l guide to CQRS data patterns

guides to CQRS for simpler command and query models while akes.

nan (Red Hat, Lead Contributor)

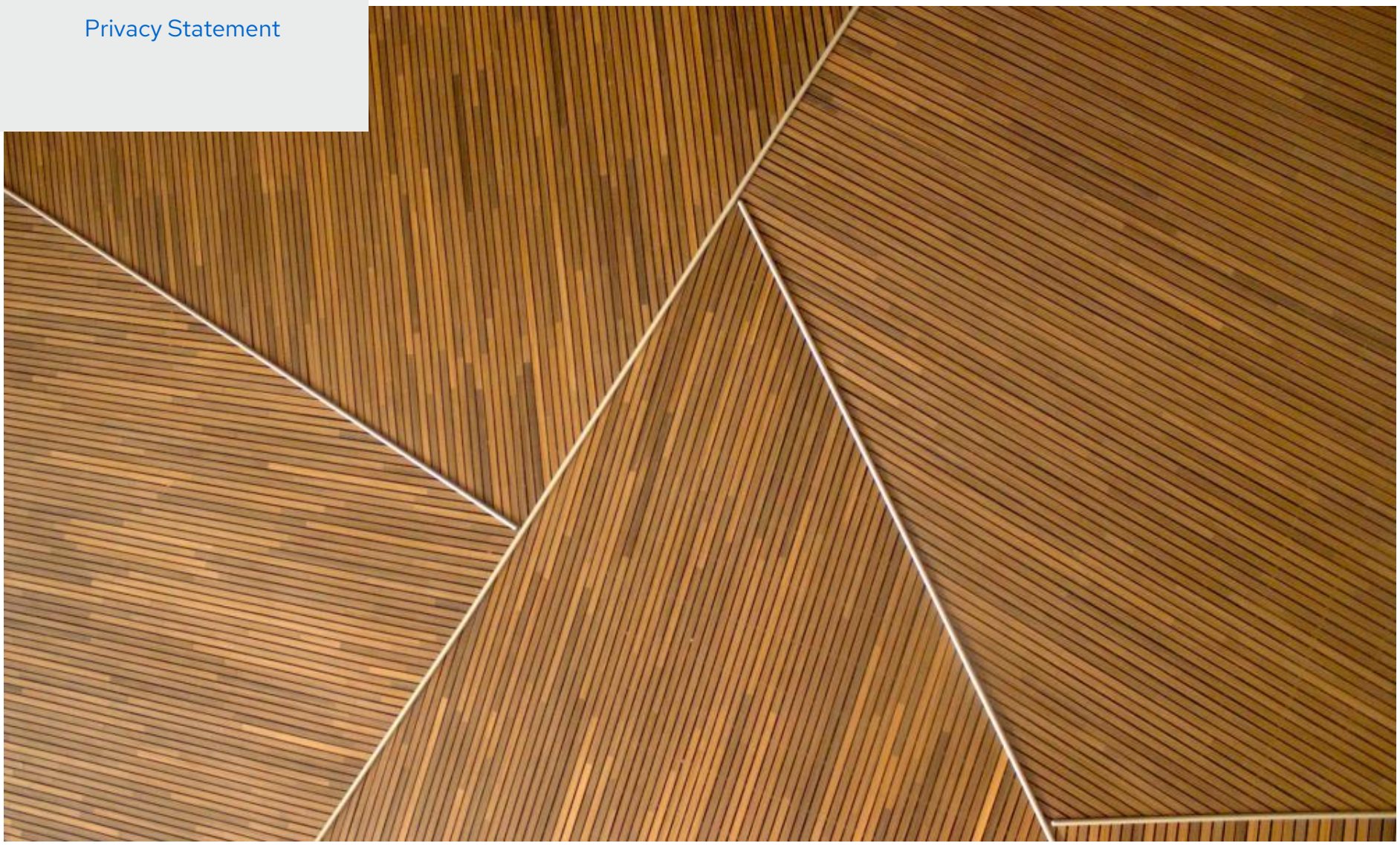

Photo by Teo Duldulao

Today millions of users can be reading and writing data into applications in near simultaneity. The variety of such applications is broad, well beyond the traditional high volume scenarios typically found in banking and other financial applications. New approaches are needed.

