Software Architecture Document

Plan and Go:

Visual Public Transportation Analysis Tool

Group 8

DIT355 Distributed Systems Mini Project

Architectural Forces

Extendability - The system should be able to easily add to its functionality.
Compatibility - The system shall function on a variety of devices and operating systems.
Interoperability . The system shall function with a variety of tools and technologies.
Maintainability - The system should be able to be kept up to date with new versions of technology. It also needs to be able to be repaired when necessary.
Usability - The system should be easy to navigate with little instruction needed.
Availability - The system should have very little downtime as it affects the reliability of the data.
Reliability - The system needs to be able to display data when needed and should be trusted to perform consistently.

Architectural Drivers

The forces that act as the main architectural drivers in our system are **extendability, availability** and **interoperability.**

Having an **extendable** system is a key requirement of our product, as our Stakeholder is very prone to making new requirements. We want our system to be flexible enough to adapt to different needs and priorities. Also, we are using a modular design pattern within our components and the separation of concerns pattern between our components through the Publish Subscribe architectural style. This allows us to easily extend functionality and adapt the software architecture.

As our system needs to be able to handle many events and messages, it needs to be **available.** The system needs to be up and running at all times as the quality of the analysis depends on accurate data. If some functionalities are not working, the overall purpose of informing the city planners about demands on the transport network will not be achieved.

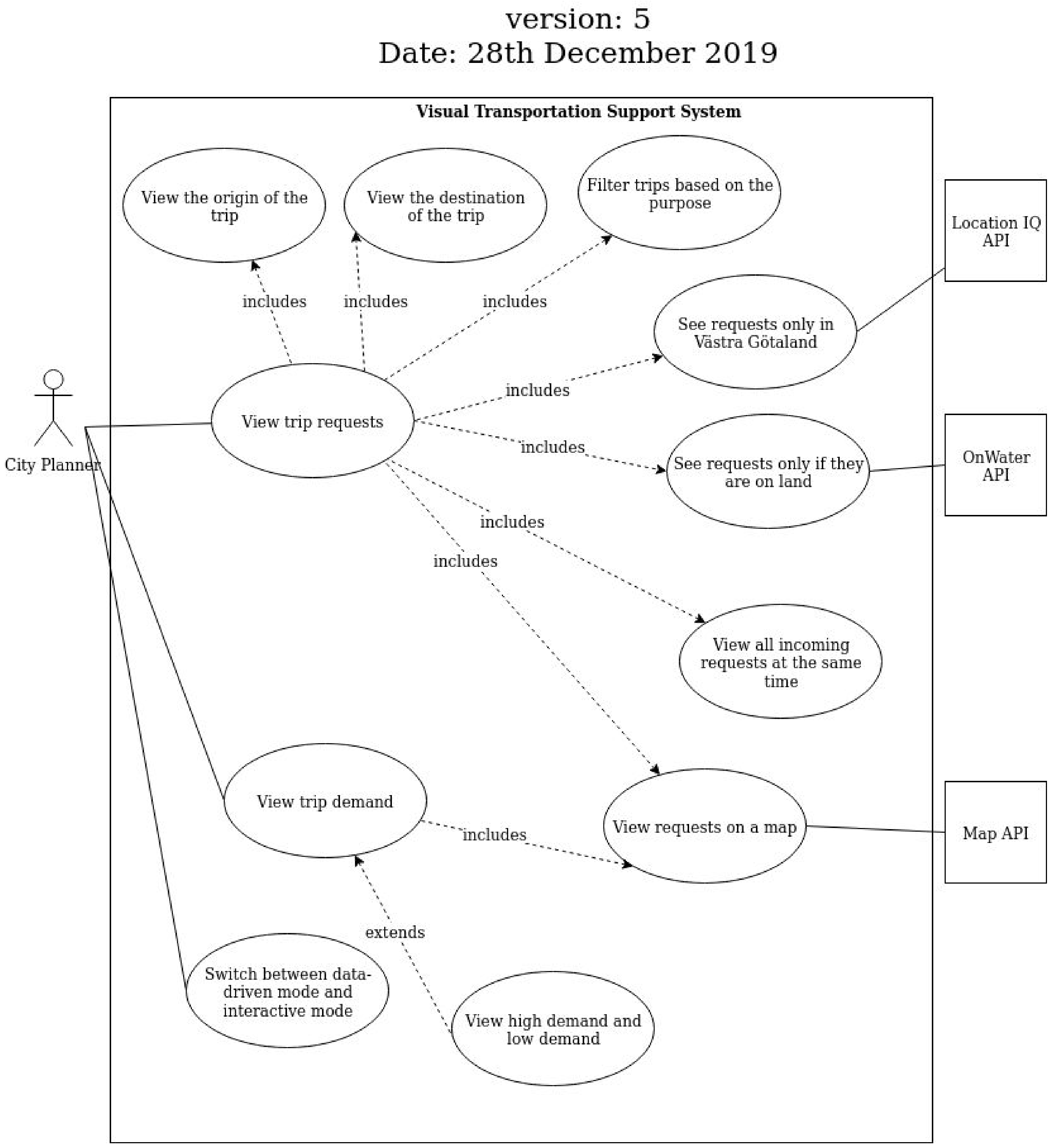
The system uses other systems to function. Therefore, it is essential to ensure its **interoperability** by creating interfaces that are compatible with these external systems.

