A Practical Use Case of Software Architecture Evolution

Ionut Balosin

www.IonutBalosin.com | ionut.balosin@gmail.com | @ionutbalosin

Agenda

01 Layered Architecture

02 Hexagonal Architecture

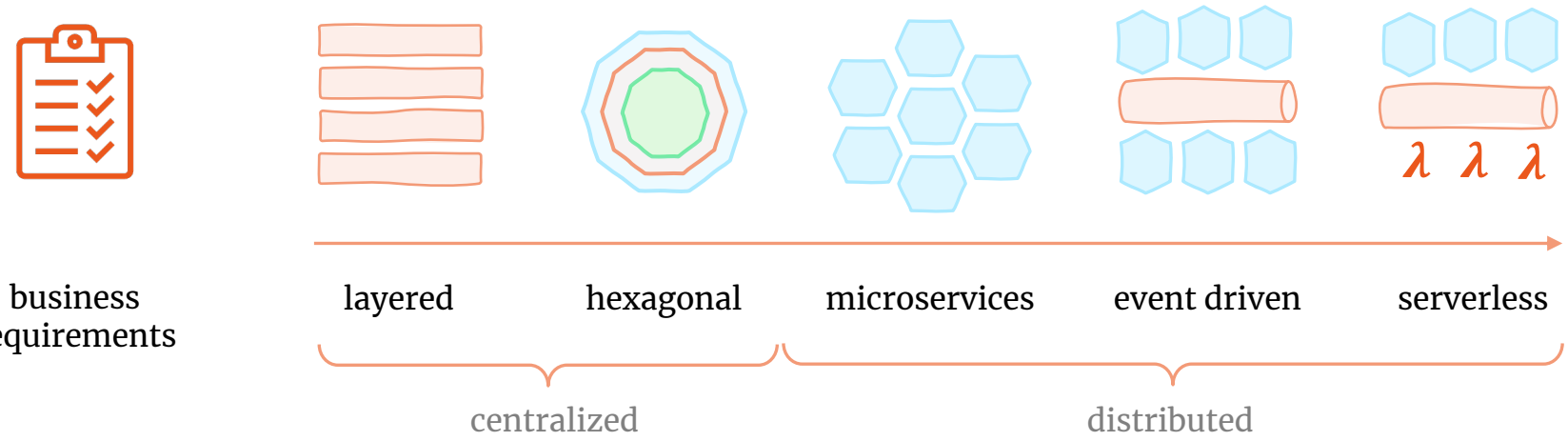
03 Microservices Architecture

04 Event Driven Architecture

05 Serverless Architecture

The Idea Behind This Talk

eCommerce use case: the evolution of software architecture driven by the continuously evolving business requirements.



Ionut Balosin

- Software Architect @ Raiffeisen Bank International
- Technical Trainer
- Security Champion
- Oracle ACE Associate
- Blogger
- Speaker

www.IonutBalosin.com | ionut.balosin@gmail.com | @ionutbalosin

My Training Catalogue

- Software Architecture Essentials
- Java Performance Tuning
- Designing High-Performance, Scalable, and Resilient Applications
- Application Security for Java Developers

Training figures: 80+ sessions | 900+ trainees | 1300+ hours | 10+ clients | 4+ countries
Conference figures: 35+ sessions | 14+ countries

www.IonutBalosin.com/training

Layered Architecture

01

Use Case I: eCommerce Platform

Key eCommerce features for a Minimum Viable Product (MVP)

