


Software Architecture: The 5 Patterns You Need to Know

by Peter Morlion  MVB · Jun. 30, 18 · Microservices Zone · Tutorial

Augment your existing security architecture by building a microperimeter that assures authentication and API security at the microservice level.

When I was attending night school to become a programmer, I learned several design patterns: singleton, repository, factory, builder, decorator, etc. Design patterns give us a proven solution to existing and recurring problems. What I didn't learn was that a similar mechanism exists on a higher level: software architecture patterns. These are patterns for the overall layout of your application or applications. They all have advantages and disadvantages. And they all address specific issues.

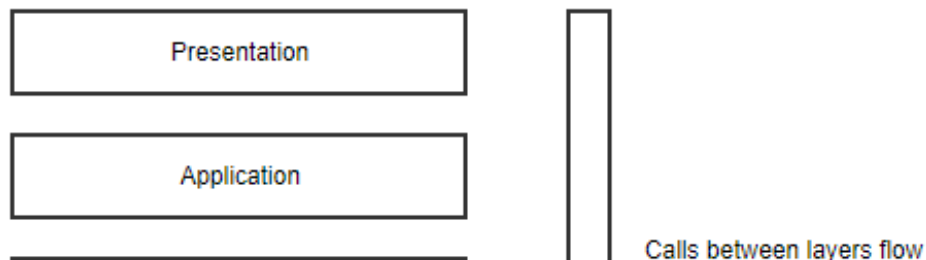
Layered Pattern

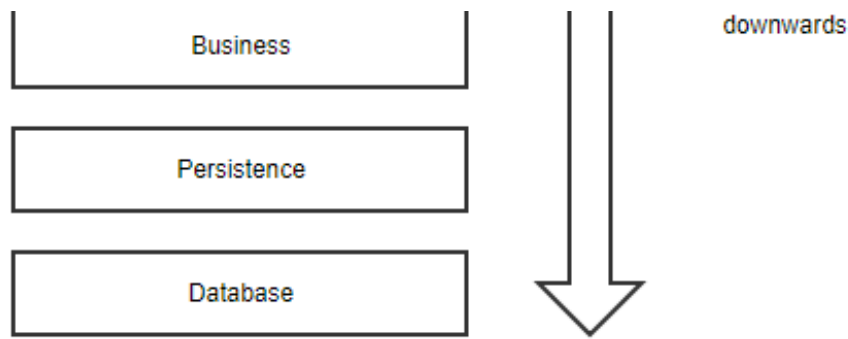
The layered pattern is probably one of the most well-known software architecture patterns. Many developers use it, without really knowing its name. The idea is to split up your code into “layers”, where each layer has a certain responsibility and provides a service to a higher layer.

There isn't a predefined number of layers, but these are the ones you see most often:

- Presentation or UI layer
- Application layer
- Business or domain layer
- Persistence or data access layer
- Database layer

The idea is that the user initiates a piece of code in the presentation layer by performing some action (e.g. clicking a button). The presentation layer then calls the underlying layer, i.e. the application layer. Then we go into the business layer and finally, the persistence layer stores everything in the database. So higher layers are dependent upon and make calls to the lower layers.





You will see variations of this, depending on the complexity of the applications. Some applications might omit the application layer, while others add a caching layer. It's even possible to merge two layers into one. For example, the ActiveRecord pattern combines the business and persistence layers.

Layer Responsibility

As mentioned, each layer has its own responsibility. The presentation layer contains the graphical design of the application, as well as any code to handle user interaction. You shouldn't add logic that is not specific to the user interface in this layer.

The business layer is where you put the models and logic that is specific to the business problem you are trying to solve.

The application layer sits between the presentation layer and the business layer. On the one hand, it provides an abstraction so that the presentation layer doesn't need to know the business layer. In theory, you could change the technology stack of the presentation layer without changing anything else in your application (e.g. change from WinForms to WPF). On the other hand, the application layer provides a place to put certain coordination logic that doesn't fit in the business or presentation layer.

Finally, the persistence layer contains the code to access the database layer. The database layer is the underlying database technology (e.g. SQL Server, MongoDB). The persistence layer is the set of code to manipulate the database: SQL statements, connection details, etc.

Advantages

- Most developers are familiar with this pattern.
- It provides an easy way of writing a well-organized and testable application.