One such approach is the *Command Query Responsibility Segregation* (CQRS) pattern.

Take the CQRS interactive tutorials on Katacoda

In addition to the information provided in this article, we've published a set of three tutorials on the Katacoda interactive learning environment that will help you understand the CQRS pattern at an operational level.

The Katacoda learning environment provides an interactive virtual machine where you can execute the commands and techniques described in the tutorials directly in a terminal window. The Katacoda environment allows you to learn by doing!

You can access the tutorials at the following links:

- Part 1-Examining a Single Source Database Architecture
- Part 2-Implementing the CQRS Pattern
- Part 3-Taking an Event-Driven Approach to CQRS

What is CQRS?

The Command Query Responsibility Segregation pattern is, as the name implies, a software design pattern that from query activities. In CQRS parlance, a command *writes* data to a data source. A ource. CQRS addresses the problem of data access performance degradation when ale have too much burden placed on the physical database and the network on which it

rgence of ubiquitous computing brought about by the internet, having a single for read and write activities in an application was a reasonable approach to data

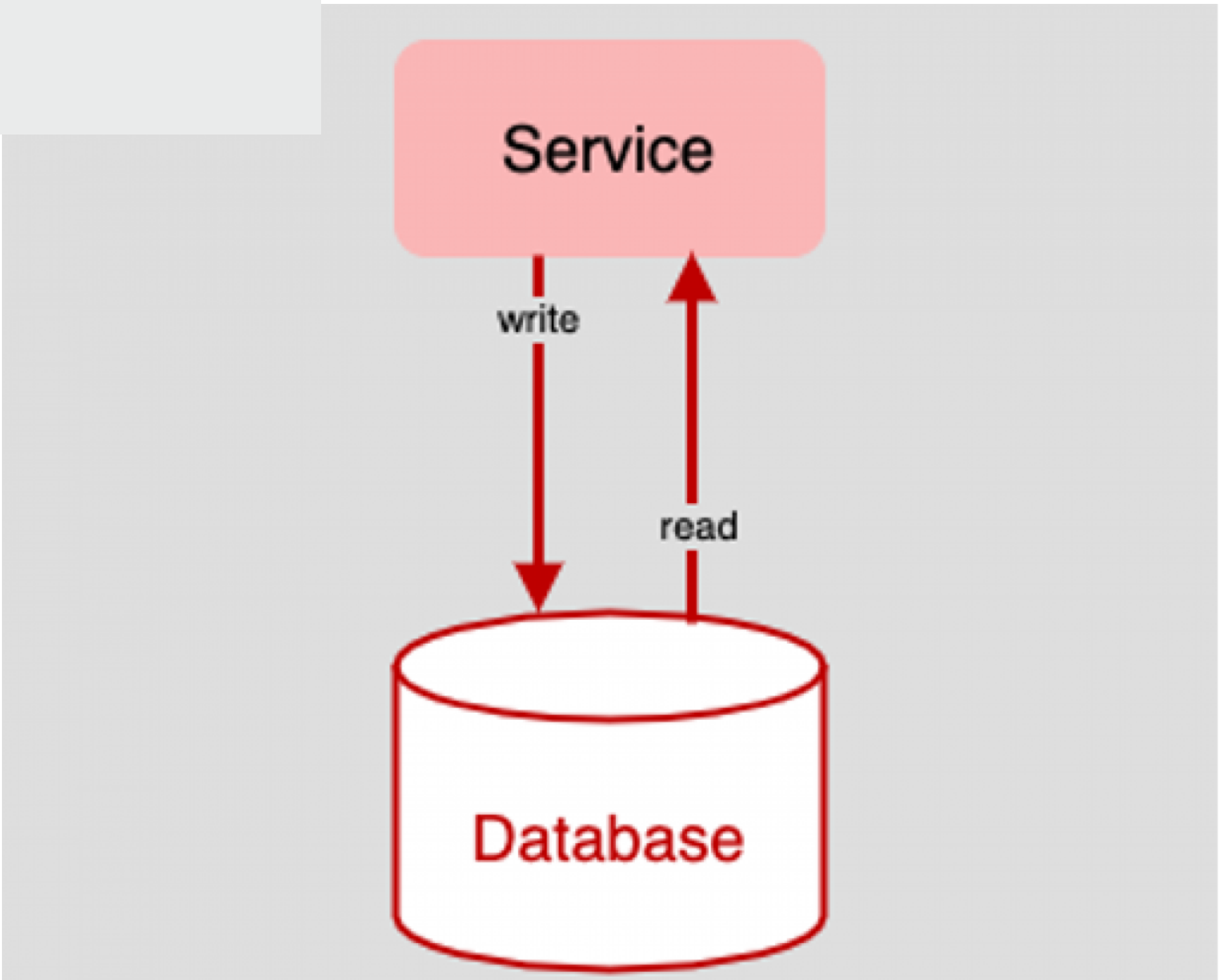

Figure 1: In the days before the commercialized internet, reading and writing data to a single database was commonplace

If performance started to degrade, you could always convert the single database into a database cluster and thus spread access activities across the cluster. However, as long as the cluster was exposed as a single point on the network, there was still the risk of high latency due to the bottleneck around that access point. All read requests (the query) and all write requests (the command) went to the same target.

The benefit that CQRS brings to data design is that it separates read and write behaviors in the application logically and physically. (See Figure 2, below)