Use Cases

Use Cases

1. As a City Planner, I want to see the trips of travellers, so that I can improve the transportation network.
2. As a City Planner I want to see the demand of travel requests, so that I can adjust the static infrastructure and decide on public transport to meet demand.
3. As a City Planner, I want to see the places that many travellers wish to travel to displayed on a map, so that I can identify where public transport might be in high demand.
4. As a City Planner, I want to be able to see the purpose of the different trip requests, so that I can
5. As a City Planner, I want to see all traveller requests at the same time on the map, so that I can quickly see all data and improve the transportation design.
6. As a City Planner, I want to switch between data-driven mode and interactive mode, so that I can analyze the incoming data in different ways.

Use Case Diagram

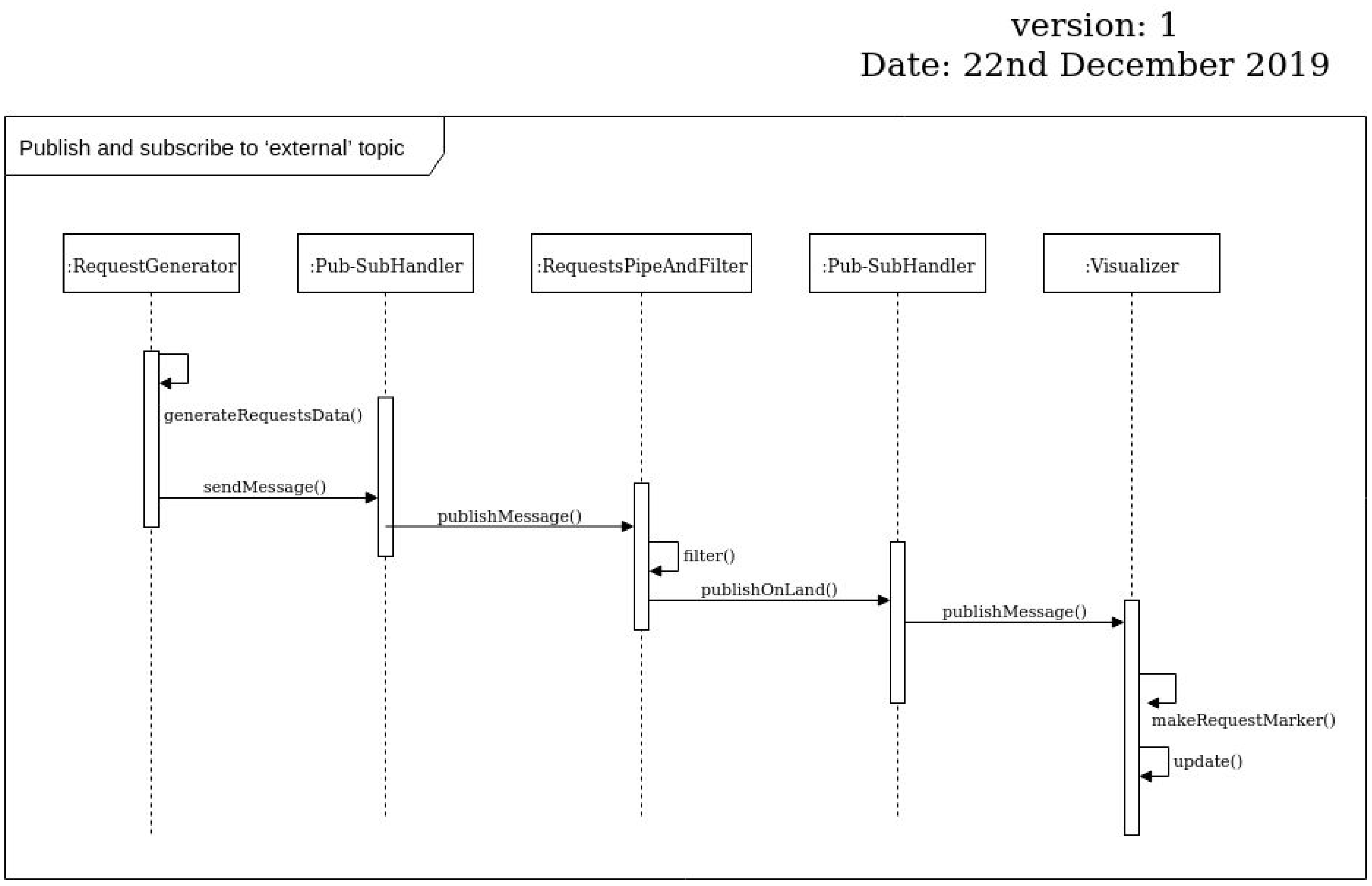

version: 5
Date: 28th December 2019
Visual Transportation Support System
View the origin of the trip
View the destination of the trip
