University of Groningen

Enterprise Application Integration

Scooter Maintenance System

Group 4

Luc Breeman	s3212335
Wouter Hertsenberg	s2992795
Ruben Kip	s2756781
Robert Riesebos	s3220672
Kaavyaa Stalin Thara	s4075269

January 11, 2021

Version 1.0

Version history

Author legend

Author	Full name
LB	Luc Breeman
WH	Wouter Hertsenberg
RK	Ruben Kip
RR	Robert Riesebos
KT	Kaavyaa Stalin Thara

Table 1: Author information

Revisions

Version	Date	Author(s)	Description
1.0	23.11.20	RR	Document creation and structuring
1.0	02.12.20	RK	Architectural patterns
1.0	04.12.20	RR	Added functional requirements
1.0	05.12.20	KT	Explained system description
1.0	06.12.20	RK	Extended system description
1.0	14.12.20	RK	Tiered architecture
1.0	21.12.20	RR	Added system patterns diagram
1.0	04.01.21	RK	Architecture Technologies, updated ch 1 and 3
1.0	05.01.21	RK	Mapping implementation, applying feedback
1.0	10.01.21	RR	Rewrote and added patterns to the Architectural patterns section
1.0	11.01.21	LB, WH, RK, RR, KT	Worked on finishing the report

Contents

1 System description

The Scooter Maintenance System (SMS) aims to ease the process of managing scooters by providing a real-time notification system in which scooters can report their difficulties. Maintenance personnel are able to receive notifications of broken scooters. This combined with the location of the scooter allows the maintenance personnel to quickly and efficiently find and repair the broken scooter. A statboard is also provided, showing several statistics of the system. The overview of the SMS design is shown in Figure 9. An important thing to note is the fact that all four applications are fully designed and created by us, but with the idea that each system could act as an independent application.

In order to provide the user with our main system, the Scooter Maintenance System, the following four systems will be connected and integrated with one-another:

1. **Scooter OBD (on-board diagnostic)**, for diagnosing and reporting issues with the scooter. Each scooter has its own OBD system running, which is able to send notification messages in case of any trouble. Marked with orange in the system diagram.

2. **Statistics dashboard**, also referred to as the statistics dashboard or statboard, a dashboard for showing interesting statistics of the Scooter Maintenance System. Marked with purple in the system diagram.

3. **Management system**, a component that is routes broken scooter messages, and that communicates with the maintenance app. Marked with red in the system diagram.

4. **Maintenance app**, the mobile application that is used by maintenance personnel to receive broken scooter notifications and mark repaired scooter as functional. Marked with blue in the system diagram.

If the OBD system of a scooter detects a problem, the OBD system will send a notification to the SMS. This message reports that the scooter is broken, and also includes additional data such as the corresponding error-code, an error message and the coordinates of the specific scooter. The message format is JSON, and all messages make use of the following format:

```
{
    timestamp: string,
    scooter_id: string,
    status: string,
    error_code: int,
    error_message: string,
    latitude: double,
    longitude: double
}

Example:
{
    timestamp: "2018-04-26 12:59",
    scooter_id: "QVYR8MIRG7",
    status: "BROKEN",
    error_code: 1122,
    error_message: "Low tire pressure",
    latitude: 10.1234,
    longitude: 19.1234"
}
```

1.1 Brief overview of the system

We will be using a publish-subscribe channel as there are two applications (the statistics dashboard and the management system) that are interested in the same (broken scooter) messages. The message will be inserted into a database by the statistics system adapter, while the management system will handle and forward the error messages to their corresponding queue. The error messages are forwarded to a queue based on the contents of the error message, specifically the location of a broken scooter. The maintenance app will then consume messages from the queues in a competing consumer fashion.

The maintenance app, which is run on Android, is the app where maintenance personnel can see any broken scooters and possibly fix them. This app gets its messages from different message queues bound to the regions of Groningen. The maintenance personnel can then decide from which region they want to start consuming messages so that they can evenly distribute the work between themselves.

Finally, to make life easier for the maintenance personnel, we integrated the system with Google maps. This allows the maintenance personnel to lookup the location of broken scooters. If the scooter has been repaired, the maintenance app becomes a producer and sends a message with an updated status back to the publish-subscribe system to which both the management system and statistics dashboard are subscribed.

2 Functional requirements

This section lists the functional requirements for each subsystem of the complete system. For each requirement the implementation priority is given. This priority can be either **Must** or **Optional**, and indicates whether the functional requirement must be implemented in the system.

2.1 Scooters

FR-1	**Must**	Scooters must send a failure notification to the management system when a critical component/system fails.

FR-2	**Must**	Scooter failure notifications must contain the scooter's location.

FR-3	**Must**	Scooter failure notifications must contain the failure reason(s) represented by an error code.

FR-4	**Optional**	Scooters may periodically send location updates to the management system.

FR-5	**Optional**	Scooters may periodically send the total distance driven to the management system.

2.2 Statistics system

FR-6	**Must**	The statistics system must be able to show the number of past failures for each scooter.

FR-7	**Must**	The statistics system must be able to show the total number of all scooter failures.

FR-8	**Must**	The statistics system must be able to show the total number of repaired scooters.

