Getting Started with .NET Core, Docker, and RabbitMQ — Part 3

Matthew Harper Following
Oct 25, 2019 · 7 min read

Picking up from Part2, we're going add the final component of our application — a message queue. The specifics of this example will seem bit contrived, but the goal is to decouple our publisher and worker, use Docker to launch a RabbitMQ container, and use Docker Compose to orchestrate the system. And of course, go line by line through the changes required to accomplish these tasks.

https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html

Why use a message queue?

Microservices are all the rage in today's distributed systems. But an unfortunate side effect of breaking down monolithic applications into multiple single-purpose services is high coupling between the new services. What appears to a client as a single RESTful API call may actually become many calls between interdependent services behind the scenes. If one critical service goes down, the negative effects (latency and more failures) will quickly be felt throughout a highly coupled system.

This is where message queues come into play. Instead of direct HTTP connections between services, we introduce a middleman. Services can publish messages to a queue, and consumers will pull messages from that queue and process them. This can allow asynchronous communication for long-running tasks (enqueue and forget), or the caller can enqueue the message and wait for a consumer to return a response (Remote Procedure Call pattern). Either way, we've decoupled the two services, in the sense that if one goes down, the message still persists in the queue, and as long as we've scaled horizontally (multiple copies of the services) on both sides of the queue, we don't have a single point of failure (besides the queue itself, but there are ways to ensure the queue is highly available).

Let's write some code!

. . .

We finished Part2 of this series with two applications running in Docker containers and orchestrated with Docker-Compose. Our 'worker' app calls our 'publisher_api' app via RESTful web service, and prints the response. Our next goal is to modify the program so that when we post a message from the worker to the publisher, instead of receiving an immediate response, we'll simulate a situation where the server needs more time to process the request. The worker will not wait synchronously for the response to each request; it will fire off messages, and then poll our message queue awaiting responses.

Hopefully you've completed part 2 of the tutorial so we can pick up right where we left off. If you need the code for a starting point, look for the v2.0 tag in the Github repo.

To get started, we're going to add a rabbitMQ container to our project by modifying docker-compose.yml. Here's the updated file:

```
version: '3.4'

services:
  publisher_api:
    build: ./publisher_api
    restart: always

  worker:
    build: ./worker
    restart: always
    depends_on:
      - "publisher_api"
      - "rabbitmq"

  rabbitmq: # login guest:guest
    image: rabbitmq:3-management
    hostname: "rabbitmq"
    labels:
      NAME: "rabbitmq"
    ports:
    - "4369:4369"
    - "5671:5671"
    - "5672:5672"
    - "25672:25672"
    - "15671:15671"
    - "15672:15672"
```

docker-compose.yml hosted with ♡ by **GitHub** **view raw**

docker-compose.yml

What we've done above is tell docker-compose to add a new service to our application, named rabbitmq. It pulls an existing image (rabbitmq:3-management) from the public Docker repository.; this image includes the message queue and the management console that you can access in a web browser. We've also exposed the ports RabbitMQ operates on and told our other services to always restart on shutdown or failure. Docker-compose

doesn't manage the startup order of services terribly well, so we'll just keep restarting our worker/publisher until the queue is in place.

Next, we want to modify our code to use the message queue. Let's import the NuGet package for the RabbitMQ client. Navigate to the publisher_api directory in a command prompt, and run the following command:

```
dotnet add package RabbitMQ.Client
```

cmd hosted with ♡ by GitHub view raw

Now we will create a class to post messages to RabbitMQ, Create a new folder inside the publisher_api directory called services, and a new file there called MessageService.cs. Here's the file:

```
using System;
using System.Text;
using RabbitMQ.Client;

namespace publisher_api.Services
{
    // define interface and service
    public interface IMessageService
    {
        bool Enqueue(string message);
    }

    public class MessageService : IMessageService
    {
        ConnectionFactory _factory;
        IConnection _conn;
        IModel _channel;
        public MessageService()
        {
            Console.WriteLine("about to connect to rabbit");

            _factory = new ConnectionFactory() { HostName = "rabbitmq", Port = 5672 };
            _factory.UserName = "guest";
            _factory.Password = "guest";
            _conn = _factory.CreateConnection();
            _channel = _conn.CreateModel();
            _channel.QueueDeclare(queue: "hello"
```

```
            _channel.QueueDeclare(queue: "hello",
                                  durable: false,
                                  exclusive: false,
                                  autoDelete: false,
                                  arguments: null);
        }
        public bool Enqueue(string messageString)
        {
            var body = Encoding.UTF8.GetBytes("server processed " + messageString);
            _channel.BasicPublish(exchange: "",
                                  routingKey: "hello",
                                  basicProperties: null,
                                  body: body);
            Console.WriteLine(" [x] Published {0} to RabbitMQ", messageString);
            return true;
        }
    }
}