Filter trips based on the purpose
Location IQ API
See requests only in Västra Götaland
includes
includes
includes
includes
City Planner
View trip requests
includes
See requests only if they are on land
OnWater API
includes
includes
View all incoming requests at the same time
View trip demand
includes
View requests on a map
Map API
extends
Switch between data-driven mode and interactive mode
View high demand and low demand

Description of diagram

By using Visual Transportation Support System, the City Planners can view all coming trip requests on a map and differentiate between origin and destination. Since the requests are randomly generated, they will be filtered in two ways: the locations will be in Västra Götaland and on land. In Data Driven mode, users can see areas of high and low demand. In Interactive mode, users can filter on a purpose to view specific requests. The City Planners can also switch between two modes on the map.

Sequence Diagrams

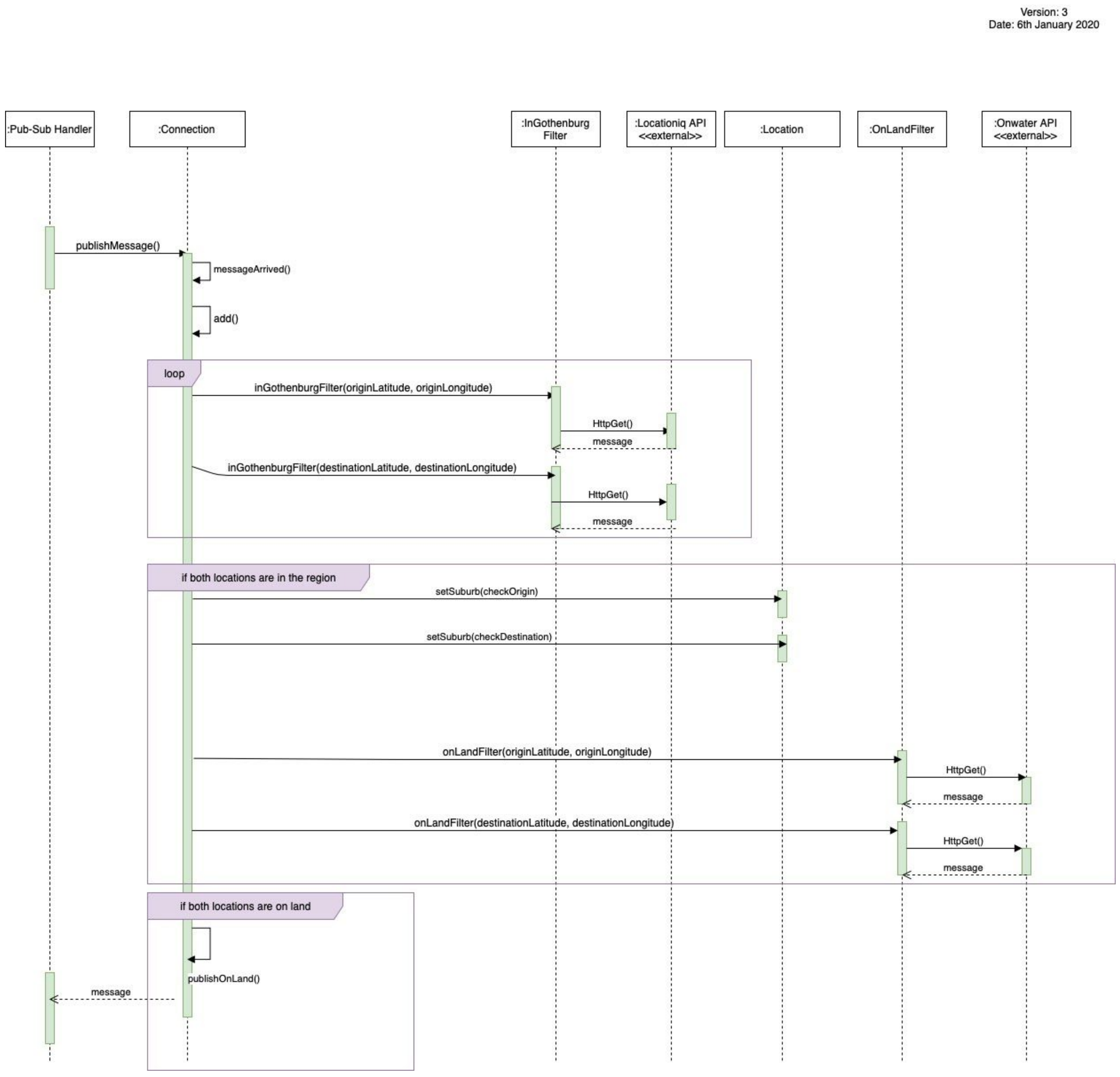
Sequence Diagram - Publish and subscribe to a message under the topic ‘external’



