**Chapter 4. Event-driven microservices: It’s not just request/response**

*This chapter covers*

* Using the request/response programming model
* Using the event-driven programming model
* Considering both models for cloud-native software
* Understanding the models’ similarities and differences
* Using Command Query Responsibility Segregation

One of the main pillars on which cloud-native software stands is microservices. Breaking what once was a large, monolithic application into a collection of independent components has shown many benefits, including increased developer productivity and more-resilient systems, provided the right patterns are applied to these microservice-based deployments. But an overall software solution is almost never made up of a single component; instead, microservices are brought together into a collection to form a rich digital offering.

But here’s the risk: if you aren’t careful, you could glue the components back together in such a way that the microservices you’ve created only give the illusion of loose coupling. You have to be careful not to re-create the monolith by coupling the individual pieces too tightly or too early. The patterns covered through the remainder of this book are designed to avoid this pitfall and to produce robust, agile software as a collection of independent components, brought together in a way that maximizes agility and resilience.

But before we jump deep into those topics, I need to cover one more umbrella topic—a cross-cutting concern against which all of the other cloud-native patterns will be applied: *the basic invocation style used in your software architecture*. Will the interaction between microservices be in a request/response or an event-driven style? With the former, a client makes a request and expects a response. Although the requestor may allow for that response to come asynchronously, the very expectation that a response will come establishes a direct dependence of one on the other. With the latter, the parties consuming events can be completely independent from those producing them. This autonomy gets to the heart of the difference between these two styles of interaction.

Large, complex software deployments employ both approaches—sometimes request/response, sometimes event-driven. But I suggest the factors that drive our choices are far more nuanced than we’ve likely given them credit for in the past. And once again, the highly distributed and constantly changing cloud context our software is now running in brings an added dimension that requires us to reexamine and challenge our previous understandings and assumptions.

In this chapter, I start with the style that seems to be the most natural for most developers and architects: request/response. It’s so natural that you might not have even noticed that the basic model I presented in [chapter 1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01) subtly favors it. I then challenge that bias and introduce the event-driven model in our cloud context. Event-driven thinking is fundamentally different from request/response, so the implications are great. You’ll study these two models through code samples, and as a result will broaden your thinking about the interaction portion of our cloud-native software mental model.

**4.1. We’re (usually) taught imperative programming**

The vast majority of students, whether learning to program in a classroom setting or via any of the many sources available online, learn imperative programming. They learn languages such as Python, Node.js, Java, C#, and Golang, most of which are designed to allow the programmer to provide a series of instructions that are executed from start to finish. Sure, control structures allow for branching and looping, and some instructions will be calls to procedures or functions, but even the logic within a loop or a function, for example, will execute from the top to the bottom.

You surely see where I’m going with this. This sequential programming model drives you to think in a request/response manner. As you’re executing a set of instructions, you make a request of a function, anticipating a response. And in the context of a program that’s executing in a single process, this works well. In fact, procedural programming has dominated the industry for nearly a half century.**[**[**1**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fn1)**]** As long as the programming process remains up and running, someone making a request of a function can reasonably expect a response from a function also running in that same process.

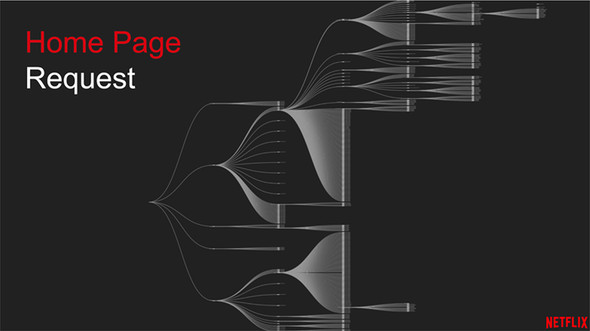
***1***

*For more information on procedural programming, see Wikipedia at*[*http://mng.bz/lp56*](http://mng.bz/lp56)*.*

But our software as a whole is no longer executing in a single process. Much of the time, different pieces of our software aren’t even running on the same computer. In the highly distributed, constantly changing environment of the cloud, a requestor can no longer depend on an immediate response when a request is made. Despite this, the request/response model has still dominated as the main programming paradigm for the web. True, with the availability of React.js and other similar frameworks, reactive programming is becoming more commonplace for code running in the browser, but server-side programming remains heavily dominated by request/response.

For example, [figure 4.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig01) shows a significant fan-out of requests to dozens of microservices that occurs when a user accesses their Netflix homepage. This slide was taken from a presentation that Senior Software Engineer Scott Mansfield has given at numerous conferences, in which he talks about patterns Netflix uses to compensate for the cases when a response from a downstream request isn’t immediately forthcoming.

**Figure 4.1. Diagram appearing in a presentation from Scott Mansfield of Netflix shows a single request to retrieve the homepage for a user results in a significant fan-out of requests to downstream microservices.**



I include this particular diagram because it does a great job illustrating the magnitude of the problem. If the homepage request were successful only when all of the cascading requests depicted in this tree were also successful, Netflix would have a great many frustrated customers. Even if each microservice could boast five nines of availability (99.999%), *and* the network was always up,**[**[**2**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fn2)**]** a single request with fewer than 100 downstream request/response dependencies loses two nines of availability, or approximately 99.9% at the root. I haven’t seen estimates of the revenue loss to Netflix when its website is offline, but if we go back to the *Forbes* estimates of the economic impact to Amazon’s bottom line, the resultant 500 minutes of downtime would cost Amazon $80 million yearly.

***2***

*See #1 on the list of fallacies of distributed computing at Wikipedia (*[*http://mng.bz/BD90*](http://mng.bz/BD90)*).*

Of course, Netflix, and many other highly successful web properties, do far better than that by implementing patterns like automatic request retries when responses aren’t received, and having multiple running instances of microservices that can fulfill requests. And as Scott Mansfield has expertly presented, caching can also help provide a bulkhead between clients and services.**[**[**3**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fn3)**]** These patterns and many more are built around the request/response invocation style, and although you can and will continue fortifying this basis with additional techniques, you should also consider a different foundation for building these resilience patterns.

