

Asp.NETCore RazorPage, WebApi, MVC + Rabbit

PHIND-70B MODEL

Let's explore how to architect a modern ASP.NET Core application combining Razor Pages, WebAPI, MVC, and RabbitMQ messaging. This architecture allows you to leverage the strengths of each component while maintaining clean separation of concerns.

Architecture Overview

The proposed architecture integrates multiple ASP.NET Core frameworks with RabbitMQ for reliable message handling. Each component serves a specific purpose:

- Razor Pages handles straightforward CRUD operations and forms
- WebAPI provides REST endpoints for external services
- MVC manages complex UI workflows
- RabbitMQ enables asynchronous communication and background processing

The diagram above illustrates the complete system architecture where:

- Blue components represent frontend interfaces (Razor Pages and MVC)
- Green components show core business logic (WebAPI and models)
- Purple indicates RabbitMQ-related components
- Orange represents data storage

The RabbitMQ components work together as follows:

1. The Message Producer component sends messages from WebAPI to RabbitMQ queues
2. Messages are stored temporarily in RabbitMQ until processed
3. The Message Consumer component retrieves messages and updates domain models

Implementation Details

Project Structure

```
src/
├── WebApp/
│   ├── Areas/
│   │   ├── RazorPages/
│   │   └── Pages/
│   └── Controllers/
└── Api/
```

```
| | | Mvc/
| | | Models/
| | | Services/
| | Tests/
```

## Core Components Implementation

1

## RabbitMQ Integration

```
public static class RabbitMqConfig
{
    public static void AddRabbitMq(this IServiceCollection services, IConfiguration
configuration)
    {
        var rabbitMqSettings =
configuration.GetSection("RabbitMq").Get<RabbitMqSettings>();

        services.AddSingleton<IConnection>(sp =>
        {
            var factory = new ConnectionFactory()
            {
                HostName = rabbitMqSettings.Hostname,
                Port = rabbitMqSettings.Port,
                Username = rabbitMqSettings.Username,
                Password = rabbitMqSettings.Password
            };

            return factory.CreateConnection();
        });
    }
}

public class RabbitMqSettings
```

```

{
    public string Hostname { get; set; }
    public int Port { get; set; }
    public string Username { get; set; }
    public string Password { get; set; }
}

```

2

## Message Producer Service

```

public interface IMessageProducer
{
    Task PublishAsync<T>(string exchange, string routingKey, T message);
}

public class MessageProducer : IMessageProducer
{
    private readonly IConnection _connection;

    public MessageProducer(IConnection connection)
    {
        _connection = connection;
    }

    public async Task PublishAsync<T>(string exchange, string routingKey, T message)
    {
        using var channel = _connection.CreateModel();
        var body = Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(message));

        var properties = new BasicProperties
        {
            Persistence = true,
            Type = "text/json"
        }
    }
}

```

```
};
```

```
await channel.BasicPublishAsync(exchange: exchange,
```

```
routingKey: routingKey,
```

```
body: body,
```

```
basicProperties: properties);
```

```
}
```

```
}
```

3

## Message Consumer Service

```
public interface IMessageConsumer
```

```
{
```

```
void StartConsuming<T>(string queueName, Func<T, Task> onMessageReceived);
```

```
}
```

```
public class MessageConsumer : IMessageConsumer
```

```
{
```

```
private readonly IConnection _connection;
```

```
private IModel _channel;
```

```
public MessageConsumer(IConnection connection)
```

```
{
```

```
_connection = connection;
```

```
_channel = connection.CreateModel();
```

```
}
```

```
public void StartConsuming<T>(string queueName, Func<T, Task>  
onMessageReceived)
```

```
{
```

```
_channel.QueueDeclare(queue: queueName,
```

```
durable: true,
```

```
exclusive: false,
```

```
autoDelete: false);
```

```
var consumer = new EventingBasicConsumer(_channel);
```

```
consumer.Received += async (model, ea) =>
```

```
{
```

```
var body = ea.Body.ToArray();
```

```
var message =
```

```
JsonConvert.DeserializeObject<T>(Encoding.UTF8.GetString(body));
```

```
await onMessageReceived(message);
```

```
_channel.BasicAck(ea.DeliveryTag, false);
```

```
};
```

```
_channel.BasicConsume(queue: queueName,
```

```
autoAck: false,
```

```
consumer: consumer);
```

```
}
```

```
}
```