Description of diagram

The RequestGenerator creates a new random request. The message is then published to the Publish-Subscribe Handler. This event triggers the Publish-Subscribe Handler to send the message to all components subscribed to the topic ‘external’. As a result, the RequestsPipeAndFilter receives the message. After the message has been filtered, it is then published to the Publish-Subscribe Handler under the topic ‘requests’. Following this, the Visualizer, which is subscribed to the topic creates a marker, which updates the user interface.

Sequence Diagram - Pipe and Filter of randomly generated coordinates


Version: 3
Date: 6th January 2020
:Pub-Sub Handler
:Connection
:InGothenburg Filter
:Locationiq API <<external>>
:Location
:OnLandFilter
:Onwater API <<external>>
publishMessage()
messageArrived()
add()
loop
inGothenburgFilter(originLatitude, originLongitude)
HttpGet()
message
inGothenburgFilter(destinationLatitude, destinationLongitude)
HttpGet()
message
if both locations are in the region
setSuburb(checkOrigin)
setSuburb(checkDestination)
onLandFilter(originLatitude, originLongitude)
HttpGet()
message
onLandFilter(destinationLatitude, destinationLongitude)
HttpGet()
message
if both locations are on land
publishOnLand()
message

Description of diagram

The Publish-Subscribe Handler publishes the message under the topic ‘external’. The RequestsPipeAndFilter then receives this through the Connection. The Connection is the pipe for the InGothenburgFilter and OnLandFilter.

When a new message is received via the broker, it adds the message to a queue via the add() method. Then, each message from the queue goes through the first filter; InGothenburgFilter. Here, each pair of coordinates is filtered using the Location IQ API, to check if it is in Västra Götaland region.

If they are, then the suburbs are added to the message if they exist, or it is set to “uncategorised”. Then the coordinates are checked to make sure they are on land, using the OnWater API.

Any messages that pass through both of these filters is published to the Publish-Subscribe Handler via the topic ‘requests’.

Components and subsystems

Components and subsystems

Broker
Uses the MQTT protocol via Mosquitto to receive and broadcast messages.

RequestGenerator
This component handles the generation of trip requests, and publishes these messages under the topic ‘external’. This component creates a set of two coordinates, randomly generated in approximately the Västra Götaland region. It also randomly generates a trip purpose.
It is made up of the following subcomponents:

> **Publisher**
> This component publishes the randomly generated trip requests to the broker under the topic ‘external’.
> **RequestGenerator**
> This creates the random request and builds the message as a JSON object. This will be published to the broker under the ‘external’ topic.