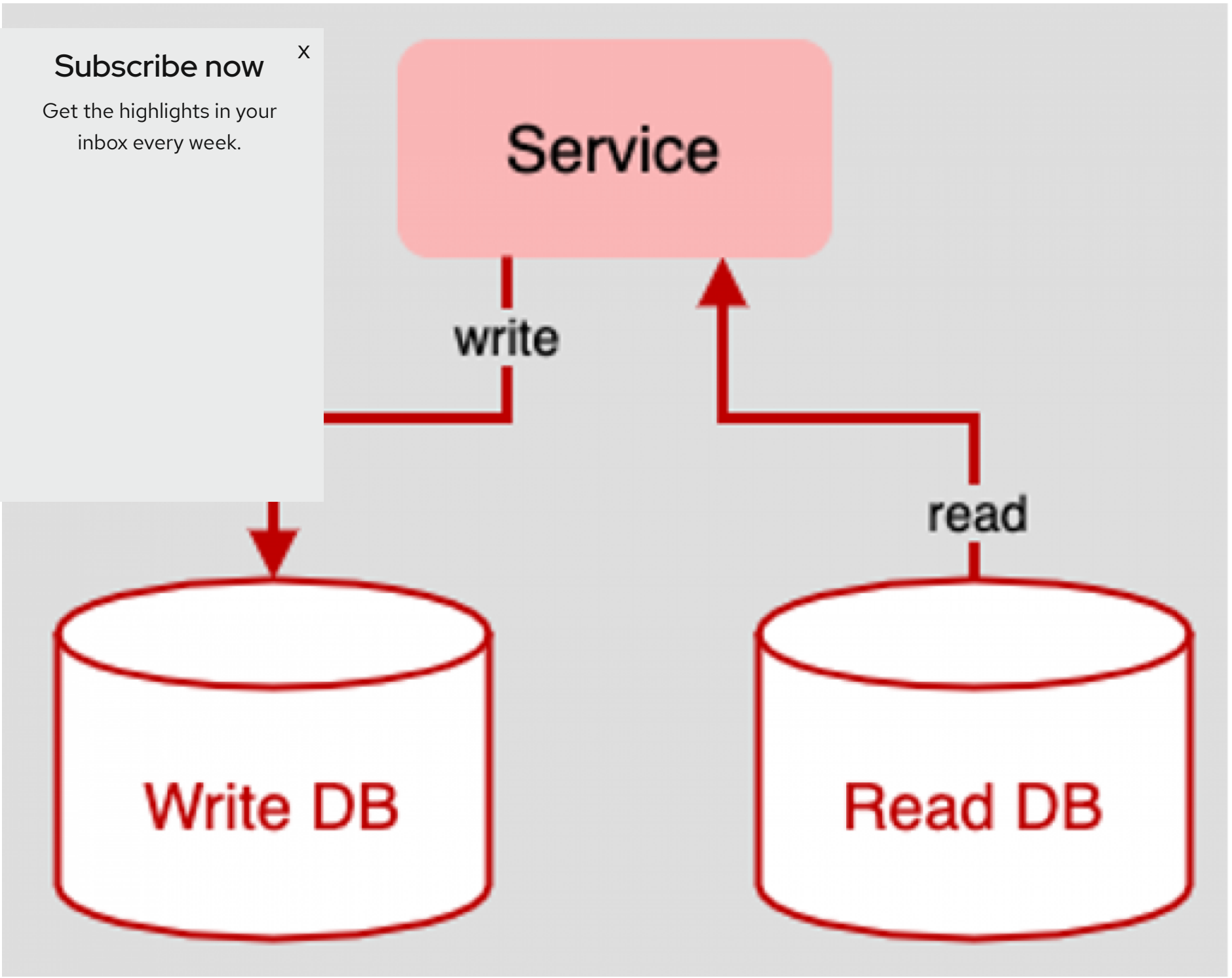

Figure 2: The underlying principle of CQRS is to completely separate *write* activity from *read* activity within the underlying application architecture

In a single-source data design, one database is the target of both read and write activity. When the CQRS pattern is used, read and write activity is separated among a variety of data sources. Also, each data source can be a database technology that is best suited for the particular need. For example, the best technology for storing write data might be a relational database such as Postgres or even DB2 on a mainframe. On the other hand, should the forces driving read behavior have to do with displaying hierarchical data on an eCommerce site, for example, using a document database such as MongoDB to store read data and view it makes sense.

While following the CQRS pattern to separate write storage from read storage provides a high degree of flexibility and efficiency for applications operating at a web-scale, implementing the pattern comes with a fundamental challenge that needs to be addressed in any situation. The challenge is data synchronization.

Addressing the challenge of data synchronization

As mentioned above, essential to the concept of CQRS is the notion of separating read behavior from write behavior. Under CQRS, an application will save data to one distinct data source and read data from another. However, given that read and write data sources are separate and isolated from each other, CQRS only works if the data is consistent among all data sources. In other words, if I have an eCommerce application that adds an order object to the write database, the information in that order had better show up somewhere in the read database in an accurate and timely manner. If not, the entire application is at risk of offering up inaccurate data and, thus, not being of much value.

Ensuring data consistency among the entire array of database technologies in a CQRS architecture is essential. The questions then become not only, how do you do it but also, how do you do it well?

First, let's take a look at the way that works but has an undesirable side-effect. Then, let's look at a more optimal

approach.

Avoiding a common mistake

nsistency under CQRS is to create intelligence in the application to inspect an QRS logic accordingly. Figure 3 below shows an example of this idea.