```

MessageService.cs hosted with ♡ by GitHub view raw

Let's walk through the code. You'll first note that we import the RabbitMQ.Client package, and that we define a namespace to hold our project's services (publisher_api.Services). Next, we declare an interface (IMessageService)that our new class (MessageService) implements. Interfaces are valuable in .NET Core because they make it easier to create unit tests against your classes, and because they allow other classes in your application to depend on abstractions rather than concrete types (more about this when we discuss dependency injection later.)

Our actual MessageService class only implements a single public method — Enqueue. But there's some setup that takes place in the constructor. We create a ConnectionFactory object to establish and return a connection to the RabbitMQ server. Then we use a channel to send and receive messages over that connection. The end result is that we invoke the QueueDeclare method on our channel — this will create a queue named "hello" if one does not already exist, and establish a connection to it.

In our Enqueue method, we create the string we want to return to our client ("server processed" plus the original message), and invoke BasicPublish on the channel. We're

not specifying an exchange, so we leave that parameter as an empty string, and our routingKey is the name of the queue we're sending the message to ("hello".)

Now, we want to modify the POST handler in ValuesController.cs to publish to our message queue. But how do we access MessageService from ValuesController? We don't want to create a new instance of MessageService every time a POST request comes in — that's not going to scale well. This is where we need Dependency Injection. In short, Dependency Injection is a pattern that allows you to decouple pieces of your application and helps us achieve the Dependency Inversion Principle (higher-level modules should not depend on lower-level modules, and modules should depend on abstractions rather than on concrete details.) My colleague Chris Tanghere has put together a deep dive on Dependency Injection if you want more info.

To configure our application to access MessageService via Dependency Injection, we have to modify Startup.cs. Add a using directive to reference Publisher_api.Services to the top of the file. Then look for the ConfigureServices method, and add the following line to the end of the method, after AddMvc is invoked:

```
1  services.AddSingleton<IMessageService, MessageService>();
```

Startup.cs hosted with ♡ by GitHub view raw

What we're doing is basically telling our program that if anyone requests an instance of type IMessageService, they will be given a reference to a singleton MessageService object. This means our MessageService class will only be instantiated once, no matter how many times it is referenced.

Now let's modify ValuesController.cs to take advantage of Dependency Injection and enqueue our message via our MessageService. Again, add the using directive for publisher_api.Services to the top of the file. Then, modify the class and it's constructor to inject an instance of IMessageService, and store a reference to it in a private member variable:

```
1  private readonly IMessageService _messageService;
2
3  public ValuesController(IMessageService messageService)
4  {
```

```
        _messageService = messageService;
    }
```

ValuesController.cs hosted with ♡ by GitHub view raw

All we have to do now is modify the POST method to invoke Enqueue on our MessageService member variable:

```
[HttpPost]
public void Post([FromBody] string payload)
{
    Console.WriteLine("received a Post: " + payload);
    _messageService.Enqueue(payload);
}
```

ValuesController.cs hosted with ♡ by GitHub view raw

Publisher_api is complete. If you want to test the application now, you can run “docker-compose up — build” from our solution directory, and you should see output similar to this:

```
rabbitmq_1       |  * rabbitmq_management
rabbitmq_1       |  * rabbitmq_management_agent
rabbitmq_1       |  * rabbitmq_web_dispatch
rabbitmq_1       |  completed with 3 plugins.
worker_1         | Posting a message!
publisher_api_1  | warn: Microsoft.AspNetCore.HttpsPolicy.HttpsRedirectionMiddleware[3]
publisher_api_1  |       Failed to determine the https port for redirect.
publisher_api_1  | about to connect to rabbit
rabbitmq_1       | 2019-10-24 14:49:56.376 [info] <0.576.0> accepting AMQP connection <0.576.0> (172.18.0.3:37517 -> 172.18.0.4:5672)
rabbitmq_1       | 2019-10-24 14:49:56.408 [info] <0.576.0> connection <0.576.0> (172.18.0.3:37517 -> 172.18.0.4:5672): user 'guest' authenticated and
 granted access to vhost '/'
publisher_api_1  | received a Post: test message
publisher_api_1  |  [x] Published test message to RabbitMQ
worker_1         | Server returned:
```

