

6/21/2024

# Distributed Data and GDPR Implementation

Individual Track

Name: Lucas Jacobs

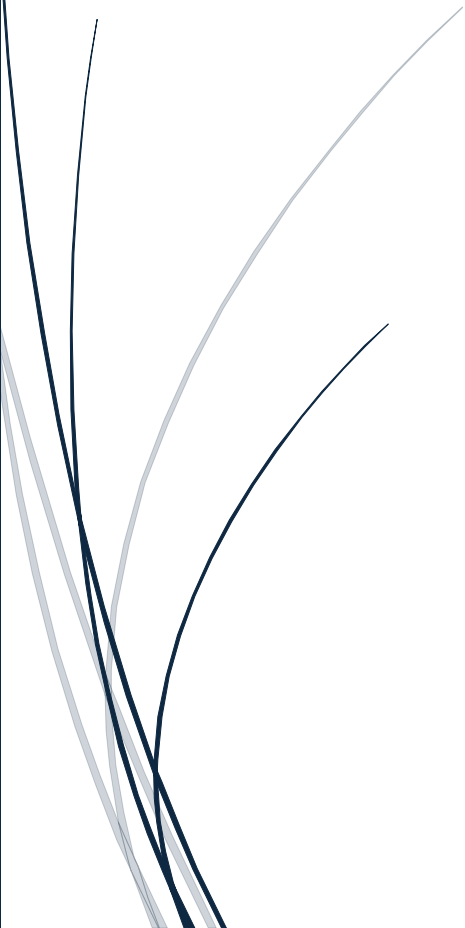
Class: S-A-RB06

PCN: 490692

Student number: 4607368

Technical teachers: Felipe Ebert, Bartosz Paszkowski

Semester coach: Gerard Elbers



## Table of Contents

Introduction.....	1
Data storage .....	2
Architecture .....	2
RabbitMQ (Message Queue) .....	3
MySQL Database (Relation Database) .....	5
Redis (Caching) .....	6
GDPR .....	7
Update and Delete a User .....	7
References .....	8

## Introduction

This document contains the implementation of all the data storages and functionalities for GDPR. All the things that are implemented are argued and specified in (Jacobs, Storage and GDPR Compliance Strategy).

# Data storage

## Architecture

See the following image of where I implemented the different data stores. To focus on the user service, I outlined the implementation of the data storage for the user services.

