# Microservice development using RabbitMQ message broker

Amar Ćatović[1], Nevzudin Buzađija[2], Samir Lemeš[2]

[1] *Isatis Software Solutions, Kralja Tvrtka 12, Sarajevo 71000, Bosnia and Herzegovina*

[2] *University of Zenica, Fakultetska 1, Zenica 72000, Bosnia and Herzegovina*

## Abstract

Nowadays, when applications are being developed faster with the introduction of agile methodologies and new technologies, microservices are emerging. The microservices make applications easier to create and maintain when broken down into smaller parts, which form a whole application. RabbitMQ acts as an intermediary between the various services. It reduces the load and delivery time on server web applications by delegating tasks that would typically take a lot of time and resources. Message queuing allows web servers to respond quickly to requests rather than being forced to perform complex procedures that can take more time and resources. AMQP (Advanced Message Queuing Protocol) is a message protocol that deals with publishers and consumers like any other messaging system. Publishers produce messages while consumers download and process them. The job of message brokers, such as RabbitMQ, is to ensure that messages from publishers go to the right consumers. To do this, the broker uses two key components: exchange and order. We demonstrated that the style of microservice architecture is an approach to the development of an application as a set of small services, each in charge of its own process and communication with other services.

***Keywords***: *Microservices, Software Development, RabbitMQ, Message Queuing*

## 1 Introduction

Modern application architecture is increasingly migrating towards microservice technology due to the rapid development of applications and shortening the lifetime of applications. By using microservices, different development frameworks can be used for individual microservice, on the other hand, failure of one microservice does not affect the functionality of other microservices. Microservice architectures allow developers to achieve greater agility and can significantly improve development time. The advantages of microservice architectures are the following [1]: independence of developers - small teams work in parallel and can go through more iterations of the development cycle than large teams; isolation - if one microservice stops working, the rest of the application will not crash, but only one part of it becomes inaccessible; life cycle automation - individual components are easier to fit into Continuous delivery pipelines.

RabbitMQ is a free open-source solution that serves as a message broker via the AMQP (Advanced Message Queuing Protocol) protocol. It is highly accessible, fault-tolerant and scalable. In today's microservice applications, RabbitMQ is used as an intermediary between individual microservices, allowing them to communicate with each other without worrying about message loss [2]. Like bees, each development team makes a separate component using one of the technologies. Each of the individual components forms a strong structure of the so-called microservice hive and performs a separate role in the system.

Several authors investigated the use of RabbitMQ in microservice development. Kwon et al. in [3] found that RabbitMQ is vulnerable and presented how it can be exploited by protocol fuzzing, which is a common way to find unknown vulnerabilities inherent in software. Nugroho and Kusumawardani in [4] demonstrated that RabbitMQ as a load-balancer can divide the workload equally, thus reducing the latency time of the Naïve Bayes Classifier classification process. Dixit and Madhu in [5] used AMQP and different types RabbitMQ exchanges to send messages to single subscriber or multiple subscribers, demonstrating the advantages of RabbitMQ as the message-oriented middleware.

Hong et al. in [6] have shown that when many users send requests to the web application at the same time, it is more stable to use RabbitMQ as the message-oriented middleware than the REST (Representational State

*Published: 30 April 2022*
Corresponding author: *Samir Lemeš (samir.lemes@unze.ba)*

Transfer) API (Application Programming Interface) communication method. Dragoni et al. in [7] presented a comprehensive literature review about the microservices, explaining in detail the importance of microservices. Indrasiri and Siriwardena in [8] described the architectural principles and how to use microservices in real-world scenarios. Akbulut and Perros in [9] obtained performance results related to query response time, efficient hardware usage, hosting costs, and packet-loss rate, for three microservice design patterns practiced in the software industry. Richter et al. in [10] evaluated the general properties of a microservice architecture and its dependability with reference to the legacy system, achieving high availability with the help of replication. Bakshi in [11] discussed the benefits and the challenges of using microservices architecture.