***3***

*See “Caching at Netflix: The Hidden Microservice” on YouTube (*[*http://mng.bz/dPvN*](http://mng.bz/dPvN)*).*

**4.2. Reintroducing event-driven computing**

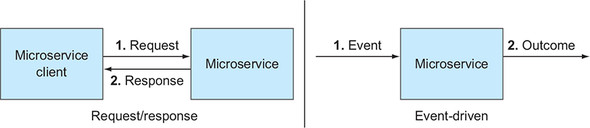
When you get into the details, you may find differing opinions on what makes up an event-driven system.**[**[**4**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fn4)**]** But even through this variability, one thing is common. The entity that triggers code execution in an event-driven system doesn’t expect any type of response—*it’s fire and forget*. The code execution has an effect (otherwise, why would you run it at all), and the outcome may cause other things to happen with the software solution, but the entity that triggered the execution doesn’t expect a response.

***4***

*“What Do You Mean by Event-Driven?” by Martin Fowler (*[*http://mng.bz/YPla*](http://mng.bz/YPla)*) explores this topic further.*

This concept is easily understood with a simple diagram, particularly when you set it in contrast to the request/response pattern. The left side of [figure 4.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig02) depicts a simple request and response: the code that’s executed when a request is received is on the hook for providing some type of a response to the requestor. By contrast, the right-hand side of this figure shows an event-driven service: the outcome of the code execution has no direct relationship to the event that triggered it.

**Figure 4.2. Contrast the base primitive of request/response to event-driven invocation styles.**



A couple of points are interesting to note in these two diagrams. First, on the left, two parties are involved in the dance: the microservice client and the microservice itself. Partners in the dance depend on each other to make things go smoothly. On the right-hand side, only one party is depicted, and this is significant. The microservice executes as a result of an event, but what triggered that event isn’t of concern to the microservice at all. As a result, the service has fewer dependencies.

Second, notice that the event and the outcome are entirely disconnected. The lack of coupling between the former and the latter even allows me to draw the arrows on different sides of the microservice. That’s something I couldn’t do in the request/response style on the left.

The implications of these differences run fairly deep. The best way for you to start to wrap your head around them is with a concrete example. Let’s jump into the first bit of code in the book.

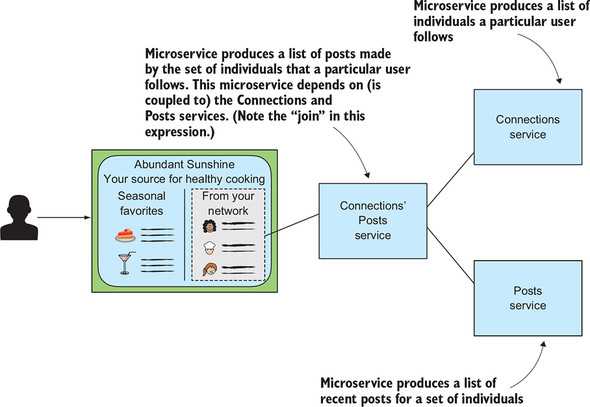
**4.3. My global cookbook**

I love to cook, and I spend far more time browsing food-related blogs than I care to admit. I have my favorite bloggers ([https://food52.com](https://food52.com/) and [https://smittenkitchen.com](https://smittenkitchen.com/), I’m looking at you) as well as my favorite “official” publications ([www.bonappetit.com](http://www.bonappetit.com/)). What I want to do now is build a site that pulls together content from all of my favorite sites and allows me to organize that content. Yeah, basically I want a blog aggregator, but perhaps something that’s specialized for my *addiction*, er, hobby.

One of the content views I’m interested in is a list of the latest posts that come from my favorite sites: given a network of people or sites that I follow, what are their latest posts? I’m going to call this set of content the *Connections’ Posts* content, and it’ll be produced by a service I’m going to write. Two parts come together to form this content: a list of the people or sites that I follow and a list of content provided by those individuals. The content for each of these two parts is provided by two additional services.

[Figure 4.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig03) depicts the relationship between these components. This diagram doesn’t depict any particular protocol between the various microservices, but only the relationships between them. This is a perfect example for you to have a deeper look at the two protocols: request/response and event-driven.

**Figure 4.3. The Abundant Sunshine website will display a list of posts made by my favorite food bloggers. The aggregation is a composition of the network of people I follow and posts made by those individuals.**

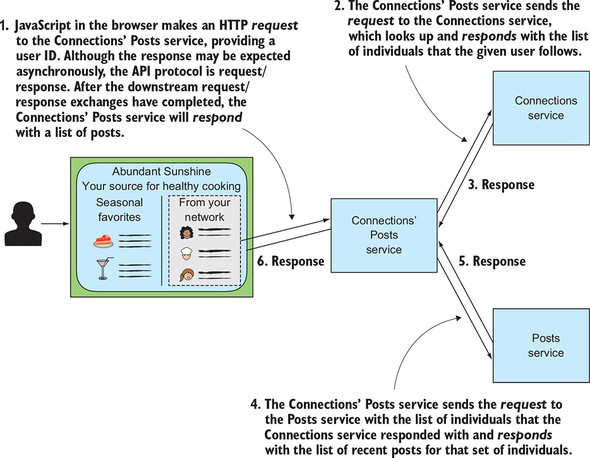


**4.3.1. Request/response**

As I talked about earlier in the chapter, using the request/response protocol to concretely draw together the components depicted in [figure 4.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig03) is, for most people, the most natural. When we think about generating a set of posts written by my favorite bloggers, it’s easy to say, let’s first get a list of the people I like, and then look up the posts that each of those individuals has made. Concretely, the flow progresses as follows (see [figure 4.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig04)):

1. JavaScript in the browser makes a request to the Connections’ Posts service, providing an identifier for the individual who has requested the web page (let’s say that’s you), and waits for a response. Note that the response may be returned asynchronously, but the invocation protocol is still request/response in that a response is expected.
2. The Connections’ Posts service makes a request to the Connections service with that identifier and waits for a response.
3. The Connections service responds with the list of bloggers that you follow.
4. The Connections’ Posts service makes a request to the Posts service with that list of bloggers just returned from the Connections service, and awaits a response.
5. The Posts service responds with a list of posts for that set of bloggers.
6. The Connections’ Posts service creates a composition of the data it has received in the responses and itself responds to the web page with that aggregation.