Individual user accounts for each customer

A comprehensive catalogue of available products

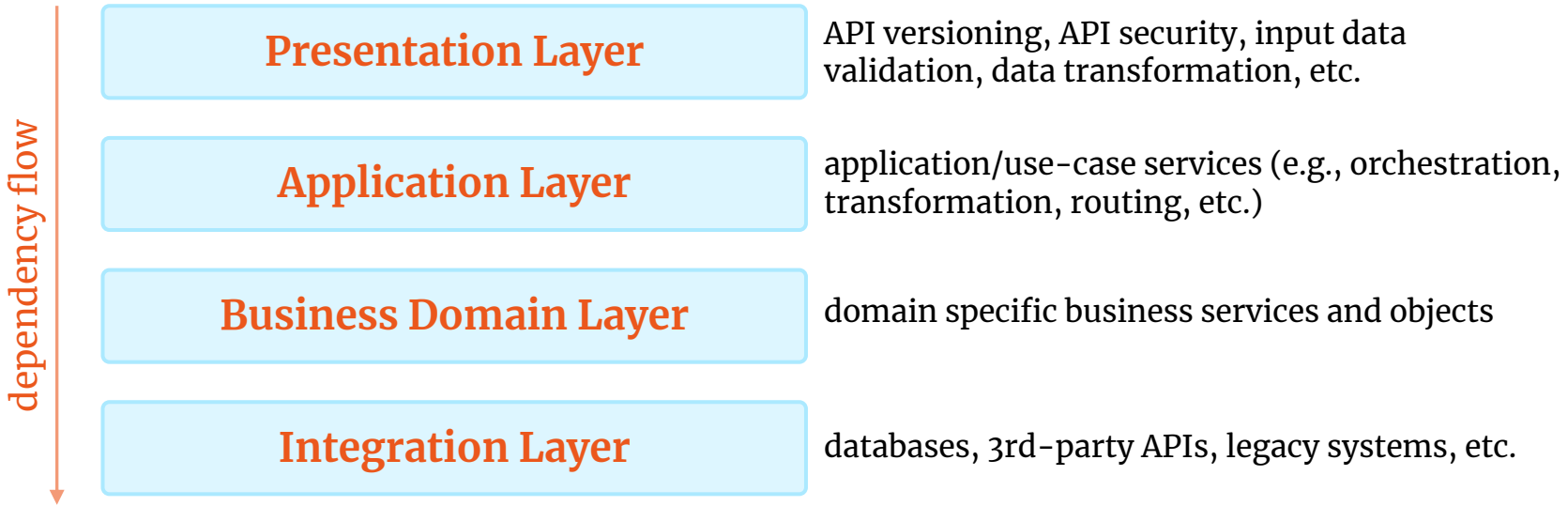
User-friendly search for adding products to shopping cart

Seamless checkout via 3rd party payment system

Expecting thousands of users daily within the first 3-6 months

Limited budget, maximum return of investment (ROI)

Technical Assumption: Layered Architecture



At this stage, a simple **relaxed layered architecture** looks suitable for the MVP.

Account | Product Catalog | Order | Shopping Cart | Payment

Domain

relaxed layered

Presentation

RESTful API | API versioning | DTO Mapper

API input data validation | API security

Application

Service | DTO Mapper

Client to External Service | External Listener

DTO

Business Domain

Service | Domain Event

Entity, VO

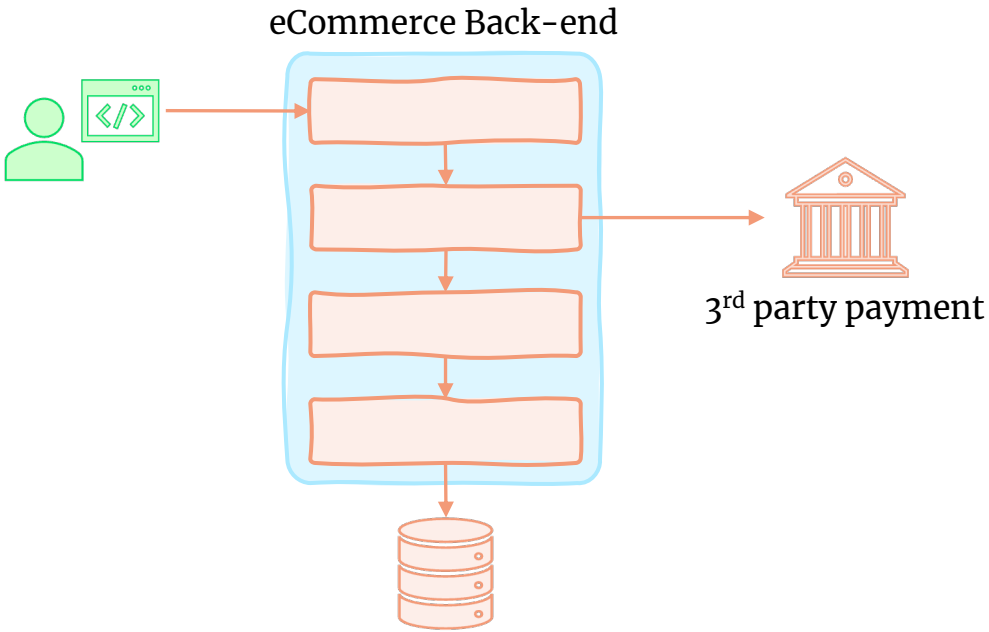
Repository

External Service

3rd party payment

Legend:
DTO - Data Transfer Object
VO - Value Object

Monolithic Layered Architecture



The Key Benefits of Layered Architecture

☺ **Simple and straightforward (modular) architecture**, suitable for current business requirements.