FR-9	**Optional**	The statistics system may be able to show the count per failure reason for each scooter.

FR-10	**Optional**	The statistics system may be able to show the total count per scooter failure reason.

FR-11	**Optional**	The statistics system may be able to show the scooter fleet size.

FR-12	**Optional**	The statistics system may be able to show the distance driven for each scooter.

FR-13	**Optional**	The statistics system may be able to show the total distance driven by all scooters.

2.3 Management system

FR-14	**Must**	The management system must notify the relevant maintenance personnel (through the mobile application) when a scooter breaks down. Relevant maintenance personnel includes personnel within the designated region that the scooter breaks down in.

FR-15	**Must**	The management system must store scooter failures in a database.

FR-16	**Must**	The management system must store scooter repairs in a database.

2.4 Maintenance (mobile) application

FR-17	**Must**	The maintenance app must show a list of defect scooters assigned to the corresponding maintenance person.

FR-18	**Must**	The maintenance app must show the location of a defect scooter on a map.

FR-19	**Optional**	The maintenance app may show the locations of functioning scooters on a map.

FR-20	**Must**	The maintenance app must allow maintenance personnel to view the error code for a scooter failure.

FR-21	**Must**	The maintenance app must allow maintenance personnel to view the reason for a scooter failure.

FR-22	**Must**	The maintenance app must allow maintenance personnel to manually update the status of a scooter.

FR-23	**Must**	The maintenance app must allow maintenance personnel to log in.

FR-24	**Must**	The maintenance app must allow maintenance personnel to log out.

FR-25	**Optional**	The maintenance app may allow the maintenance personnel to comment on the reason of a scooter failure.

3 Tiered architecture

The layered architecture, as shown in Figure 1, is split into three layers. The first layer is the Presentation layer, this contains the repair man app which is a simple UI with a map and button, and the stat board showing the statistical data from the repairs. The repair man app shows accepted job data from the maintenance system in the application layer, but sends back input to the maintenance system e.g. when updating the status of a broken scooter to “functional”.

In the application layer we find the scooter logic that reports errors, acting as a producer and error generator. These errors are reported to the maintenance system logic in order to dispatch the job to a repair man (in the repair man app). The maintenance system logic is also responsible for adding/deleting new areas of Groningen. Furthermore, these scooters errors are also reported and stored in a database for statistical purposes, so the statboard logic can use this information. Finally, the application layer also contains the statboard logic. This logic simply calculates the statistical data which is shown on the statboard.

Lastly, we have the database layer containing the management system database for storing/updating the status of scooters and the statistics system database that stores all messages about the scooter errors for statistical purposes.

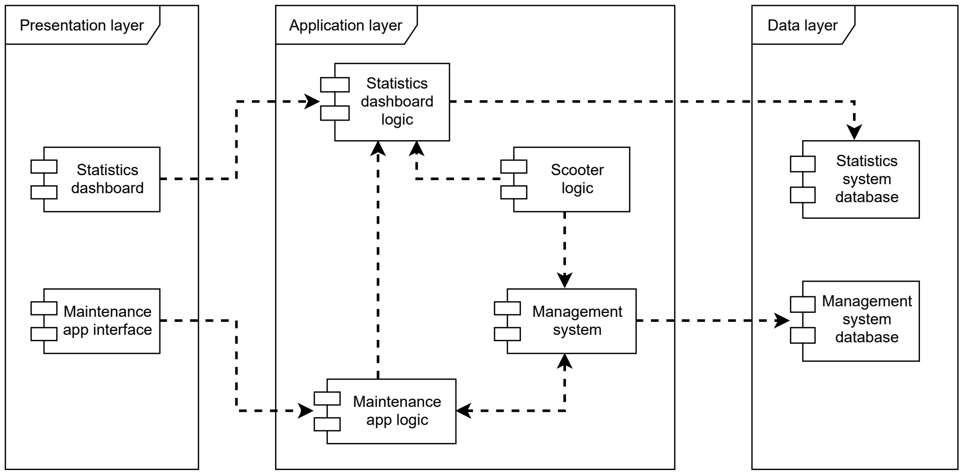


Figure 1: Layered architecture of the system

The tiered architecture that show how the different system are physically split amongst computing nodes is shown in Figure 2 below. Each square containing different components represents a different computing node, and the arrows indicate the interactions with our messaging broker.