The objective of this paper is to demonstrate how microservices can be used to create an application for task management. The process presented can be improved by introducing nanoservices which would be called through HTTP (Hypertext Transfer Protocol) request from each microservice. Another improvement could be obtained by using Cache memory to optimize the access to multiple databases simultaneously

## 2 RabbitMQ message broker

RabbitMQ is a lightweight and scalable message broker based on AMQP [12]. RabbitMQ acts as an intermediary between the various services. It is used to reduce the load and delivery time on server web applications by delegating tasks that would normally take a lot of time and resources. The job of message comparators, such as RabbitMQ, is to ensure that messages from publishers go to the right consumers. To do this, the broker uses two key components, namely Exchange and Queue [13].

Figure 1 shows the working principle of the RabbitMQ intermediary. The process is quite simple. The publisher sends messages to the exchange, and the consumer receives messages from the queue. Before sending messages, one needs to establish all kinds of links between exchanges and queues and has to configure publisher and consumer applications.

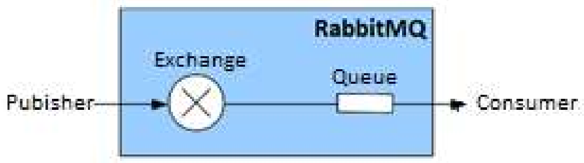

**Figure 1.** Working principle of RabbitMQ intermediary

Usually, a publisher or consumer creates an exchange with a specific name. If these are applications that are not developed by the same team of developers, the name of the exchange is usually put in the documentation or sent to another team with the appropriate documentation.

The content of an AMQP message consists of headers, properties, and data. The header and message properties are object data types. Header is defined by the AMQP specification, while the properties contain arbitrary application-specific information. Data is a sequence of bits, and each message, whether object-type, character or otherwise, requires UTF-8 conversion. Header and property conversion is not that common, but it can be done. One of the header attributes is the "routing-key" that the broker uses to pass the message to the queues. Each row contains a "binding-key" attribute, and if the values of these attributes are the same, the row receives a message.

### 2.1 Exchange types in RabbitMQ

The AMQP protocol supports multiple types of exchanges. Instead of forwarding messages directly to queues, publishers' forward messages to exchanges. The exchange oversees redirecting messages to different queues via bindings and routing keys. Binding is the link between order and exchange. The AMQP protocol defines the following types of exchanges:

- Direct – a direct exchange forwarding the message to a queue whose routing key matches the binding key.
- Topic – topic-based exchange. It is similar to direct exchange but includes a template. Queues that meet the condition from the template receive a message.
- Fanout – messages are forwarded to all queues.
- Headers – header exchange. Messages are passed in queues by header, and the routing key is ignored.

## 3 A practical example of an application

To demonstrate the creation of microservices, using the C# programming language and RabbitMQ message broker, the application "TaskApp" was created. The application is used to create and manage tasks. On the home page of the application, the user has an insight into all the tasks that need to be done. Tasks contain information about their creator, the task date, a detailed description of the task, the person in charge of its completion and the date by which the task should be performed. The user can create a new task if he is logged in to the application. When a new task is created, an email is sent to the accountable user. Users can also view their assigned tasks on the profile page and change some of their personal information.

## 3.1 RabbitMQ Infrastructure

For the development of the application, it was necessary to develop a configurational NuGet package that will retain the definitions of classes and methods to facilitate the configuration of RabbitMQ intermediaries. The package was developed to make code easier to organize, and to comply with the DRY (Don't Repeat Yourself) code writing rule. In this case, the entire RabbitMQ broker configuration code is in one place and will only be installed as a separate package in each project, thus avoiding rewriting the program code. There are three basic types in the package: message, command, and event. Code example 1 shows the message class.