Disadvantages

- It tends to lead to monolithic applications that are hard to split up afterward.
- Developers often find themselves writing a lot of code to pass through the different layers, without adding any value in these layers. If all you are doing is writing a simple CRUD application, the layered pattern might be overkill for you.

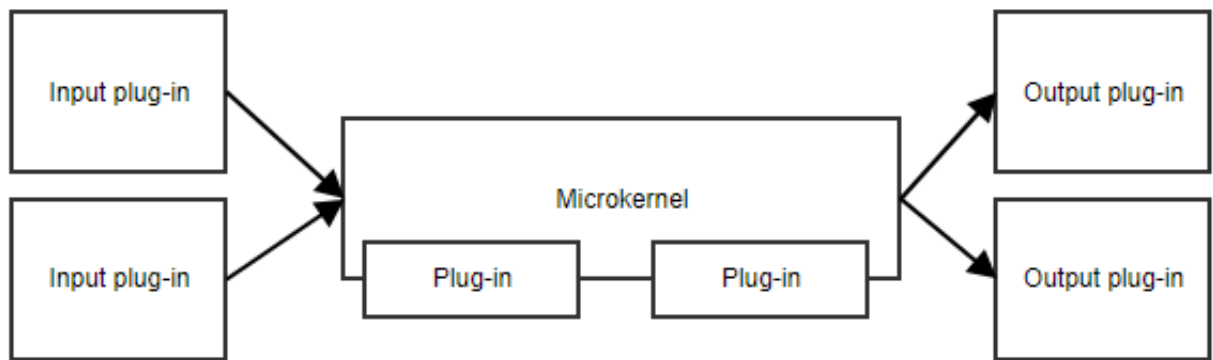
Ideal for

- Standard line-of-business apps that do more than just CRUD operations

Microkernel

The microkernel pattern, or plug-in pattern, is useful when your application has a core set of responsibilities and a collection of interchangeable parts on the side. The microkernel will provide the entry point and the general flow of the application, without really knowing what the different plug-ins are doing.

without really knowing what the different plug-ins are doing.



An example is a task scheduler. The microkernel could contain all the logic for scheduling and triggering tasks, while the plug-ins contain specific tasks. As long as the plug-ins adhere to a predefined API, the microkernel can trigger them without needing to know the implementation details.

Another example is a workflow. The implementation of a workflow contains concepts like the order of the different steps, evaluating the results of steps, deciding what the next step is, etc. The specific implementation of the steps is less important to the core code of the workflow.

Advantages

- This pattern provides great flexibility and extensibility.
- Some implementations allow for adding plug-ins while the application is running.
- Microkernel and plug-ins can be developed by separate teams.

Disadvantages

- It can be difficult to decide what belongs in the microkernel and what doesn't.
- The predefined API might not be a good fit for future plug-ins.

Ideal for

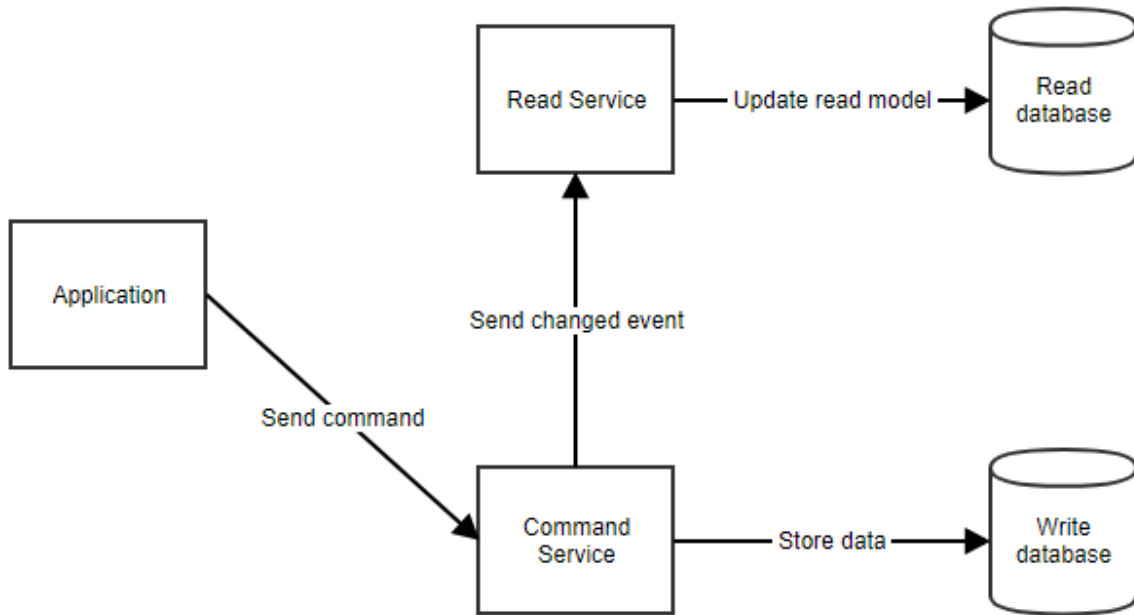
- Applications that take data from different sources, transform that data and writes it to different destinations
- Workflow applications
- Task and job scheduling applications

