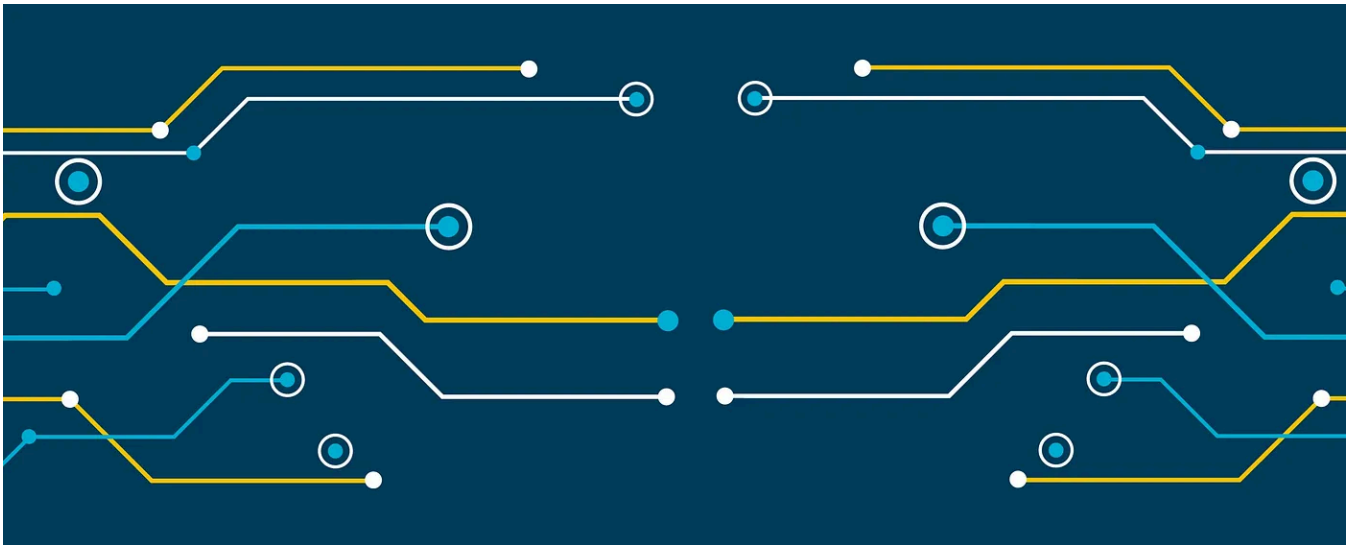
10 microservices design patterns for better architecture

Consider using these popular design patterns in your next microservices app and make organization more manageable.

Capital One Tech · Follow
Published in Capital One Tech
11 min read · Jan 10, 2024



The monolithic architecture was historically used by developers for a long time — and for a long time, it worked. Unfortunately, these architectures use fewer parts that are larger, thus meaning they were more likely to fail in entirety if a single part failed. Often, these applications ran as a singular process, which only exacerbated the issue.

Microservices solve these specific issues by having each microservice run as a separate process. If one cog goes down, it doesn't necessarily mean the whole

machine stops running. Plus, diagnosing and fixing defects in smaller, highly cohesive services is often easier than in larger monolithic ones.

Microservices design patterns provide tried-and-true fundamental building blocks that can help write code for microservices. By utilizing patterns during the development process, you save time and ensure a higher level of accuracy versus writing code for your microservices app from scratch. In this article, we cover a comprehensive overview of 10 microservices design patterns you need to know, as well as when to apply them.

Key benefits of using microservices design patterns

Knowing the key benefits of microservices will help you understand the design patterns. The exact benefits may vary based on the microservices being used and the applications they're being used for. However, developers and software engineers can generally expect the following advantages when using microservices design patterns:

- Creation of an application architecture that's independently deployable and decentralized
- Massive scalability when and if needed
- New versions of microservices that can be rolled out incrementally, thus reducing downtime
- Detecting unwanted behavior before an old application version is completely replaced
- Use of multiple coding languages
- Prevention of systemic failure due to a root cause in an isolated component
- Real-time load balancing

At Capital One, we've applied microservices architecture to help increase our speed of delivery without compromising quality, so we have experience using types of design patterns like these firsthand. Of course, understanding microservices best practices will help you reap the most benefits. Before incorporating any best practice the first step is to understand the microservices design practices you might frequently use during development.

