Advanced Software Engineering  
FINAL REPORT

|  |  |
| --- | --- |
| **Team number:** | 0508 |

|  |  |
| --- | --- |
| **Team member 1** | |
| **Name:** | Klavio Tarka |
| **Student ID:** | 12146253 |
| **E-mail address:** | a12146253@unet.univie.ac.at |

|  |  |
| --- | --- |
| **Team member 2** | |
| **Name:** | TODO |
| **Student ID:** | TODO |
| **E-mail address:** | TODO |

|  |  |
| --- | --- |
| **Team member 3** | |
| **Name:** | TODO |
| **Student ID:** | TODO |
| **E-mail address:** | TODO |

|  |  |
| --- | --- |
| **Team member 4** | |
| **Name:** | TODO |
| **Student ID:** | TODO |
| **E-mail address:** | TODO |

|  |  |
| --- | --- |
| **Team member 5** | |
| **Name:** | TODO |
| **Student ID:** | TODO |
| **E-mail address:** | TODO |

# Final Design

## Design Approach and Overview

### Assumptions

Assumptions:

* The maintainer will only have access to the health check system, he will check if the service is alive or dead.
* There can be multiple devices in the house that can control rooms and the devices in each room.
* The user can only interact with their devices once he’s logged in.
* Each service shall have it’s own database, unrestricted by other databases. The notification service will have an in memory database as well.
* Each service shall have it’s own API endpoints and can only communicate with other classes via these points.
* Each action will be followed by a notification via E-Mail.

### Design Decisions

Design Decisions on the logical level:

* Decision for using an n-to-m mapping between devices and rooms and not simply devices being contained in rooms. This would allow us to also support devices that span multiple rooms like a heating system.
* Decision to have time-based actions service only store the operations and the time events to then address the controller. This reduces the need to have any too tight coordination and we can mostly just use conformist patterns (with ACL) and not worry about any shared kernels, etc.

Decision to use sequence diagrams because they allow the detailed description of communication flows.

### Birds-eye view

We have finished developing all the architectural views for the 4 + 1 model, and beforehand we also worked with the context map which gave us a basic idea of how the system should work and communicate with each other.

To documentmyteam’s progression we have included the team’s evolving ubiquitous language in the following context map iterations.

Graphical user interface, text

Description automatically generated

Figure 1: Context Map 1.0

Diagram

Description automatically generated

Figure 2: Context Map 2.0

## Architectural Design Decisions (ADDs)

We decided to use C# asmycoding language as it is being developed by Microsoft and thus it has well written documents, a large amount of resources both in their official website and spread on the internet. For the website which acts as a simple interface I used simple HTML and Javascript. As for my architecture builds, Microsoft provides official documentation for many elements. In particular we we read information about the publish/subscribe event system, additional information on DDD as explained on the official docs.microsoft website, the microservice system which is thoroughly explained using an example GitHub project about online shopping, and lastly, a few books which are offered for free, again by the official Microsoft website (we are not sponsored by Microsoft).

.NET 6.0 was used for most simple microservice examples that were created by the team, because it is the latest version of .NET and the one that holds the most up to date resources.

dotnet will most likely be the nuget providing system we are going to use as it is easy to implement external libraries in my system, and it also helps the developer with creating the UI part of the microservices.

Links:

* <https://docs.microsoft.com/en-us/>
* <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/microservices-architecture>
* <https://docs.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber>
* <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice>
* <https://docs.microsoft.com/en-us/microsoft-365/solutions/cloud-architecture-models?view=o365-worldwide>
* <https://www.youtube.com/watch?v=r8ucofiI8vY&t=2536s>

## Major Changes Compared to DESIGN

After the feedback from my instructor, I decided that not every single notification should proc an email notification to the user. And even though I would have liked to make many more architectural design changes from the first phase, I didn’t have enough time to implement all the changes in code and in the architecture. Some code decisions I made specifically were deciding the communication between the APIGateway with the other services was synchronous while the communication between services that aren’t reachable directly without the APIGateway is asynchronous. Another change was that I decided to include a repository for every single service. After reading some Microsoft official documents, I decided that it was a good coding architecture to not have the api endpoints of a service to communicate directly with the database, hence the existence of the repositories.