## Usage Example

```
// In your controller
```

```
public class OrderController : ControllerBase
```

```
{
```

```
private readonly IMessageProducer _producer;
```

```
public OrderController(IMessageProducer producer)
```

```
{
```

```
_producer = producer;
```

```
}
```

```
[HttpPost]
```

```
public async Task<IActionResult> PlaceOrder(Order order)
```

```
{
```

```
// Process order synchronously
```

```
await _producer.PublishAsync("orders_exchange",
```

```
"order.created",
```

```
order);
```

```
return Accepted();
```

```
}
```

```
}
```

```
// In your Razor Page
```

```
public class OrderModel : PageModel
```

```
{
```

```
private readonly IMessageProducer _producer;
```

```
[BindProperty]
```

```
public Order Order { get; set; }
```

```
public OrderModel(IMessageProducer producer)
```

```
{
```

```
_producer = producer;
```

```
}
```

```
public async Task<IActionResult> OnPostAsync()
```

```
{
```

```
await _producer.PublishAsync("orders_exchange",
```

```
"order.created",
```

```
Order);
```

```
return RedirectToPage("./Index");
```

```
}
```

```
}
```

Best Practices

1

## Message Handling

- Implement retry policies for failed messages
- Use dead-letter queues for unprocessable messages
- Consider implementing message acknowledgment patterns

2

## Error Handling

- Log all message processing failures
- Implement circuit breakers for RabbitMQ connections
- Monitor queue lengths and consumer lag

3

## Performance Optimization

- Batch messages when possible
- Use connection pooling for RabbitMQ connections
- Implement proper prefetch settings for consumers

|  
|

## Sources

1  **Integrate ASP.NET Core Razor components with MVC or Razor Pages**

Razor components can be integrated into Razor Pages or MVC a...

[learn.microsoft.com](https://learn.microsoft.com)

2  **Razor pages and webapi in the same project - Stack Overflow**

I created a web app (razor pages) in .net core 3.0. Then I a...

[stackoverflow.com](https://stackoverflow.com)

3  **Getting Started with RabbitMQ in ASP.NET**

In this example we'd like to make our user creation a backgr...

[aspnetmonsters.com](https://aspnetmonsters.com)

4  **I left the Web Form and switched to Razor Pages. | by Runaho | CodeX**

In this article, I will talk about my thoughts about .NET Co...

[medium.com](https://medium.com)

+1 more

Переведи все это на русский

PHIND-70B MODEL

Давайте рассмотрим архитектуру современного веб-приложения ASP.NET Core, объединяющую Razor Pages, WebAPI, MVC и RabbitMQ для обработки сообщений. Такая архитектура позволяет использовать преимущества каждого компонента при сохранении чистого разделения ответственности.

## Обзор Архитектуры

Предлагаемая архитектура интегрирует несколько фреймворков ASP.NET Core с использованием RabbitMQ для надежной обработки сообщений. Каждый компонент выполняет определенную задачу:

- Razor Pages обрабатывает простые операции CRUD и формы
- WebAPI предоставляет REST-конечные точки для внешних сервисов
- MVC управляет сложными рабочими процессами интерфейса
- RabbitMQ обеспечивает асинхронное взаимодействие и фоновой обработку данных

На диаграмме выше показана общая структура системы, где:

- Синие компоненты представляют собой фронтенд (страницы Razor и представления MVC)
- Зеленые компоненты отображают бизнес-логику (WebAPI и модели)
- Фиолетовые компоненты связаны с RabbitMQ
- Оранжевые компоненты представляют хранилище данных

Компоненты RabbitMQ работают следующим образом:

1. Производитель сообщений отправляет данные из WebAPI в очереди RabbitMQ
2. Сообщения временно хранятся в RabbitMQ до обработки
3. Потребитель сообщений получает их и обновляет модели домена

## Детали Реализации

### Структура Проекта

```
src/
├── WebApp/
├── Areas/
│   ├── RazorPages/
│   └── Pages/
├── Controllers/
├── Api/
└── Mvc/
```