RequestPipeAndFilter
This component uses the Pipe and Filter architecture style to handle incoming messages from the RequestGeneratorComponent. It is made up of the following components:

> **RequestsPipe**
> This component handles the communication between the RequestGenerator, the RequestsPipeAndFilter and the Visualizer. It subscribes to the topic ‘external’, receiving requests from the RequestGenerator. First, it passes the message to the InGothenburgFilter then, to the OnLandFilter. Lastly, it publishes these messages under the topic ‘requests’.
> **InGothenburgFilter**
> This component filters the messages received, only passing through messages where both coordinates are within the Västra Götaland region and it adds the the field “suburb” if available from the API or sets it to “uncategorised” otherwise. This component uses an API called LocationIQ to handle this. If both coordinates are within the given area, the filter will return the message back to the pipe, adding the suburb of the coordinate to the message.
> **OnLandFilter**
> This component is responsible for ensuring that both the origin and destination coordinates are on land. It filters the coordinates using the onWater API to check if the coordinates are in water. It then returns messages where both sets of coordinates are confirmed to be on land to the Connector.

Visualizer
This component is where city planners can look to view the demands on a map.
It is made up of the following subcomponents:

> **Subscriber**
> This is the subscriber component. It connects to the broker via MQTT and subscribes to the ‘requests’ topic.

User Interface

Handles user interactions and visualises data to the user. It consists of the following subcomponents:

MapPage

Interfaces the Google Maps API to display a map. This is where the city planner will be able to view the requests and demands.

InteractiveMode

This component handles the messages received from the ‘requests’ subscription. It extracts the data needed to display information on the map.

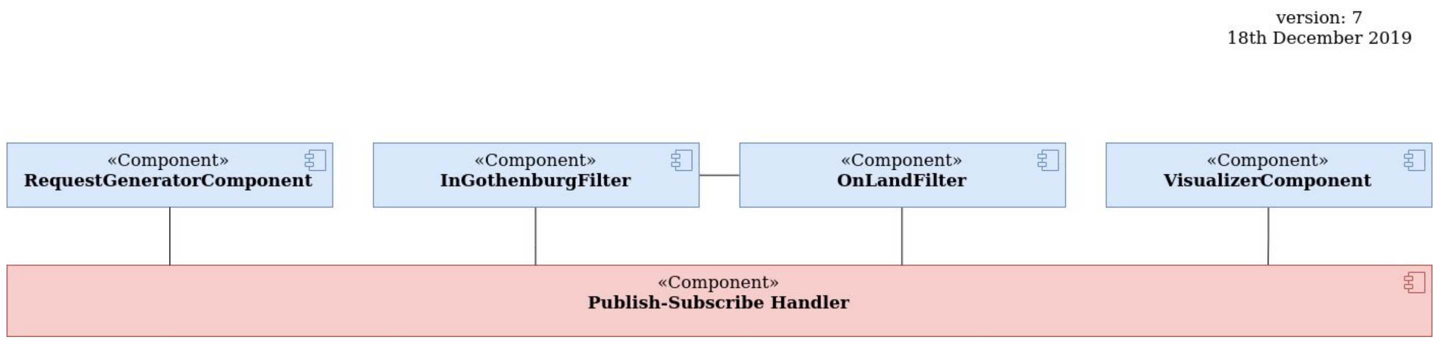
DataDrivenMode

This component handles the messages received from the ‘requests’ subscription. It extracts the data needed to display information on the map. Based on the requests being received, it automatically updates the **UserInterface** with data showing the number of requests in different areas in Gothenburg.

Dashboard

Displays information about the requests, other than the map. This component also handles the user input to enable the user to switch modes.

Component Diagram - Conceptual View



Description of diagram

As we are using a Publish Subscribe style to facilitate communication, our components share data using a **Publish-Subscribe Handler**. Above this, we have all our other components.