## Development Stack and Technology Stack

I decided to use C# as my coding language as it is being developed by Microsoft and thus it has well written documents, a large amount of resources both in their official website and spread on the internet. As for my architecture builds, Microsoft provides official documentation for many elements. In particular I read information about the publish/subscribe event system, additional information on DDD as explained on the official docs.microsoft website, the microservice system which is thoroughly explained using an example GitHub project about online shopping, and lastly, a few books which are offered for free, again by the official Microsoft website.

Additionally, as for other technologies I used I should add Kubernetes which acts as a host for my docker images and it keeps them up and running at all times.

Dotnet was used to provide the needed packages for my system such as

* Automapper
* Sql and InMemory databases
* Dependency Injection
* FrameworkCore packages

I would like to point out that the first link in the resources used was a huge help and it helped with setting up the structure of my project.

Links:

* <https://www.youtube.com/watch?v=DgVjEo3OGBI&t=21134s>
* <https://docs.microsoft.com/en-us/>
* <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/microservices-architecture>
* <https://docs.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber>
* <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice>
* <https://docs.microsoft.com/en-us/microsoft-365/solutions/cloud-architecture-models?view=o365-worldwide>
* <https://www.youtube.com/watch?v=r8ucofiI8vY&t=2536s>

### Development Stack

I mainly used GitLab, Docker and Kubernetes.

### Technology Stack

C#, Draw.io, dotnet, .NET 6.0, GitHub for repository testing, GitLab for setting up the CICD pipeline, Kubernetes for having a way to keep the docker images up and running at all the times even when errors happen. Docker hub for a place where I could organize my containers. I used Insomnia to test requests (GET, POST) and if the URLs Ire functional. I used Discord to get extra information about the deployment phase of the project.

# **Updated Requirements**

The specified system requirements have not been explicitly split up among the team members, but as we decided to go for a domain driven approach, all the system requirements are relevant and should be considered by all the team members wherever their consideration is required to design a particular facet of the system or implement it. This being said, we will ensure that every person implement their part of the system requirements. We have also not yet defined test cases, however, based on the interaction of the services/components as we have defined and discussed in our internal discussions, the required inputs and outputs can already be anticipated.

A big resource usage in the performance of the application would be the docker images (this could maybe be because my laptop is overall not that good either), however each docker image is around 300MB and according to my laptop, all the resources are max-ed out. CPU, RAM and storage. However, after publishing the docker images on Kubernetes and running the services from Kubernetes, the resource usage dropped by a lot which made the application run much more smoothly.

# **Updated 4+1 Views Model**

In the following sections we present use case diagrams that depict the main features / scenarios covered and provided by our solution. Whenever we refer to *UC#*, then the main use cases that are required in the original assignment sheet are meant. For each of these *UC#*s we have defined a separate use case diagram, as well as for a few additional ones, we added to illustrate additional key functionality. In the use case description section following the diagrams we then describe in depth the key use cases from the use case diagrams.

For example, when we refer to use case***3.1*** then this means the description refers to the sub-use-case 3.1 in the “main use case” ***UC3***.

To keep the balance between readable and helpful, we have decided to abstract from some very basic or obvious use cases, which don’t require further explanation.