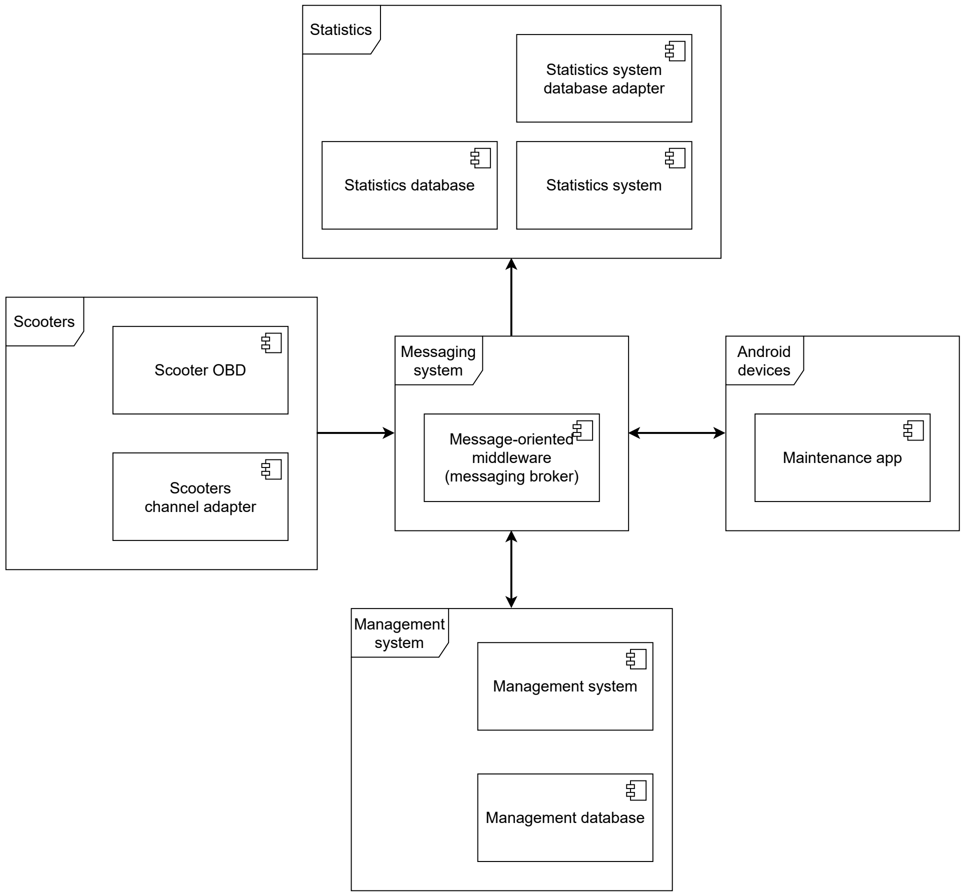


Figure 2: Tiered architecture of the system

4 Architectural patterns

In this section we will discuss the enterprise integration patterns that we used to integrate the various subsystems. Figure 9 in Appendix A shows the complete system diagram with all the patterns that we used. As we discuss the patterns in the following subsections, we will include relevant parts of this diagram.

4.1 Publish-subscribe channel

We start off with the publish-subscribe channel, which we use to broadcast scooter-related messages. These scooter-related messages include the broken scooter messages generated when a scooter breaks down, as well as “functional” scooter messages that are send when a maintenance person marks a scooter as functional. In Figure 3, we can see the publish-subscribe channel in the system’s context.

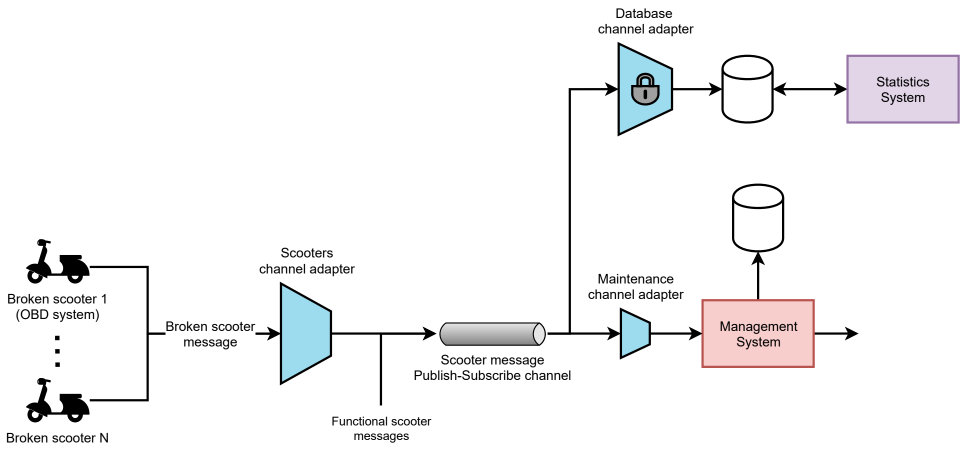

Database channel adapter
Statistics System
Broken scooter 1 (OBD system)
Broken scooter message
Scooters channel adapter
Maintenance channel adapter
Management System
Scooter message Publish-Subscribe channel
Broken scooter N
Functional scooter messages

Figure 3: Publish-subscribe channel within the system’s context

Broken and functional scooter messages are published by this channel to its subscribers. The subscribers include the database channel adapter in front of the statistics system and the maintenance system. Furthermore any future systems that might be interested to know that a scooter has broken down or is repaired could also subscribe to this topic.

4.2 Channel adapter

As seen in Figure 3, we use various channel adapters to connect subsystems to the messaging system — in order to connect broken scooters to the messaging system we used a channel adapter, as the on-board diagnostics (OBD) is obviously not intended to connect directly to a messaging system. Furthermore, the database that the statistics system uses as a source of its data is also connected to the messaging system with a channel adapter. This adapter directly stores the scooter-related messages in the database.