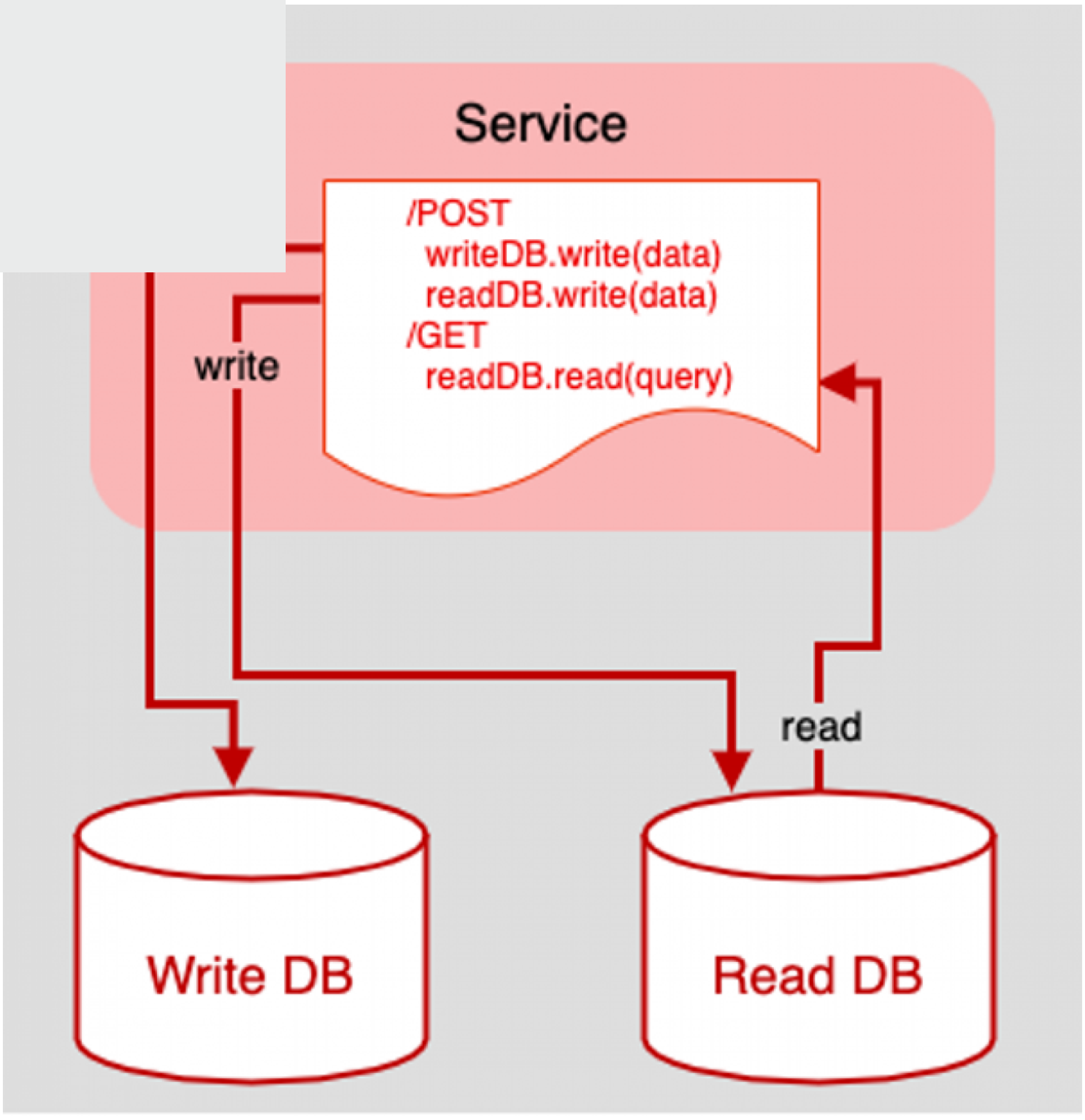

Figure 3: Putting data routing within the scope of an HTTP request creates a tightly bound architecture that will be difficult to refactor

Figure 3 above is an example of a fictitious API that supports read (a.k.a. query) behavior via an HTTP `GET` request on a single endpoint. Write (a.k.a. command) behavior is supported by an HTTP `POST` against that same endpoint.

When a read query comes in, the HTTP service retrieves the data from the Read database. However, when write data comes in via the `POST` request, things get a little more involved. The `POST` logic makes two writes, one to the Write DB as expected and one to the Read DB. Thus, data consistency among the data sources is achieved. That's the good news. The bad news is that the technique has incurred risk and technical debt.

The biggest risk is one of transactional failure. There is an underlying assumption in play that the HTTP `POST` write

activities that attempt to add data to both the Write DB and the Read DB are successful. As it stands in the Figure 3 code, there is no failure check on either write and no fallback behavior to save the incoming data on failure. Obviously, this problem needs to be addressed in order to have a viable implementation of CQRS. Yet, there is another problem,

ting CQRS directly in the web server makes the application brittle. A fundamental is the clear separation of concerns. Putting CQRS logic in the web server violates the

is to accept and route requests onto relevant logic. The concern of CQRS is to es in the application's data access framework. Mixing these two creates a technical ay back when the time comes.

y.

en data persistence

RS is to use the mediator pattern in conjunction with an event-driven data persistence architecture. Let's start with the latter.