1. Database per service pattern

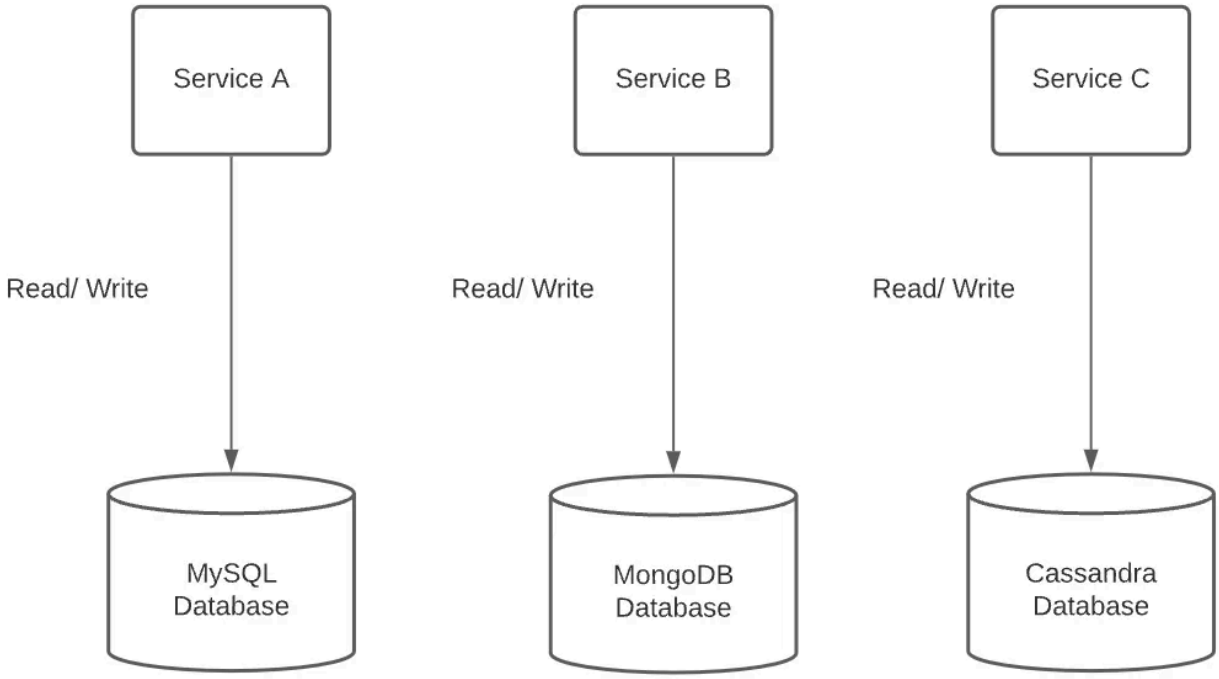
The database is one of the most important components of microservices architecture, but it isn't uncommon for developers to overlook the database per service pattern when building their services. Database organization will affect the efficiency and complexity of the application. The most common options that a developer can use when determining the organizational architecture of an application are:

Dedicated database for each service:

A database dedicated to one service can't be accessed by other services. This is one of the reasons that makes it much easier to scale and understand from a whole end-to-end business aspect.

Picture a scenario where your databases have different needs or access requirements. The data owned by one service may be largely relational, while a second service might be better served by a NoSQL solution and a third service may require a vector database. In this scenario, using dedicated services for each database could help you manage them more easily.

This structure also reduces coupling as one service can't tie itself to the tables of another. Services are forced to communicate via published interfaces. The downside is that dedicated databases require a failure protection mechanism for events where communication fails.



Single database shared by all services:

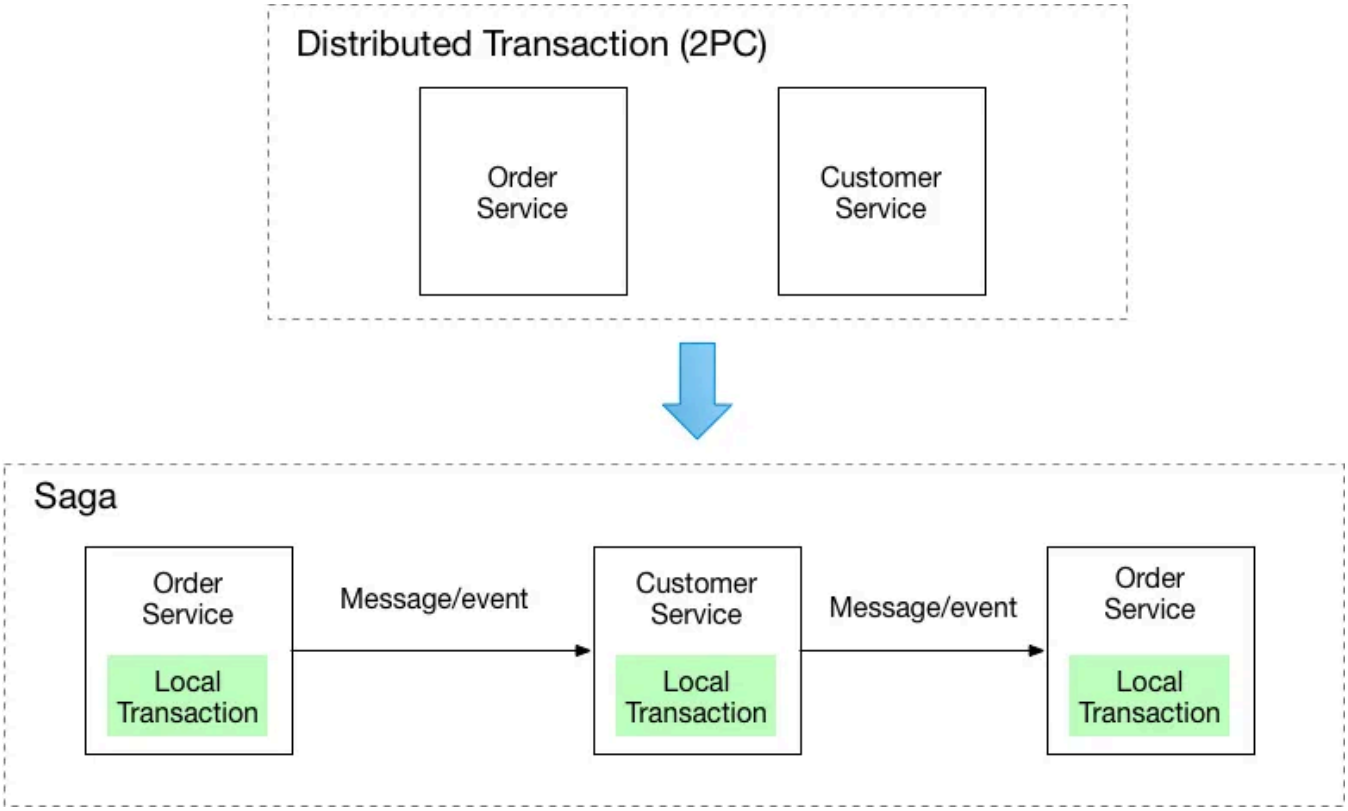
A single shared database isn't the standard for microservices architecture but bears mentioning as an alternative nonetheless. Here, the issue is that microservices using a single shared database lose many of the key benefits developers rely on,