### Use Case Diagram(s)

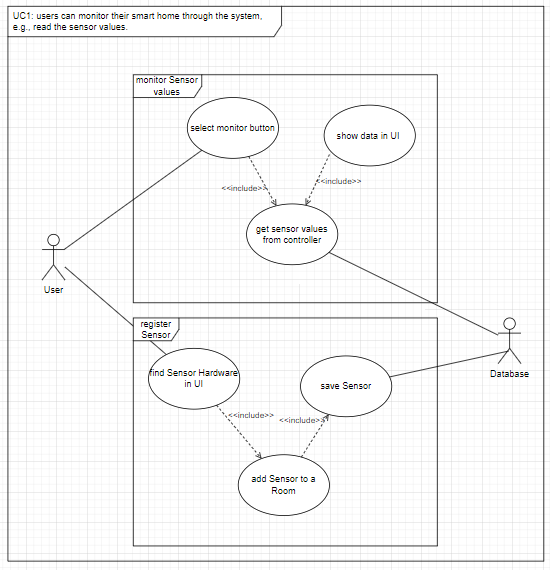


Figure 3: UC1, users can monitor their smart home through the system

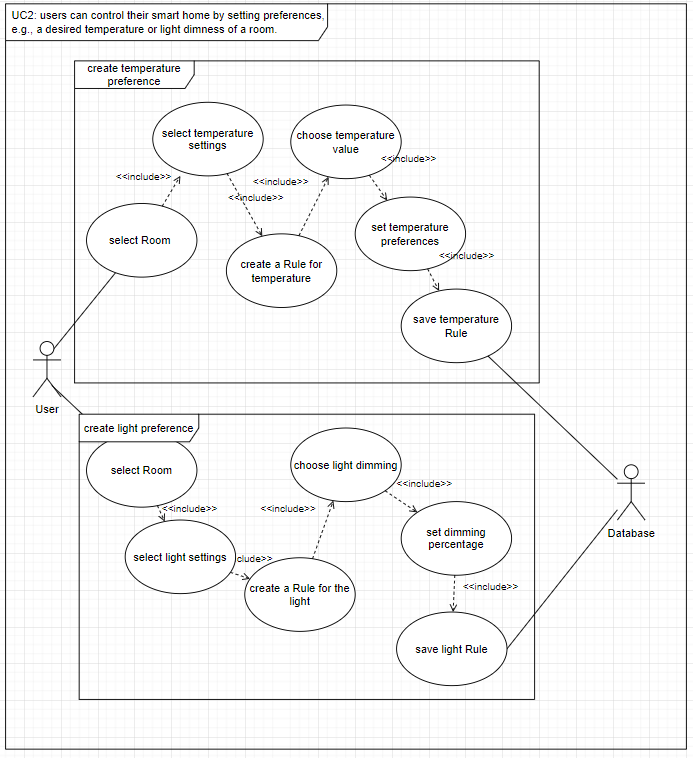


Figure 4: UC2, user can control their smart home by setting preferences

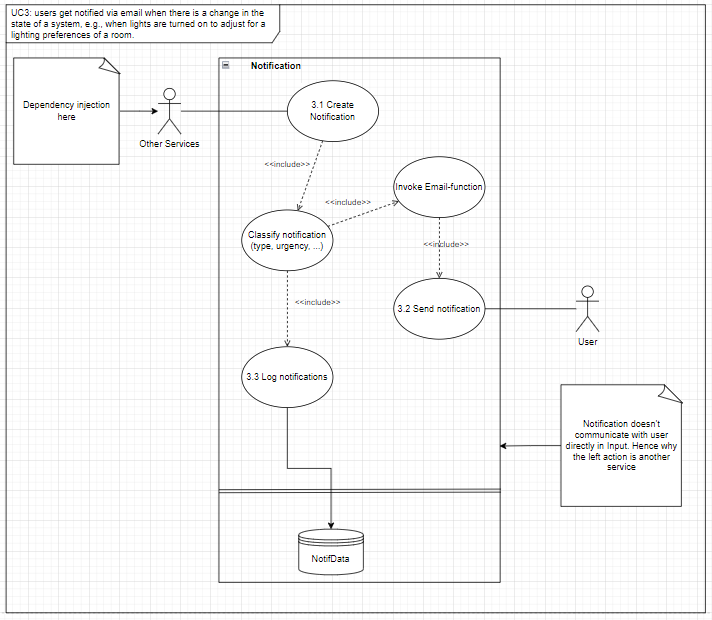


Figure 5: UC3, user get notified via email when there is a change in the state of a system