Models/

Services/

Tests/

## Основные Компоненты

1

## Интеграция RabbitMQ

```
public static class RabbitMqConfig
{
    public static void AddRabbitMq(this IServiceCollection services, IConfiguration
configuration)
    {
        var rabbitMqSettings =
configuration.GetSection("RabbitMq").Get<RabbitMqSettings>();

        services.AddSingleton<IConnection>(sp =>
        {
            var factory = new ConnectionFactory()
            {
                HostName = rabbitMqSettings.Hostname,
                Port = rabbitMqSettings.Port,
                Username = rabbitMqSettings.Username,
                Password = rabbitMqSettings.Password
            };

            return factory.CreateConnection();
        });
    }
}

public class RabbitMqSettings
{
}
```

```

    public string Hostname { get; set; }

    public int Port { get; set; }

    public string Username { get; set; }

    public string Password { get; set; }
}

```

2

## Сервис Производителя Сообщений

```

public interface IMessageProducer
{
    Task PublishAsync<T>(string exchange, string routingKey, T message);
}

public class MessageProducer : IMessageProducer
{
    private readonly IConnection _connection;

    public MessageProducer(IConnection connection)
    {
        _connection = connection;
    }

    public async Task PublishAsync<T>(string exchange, string routingKey, T message)
    {
        using var channel = _connection.CreateModel();

        var body = Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(message));

        var properties = new BasicProperties
        {
            Persistence = true,
            Type = "text/json"
        };
    }
}

```

```
await channel.BasicPublishAsync(exchange: exchange,
```

```
routingKey: routingKey,
```

```
body: body,
```

```
basicProperties: properties);
```

```
}
```

```
}
```

3

### Сервис Потребителя Сообщений

```
public interface IMessageConsumer
```

```
{
```

```
void StartConsuming<T>(string queueName, Func<T, Task> onMessageReceived);
```

```
}
```

```
public class MessageConsumer : IMessageConsumer
```

```
{
```

```
private readonly IConnection _connection;
```

```
private IModel _channel;
```

```
public MessageConsumer(IConnection connection)
```

```
{
```

```
_connection = connection;
```

```
_channel = connection.CreateModel();
```

```
}
```

```
public void StartConsuming<T>(string queueName, Func<T, Task>  
onMessageReceived)
```

```
{
```

```
_channel.QueueDeclare(queue: queueName,
```

```
durable: true,
```

```
exclusive: false,
```

```
autoDelete: false);
```

```
var consumer = new EventingBasicConsumer(_channel);
```

```
consumer.Received += async (model, ea) =>
```

```
{
```

```
var body = ea.Body.ToArray();
```

```
var message =
```

```
JsonConvert.DeserializeObject<T>(Encoding.UTF8.GetString(body));
```

```
await onMessageReceived(message);
```

```
_channel.BasicAck(ea.DeliveryTag, false);
```

```
};
```

```
_channel.BasicConsume(queue: queueName,
```

```
autoAck: false,
```

```
consumer: consumer);
```

```
}
```

```
}
```

## Пример Использования

```
// В вашем контроллере
```

```
public class OrderController : ControllerBase
```

```
{
```

```
private readonly IMessageProducer _producer;
```

```
public OrderController(IMessageProducer producer)
```

```
{
```

```
_producer = producer;
```

```
}
```

```
[HttpPost]
```

```
public async Task<IActionResult> PlaceOrder(Order order)
```

```
{
```

```
// Обработка заказа синхронно
```

```
await _producer.PublishAsync("orders_exchange",
```

```
"order.created",
```

```
order);
```

```
return Accepted();
```

```
}
```

```
}
```

```
// В вашей странице Razor
```

```
public class OrderModel : PageModel
```

```
{
```

```
private readonly IMessageProducer _producer;
```

```
[BindProperty]
```

```
public Order Order { get; set; }
```

```
public OrderModel(IMessageProducer producer)
```

```
{
```

```
_producer = producer;
```

```
}
```

```
public async Task<IActionResult> OnPostAsync()
```

```
{
```

```
await _producer.PublishAsync("orders_exchange",
```

```
"order.created",
```

```
Order);
```

```
return RedirectToPage("./Index");
```

```
}
```

```
}
```

Лучшие Практики

## Обработка Сообщений

- Реализуйте политику повторных попыток для неудачных сообщений
- Используйте очередь мертвых писем для непроцессуемых сообщений
- Рассмотрите реализацию паттернов подтверждения сообщений