logical boundaries within it. For example, each service should own its have schema and read/write access should be restricted to ensure that services can't poke around where they don't belong.

2. Saga pattern

A saga is a series of local transactions. In microservices applications, a saga patterncan help maintain data consistency during distributed transactions.

The saga pattern is an alternative solution to other design patterns that allows for multiple transactions by giving rollback opportunities.



A common scenario is an e-commerce application that allows customers to purchase products using credit. Data may be stored in two different databases: One for orders and one for customers. The purchase amount can't exceed the credit limit. To implement the Saga pattern, developers can choose between two common approaches.

1. Choreography:

Using the choreography approach, a service will perform a transaction and then publish an event. In some instances, other services will respond to those published events and perform tasks according to their coded instructions. These secondary tasks may or may not also publish events, according to presets. In the example above, you could use a choreography approach so that each local e-commerce transaction publishes an event that triggers a local transaction in the credit service.

2. Orchestration:

An orchestration approach will perform transactions and publish events using an object to orchestrate the events, triggering other services to respond by completing their tasks. The orchestrator tells the participants what local transactions to execute.

Saga is a complex design pattern that requires a high level of skill to successfully implement. However, the benefit of proper implementation is maintained data consistency across multiple services without tight coupling.

3. API gateway pattern

For large applications with multiple clients, implementing an API gateway pattern is a compelling option One of the largest benefits is that it insulates the client from needing to know how services have been partitioned. However, different teams will value the API gateway pattern for different reasons. One of these possible reasons is because it grants a single entry point for a group of microservices by working as a reverse proxy between client apps and the services. Another is that clients don't need to know how services are partitioned, and service boundaries can evolve independently since the client knows nothing about them.

The client also doesn't need to know how to find or communicate with a multitude of ever-changing services. You can also create a gateway for specific types of clients (for example, backends for frontends) which improve ergonomics and reduce the number of roundtrips needed to fetch data. Plus, an API gateway pattern can take care of crucial tasks like authentication, SSL termination and caching, which makes your app more secure and user-friendly.

Another advantage is that the pattern insulates the client from needing to know how services have been partitioned. Before moving onto the next pattern, there's one more benefit to cover: Security. The primary way the pattern improves security is by reducing the attack surface area. By providing a single entry point, the API endpoints aren't directly exposed to clients and authorization and SSL can be efficiently implemented.

Developers can use this design pattern to decouple internal microservices from client apps so a partially failed request can be utilized. This ensures a whole request won't fail because a single microservice is unresponsive. To do this, the encoded API gateway utilizes the cache to provide an empty response or return a valid error code.

4. Aggregator design pattern

An aggregator design pattern is used to collect pieces of data from various microservices and returns an aggregate for processing. Although similar to the backend-for-frontend (BFF) design pattern, an aggregator is more generic and not explicitly used for UI.

To complete tasks, the aggregator pattern receives a request and sends out requests to multiple services, based on the tasks it was assigned. Once every service has answered the requests, this design pattern combines the results and initiates a response to the original request.