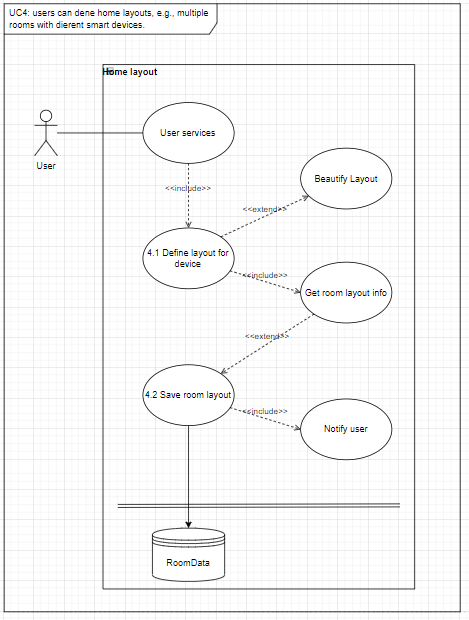


Figure 6: UC4, user can define home layouts

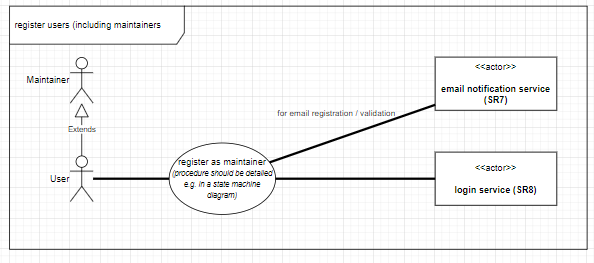


Figure 7: use case, user can register in the System

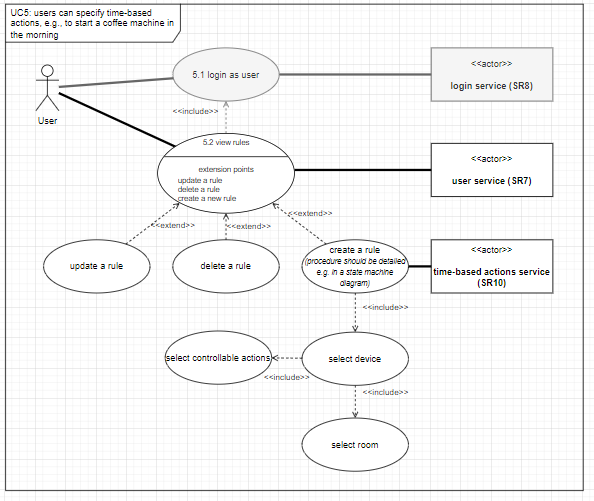


Figure 8: UC5, user can specify time-based actions

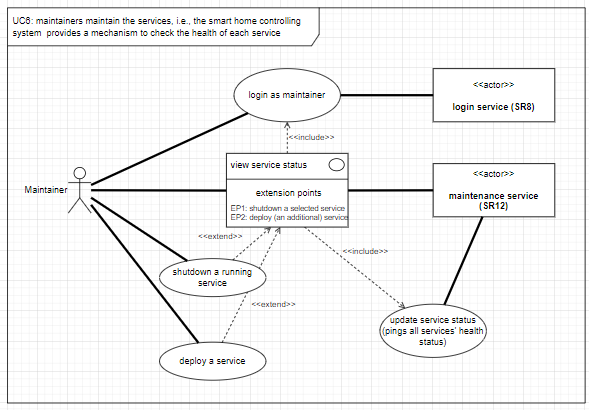


Figure 9: UC6, maintainers maintain the services

## Scenarios / Use Case View

### Use Case Diagram(s)

Insert Use Case Diagrams here.