CQRS

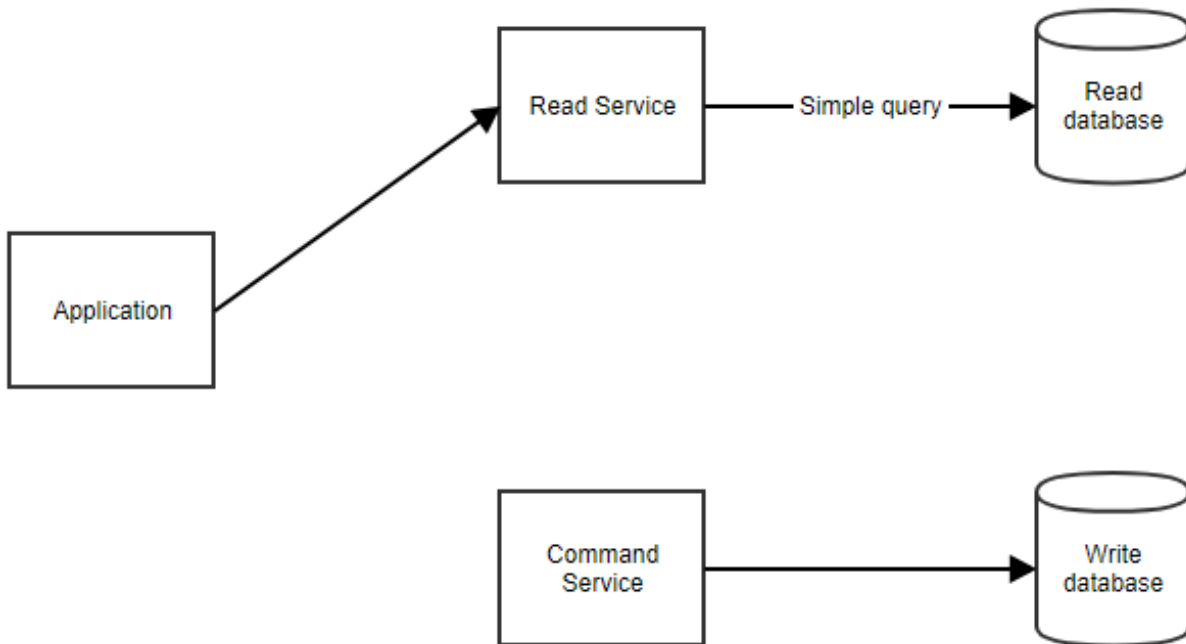
CQRS is an acronym for Command and Query Responsibility Segregation. The central concept of this pattern is that an application has read operations and write operations that must be totally separated. This also means that the model used for write operations (commands) will differ from the read models (queries). Furthermore, the data will be stored in different locations. In a relational database, this means there will be tables for the command model and tables for the read model. Some implementations even store the different models in totally different databases, e.g. SQL Server for the command model and MongoDB for the read model.

This pattern is often combined with event sourcing, which we'll cover below.

How does it work exactly? When a user performs an action, the application sends a command to the command service. The command service retrieves any data it needs from the command database, makes the necessary manipulations and stores that back in the database. It then notifies the read service so that the read model can be updated. This flow can be seen below.



When the application needs to show data to the user, it can retrieve the read model by calling the read service, as shown below.