4.3 Content-based router

The zone in which scooters can break down is divided into different regions to accommodate maintenance personnel that operates in a single region. As such, broken scooter messages have to be put in the channel that corresponds to the region in which the scooter breaks down. This is done with a content-based router. It inspects the message contents, specifically the longitude and latitude of the broken scooter, and based on these coordinates it decides in which region the scooter is located. Then it routes the message to the correct channel, as seen in Figure 4.

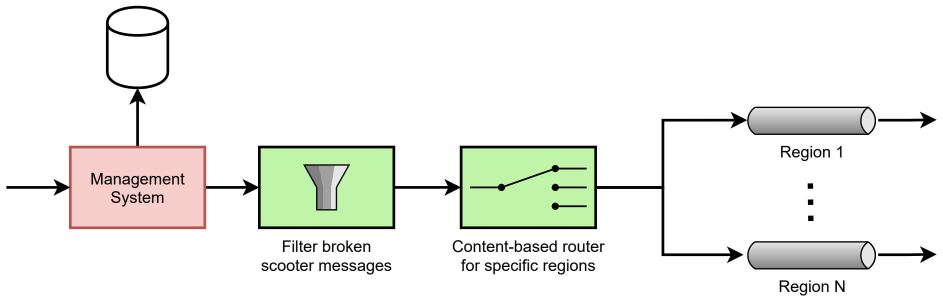

Management System
Filter broken scooter messages
Content-based router for specific regions
Region 1
Region N

Figure 4: Routing broken scooter messages to different regions

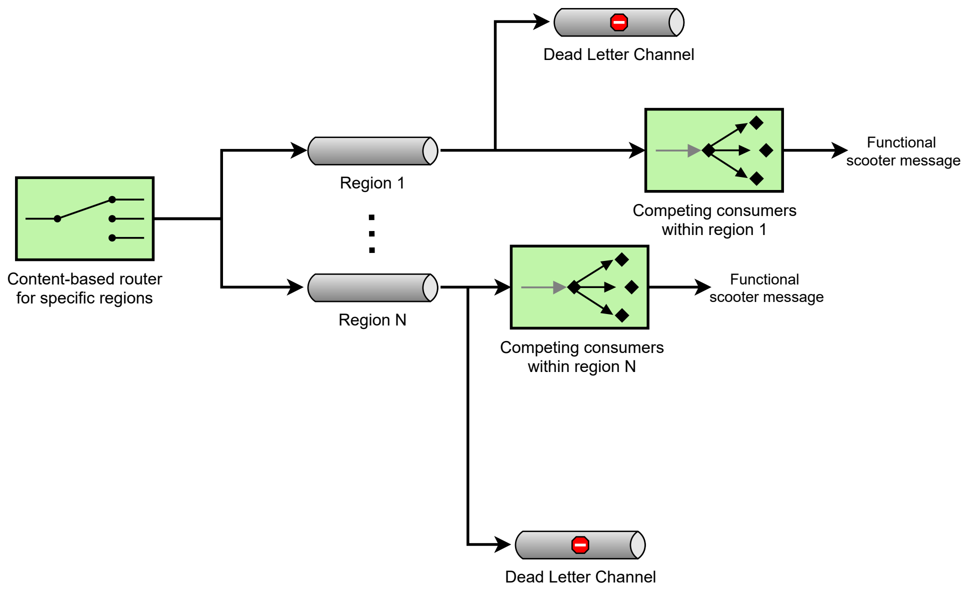
The broken scooter messages are forwarded to the content-based router by the maintenance system.

4.4 Point-to-point channel

The channels for each region discussed in the previous subsection are point-to-point channels. This ensures that only a single receiver (maintenance person) will receive the message.

4.5 Competing consumers

Because the channels for each region are point-to-point channels, when there are multiple consumers per region they compete for each broken scooter message since only one can actually consume it. The consumers in this case are the maintenance people that are currently available (i.e. maintenance personnel that has logged in into the maintenance app). The competing consumers are depicted in Figure 5.



Dead Letter Channel
Region 1
Functional scooter message
Competing consumers within region 1
Content-based router for specific regions
Region N
Functional scooter message
Competing consumers within region N
Dead Letter Channel

Figure 5: Competing consumers

One important detail to mention is that the specific messaging broker that we chose divides the messages between competing consumers in a round-robin fashion, which is ideal for our case, as a single maintenance person should not be overwhelmed with to many broken scooter messages.

4.6 Dead letter channel

When a notification from one of the scooters does not get delivered to a receiver we cannot let it timeout and disappear, as the scooter is apparently broken and might not send another notification (for example because of a dead battery). It can also not stay on the channel. Hence we opted to use Dead letter channels for any messages that could not be delivered, in order to not lose any messages.

4.7 Durable subscriber

The systems that consume messages from the publish-subscribe channel are implemented as durable subscribers. The reason for this is fairly simple: all messages published on the publish-subscribe channels might be of potential interested for all subscribed parties. Even if said parties are not online at the time the message is published. By using durable subscribers the messages will be consumed by the party, even if the party was not online at the time the message was published.