### Use Case Descriptions

Note: One description table per use-case!

|  |  |
| --- | --- |
| **Use Case:** | Privilege Service |
| **Use Case ID:** | Privilege Service |
| **Actor(s):** | The client |
| **Brief Description:** | This service severs as the mechanism for the user to register or log in |
| **Pre-Conditions:** | You need to provide a username/email and password |
| **Post-Conditions:** | Username/email & password need to match that in the database |
| **Main Success Scenario:** | User logs in or registers |
| **Extensions:** | None |
| **Priority:** | - |
| **Performance Target:** | - |
| **Issues:** | There’s no hash encryption |

|  |  |
| --- | --- |
| **Use Case:** | Notification Service |
| **Use Case ID:** | Notification Service |
| **Actor(s):** | Client |
| **Brief Description:** | Once the user creates a time based action a notification is created in the database. |
| **Pre-Conditions:** | An object action needs to be registered and the user needs to register the type as getting notifications from |
| **Post-Conditions:** | Notification is saved successfully in database |
| **Main Success Scenario:** | Notification proctime happens and user gets notified |
| **Extensions:** | None |
| **Priority:** | - |
| **Performance Target:** | - |
| **Issues:** | - |

|  |  |
| --- | --- |
| **Use Case:** | Home Layout Service |
| **Use Case ID:** | Home Layout Service |
| **Actor(s):** | Client |
| **Brief Description:** | The user can change the layout of his house, add rooms, remove rooms, get the home layout etc. |
| **Pre-Conditions:** | - |
| **Post-Conditions:** | Homelayout needs to be successfully saved in database. |
| **Main Success Scenario:** | Homelayout is saved successfully in database. |
| **Extensions:** | None |
| **Priority:** | - |
| **Performance Target:** | - |
| **Issues:** | - |

|  |  |
| --- | --- |
| **Use Case:** | Api Gateway Service |
| **Use Case ID:** | Api Gateway Service |
| **Actor(s):** | Client |
| **Brief Description:** | A direct request redirector for the between the user and other services. |
| **Pre-Conditions:** | - |
| **Post-Conditions:** | Request is successfully redirected and an answer is received. |
| **Main Success Scenario:** | Request is successfully redirected and answered is received. |
| **Extensions:** | None |
| **Priority:** | - |
| **Performance Target:** | - |
| **Issues:** | - |

## Logical View

Graphical user interface

Description automatically generated with medium confidence

## Process View

The following sequence diagrams provide more information on the dynamic aspects of the system. They provide insight into the sequence of components that are being invoked.

Chart, box and whisker chart

Description automatically generated

Figure 11: Notification Sequence Diagram

Chart, box and whisker chart

Description automatically generated

Figure 12: Home Layout Sequence Diagram

Chart

Description automatically generated

Figure 13: Register + Login Sequence Diagram

A picture containing graphical user interface

Description automatically generated

Figure 14: Time-Base-Action Sequence Diagram

Chart, box and whisker chart

Description automatically generated

Figure 15: create and monitor sensor values Sequence Diagram

Chart

Description automatically generated

Figure 16: create preferences for temperature and lights

## Development View

The following component diagram provides insight about the interaction between the components of the system on a technical level.

Diagram

Description automatically generated

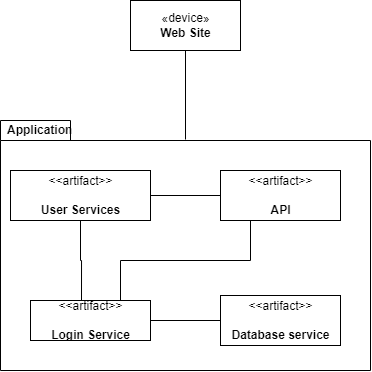
The central components which feed directly into the UI are the Maintenance Service and the User Service Components. They do invoke the Login service and save a token. Furthermore, the User service can invoke the components of home layouts and device controlling services or time-based actions. Device Monitoring services encapsulate hardware and are invoked by controlling services. E-mail notification service provides functionality to be invoked by other services. Finally, as this reflects a microservice architecture, the different components are served by their own databases which retain information on specific things e.g., stored user preferences or the allowed device operations for a specific device.