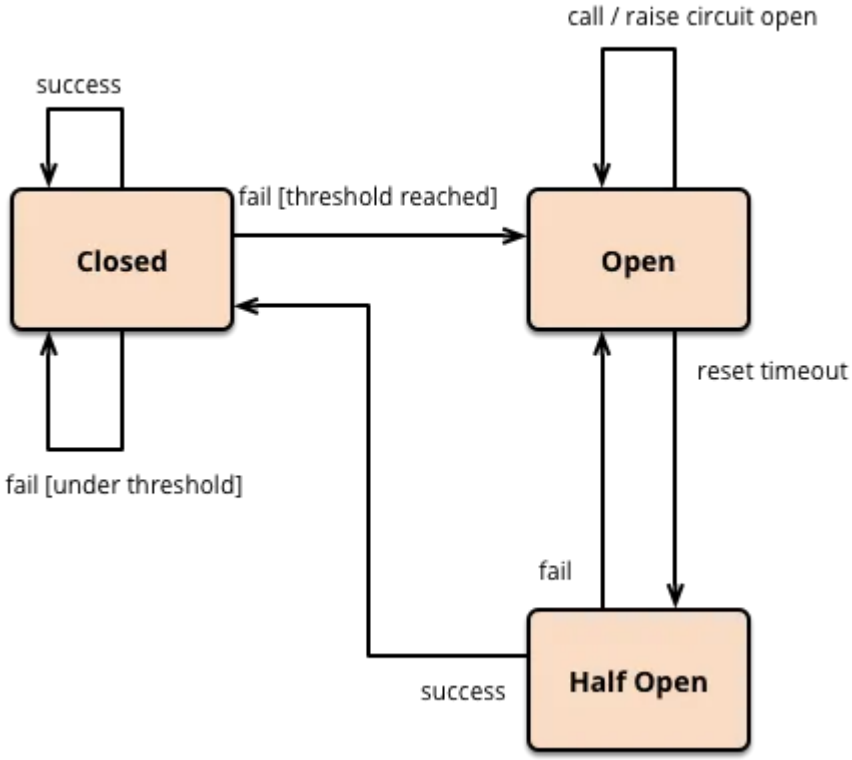
5. Circuit breaker design pattern

This pattern is usually applied between services that are communicating synchronously. A developer might decide to utilize the circuit breaker when a service is exhibiting high latency or is completely unresponsive. The utility here is that failure across multiple systems is prevented when a single microservice is unresponsive. Therefore, calls won't be piling up and using the system resources, which could cause significant delays within the app or even a string of service failures.

Implementing this pattern as a function in a circuit breaker design requires an object to be called to monitor failure conditions. When a failure condition is detected, the circuit breaker will trip. Once this has been tripped, all calls to the circuit breaker will result in an error and be directed to a different service. Alternatively, calls can result in a default error message being retrieved.

There are three states of the circuit breaker pattern functions that developers should be aware of. These are:

1. **Open:** A circuit breaker pattern is open when the number of failures has exceeded the threshold. When in this state, the microservice gives errors for the calls without executing the desired function.
2. **Closed:** When a circuit breaker is closed, it's in the default state and all calls are responded to normally. This is the ideal state developers want a circuit breaker microservice to remain in — in a perfect world, of course.
3. **Half-open:** When a circuit breaker is checking for underlying problems, it remains in a half-open state. Some calls may be responded to normally, but some may not be. It depends on why the circuit breaker switched to this state initially.



6. Command query responsibility segregation (CQRS)

A developer might use a command query responsibility segregation (CQRS) design pattern if they want a solution to traditional database issues like data contention risk. CQRS can also be used for situations when app performance and security are complex and objects are exposed to both reading and writing transactions.

The way this works is that CQRS is responsible for either changing the state of the entity or returning the result in a transaction. Multiple views can be provided for query purposes, and the read side of the system can be optimized separately from the write side. This shift allows for a reduction in the complexity of all apps by separately querying models and commands so:

- The write side of the model handles persistence events and acts as a data source for the read side
- The read side of the model generates a projections of the data, which are highly denormalized views

7. Asynchronous messaging

If a service doesn't need to wait for a response and can continue running its code post-failure, asynchronous messaging can be used. Using this design pattern, microservices can communicate in a way that's fast and responsive. Sometimes this pattern is referred to as event-driven communication.

To achieve the fastest, most responsive app, developers can use a message queue to maximize efficiency while minimizing response delays. This pattern can help connect multiple microservices without creating dependencies or tightly coupling them. While there are tradeoffs one makes with async communication (such as eventual consistency), it's still a flexible, scalable approach to designing a microservices architecture.

8. Event sourcing

The event sourcing design pattern is used in microservices when a developer wants to capture all changes in an entity's state. Using event stores like Kafka or alternatives will help keep track of event changes and can even function as a message broker. A message broker helps with the communication between different microservices, monitoring messages and ensuring communication is reliable and stable. To facilitate this function, the event sourcing pattern stores a series of state-changing events and can reconstruct the current state by replaying the occurrences of an entity.

Using event sourcing is a viable option in microservices when transactions are critical to the application. This also works well when changes to the existing data layer codebase need to be avoided.

9. Strangler

Developers mostly use the strangler design pattern to incrementally transform a monolith application to microservices. This is accomplished by replacing old functionality with a new service — and, consequently, this is how the pattern receives its name. Once the new service is ready to be executed, the old service is "strangled" so the new one can take over.