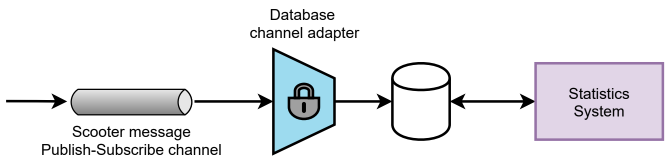


Figure 6: Durable subscriber

4.8 Message filter

Because the maintenance personnel (and by extension the maintenance app) is only interested in the broken scooter notifications, we added a message filter exclude functional scooter messages that the app itself produces. As seen in Figure 4, this happens before the messages are routed to the queues that represent the different regions.

4.9 Maintenance app

This subsection discusses the patterns that were employed within the maintenance app, as seen in Figure 7. The maintenance app allows maintenance personnel to receive broken scooters from the maintenance system, and mark them as functional after they are repaired. This leads to a ”Functional scooter” message being send to the publish-subscribe channel of the maintenance system.

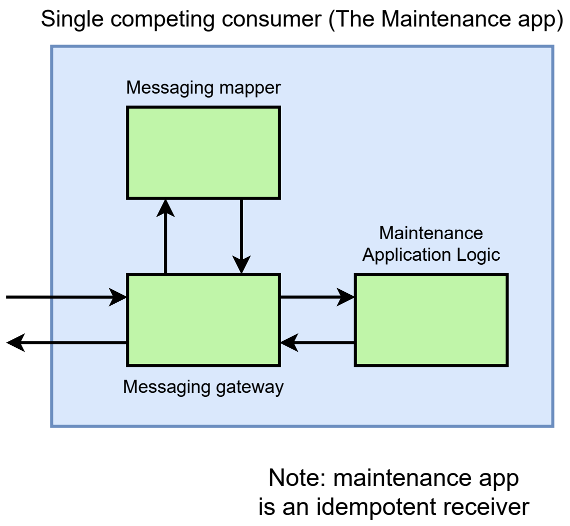

Single competing consumer (The Maintenance app)
Messaging mapper
Maintenance Application Logic
Messaging gateway
Note: maintenance app is an idempotent receiver

Figure 7: Patterns employed within the maintenance app

4.9.1 Messaging gateway

In order to encapsulate access to the messaging system from the rest of the application we used the messaging gateway pattern. We created a class that sits between the messaging system and the application logic. This class wraps all the messaging-related methods and handles both the consumption of broken scooter messages as well as the production of functional scooter messages after the maintenance person repaired a scooter and marked it as functional from within the app.

4.9.2 Messaging mapper

The structure of the domain objects in the mobile application differs from the structure of the received broken scooter messages. To remedy this, we used the messaging mapper pattern to create a separate class that is responsible for mapping domain objects to a messaging system compatible structure and vice versa. Since this class references the domain objects (Scooter objects) and messaging layer itself, neither layer is aware of the other or the messaging mapper.

4.9.3 Idempotent receiver

To account for potential duplicate broken scooter messages we made the messaging application an idempotent receiver. When a broken scooter message is received we check if the broken scooter message is already contained in the list of received broken scooters messages. If so, the broken scooter message is discarded. Because of this, broken scooters are only added once to the maintenance app after which duplicate messages are discarded — making the addition of broken scooters idempotent.

4.10 Additional patterns

Besides the implemented patterns, we also thought of additional patterns that cover more of the fault tolerance and management aspects. Because of time constraints however these are not in the final product. For completeness sake we opted to include the complete diagram in the report in Appendix B, Figure 10. In this section we will briefly go over the additional patterns.

4.10.1 Invalid message channels

Invalid message channels, are included in the complete diagram. These channel cover cases where the management system might receive data that is invalid, e.g. when a message is malformed and can not be interpreted by the recipient.

4.10.2 Test message

The test message pattern is used for testing the system. We have the test message generator for creating possible scooter messages of type broken and fixed, and a test message injector for adding these to the system. Finally we have a test message verifier at the content based router so we can verify the messages are correctly handled and passed along.

5 Technology architecture

Figure 8 shows an overview of technologies used in each separate application. In the section below, we explain where we used each technology and why we chose each particular technology.

5.1 General

These are the technologies used to used to implement the applications.

- **Docker** is used to containerize every application to simulate a distributed environment. An exception here is the maintenance app as it already runs on an android emulator.
- **JMS** we use the JMS API to interact with our chosen JMS provider. Besides it being compulsory it also offers many advantages to distributed messaging. All applications are connected and thus use JMS or an adapter with JMS to communicate.
- **Maven/Gradle** all separate systems have their dependencies handled by Maven, except for the app which uses Gradle.

5.2 Messaging system

For the messaging system we use JMS with Apache ActiveMQ as JMS provider.

- **Apache ActiveMQ (Artemis)**, as the JMS provider and to create all channels. ActiveMQ is used also to replace or hide JNDI. We have chosen ActiveMQ as its one of the more popular JMS providers that also automatically supports dead letter and invalid message channels. We specifically use the Artemis version, as this is a newer version with also more docker support.

5.3 Management system and database

The management system is the system that handles routing and filtering scooter messages.

- **MongoDB**, we choose MongoDB because it offers a simple and dynamic database which is handy. We added MongoDB to keep track of broken scooters, and to potentially use it as a maintenance personnel database in case maintenance personnel is statically assigned to regions.