## Physical View

The Physical View includes the architectural design of how our system will be connected in the real world. We have decided to follow a simple approach for the longevity of our system, and how it will run.

First, the application is composed out of microservices, each providing different services, such as:

* User services (monitoring system, home layout service, logging in, time-based service, notification system etc)
* UI which the user will interact with (simple website, potentially built using ASPNET)
* Database system which again will most likely be SQL



# **Continuous Delivery**

I use docker and Kubernetes to deploy and keep my applications running at all time. My most used commands are as below

## Docker

* Docker is used to deploy my image containers and thus works well with microservices. The commands I have used during deployment so far are as below
* docker ps -> Shows all the running containers
* docker start -> Starts a container of my choice after i specify the port (if i want to) and the name of the container
* docker stop -> Stops the container from running
* docker push -> Basically the same as github pushing, it uploads your container to docker hub.

## Kubernetes

* Kubernetes is nice to work with since I can integrate a lot of services and they can all communicate with each other once i use the API gateway. It also has good documentations and more importantly, it makes sure even if my services go down, kubernetes will always be there to restart them.

This happens using the .yaml files below: notification-depl.yaml -> the replica typeword in here lets kubernetes know how many replicas, or pods do i want to be running at the same time with my container.

* Kubectl delete deployment notif-depl.yaml -> Deletes the kubernete pod (which basically works as a restart for the docker image)
* kubectl apply -f notif-depl.yaml -> Creates a pod for container and keeps restarting it everytime it goes down
* kubectl apply -f notif-depl-srv.yaml -> Creates the service in Kubernetes and makes the service accessible via a port.

## Distribution of Work and Efforts

Contribution of Member 1: Klavio Tarka

I did all the implementation in code, deployment and databases for every single service. Databases were created for only specific services because I believe not all services require a database even though that’s part of microservice architecture. For example, I don’t believe Api Gateway requires a database since it will be redirecting requests rather than saving requests.

# 14.05.2022

## Dto (data transfer object) files

* Dto are objects used for communication between the user and the service. In other words if the user wants to communicate with the service, these objects are like information which is passed between the service and the user using the API endpoints
  + NotificationCreateDto > Whenever a client tries to create a notification for a specific action this is the file that will handle it. For example if the user wants to set the microwave in working condition for 2 hours, a notification will be created using the properties
  + **ProcTime**: DataTime -> Handles when the notification will be sent to client’s email
  + **NotificationDetails**: string -> Specific details about the notification
  + NotificationReadDto >Whenever an API GET request is sent to the microservice the service sends a Dto object that has these properties for the user to read.
  + Id: int -> The Id of the notification

## Controllers

* + NotificationController > Creates API endpoints which are used as a means of communication between the user and the service.
* API Endpoints
  + GetNotification, route: localhost:5258/api/notification (testing purposes) This is a GET request and it sends all the notifications to the user
* GetNotificationById(int id), route: localhost:5258/api/notification/{id} (testing purposes) > This is also a GET request which sends the user a specific notification based on the id
* CreateNotification, route: localhost:5258/api/notification > This is POST request which gives the user the ability to create a notification.

## Repository

* + INotificationRepo > 1. Gives the ability to save changes in database. > 2. Create notifications in db. > 3. Get notifications from the db.