Event-driven data persistence is a technique in which events are raised in accordance with a given application's purpose, and the data associated with the given event is consumed by interested parties. For example, at the conceptual level, let's imagine that we have an eCommerce API that accepts orders. A user posts a request that has order information. The API receives the order information and packages it up into a structured message that describes an event, **OnNewOrder**, as shown below in Listing 1.

```
{
    eventName: "OnNewOrder",
    orderId: "8263341f-03e0-4b89-a7b1-81670142383d",
    description: "Ceramic coffee cup",
    quantity: 1;
    firstName: "Bill",
    lastName: "Jones",
    email: "bill.jones@example.com"
}
```

Listing 1: An example of an event message

The API publishes that message to a message broker such as Kafka. Those parties interested in information around an **OnNewOrder** event consume that message and process it according to their purpose.

The value of using event-driven data persistence is that the technique creates a high degree of independence among the various data sources in play in a given application. Each component in an application does only what it's supposed to do. There is no mixing and matching.

However, to support event-driven data persistence under CQRS, there still needs to be intelligence that can mediate read and write behavior for incoming requests. Also, because data will be dispersed throughout the application in a variety of asynchronous messages that correspond to particular events, there needs to be a mechanism to store all the data for all the events generated. This storage mechanism needs to store the data for a long time, on the order of years. This way, if things go awry and read and write data goes out of sync or the data in a data source is destroyed, a sole point of truth—the event store—can be used to reconstitute the system to the last known, good state.

Let's take a look at how this works.

Taking an event-driven approach to CQRS

Figure 4 below illustrates a data architecture that implements CQRS using the mediator pattern. The first thing to notice is that all requests, both commands to add data to the application and queries to read data from the application, pass from the application's web server onto a component named **Mediator**.