**Figure 4.4. Rendering a portion of the web page depends on a series of coordinated requests and responses.**



Let’s have a look at the code that implements the steps I’ve depicted in [figure 4.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig04).

**Setting up**

This and most of the examples throughout the book require you to have the following tools installed:

* Maven
* Git
* Java 1.8

I won’t require you to write any code. You need only check it out of GitHub and execute a few commands to build and run the applications. Although this isn’t a programming book, I use code throughout to demonstrate the architectural principles that I cover.

**Obtaining and building the microservices**

You’ll begin by cloning the cloudnative-abundantsunshine repository with the following command and then changing into that directory:

git clone https://github.com/cdavisafc/cloudnative-abundantsunshine.git

cd cloudnative-abundantsunshine

Here you’ll see subdirectories containing code samples that appear in various chapters throughout this text. The code for this first example is located in the cloudnative-requestresponse directory, so I’ll have you step one level deeper into the project with the following:

cd cloudnative-requestresponse

You’ll drill into the source code of the example in a moment, but first let’s get you up and running. The following command builds the code:

mvn clean install

**Running the microservices**

You’ll now see that a new JAR file, cloudnative-requestresponse-0.0.1-SNAPSHOT.jar, has been created in the target subdirectory. This is what we call a *fat jar*—the Spring Boot application is completely self-contained, including a Tomcat container. Therefore, to run the application, you need only run Java, pointing to the JAR:

java -jar target/cloudnative-requestresponse-0.0.1-SNAPSHOT.jar

The microservices are now up and running. In a separate command-line window, you can curl the Connections’ Posts service:

curl localhost:8080/connectionsposts/cdavisafc

To obtain a response:

[

{

"date": "2019-01-22T01:06:19.895+0000",

"title": "Chicken Pho",

"usersName": "Max"

},

{

"date": "2019-01-22T01:06:19.985+0000",

"title": "French Press Lattes",

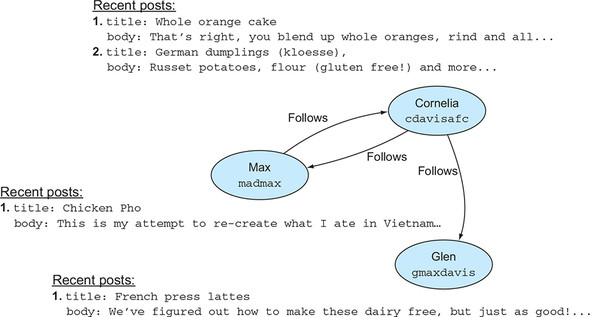
"usersName": "Glen"

}

]

As part of starting this application, I’ve prepopulated several databases with sample content. This response represents a list of posts by individuals that I, cdavisafc, follow. In this case, one post is titled “Chicken Pho,” written by someone whose name is Max; and a second post is titled “French Press Lattes,” written by someone whose name is Glen. [Figure 4.5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig05) shows how the three sample users are connected as well as the posts that each user has recently made.

**Figure 4.5. Users, the connections between them, and the posts each has recently made**



Indeed, you can see this data reflected by invoking the Connections’ Posts service for each of the users:

curl localhost:8080/connectionsposts/madmax

curl localhost:8080/connectionsposts/gmaxdavis

|  |
| --- |
|  |

**A note on the project structure**

I’ve bundled the implementations of each of the three services into the same JAR, but let me be clear: in any real setting, this would be completely discouraged. One of the advantages of a microservices-based architecture is the existence of bulkheads between the services, so that failures in one don’t cascade to others. The implementation here has none of those bulkheads. If the Connections service crashes, it will take the other two services with it.

I start the implementation with this antipattern in place for two reasons. First, it allows you to get the code sample running with a minimum number of steps; I’ve taken a shortcut for simplicity. This approach also will allow you to clearly see the benefits as you refactor the implementation by applying patterns that serve cloud-native architectures well.

|  |
| --- |
|  |

**Studying the code**

The Java program that you’re running implements all three of the microservices depicted in [figure 4.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig04). I’ve organized the code for the implementation into four packages, each a subpackage of the com.corneliadavis.cloudnative package:

* The *config* package contains the Spring Boot application and configuration, as well as a bit of code that fills the databases with sample data.
* The *connections* package contains the code for the Connections microservice, including domain objects, data repositories, and controllers.
* The *posts* package contains the code for the Posts microservice, including domain objects, data repositories, and controllers.
* The *connectionsposts* package contains the code for the Connections’ Posts microservice, including domain objects and controllers (note, no data repository).

One package draws all the pieces together into a single Spring Boot application and contains some utility implementations. Then you have one package for each of the three microservices that make up the digital solution.

The Connections and Posts microservices are similar in structure. Each contains classes that define the domain objects for the service, as well as interfaces used by Spring’s Java Persistence API (JPA) implementation to generate databases that store the content for objects of each type. Each package also contains the controller that implements the service and the core functionality of the microservice. These two microservices are basic CRUD services: they allow objects to be created, read, updated, and deleted, and data is persisted in a database.

The microservice of most interest in this first implementation is Connections’ Posts because it doesn’t only store data into and retrieve data from a database, but it also calculates a composite result. Looking at the contents of the package, you can see there are only two classes: a domain object called PostSummary and a controller.