5.4 Statistics system and database

The statboard and its database are responsible for showing statistics about the scooter problems and repairs. It uses the following technologies:

- **Apache Cassandra**, for the statboard database. This database is chosen because this database is mostly used for statistics based on timed data.
- **SpringBoot**, we use SpringBoot in the Statboard application to bootstrap many of the functionalities of the application. So that we do not start from scratch. Springboot provides an easy way to host a web application with a simple frontend.

5.5 Maintenance app

The maintenance app is the app used by the maintenance personnel in the field to check for new incoming scooter problems and mark scooters as functional again after they are repaired.

- **Android** is an obvious choice because it's one of the most popular mobile platforms. Moreover, maintenance personnel used to work a lot on the road so creating an android app for the maintenance personnel will be more useful for them to track messages continuously.
- **SwiftMQ AMQP 1.0 Java Client** to communicate with our messaging broken (message oriented middleware) from the maintenance app we had to use a different client than for the rest of the system as Android applications do not use the regular Java Virtual Machine (JVM), but a stripped-down, altered version namely the Dalvik Virtual Machine (DVM). As such the `javax` library and other common libaries are not available on Android, while they are required for JMS and the Apache ActiveMQ client to operate.
  We opted to use a different protocol, namely the Advanced Message Queuing Protocol (AMQP) which is supported by Apache ActiveMQ Artemis as it is an open standard. To use this protocol we used the SwiftMQ AMQP 1.0 Java Client.

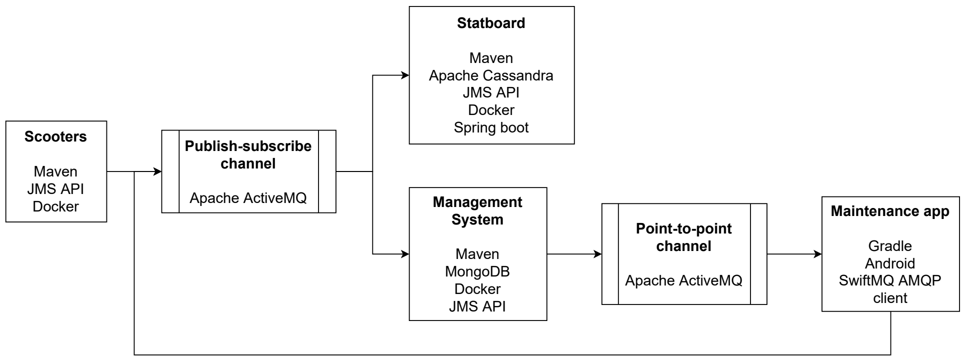


Figure 8: Technology architecture

6 Pattern - implementation mapping

In this section we will discuss how the aforementioned patterns are implemented in our final system. Where applicable, technologies will be mentioned which helped us to realise our desired architecture.

6.1 Publish-subscribe channel

The publish-subscribe channel is managed by ActiveMQ. Whereas, channel and topic will be created dynamically. In the scooter (OBD) adapter code working as a producer by using JMS connects to the ActiveMQ container. In the adapters for the statboard and the management system are both subscribed consumers of the publish subscribe channel and it's topic, which is also done with the JMS API.

6.2 Channel adapter

The channel adapter pattern is implemented in each application with the full integrated system. It has been implemented in code as an adapter class that handles the incoming JSON data from the channels, and is key for the applications to becomes aware of the incoming data and the other application. For the scooter (OBD) application it give the application the possibility to push and share its data to the pub sub channel. For the statboard it is a class that offers a way to receive new data by consuming from the channel and directly inserting the data into the database. Finally for the management system, it offers a way to receive dynamically repair jobs for the scooters, by consuming the broken scooter messages from the scooters (publish-subscribe channel).

6.3 Content-based router

The content-based router is implemented in the management system, here the logic is placed where it routes the message to certain channels based on the longitude and latitude attributes of the message. The `Router` class in `eai_rug/management-system/src/main/java/Network/` is responsible for determining to which region channel the incoming broken scooter message gets routed.

6.4 Point-to-point channel

The point-to-point channel is very similar to the publish-subscribe channel, its managed by ActiveMQ and dynamically created (of course without topic). The point-to-point channels are used in the management system to route messages to and in the maintenance app to take message from the region-specific channel.

6.5 Competing consumers

The computing consumers pattern is implemented by simply having multiple consumers (because there will be multiple maintenance apps running) attached to each point-to-point channel that represents a region, this has the effect that when a consumer consumes a message it is gone from the channel and not available to any other consumer. The consumer implementation in the maintenance app is contained in the `MessageConsumer` class located at `eai_rug/scooter-maintenance-app/app/src/main/java/com/eai/scootermaintenanceapp/messaging/`.

6.6 Dead letter channel

The dead letter channel is implemented in the ActiveMQ container. ActiveMQ can automatically create a dead letter channel for all queues that are created, this is a setting which can be adapted in the `broker.xml` file located in `eai_rug/artemis/`.