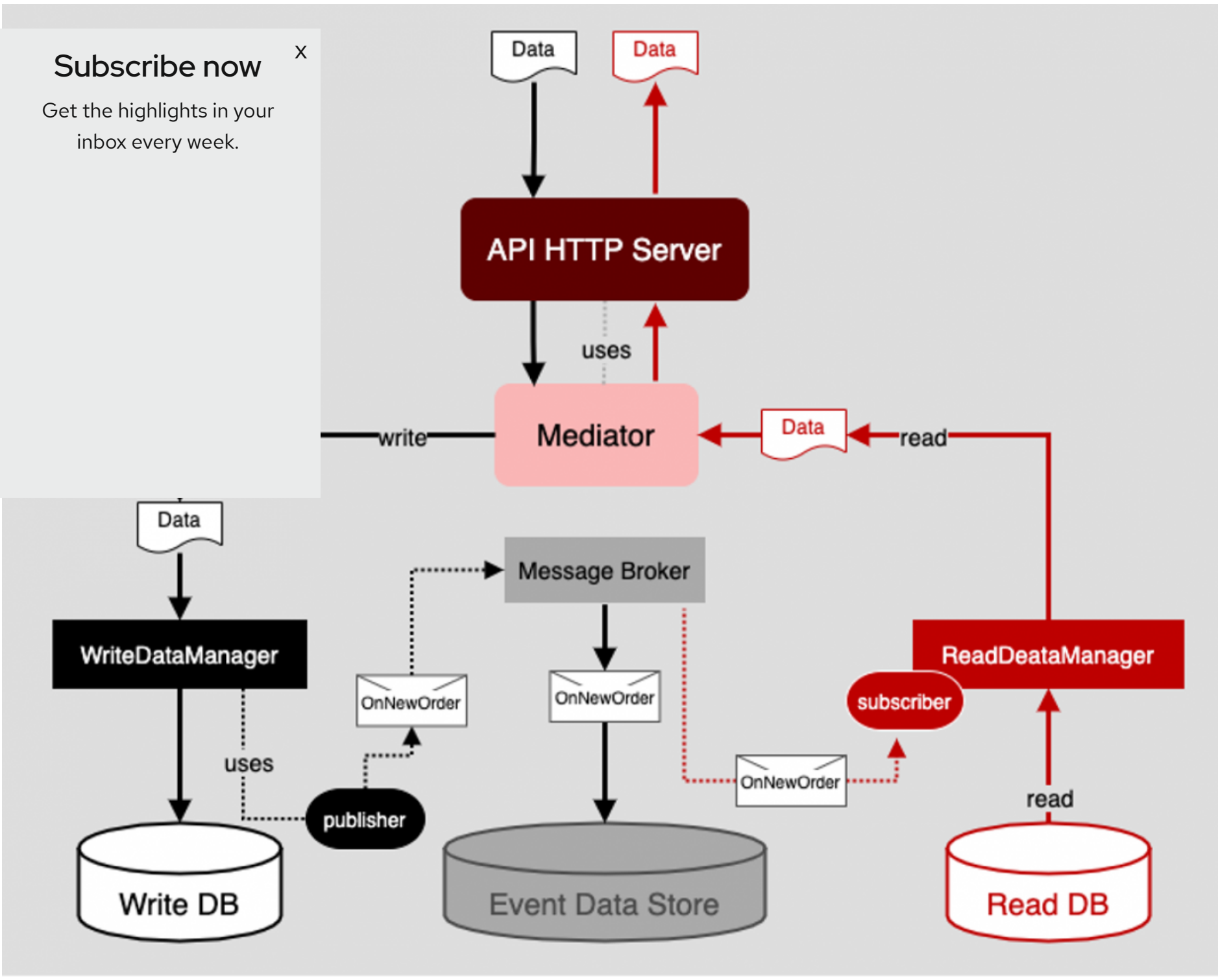

Figure 4: **Using a Mediator in an event-driven architecture provides independence and flexibility**

The purpose of the **Mediator** is to distinguish read and write requests and then route the request accordingly. If the request is a write command, that data is forwarded onto a component called a **WriteDataManager**. If the request is a read query, it is retrieved from a component called **ReadDataManager**.

The purpose of **WriteDataManager** is to connect to a write database and add the data in the request. Also, upon adding data to the write database, the **WriteDataManager** will publish an event, **OnNewOrder**, to a message broker to which it is bound. At this point, **WriteDataManager** has done its work. The actual consumption of the data associated with the **OnNewOrder** event is of no concern to **WriteDataManager**. Its job is to add data to its data store and generate an event when data is added.

When the message broker receives an **OnNewOrder** event message, it does two things. First, the message broker stores the data associated with the event in its internal event data store. This creates a permanent record of the event that can be used for audit and failure recovery purposes.

Second, the message broker passes the message to a message queue to which interested parties are subscribed. Interested parties consume the message and act on it according to their purpose.

In the case of the read data source, as mentioned previously, that functionality is represented by the component named **ReadDataManager**. **ReadDataManager** subscribes to a queue on the message broker to receive **OnNewOrder** event messages. When an **OnNewOrder** message is received, **ReadDataManager** saves the relevant information in the event message to its data store. At this point, both data as represented by both **WriteDataManager** and **ReadDataManager** is consistent. Thus, the intention of CQRS has been fulfilled in a way that is loosely coupled yet reliable and verifying between data sources.

Putting it all together

Putting it all together

The Command Query Responsibility Segregation pattern is a valuable design technique that is well suited to ort a high volume of requests made against very big data sources. Separating read ystem performance overall. And, it allows systems to scale up quickly when need to be added. When used in conjunction with the mediator pattern, CQRS makes asier to refactor.

ful, it is not simple. Separating data according to purpose means that there is always a istency either due to varying degrees of latency on the network or episodic system RS using an event-driven data persistence architecture in which the system's message e received for a very long time is a sensible way to go.

formance and efficiency. The trick is to make sure that the underlying application mented is loosely coupled and can be reconstituted to the last known good state in the

Bob Reselman

Bob Reselman is a nationally known software developer, system architect, industry analyst, and technical writer/journalist. More about me