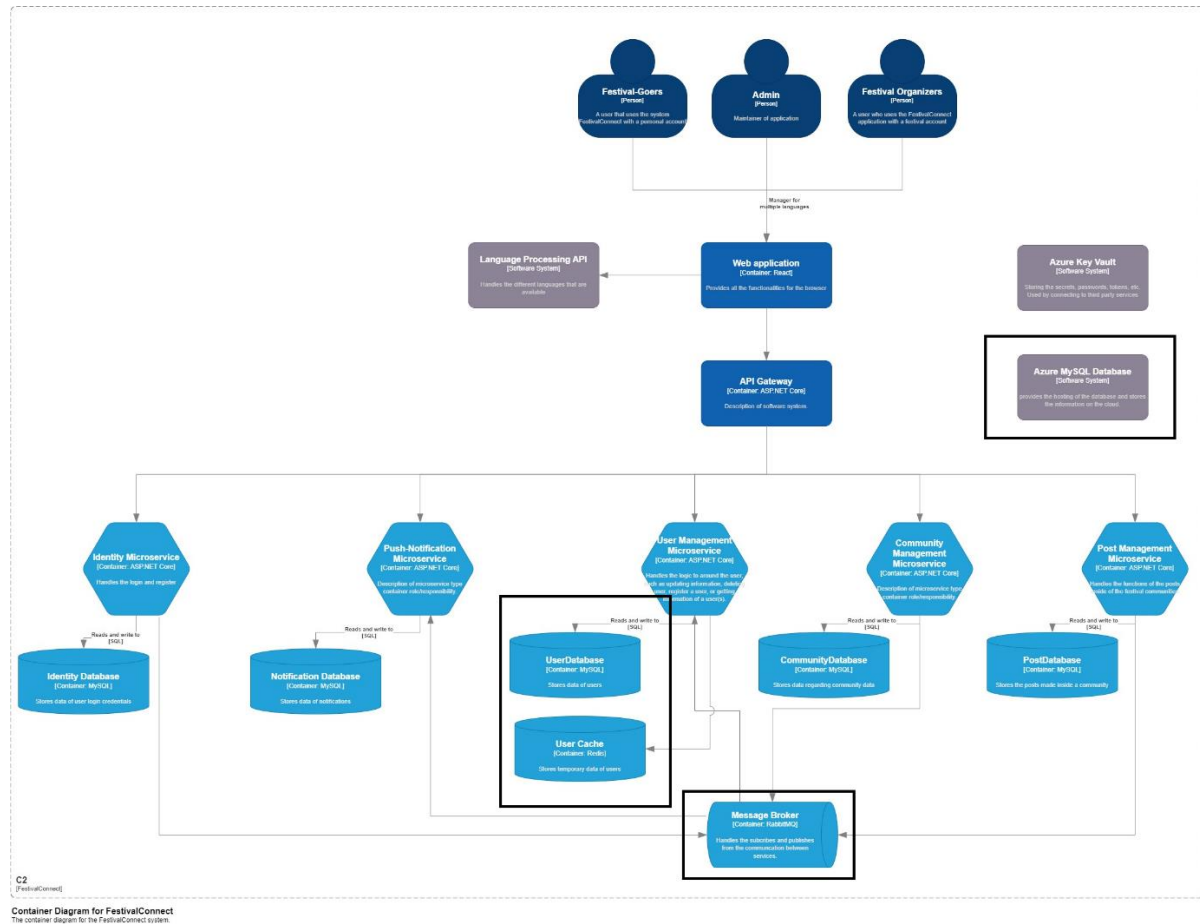


Figure 1: Architecture FestivalConnect

## RabbitMQ (Message Queue)

From the documents of (Jacobs, Technical Design) and (Jacobs, Storage and GDPR Compliance Strategy) I explained what benefits a message broker has in applying FestivalConnect. I applied a message queue for the following operations

- When registering a user, a request is sent to the Identity Service. In the request, it publishes a message to the queue. The user service consumes (listens) a message when the user registers and the needed information is saved inside of the database.
- When deleting a user, a request is sent to the identity service and publishes a message to the delete queue. The user service consumes the message from the delete queue and deletes the user. This is also used so that all the user's data is removed once the user wants to remove his account.

See the following picture of the rabbitMQ environment.

The screenshot shows the RabbitMQ management interface. At the top, it says 'RabbitMQ 3.13.1 Erlang 26.2.4' and 'Refreshed 2024-06-11'. The 'Queues and Streams' tab is selected. Under 'Queues', there are two queues listed: 'CreateUser' and 'DeleteUser'. Both are classic queues, running, and have no messages. Below the table is a form to 'Add a new queue'. The form includes fields for 'Virtual host' (set to '/'), 'Type' (set to 'Default for virtual host'), 'Name' (empty), 'Durability' (set to 'Durable'), and 'Arguments' (empty). There are also links for 'Add', 'Auto expire', 'Message TTL', 'Overflow behaviour', 'Single active consumer', 'Dead letter exchange', 'Dead letter routing key', 'Max length', 'Max length bytes', and 'Leader locator'.

Overview				Messages				Message rates			
Virtual host	Name	Type	State	Ready	Unacked	Total	incoming	deliver / get	ack		
/	CreateUser	classic	running	0	0	0					
/	DeleteUser	classic	running	0	0	0					

Figure 2: RabbitMQ User Interface

This has two queues, one for creating a user and one for deleting.'

To give an example of creating on registering a user. When a request is sent to the identity service to handle and save sensitive information of the user, it will then publish a message to the queue with the identity ID and the needed information that needs to be stored in the user service.

```
_messagingLogic.Publish("CreateUser", "CreateUser", new RabbitMessage<UserResponse>()
{
    Data = new UserResponse()
    {
        IdentityId = getRegisteredUser.Id,
        Username = request.Username
    }
});
```