**Code example 1.** Class Message

```
public class Message     {
   public readonly Guid MessageId;
   public readonly string MessageType;
   public Message() : this(Guid.NewGuid())
   {  }
   public Message(Guid messageId)   {
      MessageId = messageId;
      MessageType = this.GetType().Name; }
   public Message(string messageType) :
   this(Guid.NewGuid()) {
      MessageType = messageType;   }
      public Message(Guid messageId,
      string messageType) {
      MessageId = messageId;
      MessageType = messageType;   }
   }
```

When configuring each of the microservices, it is necessary to enter information about the RabbitMQ broker in the appsettings.json file. Depending on whether the service is used as a publisher or a consumer, two different configuration classes are used in the Startup class of each of the microservices. In the Configuration NuGet package class for the RabbitMQ broker, there are two methods that create a publisher or consumer.

Code examples 2 and 3 are methods for creating RabbitMQ consumers and publishers. Both methods use the GetRabbitMQSettings method, which provides information about the RabbitMQ broker from the appsettings.json file and use them to create new classes for consumers and publishers. In the consumer and publisher classes in the RabbitMQ broker, the connection to the RabbitMQ server is established first, and a model is created. An exchange is then created from the connection variable, whose name is inside the appsettings.json file.

**Code example 2.** Handler config method

```
public static void UseRabbitMQMessageHandler(this
 IServiceCollection services,
 IConfiguration config)
   {
    GetRabbitMQSettings(config,
    "RabbitMQHandler");
    services.AddTransient<IMessageHandler>
    (_ => new RabbitMQMessageHandler(_host,
    _userName, _password, _exchange, _queue,
    _routingKey, _port));
    }
```

**Code example 3.** Publisher config method

```
public static void UseRabbitMQMessagePublisher(this
 IServiceCollection services,
 IConfiguration config)
   {
    GetRabbitMQSettings(config,
    "RabbitMQPublisher");
    services.AddTransient<IMessagePublisher>
    (_ => new RabbitMQMessagePublisher
    (_host, _userName, _password, _exchange,
    _port));
   }
```

The exchange type is set as a fanout for easier demonstration of the application. Then rows are created, and connections are created. In the consumer class, an event is created that calls the "Consumer_Received" method of the same class, which converts the UTF-8 message into an object and calls the function that the programmer creates to perform a specific action. Also, the same method returns a confirmation to the RabbitMQ server that a message has been received (Code examples 4 and 5). In the publisher class, the message is converted to the UTF-8 equivalent, headers are added, and the message is sent via the "BasicPublish" method (Code example 6).

**Code example 4.** Part of the code to establish a connection and create an event to receive a message

```
var factory = new ConnectionFactory()
   {
     UserName = _username,
     Password = _password,
     DispatchConsumersAsync = true,
     Port = _port };
     _connection =
     factory.CreateConnection(_hosts);
     _model = _connection.CreateModel();
```

```
      _model.ExchangeDeclare(_exchange,
       "fanout", durable: true,
       autoDelete: false);

      _model.QueueDeclare(_queuename,
       durable: true, autoDelete: false,
       exclusive: false);

      _model.QueueBind(_queuename, _exchange,
       _routingKey);

      _consumer = new
       AsyncEventingBasicConsumer(_model);

      _consumer.Received +=
       Consumer_Received;

      _consumerTag =
       _model.BasicConsume(_queuename, false,
       _consumer);

   }
```

**Code example 5.** Methods for receiving and processing messages

```
private async Task Consumer_Received(object
 sender, BasicDeliverEventArgs ea)
 {
   if (await HandleEvent(ea))
    { model.BasicAck(ea.DeliveryTag, false);}
 }

   private Task<bool>
    HandleEvent(BasicDeliverEventArgs ea)
    {
       // determine messagetype
       string messageType =
        Encoding.UTF8.GetString(
        (byte[])ea.BasicProperties.Headers
        ["MessageType"]);

        // get body
        string body = Encoding.UTF8.GetString
         (ea.Body.ToArray());

         // callback to handle the message
        return _callback.HandleMessageAsync
         (messageType, body);

    }
```