## 2

## Обработка Ошибок

- Логируйте все сбойные обработки сообщений
- Внедрите предохранители для подключений к RabbitMQ
- Отслеживайте длину очередей и задержку потребителей

## 3

## Оптимизация Производительности

- Группируйте сообщения при возможности
- Используйте пул соединений с RabbitMQ
- Настройте правильные параметры предварительной выборки для потребителей

Задача вот такая. Есть Asp.Net Приложение Feeder - сервис котировок, который - генерит котировки ( DateTime DT, Tciker string, Bid double, Ask double, Last double, VolumeInLast double) - передает котировки остальным Asp.Net Приложениям. - другие Asp.Net слушают шину и принимают эти котировки Все работает в реальном времени. Для того чтобы контролировать правильную работу сервиса Котировок необходимо передавать котировки не только все Потребителям, но также и себе самому и визуализировать эти котировки на графике. Мы можем замкнуть этот цикл на уровне AspNet Core - контроллеров - Первый вариант. Второй вариант мы можем закольцевать на уровне Rabbit, то есть подписаться на свои же котировки. Как должна выглядеть архитектура такого Приложения ? 1. Asp.NetCore Feeder -> Сервис -> котировки -> контроллер Asp.NetCore Feeder -> RazorPage визуализация 2. Asp.NetCore Feeder -> Сервис -> котировки -> Rabbit -> контроллер Asp.NetCore Feeder -> RazorPage визуализация ----- Какая должна быть архитектура Asp.NetCore Приложения - в Пером случае - без Rabbit - во втором случае с Rabbit ----- Предложи варианты архитектурно. Кода пока не нужно. Если есть вопросы - Задавай

## PHIND-70B MODEL

Давайте рассмотрим архитектурные варианты для сервиса котировок с возможностью визуализации данных.

## Вариант 1: Архитектура без RabbitMQ

В этом варианте все компоненты находятся внутри одного ASP.NET Core приложения, что обеспечивает простоту реализации и минимальную задержку при передаче данных.

В этой архитектуре:

- Генератор котировок (GC) создает новые данные в реальном времени
- Сервис котировок (CS) обрабатывает и хранит данные
- Контроллер котировок (CC) управляет передачей данных между сервисом и фронтом
- Кэш используется для оптимизации производительности при частых обновлениях
- Razor Pages обеспечивает визуализацию данных в реальном времени

Преимущества этого подхода:

- Минимальная задержка при передаче данных
- Простота реализации и поддержки
- Низкие требования к инфраструктуре
- Легко масштабируется в рамках одного сервера

## Вариант 2: Архитектура с RabbitMQ

Этот вариант использует RabbitMQ как шину обмена сообщениями, что обеспечивает надежную асинхронную передачу данных и масштабируемость.

В этой архитектуре:

- Генератор котировок отправляет данные через производителя сообщений в RabbitMQ
- RabbitMQ обеспечивает надежную доставку данных всем подписчикам
- Потребитель сообщений получает данные из RabbitMQ и передает их в сервис котировок
- Кэш используется для оптимизации производительности при частых обновлениях
- Razor Pages получает данные через контроллер котировок

Преимущества этого подхода:

- Высокая надежность доставки данных
- Возможность масштабирования по горизонтали
- Гарантированная обработка всех сообщений
- Возможность добавления новых потребителей без изменения существующей архитектуры

Сравнение вариантов

ХАРАКТЕРИСТИКА	ВАРИАНТ 1 (БЕЗ RABBITMQ)	ВАРИАНТ 2 (С RABBITMQ)
Сложность реализации	Простая	Средняя
Задержка доставки	Минимальная	Немного выше
Масштабируемость	Ограничена одним сервером	Высокая
Надежность	Зависит от приложения	Высокая
Требования к инфраструктуре	Минимальные	Требуется сервер
Обработка ошибок	Простая	Профессиональная

Рекомендации

Для вашего случая рекомендуется использовать Вариант 2 (с RabbitMQ), потому что:


1. Обеспечивает надежную доставку котировок всем потребителям, включая визуализацию

|


|

Sources


1

 Building a Robust Chat Application with .NET Core 8, SignalR, and ...  
The ChatRoomWithBot project is a browser-based chat applicat...  
medium.com

