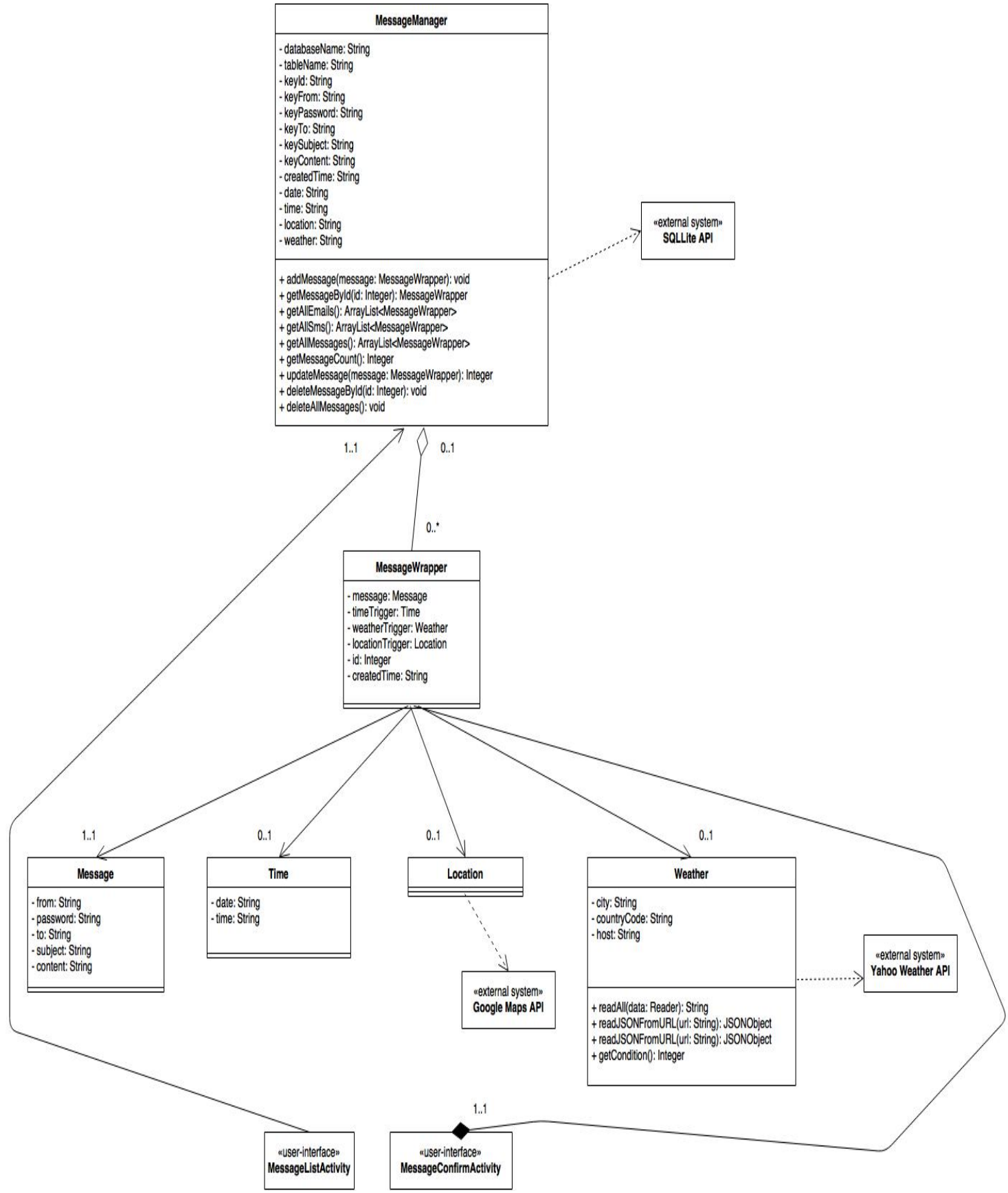
Chain of responsibility

From the moment the MessageWrapper is inputted into the database, a chain of responsibility is tied to its instance. Requests and handlers are in place directing events with regards to their properties. For instance, when the user creates a MessageWrapper (Message coupled with an event trigger), a signal is created which is fired when applicable. Proceeding a provoked signal, handlers work alongside the message object to dispatch it according to its form including SMS or Email. Instances and their signals are initialized client side; however, handlers are kept in the backend. It was necessary to assign responsibility at every instant of operation for the sake of event triggers existing as long as a message is in the queue.