**Code example 6.** The method for sending a message to the publisher

```
public Task PublishMessageAsync(
 string messageType, object message,
 string routingKey) {

   return Task.Run(() => {

      string data =
       MessageSerializer.Serialize
       (message);

      var body =
       Encoding.UTF8.GetBytes(data);

       IBasicProperties properties =
       _model.CreateBasicProperties();

       properties.Headers =
          new Dictionary<string, object> {
               {"MessageType",
                messageType } };

          _model.BasicPublish(_exchange,
          routingKey, properties, body);

          });

    }
```

### 3.2 Application arhitecture

The "TaskApp" application was created using a microservice architecture. It was written using the .NET 5 framework. The RabbitMQ message broker was used for the communication between the services. Figure 2 shows the architecture diagram of the "TaskApp" application. The application is divided into two parts, frontend and backend. The backend part of the application is divided into 6 microservices that can communicate with each other. Frontend applications communicate with four services, which include business logic for user management, task management, photo management, and authentication and authorization process management.

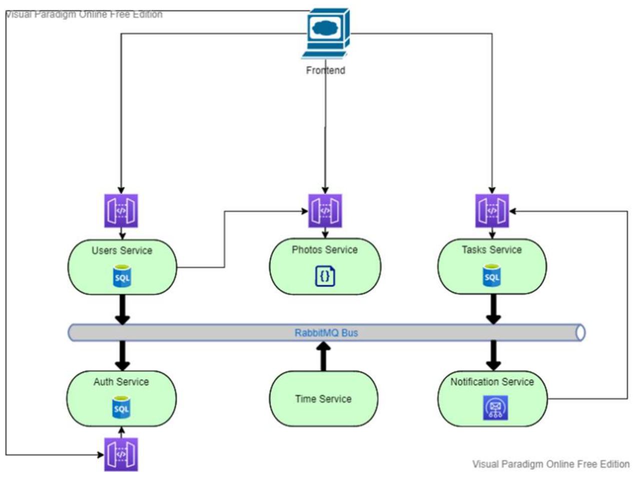

**Figure 2.** TaskApp arhitecture

#### *3.2.1 Users Microservice*

The "Users" microservice is one of the basic microservices of the "TaskApp" application. Contains an SQL (Structured Query Language) database in which user data is stored.

Microservice for users, to facilitate the demonstration of a practical solution, has one class that describes each user. The class consists of four private variables that describe: unique user identification number, user email address, and username. The "Users" service is directly dependent on the users class, and it implements four public asynchronous methods that are used to: create users, retrieve users from the database by unique identification number and email address, and retrieve all users from the database.

Figure 3 shows a sequential diagram for adding a new user. When the user fills out the registration form on the application's user interface, the frontend framework sends an HTTP POST request to the users microservice, and the "createUserAsync" method is called. The "User" object is passed to it. After the called method adds a new user to the database, the object is mapped to the "UserCreated" object, which is sent to the RabbitMQ broker. The users microservice communicates through the RabbitMQ intermediary with the "Authentication" microservice.

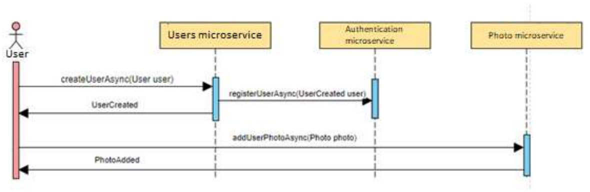
User
Users microservice
Authentication microservice
Photo microservice
createUserAsync(User user)
registerUserAsync(UserCreated user)
UserCreated
addUserPhotoAsync(Photo photo)
PhotoAdded

**Figure 3.** Sequential diagram of adding a new user

When creating a new user, in the "Users" service, the user object is forwarded to the user repository where it is placed in the SQL database. An event object is then created and passed to the PublishMessageAsync method, which converts the message to the UTF-8 equivalent and sends it to the exchange whose name is defined within the appsettings.json file (code example 7).