To accomplish this successful transfer from monolith to microservices, a facade interface is used by developers that allows them to expose individual services and functions. The targeted functions are broken free from the monolith so they can be "strangled" and replaced.

To fully understand this specific pattern, it's helpful to understand how monolith applications differ from microservices.

10. Decomposition patterns

Decomposition design patterns are used to break a monolithic application into smaller, more manageable microservices. A developer can achieve this in one of

three ways:

1. Decomposition by business capability:

Many businesses have more than one business capability. For example, an e-commerce store is likely to have capabilities that include managing product catalogs, inventory, orders, and delivery. A single monolithic application might have been used for every service in the past, but say, for example, the business decides to create a microservices application to manage these services moving forward. In this common scenario, the business might choose to use decomposition by business capability.

This may be used when an application has a large number of interrelated functions or processes. Developers may also use it when functions or processes are likely to change frequently. The benefit is that having more focused, smaller services allows for faster iterations and experimentation.

2. Decomposition by subdomain:

This is well suited for exceptionally large and complex applications that utilize a lot of business logic. For example, you might use this if an application uses multiple workflows, data models and independent models. Breaking the application into subdomains helps make managing the codebase easier while facilitating faster development and deployment. An easy-to-grasp example is a blog that's hosted on a separate subdomain (for instance, blog.companyname.com). This approach can separate the blog from the root domain's business logic.

3. Decomposition by transaction:

This is an appropriate pattern for many transactional operations across multiple components or services. Developers could choose this option when there are strict consistency requirements. For example, consider cases where an insurance claim is submitted. The claim request might interact with both a *Customers* application and *Claims* microservices at the same time.

Utilizing design patterns to make organization more manageable

Setting up the proper architecture and process tooling will help you create a successful microservice workflow. Use the design patterns described above and learn more about microservices in our blog to create a robust, functional app.

Originally published at https://www.capitalone.com.

Written by Capital One Tech

5.3K Followers · Editor for Capital One Tech

From our founding, we've used tech to change the banking industry. Today, our innovations are making banking better for tens of millions of our customers.

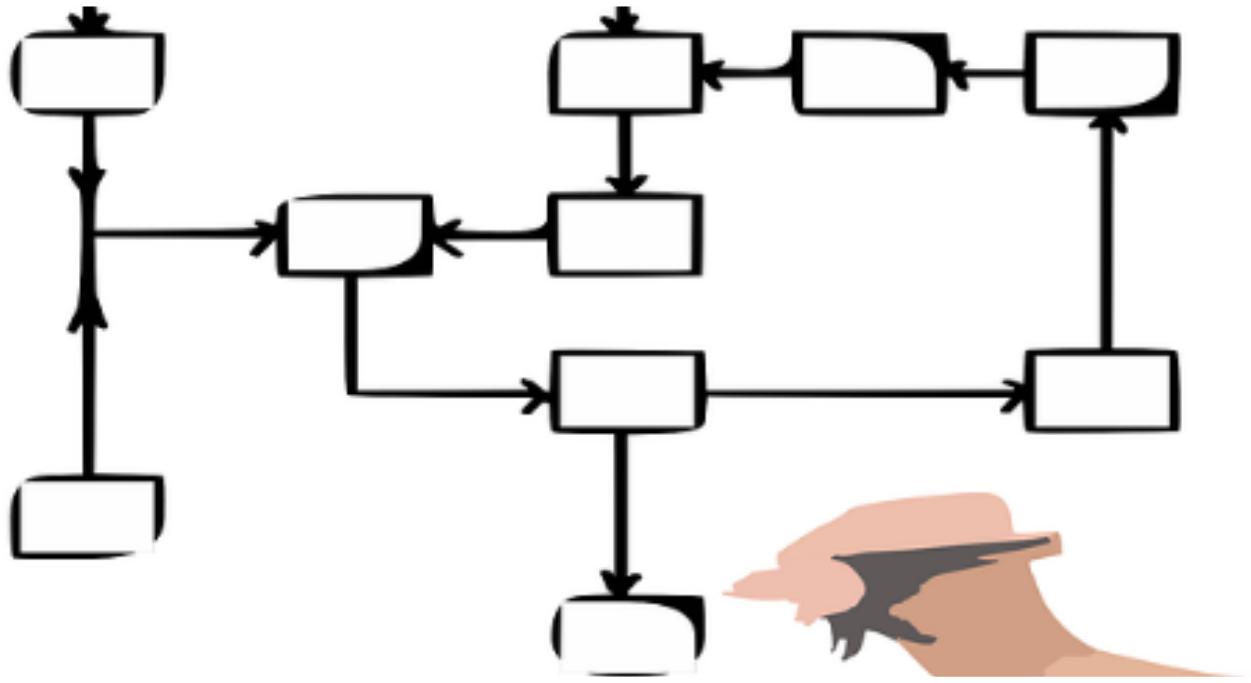
More from Capital One Tech and Capital One Tech