Advantages

- Command models can focus on business logic and validation while read models can be tailored to specific scenarios.
- You can avoid complex queries (e.g. joins in SQL) which makes the reads more performant.

Disadvantages

- Keeping the command and the read models in sync can become complex.

Ideal for

- Applications that expect a high amount of reads
- Applications with complex domains

Event Sourcing

As I mentioned above, CQRS often goes hand in hand with event sourcing. This is a pattern where you don't store the current state of your model in the database, but rather the events that happened to the model. So when the name of a customer changes, you won't store the value in a "Name" column. You will store a "NameChanged" event with the new value (and possibly the old one too).

When you need to retrieve a model, you retrieve all its stored events and reapply them on a new object. We call this rehydrating an object.

A real-life analogy of event sourcing is accounting. When you add an expense, you don't change the value of the total. In accounting, a new line is added with the operation to be performed. If an error was made, you simply add a new line. To make your life easier, you could calculate the total every time you add a line. This total can be regarded as the read model. The example below should make it more clear.

Invoice 201804	2000	
Office supplies	-120	
Train tickets	-64	
Invoice 201805	1800	
Electricity bill	-250	
Cancellation invoice 201805	-1800	<== Correction
Invoice 201805	1850	
Total	3416	<== Auto-updated read model

You can see that we made an error when adding Invoice 201805. Instead of changing the line, we added two new lines: first, one to cancel the wrong line, then a new and correct line. This is how event sourcing works. You never remove events, because they have undeniably happened in the past. To correct situations, we add new events.

Also, note how we have a cell with the total value. This is simply a sum of all values in the cells above. In Excel, it automatically updates so you could say it synchronizes with the other cells. It is the read model, providing an easy view for the user.

Event sourcing is often combined with CQRS because rehydrating an object can have a performance impact, especially when there are a lot of events for the instance. A fast read model can significantly improve the response time of the application.

Advantages

- This software architecture pattern can provide an audit log out of the box. Each event represents a manipulation of the data at a certain point in time.

Disadvantages

- It requires some discipline because you can't just fix wrong data with a simple edit in the database.
- It's not a trivial task to change the structure of an event. For example, if you add a property, the database still contains

- It's not a trivial task to change the structure of an event. For example, if you add a property, the database still contains events without that data. Your code will need to handle this missing data gracefully.