**Code example 7.** Add user method

```
public async Task<UserReadDto>
 AddUserAsync(CreateUser user) {

   var newUser = _mapper.Map<User>(user);

   newUser = await
    _userRepository.AddUserAsync(newUser);

      if (newUser == null) {  return null;  }

      var userCreatedEvent =
       _mapper.Map<UserCreated>(user);

      userCreatedEvent.Id = newUser.Id;

      await
       _messagePublisher.PublishMessageAsync
       (userCreatedEvent.MessageType,
       userCreatedEvent, string.Empty);

      return _mapper.Map<UserReadDto>
       (userCreatedEvent);

   }
```

The "Users" microservice communicates with "Photo" microservice. The photo of each user is processed by the "Photos" microservice and stored in database, and they are delivered when the user microservice sends a request to provide information about users.

Code example 8 shows a method that gets a user by an id. It first calls the user repository that retrieves the user from the SQL database. Then the user's object is mapped to an object suitable for displaying data, and a method is called that creates the user's initials based on the name and surname and places them in the attribute for the same. The "Photos" microservice is then called, which provides the URL (Uniform Resource Locator) of the user's photo, and the result is returned to the controller, which returns the data in JSON (JavaScript Object Notation) format to the frontend.

**Code example 8.** Method for supplying users by id

```
public async Task<UserReadDto>
 GetUserByIdAsync(int id) {

   var userFromDb = await
    _userRepository.GetUserByIdAsync(id);

   var result =
    _mapper.Map<UserReadDto>(userFromDb);

   CreateUserInitials(ref result);

   var photo = await
    _photosRestClient.GetPhotoByUserId(result.Id);

   if (photo != null) {
      result.PhotoUrl = photo.PhotoUrl; }

   return result;

   }
```

The frontend application, after completing the login form, sends an HTTP request to the authentication controller. Then, the "Authentication" service checks whether the user's email exists in the database, and whether the password corresponds to the encrypted password in the SQL database. If the check is successful, the "Authentication" microservice provides user data from the "Users" microservice and creates a JWT (JSON Web Token) that is used to secure individual application access points. As a result of the login method, user data and a token are returned (code example 9). The JWT token is returned to the user in the form of an HTTP Only cookie (code example 10). This has proven to be a good practice since cookies marked as HTTP Only cannot be accessed from a web browser, but only servers can decrypt and verify them. Previously, JWT tokens were returned in text form and placed in a session, plain cookies, or local repository, which allowed malicious users to access them more easily, and they used them to send requests to the application for their own benefit.

**Code example 9.** Method for user login and JWT generation

```
public async Task<AuthResultObject>
 AuthenticateUserAsync(string email,
 string password)   {
   var userFromDb = await
    _userManager.FindByEmailAsync(email);
   if (userFromDb == null || !await _userManager.
    CheckPasswordAsync(userFromDb, password)) {
      return null;
      }
    var user = await
     _userRestClient.GetUserByEmailAsync
     (email);
    var role = await
     _userManager.GetRolesAsync(userFromDb);
    var _options = new IdentityOptions();
    var tokenDescriptor = new
     SecurityTokenDescriptor  {
       Subject = new ClaimsIdentity (new Claim[]
          {
          new Claim("userId", user.Id.ToString()),
          new Claim(_options.ClaimsIdentity.
          RoleClaimType, role.FirstOrDefault())
          }
            ),
            Expires = DateTime.Now.AddYears(100),
            // Never Expires
            SigningCredentials = new
             SigningCredentials(new
             SymmetricSecurityKey(Encoding.UTF8.
             GetBytes(_configuration.GetSection
            ("AppSettings:Token").Value.
            ToString())),SecurityAlgorithms.
            HmacSha512Signature)
            };
      var tokenHandler =
       new JwtSecurityTokenHandler();
      var securityToken =
       tokenHandler.CreateToken(tokenDescriptor);
      var token = tokenHandler.WriteToken
       (securityToken);
      return new AuthResultObject() {
         Token = token, User = user };
      }
```