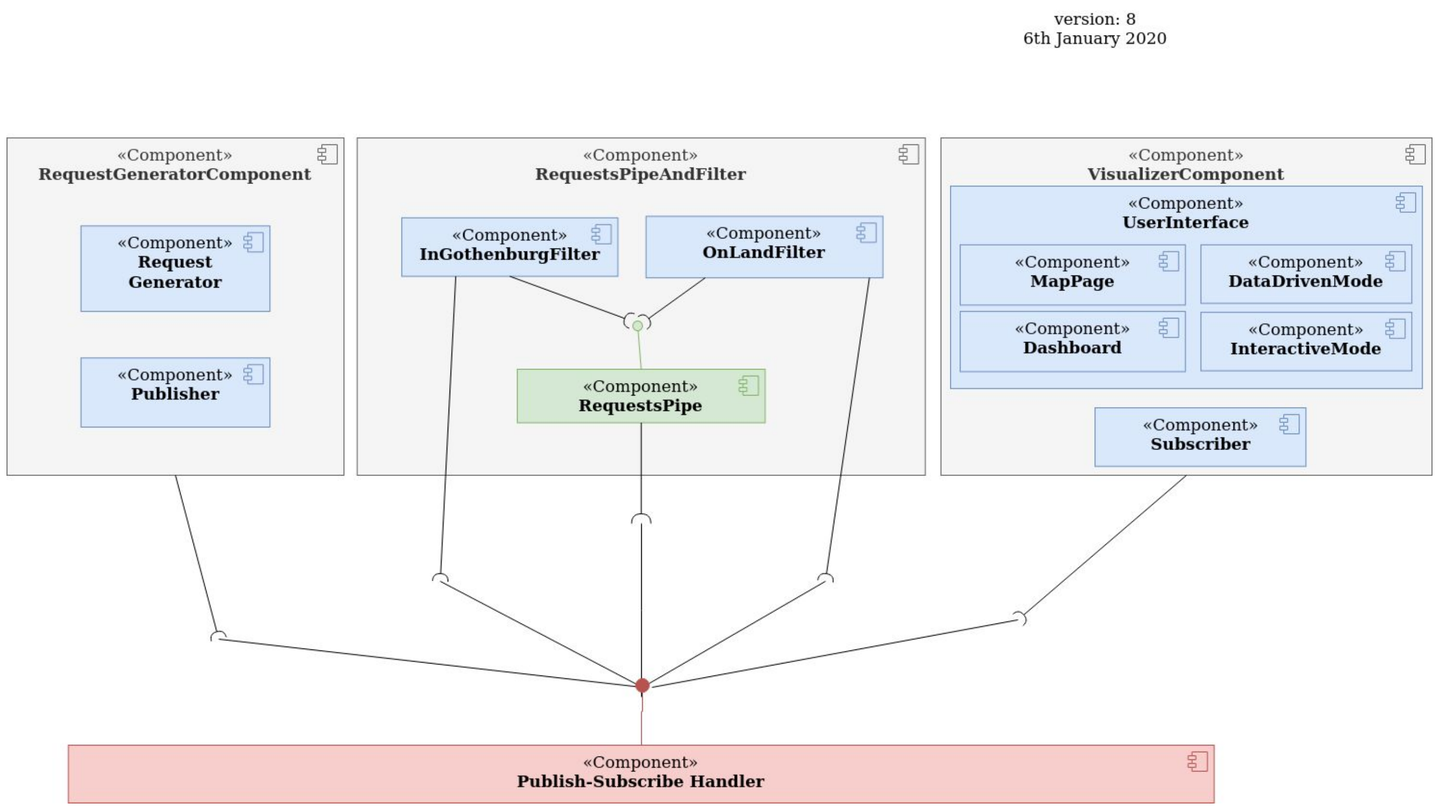
The first component in this layer is the **RequestGeneratorComponent,** which is responsible for creating the mock data and publishing this to the broker under the topic ‘external’. Within this component there are two subcomponents. First is the is the **RequestGenerator**, which generates a new request, based on randomised data. The other subcomponent is the **Publisher**, which publishes the messages created by the previous component to the Publish-Subscribe Handler.

The next component is the first filter called **InGothenburgFilter**. This filter checks the messages received from the RequestGeneratorComponent and checks if the coordinates are within the Västra Götaland region. Only the messages where both the origin and destination coordinates are within the region are forwarded to the next component after adding the suburb.

Next, is the second filter called **OnLandFilter**. This filter checks if the coordinates received are in water or on land. It forwards those that are confirmed to be on land to the Publish-Subscribe Handler which publishes these messages under the topic ‘requests’.

The final component is the **Visualizer** which handles the user input via the user interface. This component subscribes to the topic ’requests’.