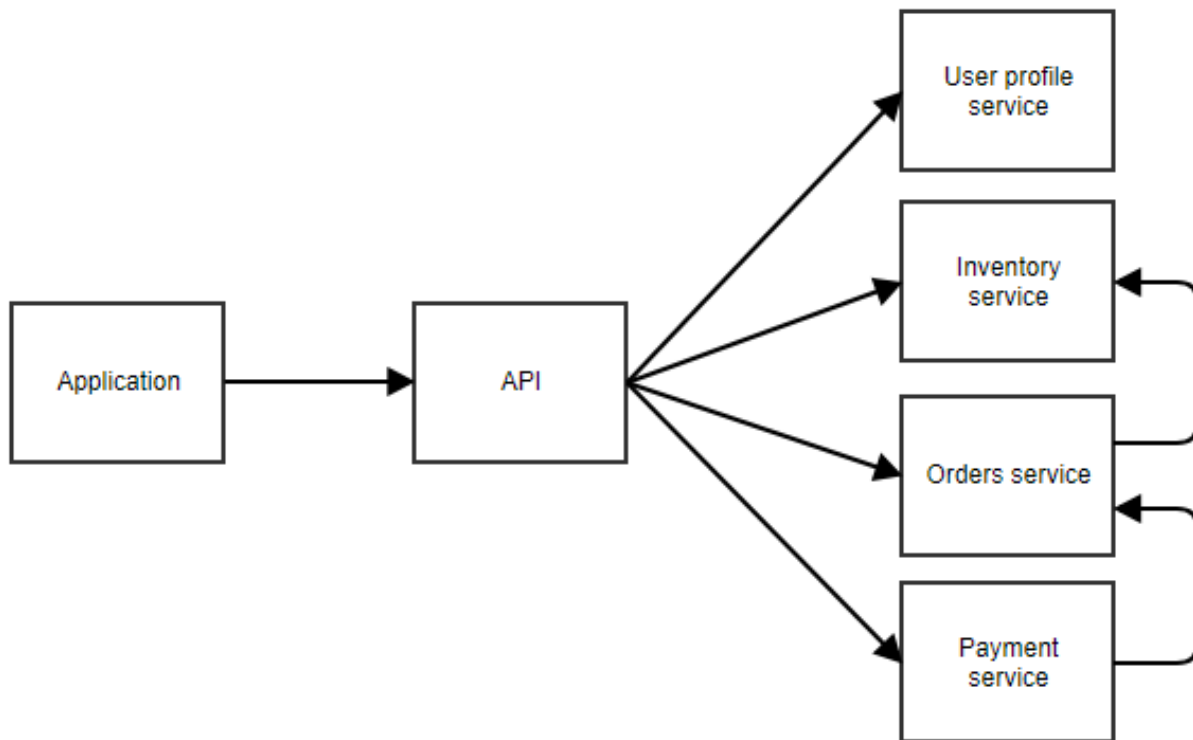
Ideal for applications that

- Need to publish events to external systems
- Will be built with CQRS
- Have complex domains
- Need an audit log of changes to the data

Microservices

When you write your application as a set of microservices, you're actually writing multiple applications that will work together. Each microservice has its own distinct responsibility and teams can develop them independently of other microservices. The only dependency between them is the communication. As microservices communicate with each other, you will have to make sure messages sent between them remain backwards-compatible. This requires some coordination, especially when different teams are responsible for different microservices.

A diagram can explain.



In the above diagram, the application calls a central API that forwards the call to the correct microservice. In this example, there are separate services for the user profile, inventory, orders, and payment. You can imagine this is an application where the user can order something. The separate microservices can call each other too. For example, the payment service may notify the orders service when a payment succeeds. The orders service could then call the inventory service to adjust the stock.

There is no clear rule of how big a microservice can be. In the previous example, the user profile service may be responsible for data like the username and password of a user, but also the home address, avatar image, favorites, etc. It could also be an option to split all those responsibilities into even smaller microservices.

Advantages

- You can write, maintain, and deploy each microservice separately.
- A microservices architecture should be easier to scale, as you can scale only the microservices that need to be scaled. There's no need to scale the less frequently used pieces of the application.
- It's easier to rewrite pieces of the application because they're smaller and less coupled to other parts.

Disadvantages

- Contrary to what you might expect, it's actually easier to write a well-structured monolith at first and split it up into microservices later. With microservices, a lot of extra concerns come into play: communication, coordination, backward compatibility, logging, etc. Teams that miss the necessary skill to write a well-structured monolith will probably have a hard time writing a good set of microservices.
- A single action of a user can pass through multiple microservices. There are more points of failure, and when something does go wrong, it can take more time to pinpoint the problem.

Ideal for:

- Applications where certain parts will be used intensively and need to be scaled
- Services that provide functionality to several other applications
- Applications that would become very complex if combined into one monolith
- Applications where clear bounded contexts can be defined

Combine

I've explained several software architecture patterns, as well as their advantages and disadvantages. But there are more patterns than the ones I've laid out here. It is also not uncommon to combine several of these patterns. They aren't always mutually exclusive. For example, you could have several microservices and have some of them use the layered pattern, while others use CQRS and event sourcing.

The important thing to remember is that there isn't one solution that works everywhere. When we ask the question of which pattern to use for an application, the age-old answer still applies: "it depends." You should weigh in on the pros and cons of a solution and make a well-informed decision.

Discover how to deploy pre-built sample microservices OR create simple microservices from scratch.

Like This Article? Read More From DZone



What Do You Mean by Event-Driven?



Event Sourcing: The Pains of Wrongly Designed Aggregates



CQRS: Understanding From First Principles



**Free DZone Refcard
Microservices in Java**

Topics: SOFTWARE ARCHITECTURE , DESIGN PATTERNS , MICROSERVICES , CQRS , EVENT SOURCING

Published at DZone with permission of Peter Morlion , DZone MVB. [See the original article here.](#) 
Opinions expressed by DZone contributors are their own.

Microservices Partner Resources

6 The Six Pillars of AI-Powered APM for Containerized Microservices

stana

|

Securing Microservice APIs new O'Reilly eBook

Al Technologies

|

Deploy Pre-Built Sample Microservices OR Create Simple Microservices From Scratch.

CloudEntity

|

Modern APM in a Containerized World

stana

|