**Code example 10.** Method for generating HTTP Only cookies

```
private void setTokenCookie(string token) {
   var cookieOptions = new CookieOptions {
      HttpOnly = true, Secure = true,
      Expires = DateTime.UtcNow.AddYears(100)
      // Never Expires
      };
   Response.Cookies.Append("jwtToken",
   token, cookieOptions);
   }
```

### *3.2.2 Tasks Microservice*

The "Task" microservice is the main microservice of the "TaskApp" application. It contains a SQL database where tasks are stored. It also contains business logic for task management.The project contains a folder for Controllers, which are also an access point to the application, a folder for data that contains models, commands, events, objects for switching objects, then services, repositories, and configuration files for the database. The project includes folder "Utilities" which contains classes for configuration, constants, scripts for migrating data to SQL database, profiles for data mapping, and REST client for communication with other services.

Code example 11 provides a method for creating a task. The task is initially mapped from the creation object to the object suitable for placement in the database. The task is then forwarded to the task repository, which stores it in the database. An event is created to add a task, which receives information from the microservice for users about the user who created the task and the user to whom it was assigned. The message is sent through the RabbitMQ intermediary, and it goes to the notification system where an email is sent to the user assigned the task.

**Code example 11.** Task creation method

```
public async Task<bool>
 CreateTaskAsync(CreateTask createTask) {
   var task = _mapper.Map<Data.Models.Task>
    (createTask);
   if (await _taskRepository.CreateTaskAsync
    (task)) {
      var taskCreatedEvent =
       new TaskCreated()    {
          Title = createTask.Title,
          Description = createTask.
           Description,
          User = await
           _usersRestClient.GetUserById
           (task.UserId),
          Assignee = await
           _usersRestClient.GetUserById
           (task.AssigneeId),
          StartDate = createTask.StartDate,
          FinishDate = createTask.FinishDate
       };
    await _messagePublisher.
     PublishMessageAsync(
     taskCreatedEvent.MessageType,
     taskCreatedEvent, string.Empty);

    return true;

  }

    return false;

 }
```

Code example 12 provides a method for processing received messages for notifications microservice. There are two types of messages that a microservice expects, and these are the task creation message and the message that one business day is over. In the case of the first type of message, a "Notification" service is called that sends an email to the user assigned the task. Initially, it is necessary to enter the information about the email address and the password of the email address in the secrets.json file. The reason for entering such information in the secrets.json file instead of the appsettings.json file is additional security. The secrets.json file will not be published to public repositories, so it remains on the local developer machine. An email message is formed by the method for sending an email, and an HTML (HyperText Markup Language) template from the "Utilities" folder is called. An email is then sent via the SMTP (Simple Mail Transfer Protocol) client to the user assigned the task. In the case of a message indicating that one business day has elapsed, "Notification" service method is called that sends an email to all users whose deadline for completing the assigned task expires in less than 24 hours.

**Code example 12.** A method for processing received messages

```
public async Task<bool> HandleMessageAsync
 (string messageType, string message) {

   try {

      JObject messageObject =
       MessageSerializer.Deserialize(message);

      switch (messageType)   {

         case "TaskCreated":
            await HandleAsync(messageObject.
             ToObject<TaskCreated>());

            break;

         case "DayHasPassed":
            await HandleAsync(messageObject.
             ToObject<DayHasPassed>());

            break;

         }

       }

   catch (Exception ex) {
```

```
      Log.Error(ex, $"Error while handling
       {messageType} event.");

      }

   return true;

   }
```

## 4 Discussion

While a monolithic application is a single entity, microservice application breaks down the application components into a set of smaller, independent entities. Each unit acts as a separate service that performs its role in the system, therefore, each unit has its own business logic and database, and they perform only certain functions. In short, the style of microservice architecture is an approach to the development of an application as a set of small services, each in charge of its own process and communication with other services, often via HTTP protocol.