The PostSummary class defines an object containing fields for the data that the Connections’ Posts service will return: for each post, it returns the title and date, and the name of the individual who made the post. There’s no JPA repository for this domain object because the microservice only uses it in memory to hold the results of its computation.

The ConnectionsPosts controller implements a single public method—the one that’s executed when a request is made to the service with an HTTP GET. Accepting a username, the implementation requests from the Connections service the list of users being followed by this individual, and when it receives that response, makes another HTTP request to the Posts service with that set of user IDs. When the response from the request to Posts is received, the composite result is produced. [Figure 4.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig06) presents the code for that microservice, annotated with the steps detailed in [figure 4.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig04).

**Figure 4.6. The composite result generated by the Connections’ Posts microservice is produced by making calls to the Connections and Posts microservices and aggregating results.**



For steps 2 and 3 and for steps 4 and 5, the Connections’ Posts microservice is acting as a client to the Connections and Posts microservices, respectively. Clearly, you have instances of the protocol depicted on the left-hand side of [figure 4.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig02)—the request/response. If you look closely, however, you’ll see that there’s one more instance of this pattern. The composite result includes the name of the user who made the post, but this data is returned neither from the request to Connections, nor from the request to Posts; each response includes only user IDs. Admittedly naïve, at the moment the implementation retrieves the name for each user of each post by making a set of additional HTTP requests to the Connections service. In a moment, when you move to an event-driven approach, you’ll see that these extra calls will go away on their own.

Okay, so this basic implementation works reasonably well, even if it could use some optimizations for efficiency. But it’s rather fragile. To generate the result, the Connections microservice needs to be up and running, as does the Posts microservice. And the network must also be stable enough for all of the requests and responses to execute without any hiccups. Proper functioning of the Connections’ Posts microservice is heavily dependent on many other things functioning correctly; it’s not truly in control of its own destiny.

Event-driven architectures, to a large extent, are designed to address the problem of systems that are too tightly coupled. Let’s now look at an implementation that satisfies the same requirements on the Connections’ Posts microservice but with a different architecture and level of resilience.

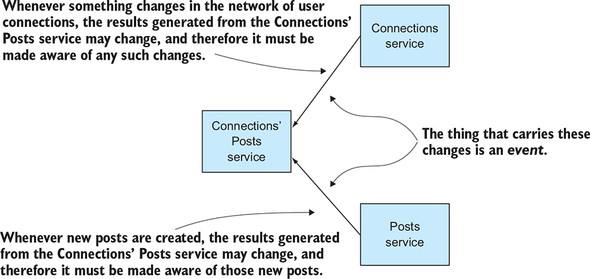
**4.3.2. Event-driven**

Instead of code being executed only when someone or some entity makes a request, in an event-driven system, code is executed when something happens. What that “something” is can vary wildly, and could even be a user request, but the main idea with event-driven systems is that events cause code to be executed that may, in turn, generate events that further flow through the system. The best way to understand the fundamentals is with a concrete example, so let’s take the same problem we’ve just solved in the request/response style and refactor it to be event-driven.

Our end goal is still to have a list of posts made by the people I follow. In that context, what are the events that could affect that result? Certainly, if one of the individuals I follow publishes a new post, that new post would need to be included in my list. But changes in my network will also affect the result. If I add to or remove from the list of individuals I am following, or if one of those individuals changes their username, that could also result in changes to the data produced by the Connections’ Posts service.

Of course, we have microservices that are responsible for posts and for user connections; they keep track of the state of these objects. In the request/response approach you just looked at, those microservices manage the state of these objects, and when requested, serve up that state. In our new model, those microservices are still managing the state of these objects, but are more proactive and generate events when any of that state changes. Then, based on the software topology, those events affect the Connections’ Posts microservice. This relationship is depicted in [figure 4.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig07).

**Figure 4.7. Events are the means through which related microservices are connected.**



Of course, you’ve already seen relationships between Connections’ Posts, Connections, and Posts in [figures 4.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig03) and [4.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig04), but of significance here is the direction of the arrows. As I said, the Connections and Posts microservices are *proactively* sending out change notifications rather than waiting to be asked. When the Connections’ Posts service is asked for data, it *already knows the answer*. No downstream requests are needed.

I want to dig into these implications in more detail, but first, let’s look at the code that implements this pattern. Doing so will help you make your way to this fundamentally different way of thinking.

**Setting up**

As with all the samples in this book, you’ll need the following tools installed on your workstation:

* Maven
* Git
* Java 1.8

**Obtaining and building the microservices**

If you haven’t already done so, you must clone the cloudnative-abundantsunshine repository with the following command:

git clone https://github.com/cdavisafc/cloudnative-abundantsunshine.git

cd cloudnative-abundantsunshine

The code for this example is housed in the cloudnative-eventdriven subdirectory, so you’ll need to move to that directory:

cd cloudnative-eventdriven

You’ll drill into the source code of the example in a moment, but first let’s get you up and running. The following command builds the code:

mvn clean install

**Running the microservices**

Just as before, you’ll see that a new JAR file, cloudnative-eventdriven-0.0.1-SNAPSHOT.jar, has been created in the target subdirectory. This is a fat jar—the Spring Boot application is completely self-contained, including a Tomcat container. Therefore, to run the application, you need only run Java, pointing to the JAR:

java -jar target/cloudnative-eventdriven-0.0.1-SNAPSHOT.jar

The microservices are now up and running. As before, I’ve loaded sample data during startup so that you can, in a separate command-line window, curl the Connections’ Posts service:

curl localhost:8080/connectionsposts/cdavisafc

You should see exactly the same output that you did when running the request/response version of the application:

[

{

"date": "2019-01-22T01:06:19.895+0000",

"title": "Chicken Pho",

"usersName": "Max"

},

{

"date": "2019-01-22T01:06:19.985+0000",

"title": "French Press Lattes",

"usersName": "Glen"

}

]

If you didn’t go through the exercise before, please look at [section 4.3.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04lev2sec1) for a description of the sample data and the other API endpoints that are available for you to send requests to in exploring the sample data. Each of the three microservices implements the same interface as before; they vary only in implementation.