Component Diagram - Implementation View


version: 8
6th January 2020
«Component» RequestGeneratorComponent
«Component» Request Generator
«Component» Publisher
«Component» RequestsPipeAndFilter
«Component» InGothenburgFilter
«Component» OnLandFilter
«Component» RequestsPipe
«Component» VisualizerComponent
«Component» UserInterface
«Component» MapPage
«Component» DataDrivenMode
«Component» Dashboard
«Component» InteractiveMode
«Component» Subscriber
«Component» Publish-Subscribe Handler

Description of diagram

As we are using a Publish Subscribe style to facilitate communication, our components share data using a **Publish-Subscribe Handler**. Above this, we have all our other components.

The first component in this layer is the **RequestGeneratorComponent,** which is responsible for creating the mock data and publishing this to the broker under the topic 'external'. Within this component there are two subcomponents. First is the is the **RequestGenerator**, which generates a new request, based on randomised data. The other subcomponent is the **Publisher**, which publishes the messages created by the previous component to the Publish-Subscribe Handler. The Publisher is the interface between the Publish-Subscribe Handler and this component.

The next component is the **RequestsPipeAndFilter** which filters the messages received from the RequestGeneratorComponent from the 'external' topic. Within this, there subcomponents consisting of two filters and a pipe.

The pipe is called **Connector**. This subcomponent handles communication between the RequestsPipeAndFilter and the Publish-Subscribe Handler. It subscribes to the topic 'external', and sends the message to the first filter. If the message passes, it then sends it to the second filter. It then publishes any messages that go through both filters to the Publish-Subscribe Handler under the topic 'requests'.

The first filter is the **InGothenburgFilter**. This filter checks the messages received from the RequestGeneratorComponent and checks if the coordinates are within the Västra Götaland region. Only the messages where both the origin and destination coordinates are within the region are forwarded to the next filter after adding the suburb string.

The second filter is called **OnLandFilter**. This filter checks if the coordinates received are in water or on land. The Connector then forwards those that are confirmed to be on land to the Publish-Subscribe Handler.

The final component is the **VisualizerComponent** and within, there are several subcomponents. The UserInterface component contains most of the functionality. **InteractiveMode** handles the features that allow the user to interact with the map. In this mode, the user can also filter on different purposes to analyse the type of request. In contrast, the **DataDrivenMode** component does not allow the user to interact with the map. Instead, it updates the view of the UserInterface as each request is received. In this mode, the map updates by displaying each request and grouping it based on area. Both of these components receive their data via the **Subscriber** component, which is subscribed to the topic 'requests'. The user interaction is facilitated by the **UserInterface** component. It implements the user's interactions and displays any updates given by the InteractionMode and DataDrivenMode components. The **MapPage** component shows incoming messages on a map and depending on the mode, has further functionality.

How Architecture Maps to Implementation

RequestsGenerator
Each subcomponent maps directly to the classes ‘Publisher’ (Publisher.java) and ‘RequestGenerator’ (RequestGenerator.java).

RequestsPipeAndFilter
The component ‘inGothenburgFilter’ maps to the class inGothenburgFilter.java and ‘OnLandFilter’ to the class OnLandFilter.java. The ‘requestsPipe’ component is handled by the class Connection.java.

Visualizer
The component ‘MapPage’ maps to the view MapPage.vue and ‘Dashboard’ to the component Dashboard.vue. Each mode is handled by the DataDrivenMode.vue and InteractiveMode.vue files. The ‘Subscriber’ is also handled by the component MapPage.vue.