In general, RabbitMQ is the choice of developers looking for a simple and traditional message broker. RabbitMQ is a better choice if you need communication between individual applications via channels, that is, queues. Apache Kafka is a better choice if data retention and data streaming is required.

The main situations when programmers prefer RabbitMQ include two cases: long-term tasks, when reliable background work is required, and communication with integration between applications as an intermediary between microservices. Developers choose Apache Kafka over RabbitMQ when there is a need for a framework for storing, reading (re-reading) and analysing streaming data, data analysis systems, and real-time data processing systems.

This example demonstrated that the style of microservice architecture is an approach to the development of an application as a set of small services, each in charge of its own process and communication with other services.

## 5 Conclusion

RabbitMQ is an open-source solution that serves as a message broker via the AMQP. It is highly accessible, fault-tolerant, and scalable.

RabbitMQ is a lightweight and scalable message broker based on AMQP. RabbitMQ acts as an intermediary between the various services. It is used to reduce the load and delivery time on server web applications by delegating tasks that would normally take a lot of time and resources.

## References

[1] Opensource.com, "What are microservices?," [Online]. Available: https://opensource.com/resources/what-are-microservices. [Accessed 29 1 2022].

[2] G. Olah, "An introduction to RabbitMQ – What is RabbitMQ?," Erlan Solutions, 3 4 2020. [Online]. Available: https://www.erlang-solutions.com/blog/an-introduction-to-rabbitmq-what-is-rabbitmq/. [Accessed 29 1 2022].

[3] S. Kwon, S.J. Son, Y. Choi, and J.H. Lee, "Protocol fuzzing to find security vulnerabilities of RabbitMQ," in *Concurrency and Computation: Practice and Experience*, vol. 33, no. 23, 2021, pp. 1-14.

[4] A. Nugroho, and S.S. Kusumawardani, "Distributed Classifier for SDGs Topics in Online News using RabbitMQ Message Broker," *Journal of Physics: Conference Series,* vol. 1577, no. 1, pp. 1-8, 2020.

[5] S. Dixit, and M. Madhu, "Distributing messages using Rabbitmq with advanced message exchanges," *Int. J. Res. Stud. Comput. Sci. Eng.*, vol. 6, no. 2, pp. 24-28, 2019.

[6] X.J. Hong, H.S. Yang, and Y.H.Kim, "Performance analysis of RESTful API and RabbitMQ for microservice web application," In *2018 International Conference on Information and Communication Technology Convergence (ICTC),* IEEE, 2018, pp. 257-259.

[7] N. Dragoni et al. "Microservices: Yesterday, Today, and Tomorrow," *In: Mazzara, M., Meyer, B. (eds) Present and Ulterior Software Engineering*, pp. 195-216, 2047.

[8] K. Indrasiri and P. Siriwardena, Microservices for the Enterprise: Designing, Developing, and Deploying, USA: Apress, 2022.

[9] A. Akbulut, and H.G. Perros, "Performance Analysis of Microservice Design Patterns," *IEEE Internet Computing*, vol. 23, no. 6, pp. 19-27, Nov. 2019.

[10] D. Richter, M. Konrad, K. Utecht, and A. Polze, "Highly-Available Applications on Unreliable Infrastructure: Microservice Architectures in Practice". in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2017, pp. 130-137.

[11] K. Bakshi "Microservices-based software architecture and approaches", *2017 IEEE Aerospace Conference*, 2017, pp. 1-8, doi: 10.1109/AERO.2017.7943959.

[12] L. Johansson, "When to use RabbitMQ or Apache Kafka", CloudAMQP, 2019. [Online]. Available: https://www.cloudamqp.com/blog/when-to-use-rabbitmq-or-apache-kafka.html. [Accessed 20 1 2022].

[13] S. Newman, Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, Inc, USA, 2015.