Figure 3: Publish a message to the queue

Where then in the user service I have a hosted service that consumes messages from the queue and handles the incoming requests.

```
public class RabbitServerManager : IHostedService
```

```
0 references
public Task StartAsync(CancellationToken cancellationToken)
{
    var scope = _serviceProvider.CreateScope();

    var messagingLogic = scope.ServiceProvider.GetRequiredService<IMessagingLogic>();

    messagingLogic.Subscribe<RabbitMessage<UserResponse>>(<
        "CreateUser",
        "CreateUser",
        "CreateUser",
        async (message) => await HandleMessageAsync(message, async (logic, msg) => await logic.RegisterUser(msg)));

    messagingLogic.Subscribe<RabbitMessage<UserResponse>>(<
        "DeleteUser",
        "DeleteUser",
        "DeleteUser",
        async (message) => await HandleMessageAsync(message, async (logic, msg) => await logic.DeleteUser(msg)));

    return Task.CompletedTask;
}
```

```
2 references
private async Task HandleMessageAsync(RabbitMessage<UserResponse> message, Func<IUserLogic, RabbitMessage<UserResponse>, Task> action)
{
    using (var scope = _serviceProvider.CreateScope())
    {
        var userLogic = scope.ServiceProvider.GetRequiredService<IUserLogic>();

        try
        {
            await action(userLogic, message);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error processing message");
        }
    }
}
```

Figure 4: Consuming and handling the incoming request

This hosted service will consume messages and trigger an action to save the information of a user inside the user database.

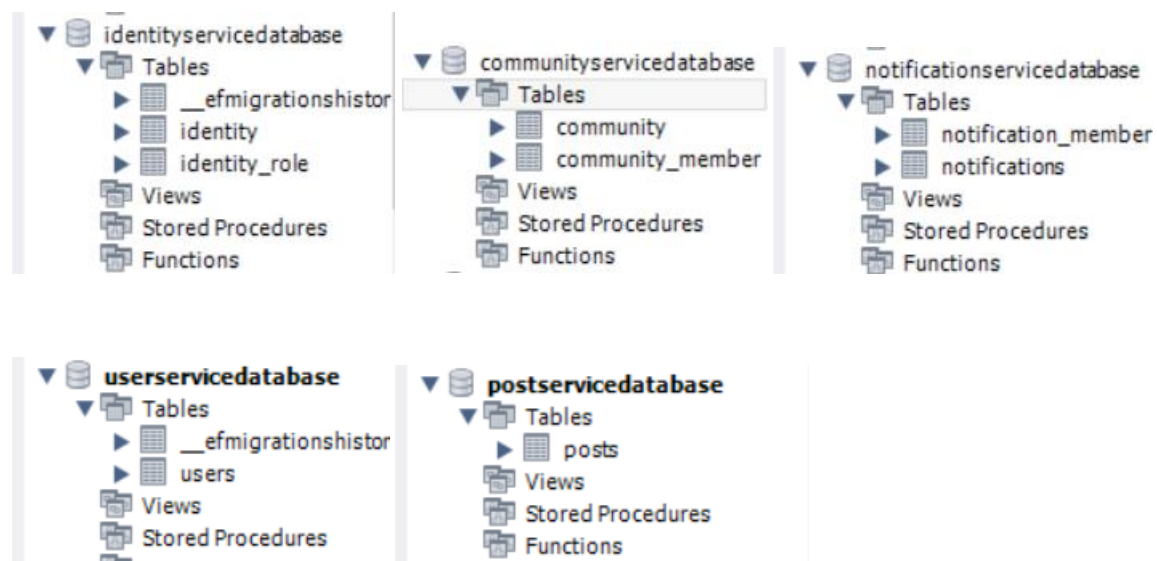
## MySQL Database (Relation Database)

From the document (Jacobs, Storage and GDPR Compliance Strategy), I specified the benefits of a relational database.

Each service in my application has a separate database because of the following:

- Easier to scale each service without also affecting other data, according to the requirements specified in (Jacobs, Software Requirement Specification).
- Contributes to security, since sensitive data can be isolated within specific microservices, reducing the risk of unwanted users accessing data.
- Based on the service, you can use different technologies (relational, NoSQL) based on the requirements.

See the following picture of the implementation of the databases, using the MySQL Workbench.



Moreover, in the production environment I am using a cloud service to host my database, this provides benefits such as pay-as-you-go, ensuring availability with replication and redundancy, backups, and having built-in monitoring options. I described this more in-depth in (Jacobs, Cloud Analysis) and in the document (Jacobs, Storage and GDPR Compliance Strategy).

## Redis (Caching)

In (Jacobs, Storage and GDPR Compliance Strategy), I explained how it benefits my application. Furthermore, I will show you my code for the implementation of Redis. I used it to get the information of a user. When you get a users information, if it is not saved in the cache it will stored it in there. When a user updates or deletes their information, the application will remove the cached information, having data consistency over the application.