## Models

* + Notification Model > Creates the base notification model which will be used in database entries and to create DTOs

# 15.05.2022

## Notification service

* I decided to add a few extra attributes to the class.
* >The model, readDto and CreateDto classes get an extra boolean Triggered and string Type
* >Triggered -> Works as a way for the system to check if the notification has already been sent to the user or not, so the system doesn’t send the same notification twice.
* >Type -> is a string which basically lets the notification service know which object in the house is causing the notification, e.g. (Lights, doors, windows)
* >Added an extra class which holds preferences of the user. After reading the notes the instructor provided for us in GitLab (bloating of notifications, I had already figured I would just make the less important objects not send a notification at all even though not explained). However, after further work, I decided to give the user the ability to pick between which notifications he wants to receive.
* >The class holds a List of device types which they want to receive notifications from.

Additionally, Notification service holds the ability to use both an InMemory database and a sql database. I implemented it this way because this service will be checking the database often for notification ProcTime, this would be incredibly slow so I mitigated this problem by implemented an InMemory database. Basically, everytime the service is started the InMemory database communicates with SQL, and adds all the needed data from SQL to InMem. For the longevity of the service, we use the InMem database to search for data and only use the SQL data if we want to modify a notification or add a new notification.

Notification service is the only service which can communicate with the gateway without client interaction. A simple example would be

“If a notification has already been created, then the notification will wait until one of the notification’s proctime triggers and then send a request to the gateway so the client can know something happened in their house. This is also followed by an email if the user wanted to receive a notification from said object.

## Docker

* Docker is used to deploy my image containers and thus works well with microservices. The commands I have used during deployment so far are as below
* docker build -t maxxburn/nameOfService . -> It builds the container for the service.
* docker ps -> Shows all the running containers
* docker start -> Starts a container of my choice after i specify the port (if i want to) and the name of the container
* docker stop -> Stops the container from running
* docker push -> Basically the same as github pushing, it uploads your container to docker hub.

## Kubernetes

* Kubernetes is nice to work with since I can integrate a lot of services and they can all communicate with each other once i use the API gateway. It also has good documentations and more importantly, it makes sure even if my services go down, kubernetes will always be there to restart them.

This happens using the .yaml files below: notification-depl.yaml -> the replica typeword in here lets kubernetes know how many replicas, or pods do i want to be running at the same time with my container. Some commands:

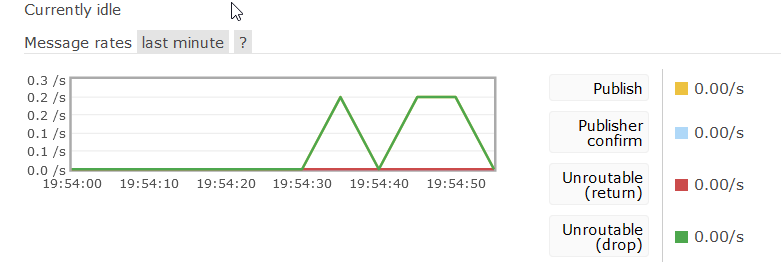
# 23.05.2022

## Implementation of RabbitMQ as a publish/sub pattern

For my publish/subscriber pattern to notify other services I am using RabbitMQ which utilizes and event Bus. This is an asynchronous communiciation, basically everytime a microservice posts something in this event bus every other microservice connected to it will get the information if they are subscribed to the publisher.

First I create a deployment file for kubernetes and then start a pod for the RabbitMQ service, I decide the ports for the event bus and then i deploy the service in kubernetes.

After the deployment is finished, any service that integrated the PublishMethod via the interface can post in the message bus.

The chart of requests being posted can be seen below [[Pasted image 20220523195519.png]] 

ApiGateway

Unlike other services I decided to code the ApiGateway so it communicates with other services using synchronous communication via HTTP requests. For example

If the client wants to get the home layout he would first need to communicate with the api gateway, the api gateway processes the request and decides in which URL will it send the request(in this case <http://localhost:5235/api/homelayout/>).

I decided not to give the ApiGateway a database because it didn’t sound logical to give a database to a service which works as a redirection of requests and holds no data of it’s own.