6.7 Durable subscriber

The durable subscriber pattern is implemented in both the management system and the statboard as these are the only systems that are subscribed to the publish-subscribe channel. To be more specific, whenever a durable subscriber is required we used the JMS API to create a durable subscriber as follows (where `session` is a JMS session object). For example in `Consumer` located at `eai_rug/Statboard_adapter/src/main/java/`:

```
session.createDurableSubscriber(destination, "statboardAdapter");
```

6.8 Message filter

The message filter patterns takes care of filtering invalid messages. There is only one system that implements this pattern as there is only one system that forwards messages, the management system. The management system makes sure to only forward messages to the maintenance app if the scooters are broken. If the scooter has been repaired it is not forwarded.

6.9 Messaging gateway

As discussed in Section 4, the messaging gateway pattern is implemented in the maintenance app to encapsulate the messaging logic from the rest of the system. Specifically, the pattern is implemented in the `MessagingGateway` class located in `eai_rug/scooter-maintenance-app/app/src/main/java/com/eai/scootermaintenanceapp/messaging/`. It delegates the actual consumption and production of messages to two other classes, `ScooterProducer` and `ScooterConsumer`, located in the same package.

6.10 Messaging mapper

Similar to the messaging gateway, the messaging mapper is also implemented in the maintenance app as a separate class called. This class is named `MessagingMapper` and is located in `eai_rug/scooter-maintenance-app/app/src/main/java/com/eai/scootermaintenanceapp/messaging/`. It is a static class that has methods to map the scooter domain objects to the JSON format used in the messaging system and vice versa.

6.11 Idempotent receiver

To make the maintenance app an idempotent receiver of the broken scooter messages, we added a simple check if the view model (used to maintain state across the scooter list and the scooter map) already contains the received scooter. This check is added to the `addScooter(Scooter scooter)` method located in the `ScooterViewModel`, which in term is located in package `eai_rug/scooter-maintenance-app/app/src/main/java/com/eai/scootermaintenanceapp/ui/maintenance/`.

7 Work distribution

The work was distributed mostly evenly distributed over the different applications. In rough lines the work was distributed in the following way:

- Wouter: Created and implemented most of the broken scooter generator and maintenance system, report and dockerization.
- Robert: Complete maintenance app, parts of the maintenance system and broken scooter generator, integrating all systems together, demo and work on the report.
- Luc: Statboard (integration and database), adapter, dockerization, presentation, report.
- Kaavyaa: Statboard, database, report and presentation.
- Ruben: Integration, statboard, messaging system and report.

For exact details of who did what see the GitHub history at `https://github.com/RUKip/eai\_rug` (code changes) and version history on the second page of this report (report changes).