See the following picture, for the logic of getting a user by identity id.

```
2 references
public async Task<UserResponse> GetUserByIdAsync(int identityId)
{
    string cacheKey = $"{_cacheKeyPrefix}{identityId}";
    // Try to get the user data from the cache
    string? chachedUser = await _cache.GetStringAsync(cacheKey);
    if (!string.IsNullOrEmpty(chachedUser))
    {
        // Deserialize and return the cached data
        return JsonSerializer.Deserialize<UserResponse>(await _cache.GetStringAsync(cacheKey));
    }
    UserModel user = _userRepository.GetByIdentityId(identityId);
    if (user != null)
    {
        UserResponse userResponse = new UserResponse()
        {
            Id = user.Id,
            Username = user.Username,
            Active = user.Active,
            RegistrationDate = user.RegistrationDate,
        };

        var cacheEntryOptions = new DistributedCacheEntryOptions
        {
            AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(5) // Cache for 5 minutes
        };
        string userJson = JsonSerializer.Serialize(userResponse);
        await _cache.SetStringAsync(cacheKey, userJson, cacheEntryOptions);

        return userResponse;
    }
    else
    {
        throw new ArgumentException("No user with such id.");
    }
}
```

Figure 5: Get User Information Method'

So it first tries to get the cached user by the cache key you created with the cache prefix and the identity id. If it exists it will return the cached value, if not it will get the user from the database. Once the user is retrieved from the database, it will save it in the cache. Note that only non-sensitive data is stored inside of the cache.

For the update and delete a user, I used the following:

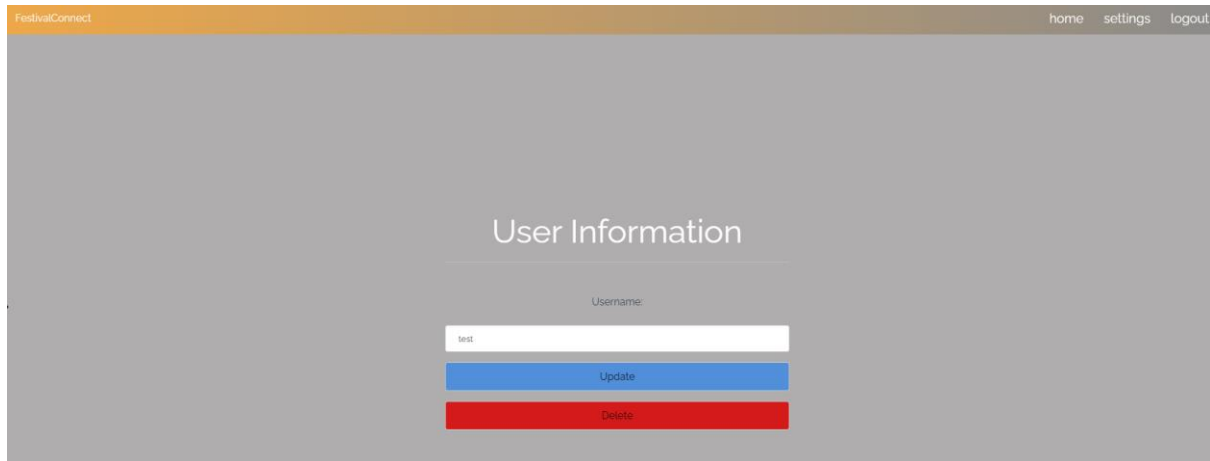
```
// Remove the user from the cache
var cacheKey = $"User_{request.Data.IdentityId}";
await _cache.RemoveAsync(cacheKey);
```

Figure 6: Code to remove the cache

# GDPR

## Update and Delete a User

The user always needs to access there information and is able to adjust them or remove it. See the following implementation of the update and delete information



The screenshot shows a web application interface for 'FestivalConnect'. At the top, there is a navigation bar with 'FestivalConnect' on the left and 'home', 'settings', and 'logout' on the right. The main content area has a grey background and is titled 'User Information'. Below the title, there is a form with a label 'Username:' above a text input field containing the text 'test'. Below the input field are two buttons: a blue 'Update' button and a red 'Delete' button.

Figure 7: Settings Page



## References

Jacobs, L. (2024). Data Storage and GDPR Compliance Strategy: Practical Guide (Unpublished manuscript), FontysICT.

Jacobs, L. (2024). Technical Design (Unpublished manuscript), FontysICT.

Jacobs, L. (2024). Software Requirements Specification (Unpublished manuscript), FontysICT.