☺ Each **layer encapsulates distinct responsibilities**, making testing easier.

Cons of Layered Architecture

☹ Without careful design, there's a **risk of tightly coupling layers** (e.g., application, integration) **to external dependencies** such as DBs or external APIs, making it challenging to swap in/out other sources in the future.

☹ In general, enforcing **strict layering rules adds unnecessary complexity** to the call chain, especially when intermediate layers are not required. A **relaxed layered architecture is preferable**.

Hexagonal Architecture

02

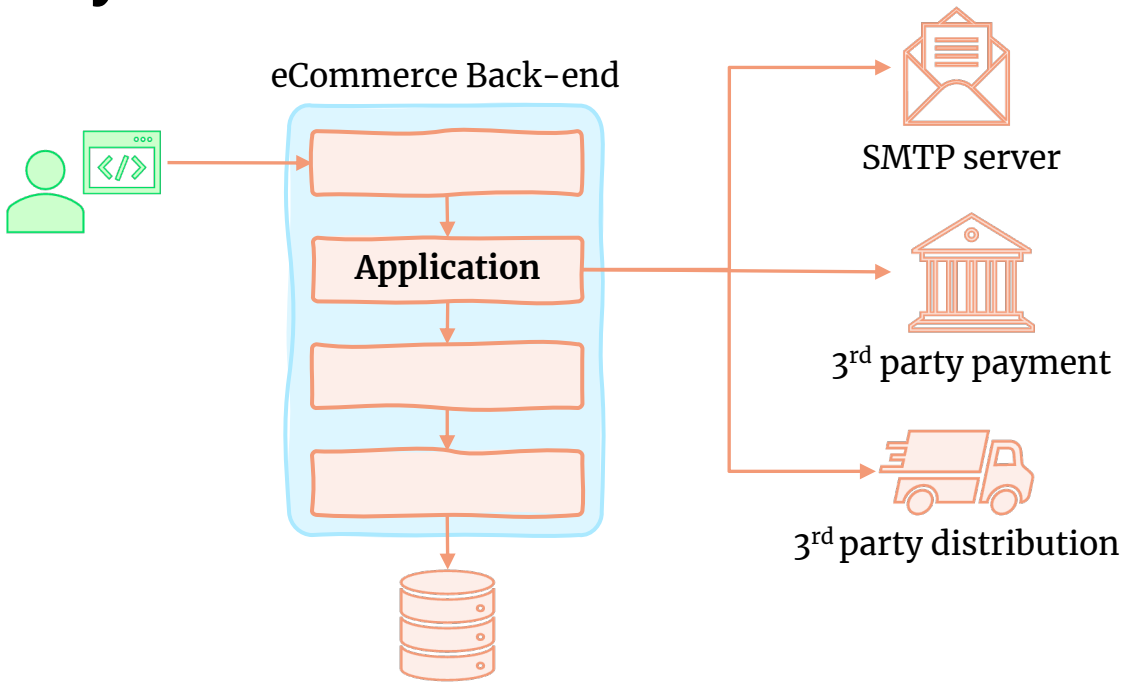
Use Case II: eCommerce Platform

After the initial release of the eCommerce platform, the management wants to add extra features to improve the user experience.