Website

The website is a simple interface which basically integrates only input fields, scrollable lists and buttons. You are first introduced with a simple login system, if the username/email and password are correct then the user logs in and they can interact with the system. So far the interface includes, creation of new users, receiving the home layout, adding new devices to a specific room in the house, receiving all devices that exist in the house currently, receiving the notifications that are currently in the database(yes, I have included a Boolean in the notification which basically lets the system know whether the notification has already been triggered or not). I have included a way to get all the users in the database, and check their personal information (email, username, role, id). Since one of the requirements of the projects was that the user would be able to change which smart device would control each room, I coded a way for the user to change which smart device would control the room, by changing the smartdeviceID that is in the record of the room table. I couldn’t implement everything in the website, however a lot of other stuff are implemented and can be tested using simple get/post requests.

# **How-to documentation**

CI Pipeline: The pipeline is ran automatically when you commit to the main branch of gitlab.   
  
Integration test: To run all the microservices locally, website included, you would first need to go to the directory of each microservice separately and run the services using dotnet. The command that would run the services is

dotnet run -> After this command is typed in the CLI, the service checks for any migration in the database and migrates the changes with the rules created by the coder(in this case me). Each microservice has a folder which is called Models, and this folder serves to create the tables of the database and the required columns for each table. For example the Privilege Database, has 1 table which stores the user and their roles. The specific table columns are, id, username, email, password and role (user, administrator). To test that each microservice is working as intended you could send simple get/post requests using insomnia or postman.

However, I tried to mimic a production environment as much as possible and thought running each microservice one by one would be troublesome, and eat a lot of time (plus each time a microservice shuts down or has an error, the user would need to restart them back). Thus, I decided (with the help of one of the youtube videos linked above) to deploy everything in Kubernetes, which would take both of starting the microservices all at the same time automatically and restart them everytime they go down.

Each microservice in Kubernetes has their own server yaml file, which dictates the name and the number of the port which will be used to access the microservice. Additionally, I decided to implement the database in Kubernetes as well, so everytime the microservices are started, the database has no need for extra work (it’s gonna start automatically as well). I decided to give the database a memory of 250-300 MB.

As for the publish/subscribe way of communication, this is also implemented in Kubernetes using RabbitMQ. All microservices have it implemented, however not all of them need it I believe. For example, I don’t think we need a communication between services when a user logs in or does user related stuff such as (creating a new user, or logging in)

The apigateway was the most time consuming microservice in my opinion, since I had to work with different microservices at the same time while also managing data models and the database at the same time. I also had to make sure each request sent to the apigateway sent back a confirmation to the user upon finishing the request. For example:

If we wanted to add a new device in a room, we would send the request to the apigateway, the gateway would redirect our request to the corresponding microservice, in this case homelayoutserice. This microservice, would add the new device in the database and link it to the specific room it’s found in, and then send a confirmation back to the apigateway that the request was well received and done its job. A more thorough description of errors that happen here can be found in the logs of Kubernetes, which basically tell you everything you need to know about the error (line which triggered the error, description of error etc).

How to run the system in production:

Simply download the project folder, and access the Kubernetes folder. In here you should type the command below to create pods for each microservice, and the database which will store the information for all microservices.

kubectl apply -f /Kubernetes/name-of-file-depl.yaml   
kubeclt apply -f /Kubernetes/name-of-file-srv.yaml

In case an error happens here, I would suggest running each microservice separately(just in case Kubernetes doesn’t create the tables in the database for each microservice)

After the process above is done, accessing every microservice should be pretty straightforward. You just type the url of the microservice and you will receive information from that microservice.  
Below I am attaching the url of the apigateway which will be used to redirect and a few calls to different microservices from the apigateway.

<http://localhost:31871/apigateway> -> Link of apigateway

<http://localhost:31871/apigateway/login> -> Logging in

<http://localhost:31871/apigateway/GetAllDevices> -> Get all room devices