Capital One Tech in Capital One Tech

Python guide: Using multiprocessing versus multithreading

An overview of the fundamental differences between multithreading and multiprocessing in Python

Sep 15, 2023 9



Andrew Bonham in Capital One Tech

2022 Open Source BPM Comparison

A compare and contrast of open source BPM products

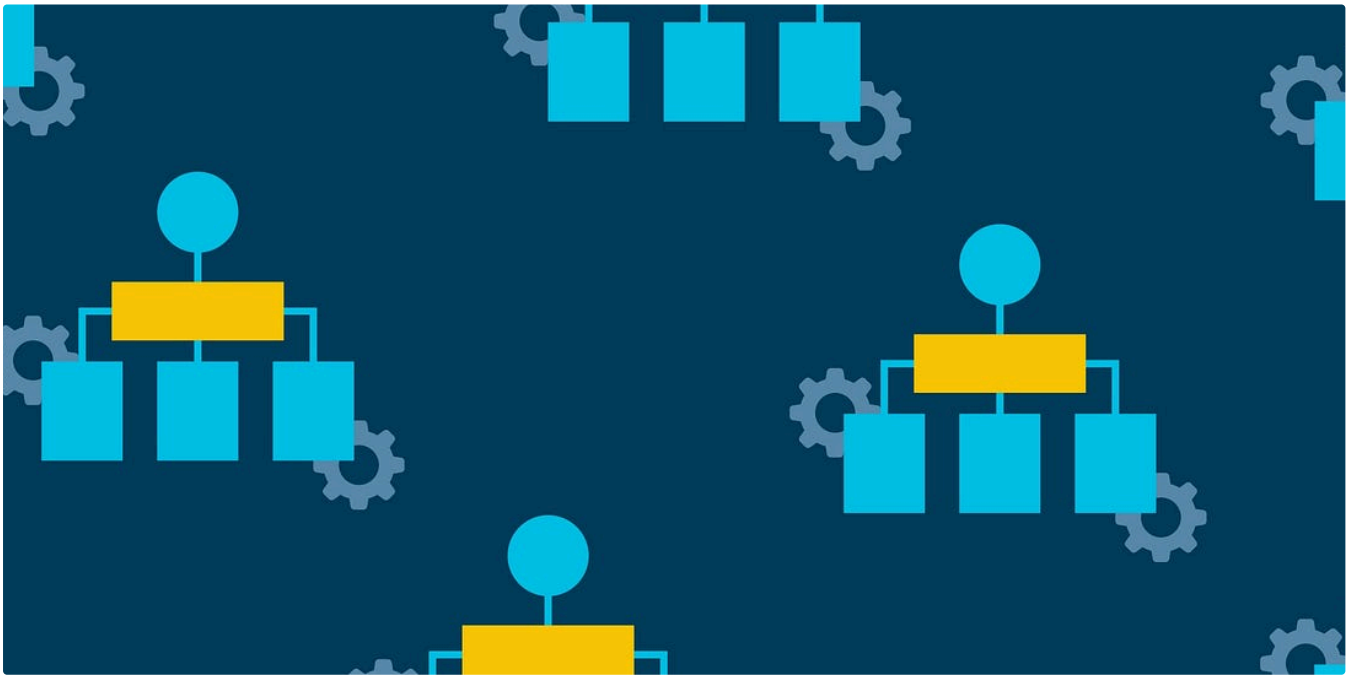
Sep 8, 2022 351 2

Jon Bodner in Capital One Tech

Learning to Use Go Reflection

Post 5 in a Series on Go

Dec 13, 2017 1.7K 4



Capital One Tech in Capital One Tech

Java programming: A deep dive into Java 21's key features

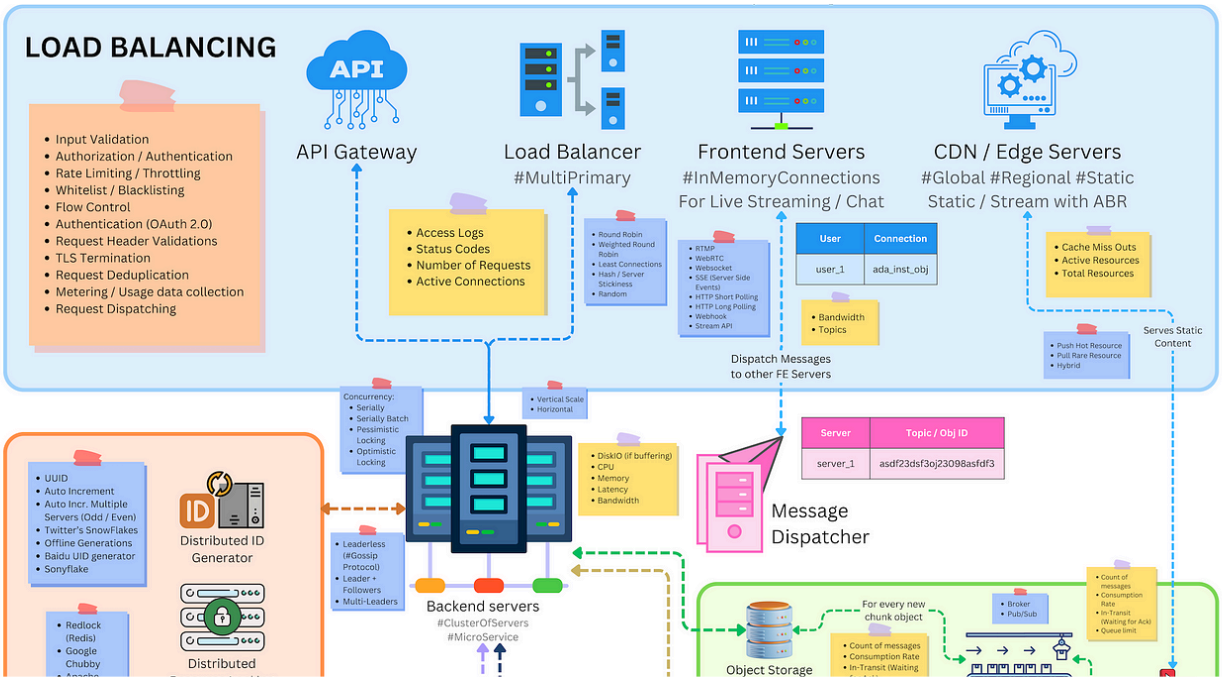
Java's ongoing relevance: A closer look at language evolution