A 3rd party distribution system to deliver purchased products

A notification feature for order and payments tracking

A Possible Layered Architecture Extension

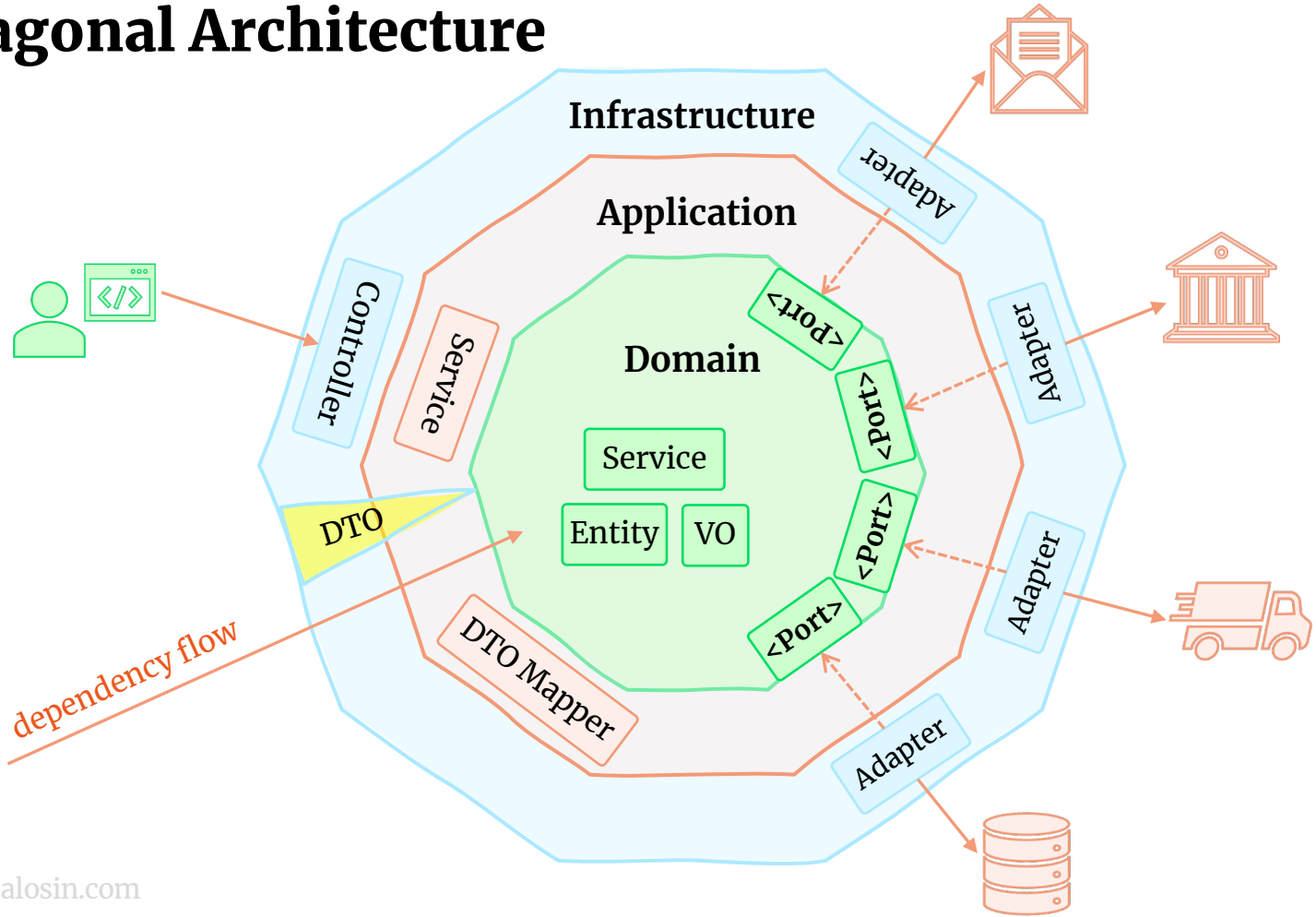


In addition to the SMTP server and third-party product distribution integrations, the application and business domains will also be extended.

A Major Drawback of Current Layered Architecture

☹ Multiple **dependencies on third-party systems from the application layer** (business logic) **complicate services decoupling** and testing (i.e., system evolvability).

Hexagonal Architecture



The Key Benefits of Hexagonal Architecture

☺ **I/O data sources are outside of any business logic**, enabling them to be easily interchangeable. This decoupling allows I/O data sources to be easily substituted without touching any business logic.

☺ **The business logic can be tested separately**, without relying on specific I/O protocols.

☺ Hexagonal architecture enables the implementation of a **domain-centric** approach, or even facilitates a **domain-driven** design (DDD).

Cons of Hexagonal Architecture

☹ **The number of interfaces (i.e., ports) increases** with the number of external I/O sources, it can make the overall functionality more complex and harder to understand.

When to Use a Hexagonal Architecture

- In the case of applications involving **multiple external integrations**.
- In the case of applications with **complex business logic or domain models**.

When to NOT Use a Hexagonal Architecture

⊗ **For applications with a simple business domain** (e.g., CRUD application) or **even without a (real) defined domain, and minimal or no integrations**, hexagonal architectures might add unnecessary complexity.