2

 Building Microservices On .NET Core – Part 6 Real time server client ...  
In this article we are going to show you how you can combine...  
altkomsoftware.com

3

 Event-Driven Microservices with .NET Core and RabbitMQ - Medium  
An effective way to manage communication between these micro...  
medium.com

Первый вариант неправильно понят В этом варианте Котировки отправляются Потребителям (например через SignalR) и самому себе. В этом варианте котировки тоже пересекают границы Сервиса, но возвращаются обратно без Rabbit. В этом случае сервис работает и как Pub и Sub, и визуализирует полученные котировки на Графике. То есть работает также как другие внешние Клиенты. Внешние Клиенты



получают котировки через SignalR и отображают их на своем Графике. Тоже самое делает и PubSuber - рисует свою работу, которую он отправили себе.

## PHIND-70B MODEL

Вы абсолютно правы. Давайте рассмотрим правильную архитектуру первого варианта, где сервис работает как паблишер и одновременно как подписчик своих же котировок.

В этой архитектуре:

- SignalR Hub работает как центральный узел для распределения котировок
- Сервис котировок одновременно является паблишером (отправляет котировки) и подписчиком (получает их обратно)
- Кэш используется для оптимизации производительности при частых обновлениях
- Razor Pages получает данные через контроллер котировок

Рассмотрим, как работает система:

### 1. Генерация и отправка котировок:

- Генератор котировок создает новые данные
- Сервис котировок получает эти данные
- Сервис отправляет котировки через SignalR Hub всем подписчикам

### 2. Внутреннее потребление:

- Тот же сервис котировок подписывается на свои собственные котировки
- Полученные котировки сохраняются в кэш
- Контроллер котировок получает данные из кэша
- Razor Pages визуализирует котировки

### 3. Работа с внешними клиентами:

- Внешние клиенты подключаются к тому же SignalR Hub
- Они получают те же самые котировки
- Визуализация происходит на стороне клиента

Таким образом, сервис работает как единое целое, где:

- Отправка и получение котировок происходит через один и тот же SignalR Hub
- Внутренняя визуализация использует те же данные, что и внешние клиенты
- Архитектура остается простой и понятной
- Нет необходимости в дополнительных компонентах типа RabbitMQ

Этот подход обеспечивает:

- Единообразие обработки данных
- Простоту поддержки и отладки
- Высокую производительность
- Надежную доставку котировок всем подписчикам

Как происходит двустороннее движение данных между сервисом котировок и другими компонентами системы?

## PHIND-70B MODEL

Давайте рассмотрим, как происходит двустороннее движение данных в системе котировок. Для этого сначала визуализируем потоки данных:

На диаграмме показаны все ключевые компоненты системы и их взаимодействие:

- Генератор котировок создает новые данные
- Сервис котировок обрабатывает и распределяет данные
- SignalR Hub обеспечивает двустороннее распространение данных
- Кэш оптимизирует доступ к данным
- Контроллер котировок управляет передачей данных на визуализацию
- Razor Pages отображает данные в реальном времени
- Внешние клиенты получают те же данные через SignalR

Рассмотрим подробно каждый поток данных:

### 1. Отправка котировок:

- Генератор котировок создает новую котировку
- Сервис котировок получает данные и обрабатывает их
- Через SignalR Hub данные отправляются всем подписчикам
- Внешние клиенты получают данные и отображают их на своих графиках

### 2. Возврат данных в сервис:

- SignalR Hub возвращает те же данные обратно в сервис котировок
- Данные сохраняются в кэш для оптимизации доступа
- Контроллер котировок получает данные из кэша
- Razor Pages получает данные для визуализации

### 3. Обновление данных:

- Razor Pages запрашивает обновление данных
- Контроллер котировок запрашивает данные из кэша
- Кэш возвращает актуальные данные
- График на Razor Pages обновляется

Такая архитектура обеспечивает:

- Единообразное распространение данных всем подписчикам
- Оптимизированный доступ к данным через кэш
- Надежную доставку данных всем компонентам системы
- Возможность масштабирования системы
- Простоту поддержки и отладки

**Важно отметить, что сервис котировок работает как единое целое, где отправка и получение данных происходит через один и тот же SignalR Hub, что обеспечивает согласованность данных во всей системе.**