I now want to demonstrate how the events we just identified (the creation of new posts and the creation of new connections) will change what is produced by the Connections’ Posts microservice. To add a new post, execute the following command:

curl -X POST localhost:8080/posts \

-d '{"userId":2,

"title":"Tuna Onigiri",

"body":"Sushi rice, seaweed and tuna. Yum..."}' \

--header "Content-Type: application/json"

Execute the original command again:

curl localhost:8080/connectionsposts/cdavisafc

This yields the following:

[

{

"date": "2019-01-22T05:36:44.546+0000",

"usersName": "Max",

"title": "Chicken Pho"

},

{

"date": "2019-01-22T05:41:01.766+0000",

"usersName": "Max",

"title": "Tuna Onigiri"

},

{

"date": "2019-01-22T05:36:44.648+0000",

"usersName": "Glen",

"title": "French Press Lattes"

}

]

**Studying the code**

This is exactly what you’d expect, and the same steps executed against the request/response implementation would yield just this. In that prior implementation, shown in [figure 4.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig06), you can clearly see how the new result is generated with calls first to obtain the list of followed individuals and then to obtain the posts for those individuals.

To see how things work in the event-driven implementation, you have to look in several places. Let’s first look at the implementation of the aggregating service—Connections’ Posts:

**Listing 4.1. Method from ConnectionsPostsController.java**

@RequestMapping(method = RequestMethod.***GET***,

value=**"/connectionsposts/{username}"**)

public Iterable<PostSummary> getByUsername(

@PathVariable("username") String username,

HttpServletResponse response) {

Iterable<PostSummary> postSummaries;

*logger*.info("getting posts for user network " + username);

postSummaries = mPostRepository.findForUsersConnections(username);

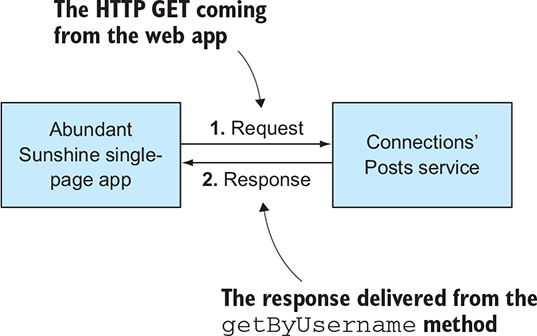
**return** postSummaries;

}

Yes, that’s it. That’s the entire implementation. To generate the result for the Connections’ Posts microservice, the only thing that the getByUsername method does is a database query. What makes this possible is that changes that have an effect on the output of the Connections’ Posts service are known to that service before a request to it is made. Rather than waiting for a request to Connections’ Posts to learn about any new posts that have been created, our software is designed to proactively make such changes known to the Connections’ Posts service through the delivery of an event.

I don’t want to get into the details of the database that the getByUsername method is querying right now. I cover it in detail in [chapter 12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12), when I talk about cloud-native data. For the moment, just know that the Connections’ Posts service has a database that stores the state that’s the result of propagated events. Summarizing, the getByUsername method *returns* a result when a *request* has come in from the Abundant Sunshine single-page app, as shown in [figure 4.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig08); you’ll recognize this as the pattern you see on the left-hand side of [figure 4.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig02). What is important is to note that the Connections’ Posts service can generate its response without depending on any other services.

**Figure 4.8. When a request is received, the Connections’ Posts service can generate a result without depending on other services in the system.**



Let’s now turn to that event that the Connections’ Posts service benefitted from, starting with the endpoint on the Posts service that allows for new posts to be created. You’ll find the newPost method in the com.corneliadavis.cloudnative.posts.write package:

**Listing 4.2. Method from PostsWriteController.java**

@RequestMapping(method = RequestMethod.***POST***, value="**/posts**")

public void newPost(@RequestBody Post newPost,

HttpServletResponse response) {

*logger*.info("Have a new post with title " + newPost.getTitle());

if (newPost.getDate() == null)

newPost.setDate(new Date());

postRepository.save(newPost);

*//event*

RestTemplate restTemplate = new RestTemplate();

ResponseEntity<String> resp =

restTemplate.postForEntity("http://localhost:8080/connectionsposts/posts",

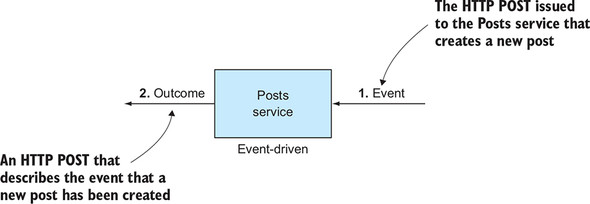
newPost, String.class);

*logger*.info("[Post] resp " + resp.getStatusCode());

}

The Posts service is first taking the HTTP POST event and storing the data for that post in the Posts repository; this is the primary job of the Posts service, to implement the create and read operations for blog posts. But because it’s part of an event-driven system, it also generates an event so that it may be picked up by any other services for which it’s relevant. In this particular example, that event is represented as an HTTP POST to a party that’s interested in that event—namely, the Connections’ Posts service. This bit of code implements the pattern shown in [figure 4.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig09), which you’ll recognize as an instance of the right-hand side of [figure 4.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig02).