⊗ **Not ideal for prototyping**, it is more suited for long-run architectures.

eCommerce Application Demo

Hexagonal Architecture in Practice

Reference: [https://github.com/ionutbalosin/ecommerce-app]

Microservices Architecture

03

Use Case IV: eCommerce Platform

The initial eCommerce platform MVP has now evolved into a profitable business, generating significant profits. Management sees even greater potential in the future but has identified two key bottlenecks.

Slower feature deliveries due to longer release cycles

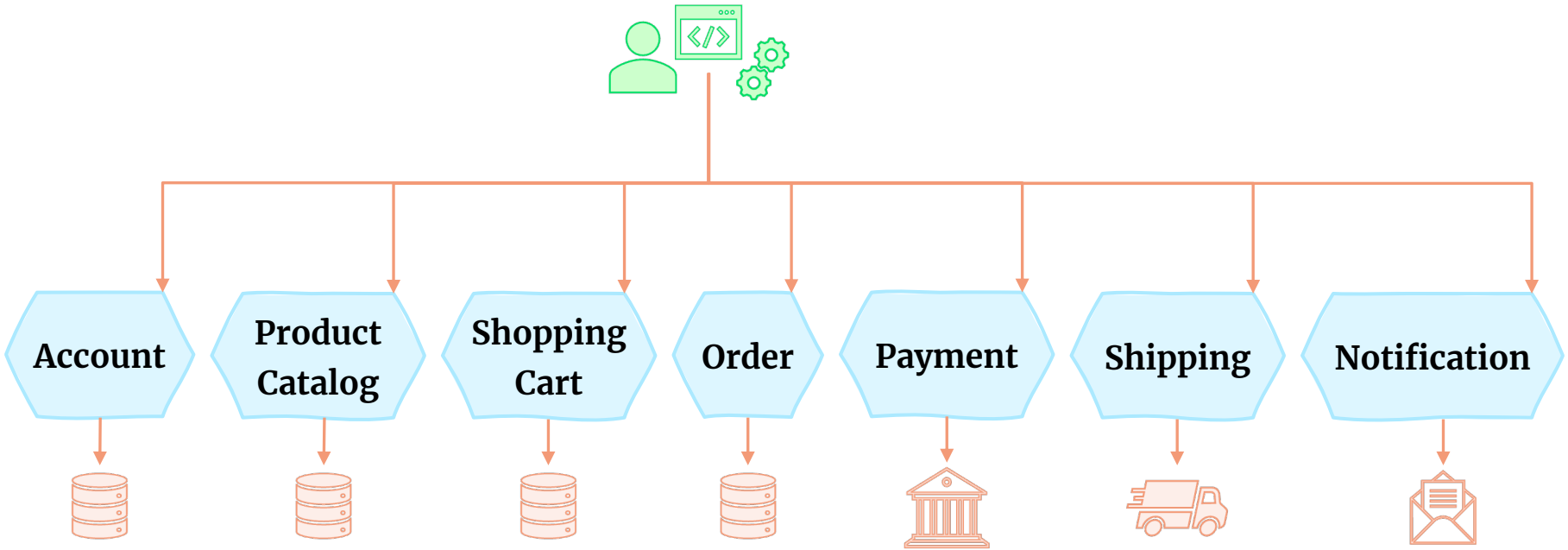
Hire new team members to cope with the backlog of features

NETFLIX

“the decision to decompose the monolith to specific services [..] was not geared by performance issues – but with setting boundaries around all of these different domains and enabling dedicated teams to develop domain – specific services independently”

Source: [Ready for changes with Hexagonal Architecture]