Jan 30 611 2

See all from Capital One Tech

See all from Capital One Tech

Recommended from Medium



Love Sharma in ByteByteGo System Design Alliance

System Design Blueprint: The Ultimate Guide

Developing a robust, scalable, and efficient system can be daunting. However, understanding the key concepts and components can make the...

Sep 17, 2023 8.8K 60

Amazon.com Seattle, WA
Software Development Engineer Mar. 2020 – May 2021

- Developed Amazon checkout and payment services to handle traffic of 10 Million daily global transactions
- Integrated Iframes for credit cards and bank accounts to secure 80% of all consumer traffic and prevent CSRF, cross-site scripting, and cookie-jacking
- Led Your Transactions implementation for JavaScript front-end framework to showcase consumer transactions and reduce call center costs by $25 Million
- Recovered Saudi Arabia checkout failure impacting 4000+ customers due to incorrect GET form redirection

Projects

NinjaPrep.io (React)
- Platform to offer coding problem practice with built in code editor and written + video solutions in React
- Utilized Nginx to reverse proxy IP address on Digital Ocean hosts
- Developed using Styled-Components for 95% CSS styling to ensure proper CSS scoping
- Implemented Docker with Seccomp to safely run user submitted code with < 2.2s runtime

HeatMap (JavaScript)
- Visualized Google Takeout location data of location history using Google Maps API and Google Maps heatmap code with React
- Included local file system storage to reliably handle 5mb of location history data
- Implemented Express to include routing between pages and jQuery to parse Google Map and implement heatmap overlay

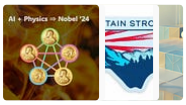
Alexander Nguyen in Level Up Coding

The resume that got a software engineer a $300,000 job at Google.

1-page. Well-formatted.

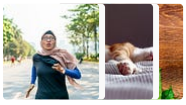
Jun 1 24K 489

Lists

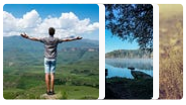


Staff Picks
749 stories · 1377 saves

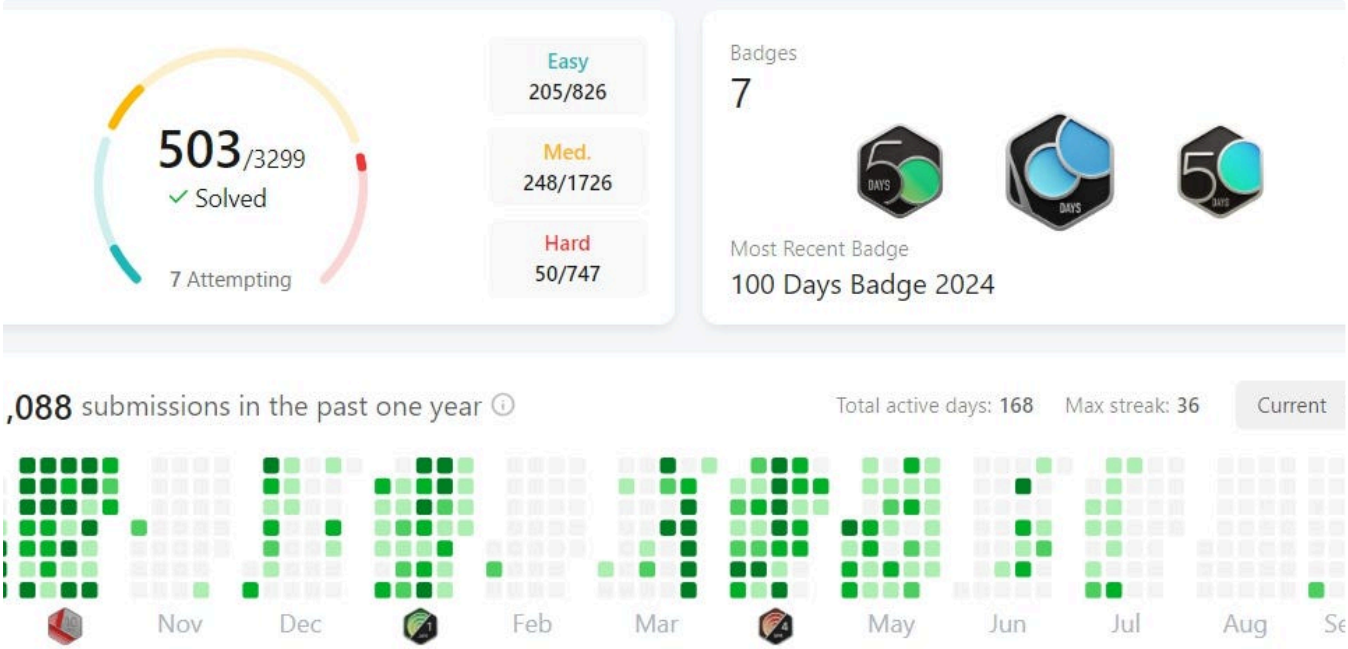
Stories to Help You Level-Up at Work
19 stories · 839 saves