The key items there are:

- rabbitmq_1 container completes initialization successfully
- worker posts a message to the publisher
- publisher_api receives a message and publishes it to RabbitMQ

Once we verify that works, let’s kill the docker-compose process (Ctrl-C) and modify the worker to poll our message queue for responses. We’ll need to import the NuGet package for the RabbitMQ client, just as we did for the publisher_api. Navigate to the worker directory in a command prompt, and run the following command:

```
dotnet add package RabbitMQ.Client
```

cmd hosted with ♡ by GitHub view raw

Now let's get into the code. In worker/Program.cs, add a couple of using directives from the RabbitMQ package:

```
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
```

Program.cs hosted with ♡ by GitHub view raw

Our PostMessage method will remain the same, we'll just modify the Main method to post 5 messages, then listen to our MessageQueue and await the responses:

```
 static void Main(string[] args)
{
    string[] testStrings = new string[] {"one", "two", "three", "four", "five"};

    Console.WriteLine("Sleeping to wait for Rabbit");
    Task.Delay(10000).Wait();
    Console.WriteLine("Posting messages to webApi");
    for(int i = 0; i < 5; i++)
    {
        PostMessage(testStrings[i]).Wait();
    }

    Task.Delay(1000).Wait();
    Console.WriteLine("Consuming Queue Now");

    ConnectionFactory factory = new ConnectionFactory() { HostName = "rabbitmq", Port = 5672 };
    factory.UserName = "guest";
    factory.Password = "guest";
    IConnection conn = factory.CreateConnection();
    IModel channel = conn.CreateModel();
    channel.QueueDeclare(queue: "hello",
                         durable: false,
                         exclusive: false,
                         autoDelete: false,
                         arguments: null);

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

```
28      consumer.Received += (model, ea) =>
29      {
30          var body = ea.Body;
31          var message = Encoding.UTF8.GetString(body);
32          Console.WriteLine(" [x] Received from Rabbit: {0}", message);
33      };
34      channel.BasicConsume(queue: "hello",
35                           autoAck: true,
36                           consumer: consumer);
37  }
```

Program.cs hosted with ♡ by GitHub view raw

The Wait() call at the start of the function is a hack to give the RabbitMQ container more time to initialize before we start sending messages. Then we loop, sending each of our test strings to the publisher_api. After that, it's time to poll the message queue and retrieve the responses.

First, we set up a connection to the queue, in the same fashion that we did for the publisher_api. The major difference is that we set up an EventingBasicConsumer object. This class exposes RabbitMQ's message handling functions as events. We add a handler to the 'Received' event where we retrieve the message body and print it to the console. Then we call the BasicConsume method on our channel object, supplying the name of the queue to poll and the EventingBasicConsumer object we just created.

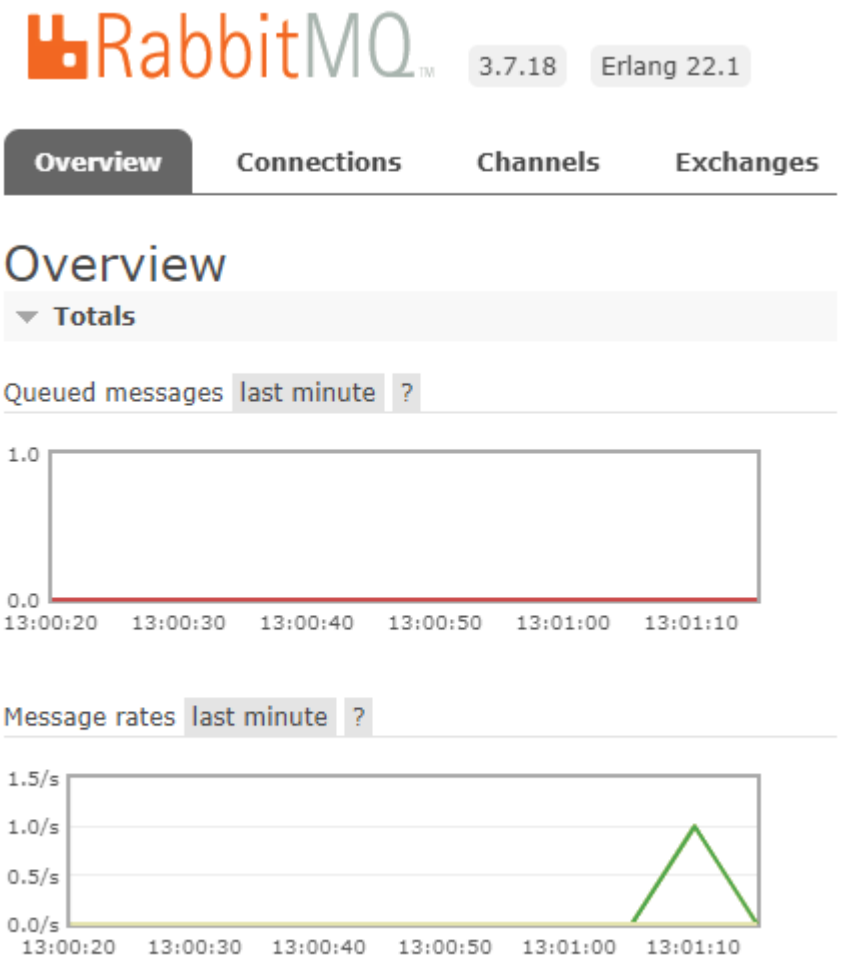
Head back to the cmd prompt and docker-compose again (don't forget the build flag to ensure your containers include the code changes you just made.) Your output should be similar to:

```
rabbitmq_1         |  completed with 3 plugins.
rabbitmq_1         | 2019-10-24 16:56:41.570 [info] <0.8.0> Server startup complete; 3 plugins started.
rabbitmq_1         |  * rabbitmq_management
rabbitmq_1         |  * rabbitmq_management_agent
rabbitmq_1         |  * rabbitmq_web_dispatch
worker_1           | Posting messages to webApi
publisher_api_1    | warn: Microsoft.AspNetCore.HttpsPolicy.HttpsRedirectionMiddleware[3]
publisher_api_1    |       Failed to determine the https port for redirect.
publisher_api_1    | about to connect to rabbit
rabbitmq_1         | 2019-10-24 16:56:46.690 [info] <0.576.0> accepting AMQP connection <0.576.0> (172.18.0.2:37867 -> 172.18.0.3:5672)
rabbitmq_1         | 2019-10-24 16:56:46.718 [info] <0.576.0> connection <0.576.0> (172.18.0.2:37867 -> 172.18.0.3:5672): user 'guest' authenticated and
 granted access to vhost '/'
publisher_api_1    | received a Post: one
publisher_api_1    |  [x] Published one to RabbitMQ
publisher_api_1    | received a Post: two
publisher_api_1    |  [x] Published two to RabbitMQ
publisher_api_1    | received a Post: three
publisher_api_1    |  [x] Published three to RabbitMQ
publisher_api_1    | received a Post: four
publisher_api_1    |  [x] Published four to RabbitMQ
publisher_api_1    | received a Post: five
publisher_api_1    |  [x] Published five to RabbitMQ
worker_1           | Consuming Queue Now
rabbitmq_1         | 2019-10-24 16:56:47.851 [info] <0.591.0> accepting AMQP connection <0.591.0> (172.18.0.4:36781 -> 172.18.0.3:5672)
rabbitmq_1         | 2019-10-24 16:56:47.881 [info] <0.591.0> connection <0.591.0> (172.18.0.4:36781 -> 172.18.0.3:5672): user 'guest' authenticated and
 granted access to vhost '/'
worker_1           |  [x] Received from Rabbit: server processed one
worker_1           |  [x] Received from Rabbit: server processed two
```

```
worker_1        | [x] Received from Rabbit: server processed three
worker_1        | [x] Received from Rabbit: server processed four
worker_1        | [x] Received from Rabbit: server processed five
```

You should see the publisher_api receive five POSTs, and the worker consume the message queue, printing out five messages it retrieved. Another way you can verify the queue is functional is to visit the RabbitMQ Management Console by visiting localhost:15672 in your web browser. This site will only be accessible while your docker-compose is running because it is hosted in the RabbitMQ container.

Once you login to the site with name: guest and password: guest (defined when you created the queue in MessageService.cs), you should see some statistics. The clearest indication of activity is the message rates graph; as you can see below, there was a small spike in activity when our worker posted the five messages.



• • •

That's the end of this series — to recap, we built .NET core applications, containerized them, decoupled them using RabbitMQ, and used docker-compose to orchestrate all the pieces of this project. You can find all of the code in the Github repo. Please share any feedback, and also any ideas you'd like to see explored in future articles. Thanks!

Interested in joining our team at Trimble Maps? Click here!

Dotnet Core Csharp Docker Rabbitmq Web Development