**Figure 4.9. The Posts service not only persists the new post that was created, but also delivers an event reporting that this post has been created.**



Turning now to the recipient of that event, let’s look at the newPost method in the Connections’ Posts service; you’ll find this in the com.corneliadavis.cloudnative.newpostsfromconnections.eventhandlers package:

**Listing 4.3. Method from EventsController.java of Connections’ Posts service**

@RequestMapping(method = RequestMethod.***POST***, value=**"/posts"**)

public void newPost(@RequestBody Post newPost, HttpServletResponse

response) {

*logger*.info("[NewFromConnections] Have a new post with title "

+ newPost.getTitle());

MPost mPost = new MPost(newPost.getId(),

newPost.getDate(),

newPost.getUserId(),

newPost.getTitle());

MUser mUser;

mUser = mUserRepository.findOne(newPost.getUserId());

mPost.setmUser(mUser);

mPostRepository.save(mPost);

}

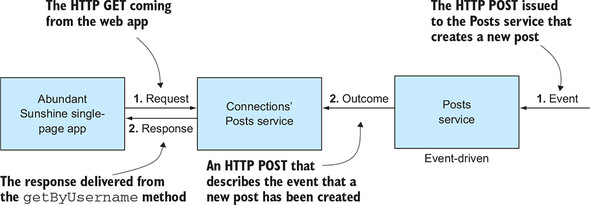
As you can see, that method simply takes the event as the body of an HTTP POST and stores the result of that event for future use. Remember that when requested, the Connections’ Posts service need only do a database query to generate the result. This is the code that allows that to happen. You’ll notice that the Connections’ Posts service is interested in only a few of the fields from the message it receives, the ID, date, user ID, and title, storing those in a locally defined post object. It also establishes the right foreign-key relation between this post and the given user.

Wow—there’s a lot in there. Most of the elements of this solution are covered in great depth later in the book. For example, the fact that each microservice in this solution has its own data stores, and the point about Connections’ Posts needing only a subset of the content in the post event—these are topics in their own right. But don’t worry about those details right now.

What I do want to draw your attention to at this juncture is the independence of the three microservices. When the Connections’ Posts service is invoked, it doesn’t reach out to the Connections or Posts services. Instead, it operates on its own; it’s autonomous. It will function even if a network partition cuts Connections and Posts off in the very moment of the request to Connections’ Posts.

I also want to point out that the Connections’ Posts service is handling both requests and events. When you issued the curl for the list of posts from individuals a given user follows, the service generated a response. But when the new post event was generated, the service handled that event without a response. Instead, it generated only a specific outcome of storing the new post in its local repository. It generated no further event. [Figure 4.10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig10) composes the patterns depicted in [figure 4.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig02) to diagram what I’ve just laid out here.

**Figure 4.10. Microservices can implement both request/response and event-driven patterns. Events are primarily used to loosely couple microservices.**



You’ll notice in this diagram that the two services are loosely coupled. Each executes autonomously. The Posts microservice does its thing anytime it receives a new post, generating a subsequent event. The Connections’ Posts microservice processes requests by simply querying its local data stores.

|  |
| --- |
|  |

**You got me!**

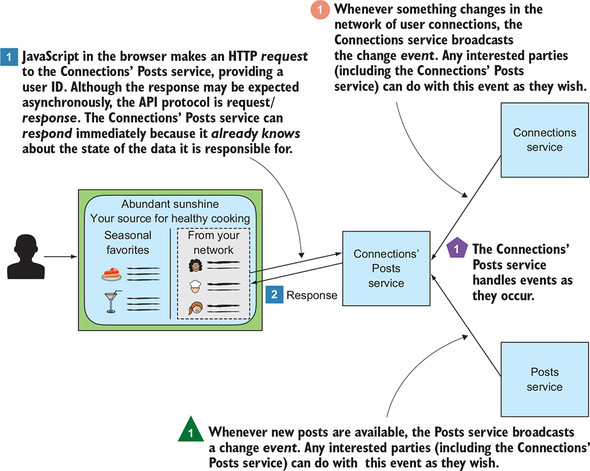
You might be thinking that my claims of loose coupling are a bit exaggerated, and with the current implementation you’re absolutely correct. The far-too-tight binding exists with the implementation of the arrow labeled “Outcome” in [figure 4.10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig10). In the current implementation, I’ve implemented that “event” from the Posts microservice as an HTTP POST that makes a call directly to the Connections’ Posts microservice. This is brittle; if the latter is unavailable when the former issues that POST, our event will be lost and our system is broken.

Ensuring that this event-driven pattern works in a distributed system requires more sophistication than this, and those are exactly the techniques that are covered in the remainder of this book. For now, I’ve implemented the example in this manner only for simplicity.

|  |
| --- |
|  |

Putting all of the pieces together, [figure 4.11](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig11) shows the event-driven architecture of our sample application. You can see that each microservice is operating independently. Notice the many “1” annotations; there’s little sequencing. Just as you saw in the Posts example, when events affecting users and connections occur, the Connections service does its work, storing data and generating an event for the Connections’ Posts service. When events affecting the outcome of the Connections’ Posts service occur, the event handler does the work of drawing those changes into its local data stores.

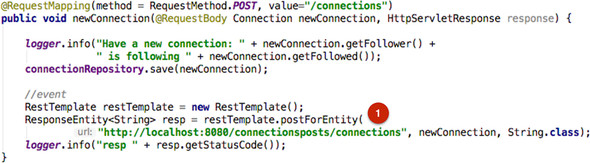
**Figure 4.11. Rendering the web page now only requires execution of the Connections’ Posts microservice. Through event handling, it already has all of the data it needs to satisfy the request, so no downstream requests and responses are required.**



What’s particularly interesting is that the work of aggregating data from the two sources has shifted. With the request/response model, it’s implemented in the NewFromConnectionsController class. With the event-driven approach, it’s implemented through the generation of events and through the implementation of event handlers.

The code corresponding to the annotations on the microservices shown in [figure 4.11](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig11) appear in the three figures that follow. [Figure 4.12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig12), from the Connections service, shows the ConnectionsWriteController. [Figure 4.13](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig13), from the Posts service, shows the PostsWriteController. And [figure 4.14](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig14), from the Connections’ Posts service, shows the EventsController (event handler).