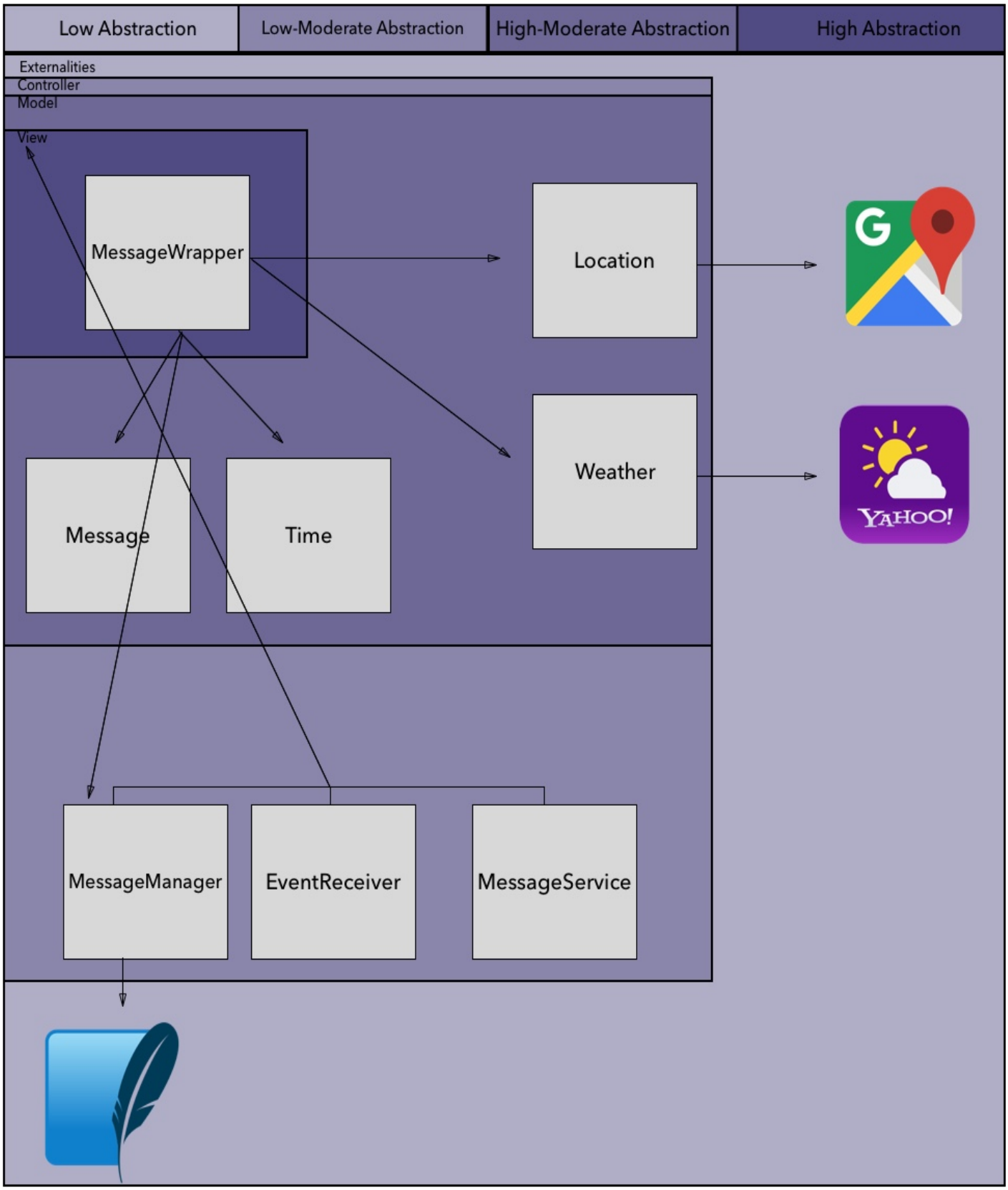
Part 2.1: Basic Scenario Sequence Diagram



Part 2.2: Class Diagram



Part 2.3: Façade



Part 3: Task Overview:

For the reason that initial complications emerged, system reconstructions were necessary. As a result of this, following early preparations and studies, roles were set. Roles were appointed with relation to outstanding development time, particular developer strengths, and developer interests. Subsequently, divisions of tasks were split in this way:

James

James takes control of the entire backend, consisting of the Database and Models. He administers the event handlers as well as takes care of the Weather event trigger. Finally, he yields the documentation including D0, D1, D2, D3, and their respective diagrams, as well as the presentation slides for the proposal, and the demonstration script.

Yue

Yue manages the entire frontend user-interface. He is to some extent familiar with the backend as James, and Devon supply him with the functionality he demands. Yue develops client initiated event triggers and finally contributes to most documentation. By cause of Yue requiring the backend interface yet not vice versa, the backend developers have a general understanding of the front-end, however not enough to develop upon, but rather to document its purpose.

Devon

Devon is in charge of the Location event trigger. Devon also contributed to the component diagram in D2, as well as the presentation slides for the proposal.