Self-Improvement 101
20 stories · 2890 saves



Productivity 101
20 stories · 2454 saves



Surabhi Gupta in Code Like A Girl

Why 500 LeetCode Problems Changed My Life

How I Prepared for DSA and Secured a Role at Microsoft

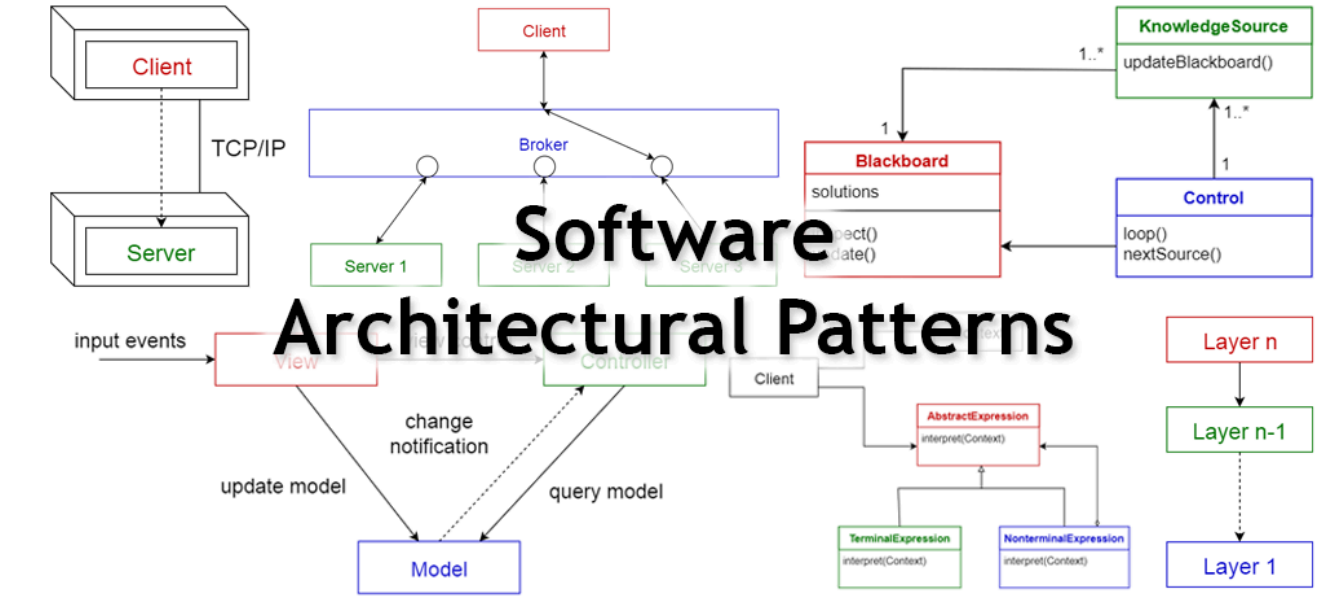
Sep 26 2.6K 55

Oliver Foster

Spring Boot: How Many Requests Can Spring Boot Handle Simultaneously?

My article is open to everyone; non-member readers can click this link to read the full text.

Jun 17 1.1K 14



Vijini Mallawaarachchi in Towards Data Science

10 Common Software Architectural Patterns in a nutshell

Ever wondered how large enterprise scale systems are designed? Before major software development starts, we have to choose a suitable...

Sep 4, 2017 41K 138

Rabinarayan Patra

Why 1==1 is true but 128==128 is false in Java

Ever wondered why comparing 1==1 returns true, but 128==128 returns false in Java? Let’s dive into the magic of Integer caching to find...

Sep 11 1.6K 42