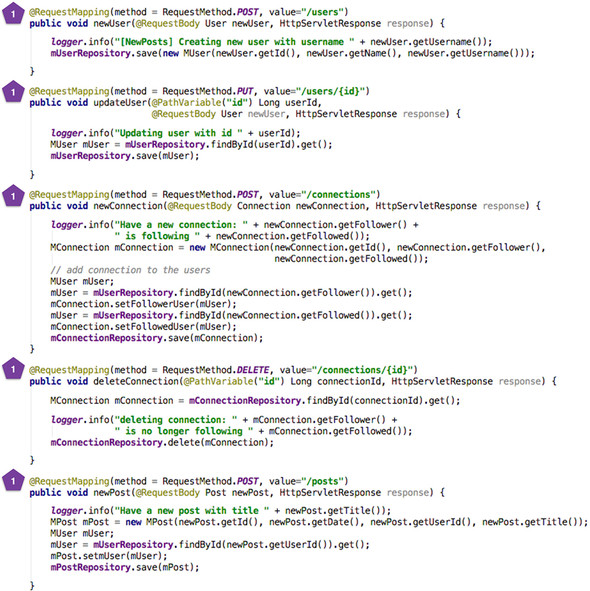
**Figure 4.12. The Connections service generates an event when a new connection has been recorded.**



**Figure 4.13. The Posts service generates an event when a new post has been recorded.**

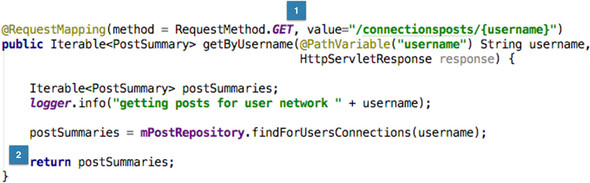


**Figure 4.14. The event handler for the Connections’ Posts service processes events as they occur.**



And finally, in [figure 4.15](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig15), the Connections’ Posts service responds to requests, in the ConnectionsPostsController.

**Figure 4.15. The Connections’ Posts service generates and delivers a response when a request is received. This is completely independent from the operations of the other microservices in our solution.**



Although it’s interesting to see how the processing that ultimately generates the Connections’ Posts result is distributed across the microservices, even more significant are the temporal aspects. With the request/response style, the aggregation occurs when the user makes a request. With the event-driven approach, it’s whenever the data in the system changes; it’s asynchronous. As you’ll see, asynchronicity is valuable in distributed systems.

**4.4. Introducing Command Query Responsibility Segregation**

I want to look at another aspect of the code for this example, starting with the Posts and Connections services. These are essentially CRUD services; they allow for posts, users, and connections to be *created*, *updated*, and *deleted*, and, of course, they allow values to be *read*. A database stores the state for the service, and the RESTful service implementation supports HTTP GET, PUT, POST, and DELETE operations that essentially interact with that data store.

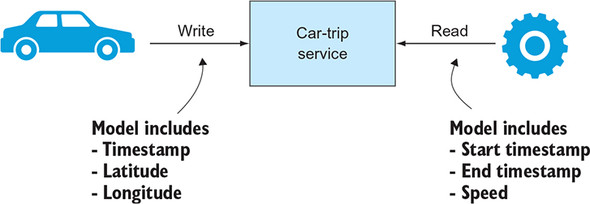
In the request/response implementation, all of this functionality is found in a single controller; for example, for the Posts service in the PostsController. But in the event-driven implementation, you can see that there’s now a read controller in the com.corneliadavis.cloudnative.posts package and a write controller in the com.corneliadavis.cloudnative.posts.write package. For the most part, these two controllers have methods that implement what was previously in that single Posts controller, except that I’ve added the delivery of events for any state-changing operations. But you might be wondering why I’ve gone through the trouble of breaking that single body of code into two.

Truth be told, in this simple example—the Posts and Connections services—that separation has little value. But for even only slightly more sophisticated services, the separation of read from write allows us greater control over our service design. If you’re familiar with the Model, View, Controller (MVC) pattern that has been in widespread use over the past several decades, you know that the controller, which is the business logic for a service (that’s why you find ...Controller classes throughout my implementations), operates against a model. When you have a single controller, the model for both read and write operations is the same, but when you separate the business logic into two separate controllers, each can have its own model. This can be powerful.

Our services don’t even need to be complex in order for you to see this power. Consider a connected car scenario in which sensors on a vehicle are collecting data every second. That data will include a timestamp and GPS coordinates—lat and long—and a service allows for those values to be stored. The model for that service includes these data fields and more: a timestamp, the latitude, and the longitude. One of the things that you want to support is the ability to access trip data speeds; for example, you may want to analyze segments of a car trip in which the speed was greater than 50 miles per hour. That velocity data isn’t directly provided via the write controller, yet obviously it can be computed from the data that was provided; part of the business logic for that service is to generate this derived data.

It’s clear that the models on the read and the write sides are different. Although some fields exist in both, others are meaningful on only one side or the other ([figure 4.16](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig16)). This is one of the first benefits of having two separate models; it allows you to write code that’s easier to understand and maintain, and greatly reduces the surface area in which bugs can creep in.

**Figure 4.16. Separating the write logic from the read logic allows you to support different models for these different concerns. This leads to more elegant, maintainable code.**



In summary, what you’re doing is separating write logic (commands) from read logic (queries). This is the root of the *Command Query Responsibility Segregation* (CQRS) pattern. At its core, CQRS is just about separating those two concerns. Numerous advantages can come from CQRS. The preceding example of more-elegant code is just one. Another is beginning to show itself in the event-driven implementation of our sample.

With the Posts service, for example, the code that was distributed across the read and the write controllers in the event-driven implementation was, for the most part, in the single controller of the request/response solution. But if you turn to the implementation of the Connections’ Posts service, there are greater differences.