A System design diagram

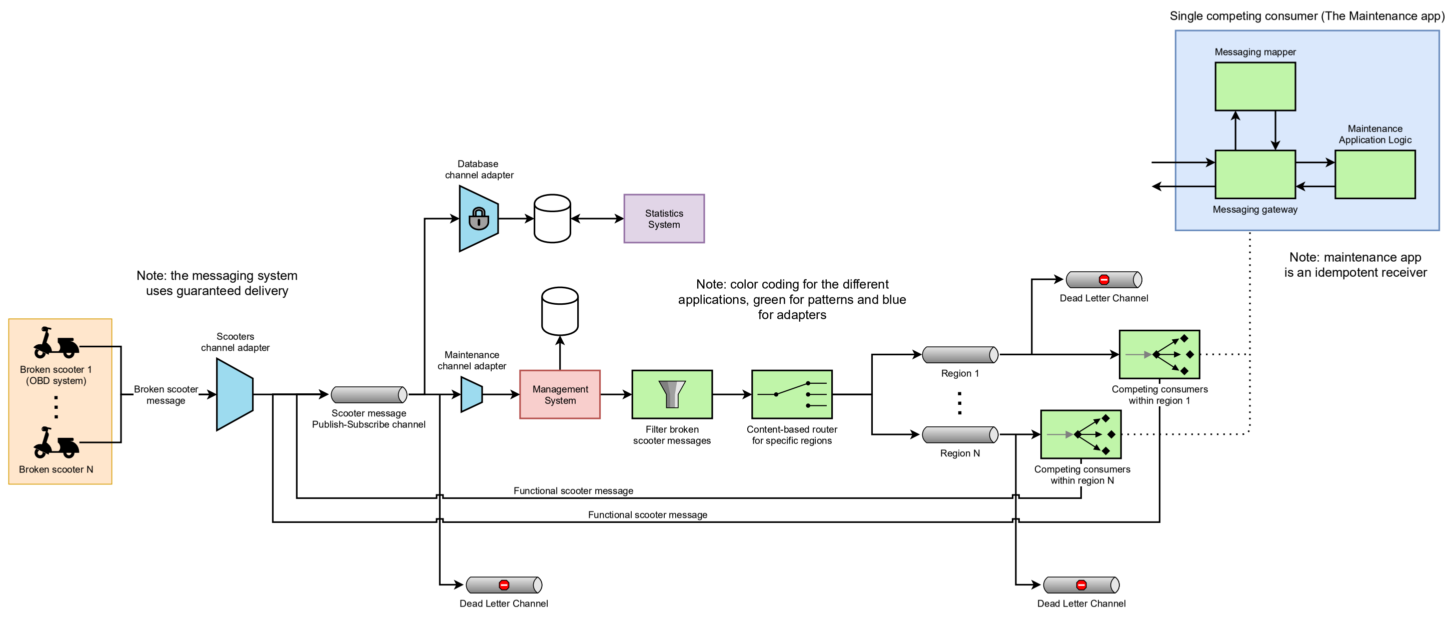

Single competing consumer (The Maintenance app)
Messaging mapper
Maintenance Application Logic
Messaging gateway
Note: maintenance app is an idempotent receiver
Database channel adapter
Statistics System
Note: the messaging system uses guaranteed delivery
Note: color coding for the different applications, green for patterns and blue for adapters
Dead Letter Channel
Broken scooter 1 (OBD system)
Broken scooter N
Broken scooter message
Scooters channel adapter
Scooter message Publish-Subscribe channel
Maintenance channel adapter
Management System
Filter broken scooter messages
Content-based router for specific regions
Region 1
Region N
Competing consumers within region 1
Competing consumers within region N
Functional scooter message
Functional scooter message
Dead Letter Channel
Dead Letter Channel

Figure 9: System design

B Full system design diagram

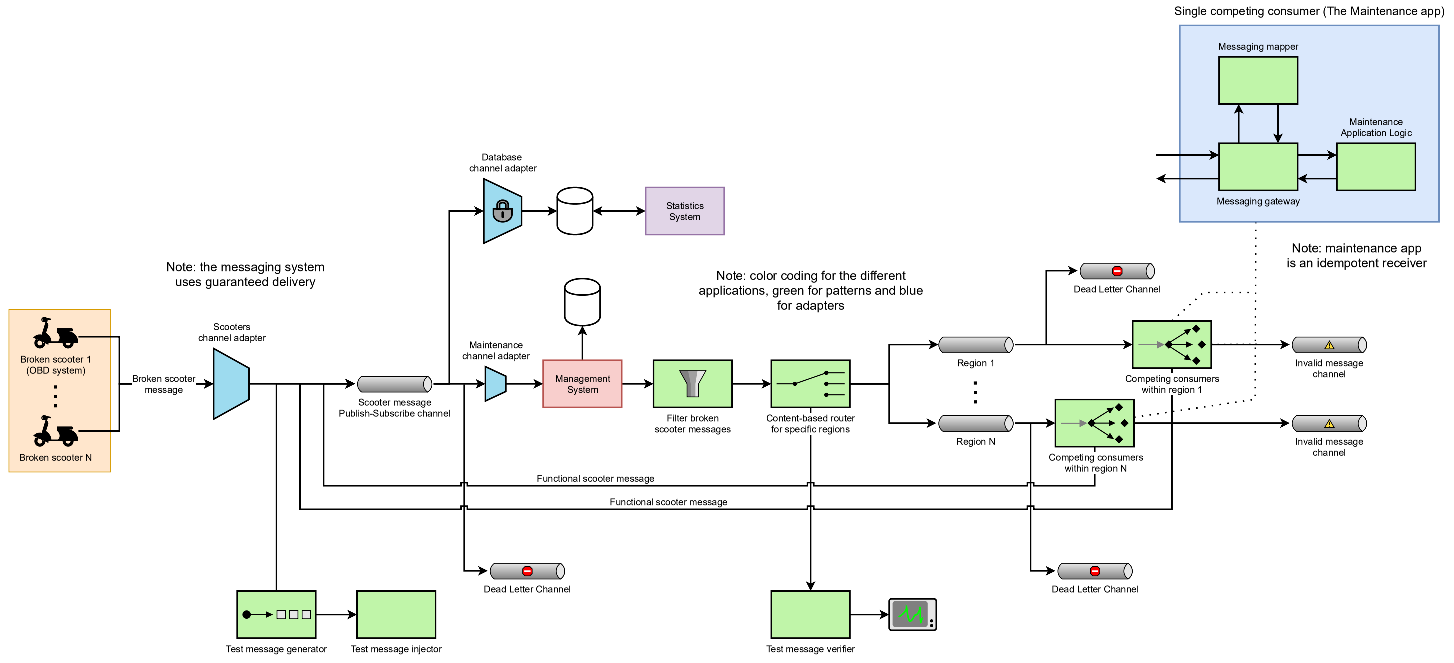

Single competing consumer (The Maintenance app)
Messaging mapper
Maintenance Application Logic
Messaging gateway
Note: maintenance app is an idempotent receiver
Database channel adapter
Statistics System
Note: the messaging system uses guaranteed delivery
Note: color coding for the different applications, green for patterns and blue for adapters
Broken scooter 1 (OBD system)
Broken scooter N
Broken scooter message
Scooters channel adapter
Scooter message Publish-Subscribe channel
Maintenance channel adapter
Management System
Filter broken scooter messages
Content-based router for specific regions
Region 1
Region N
Dead Letter Channel
Competing consumers within region 1
Competing consumers within region N
Invalid message channel
Invalid message channel
Functional scooter message
Functional scooter message
Dead Letter Channel
Dead Letter Channel
Test message generator
Test message injector
Test message verifier

Figure 10: Full system design