Architectural Style

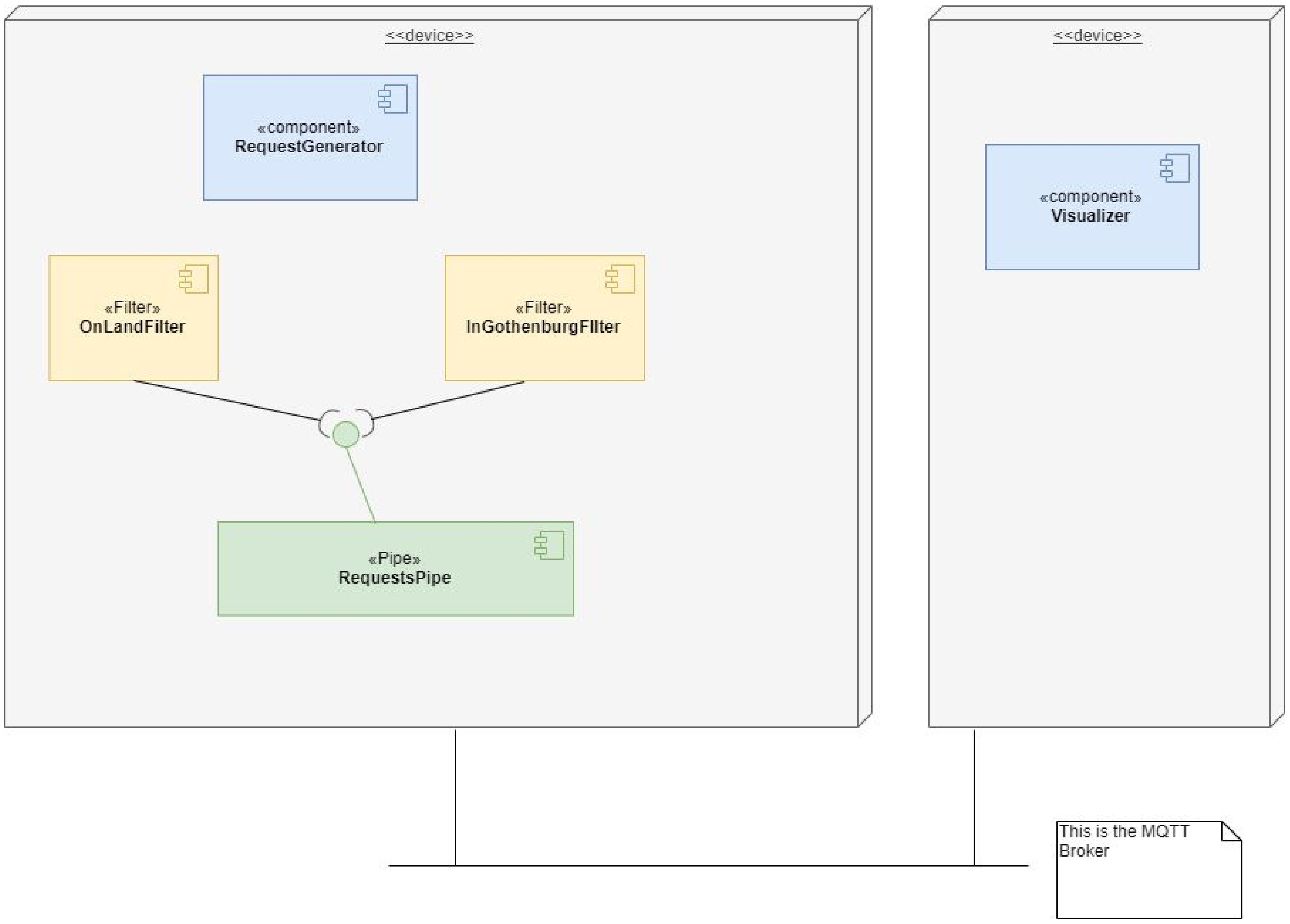
Justification of Architectural Styles

The overall system will use a **Publish Subscribe** architectural style. We chose this style because the data we will be transmitting will be of the same type: JSON. Also, our system is based on events. For example, when the user is in data driven mode, the map will automatically change and update based on the messages received. Once a request has been generated, the publisher component will send a message to all subscribing components. Currently, the only subscriber is the visualizer but this style will allow us to add more subscribers in the future.

Within the RequestsPipeAndFilter component we will use the **Pipe and Filter** architectural style to concurrently process the randomly generated trip requests from the RequestGenerator. This is essential, as we want our visualiser to show useful data. As we scale the number of requests, we can use multiple streams to publish the data through parallel processing. In our implementation however, we will filter messages through the pipe one at a time.

Deployment

Deployment Diagram



Description of diagram

The system will be deployed onto two devices.

The first device will host the RequestGenerator and the two filters. **RequestGenerator** has the publish function, which can create and publish random user requests. There are two filters, **InGothenburgFilter** and **OnLandFilter**. The messages from **RequestGenerator** will be filtered by **InGothenburgFilter**. Then the filtered data will pass to **OnLandFilter** via **RequestsPipe**.

The two devices communicate with each other via the broker over MQTT. This means that we only deploy each component once.

The second device is responsible for the **Visualizer**, which displays the user requests received on the map. The RequestGenerator Pipe & Filter and the Visualizer communicate with each other via the broker.