In the event-driven solution, the query implementation is found in the com.corneliadavis.cloudnative.connectionsposts package, and the code for the command implementation is found in the com.corneliadavis.cloudnative.connectionsposts.eventhandlers package. Comparing that to the request/response implementation, the first thing to note is that in the request/response solution, there’s no command code. Because the Connections’ Posts service was completely stateless, with its result being an aggregation of data from two other services, no command implementation was needed. As you saw, in the event-driven solution, the query implementation is greatly simplified; all of the aggregation logic disappeared. The aggregation now appears, albeit in a fairly different form, in the command controller. Whereas in request/response you saw calls to downstream resources, users, connections, and posts, in the event-driven command controller you now see event handlers that deal with changes to those same objects.

Those commands are changing state, and although I won’t talk in detail about that data store here (that will come in [chapter 12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_024.html#ch12)), it’s interesting to note other CQRS implications. Although my current implementation of the event processors that make up the command side of the Connections’ Posts service are implemented as HTTP endpoints, as is the query side, separating out the query from the command processing will ultimately allow us to use different protocols for the two sides. On the query side in this case, implementing a REST over HTTP protocol is ideal for the web apps that will access the service. On the command side, however, an asynchronous or even function-as-a-service (FaaS) implementation might be better.**[**[**5**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fn5)**]** The separation of command from query implementations offers this flexibility. (For those of you who are inclined to jump ahead, look at the implementation of the event handler for the Connections’ Posts service in the cloudnative-eventlog module of our code repository. You’ll see that a different approach has replaced the HTTP-based one you see here.)

***5***

*FaaS is an approach that spins up compute for a bit of logic, such as updating a record in a database, on demand and is well suited for event-driven designs.*

I’d like to share one final observation on this topic. I’ve found that CQRS is often conflated with event-driven systems. If the software is using an event-driven approach, then CQRS patterns may be employed, but otherwise CQRS isn’t really considered. I encourage you to consider CQRS independently from event-driven systems. Yes, they complement one another and used together are powerful, but the separation of command logic from query logic is also applicable in designs that are not event-driven.

**4.5. Different styles, similar challenges**

These two implementations yield exactly the same outcome—on the happy path. The choice of using a request/response or event-driven invocation style is arbitrary if all of the following conditions are true:

* No network partitions are cutting off one microservice from another.
* No unexpected latency occurs in producing the list of individuals that I follow.
* All the containers my services are running in maintain stable IP addresses.
* I never have to rotate credentials or certificates.

But these conditions, and many more, are exactly the things that characterize the cloud. Cloud-native software is designed to yield the needed outcomes even when the network is unstable, some components suddenly take longer to produce results, hosts and availability zones disappear, and request volumes suddenly increase by an order of magnitude.

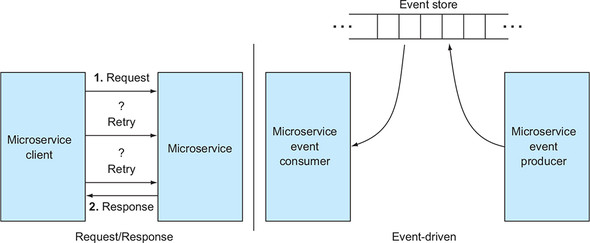
In this context, the request/response and event-driven approaches can yield very different outcomes. But I’m not suggesting that one is always more suitable than the other. Both are valid and applicable approaches. What you must do, however, is apply the right patterns at the right times, to compensate for the unique challenges the cloud brings.

For example, to be ready for spikes and valleys in request volume, you design so that you can scale capacity by creating or deleting instances of your services. Having those multiple instances also lends a certain level of resilience, especially when they’re distributed across failure domains (availability zones). But when you then need to apply new configuration to what could be hundreds of instances of a microservice, you need some type of a configuration service that accounts for their highly distributed deployment. These patterns and many more apply equally to microservices, regardless of whether they’re implementing a request/response or an event-driven protocol.

But some concerns may be handled differently depending on that protocol. For example, what type of compensating mechanisms must you put in place to account for a momentary (could be subsecond or might be last-minute) network partition that cuts related microservices off from one another? The current implementation of the Connections’ Posts service will outright fail if it can’t reach the Connections or the Posts microservices. A key pattern used in this type of scenario is a *retry*: a client making a request to a service over the network will try again if a request they’ve made fails to yield any results. A retry is a way of smoothing out hiccups in the network, allowing the invocation protocol to remain synchronous.

On the other hand, given that the event-driven protocol is inherently asynchronous, the compensating mechanisms to address the same hazards can be quite different. In this architecture, you’ll use a messaging system, such as RabbitMQ or Apache Kafka, to hold onto events through network partitions. Your services will implement the protocols to support this architecture, such as having a control loop that continuously checks for new events of interest in the event store. Recall the HTTP POST from the Posts microservice directly to the event handler of the Connections’ Posts service: you’d use a messaging system to replace that tight coupling. [Figure 4.17](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_016.html#ch04fig17) depicts the differences in the patterns used to handle this characteristic of distributed systems.

**Figure 4.17. The retry is a key pattern used in request/response microservice architectures to compensate for network partitions. In event-driven systems, the use of an event store is a key technique that compensates for network instability.**



The remainder of the book dives into these various patterns, always with a focus on the problems that the cloud context brings. Choosing one of the invocation styles over the other isn’t, on its own, a solution. That choice, along with the patterns that complement them, is. I’ll be teaching you how to select the right protocol and apply the additional patterns that support them.

**Summary**

* Both request/response and event-driven approaches are used to connect components that make up cloud-native software.
* A microservice can implement both request/response and event-handling protocols.
* Under ideal, stable circumstances, software implemented using one approach can yield exactly the same outcomes as software implemented using the other approach.
* But in a cloud setting, where the solution is a distributed system and the environment is constantly changing, the outcomes can vary wildly.
* Some architectural patterns are applied equally to cloud-native software following request/response and event-driven styles.
* But other patterns specifically serve one invocation protocol or the other.
* CQRS plays an important role in event-driven systems.