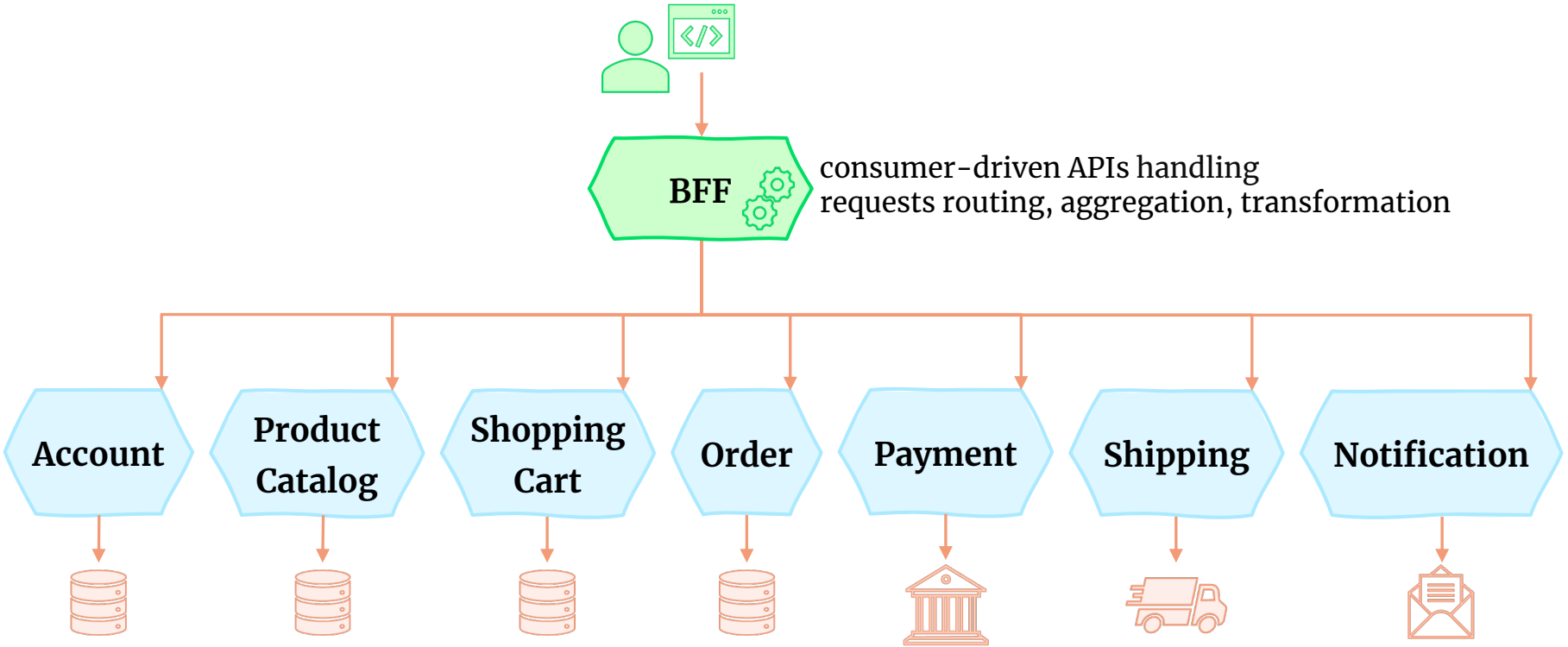
Microservices Architecture



A Major Drawback of Current Microservices Architecture

☹ The **requests orchestration logic has now been shifted to the client side**, resulting in high coupling and excessive communication with the back-end services.

Backend-For-Frontend (BFF)



Backend-For-Frontend (BFF)

BFF is tightly coupled to a specific user experience (i.e., consumer-driven APIs).

Key Benefits

✅ loose coupling between client and domain services.
✅ keeps the domain services logic cleaner and agnostic of the user experience.

Notes

⚠️ there must be a 1:1 relationship between BFF and one consumer type.
⚠️ the BFF should contain only behaviour logic and not any business logic (neither any persistent state).

The Key Benefits of Microservices Architecture

- Easier workload distribution across teams. (i.e., each team owns its microservices).
- Faster and independent development (i.e., quicker release cycles).
- Improved data segregation and governance (i.e., each domain service owns its data).
- Services adhering to DDD could create a highly decoupled architecture.
- Enhanced scalability and availability of individual services.

Before adopting the microservices style make sure the infrastructure is ready to support it.

Distributed architectures impose other challenges such as **monitoring**, **aggregated logging**, **tracing**, etc.

When to NOT Use a Microservices Architecture

- ⊗ **A few domains with limited capabilities** might benefit more from a modular monolith.
- ⊗ **Few users or requests** may not justify the overhead of multiple microservices.
- ⊗ **Tight data coupling** makes it challenging to segregate across distributed services.
- ⊗ **High networking overhead** can hinder efficient data movement.
- ⊗ Enhancing modularity within an existing monolith or optimizing its components might be preferable before transitioning to microservices.

Event Driven Architecture

04

Use Case V: eCommerce Platform

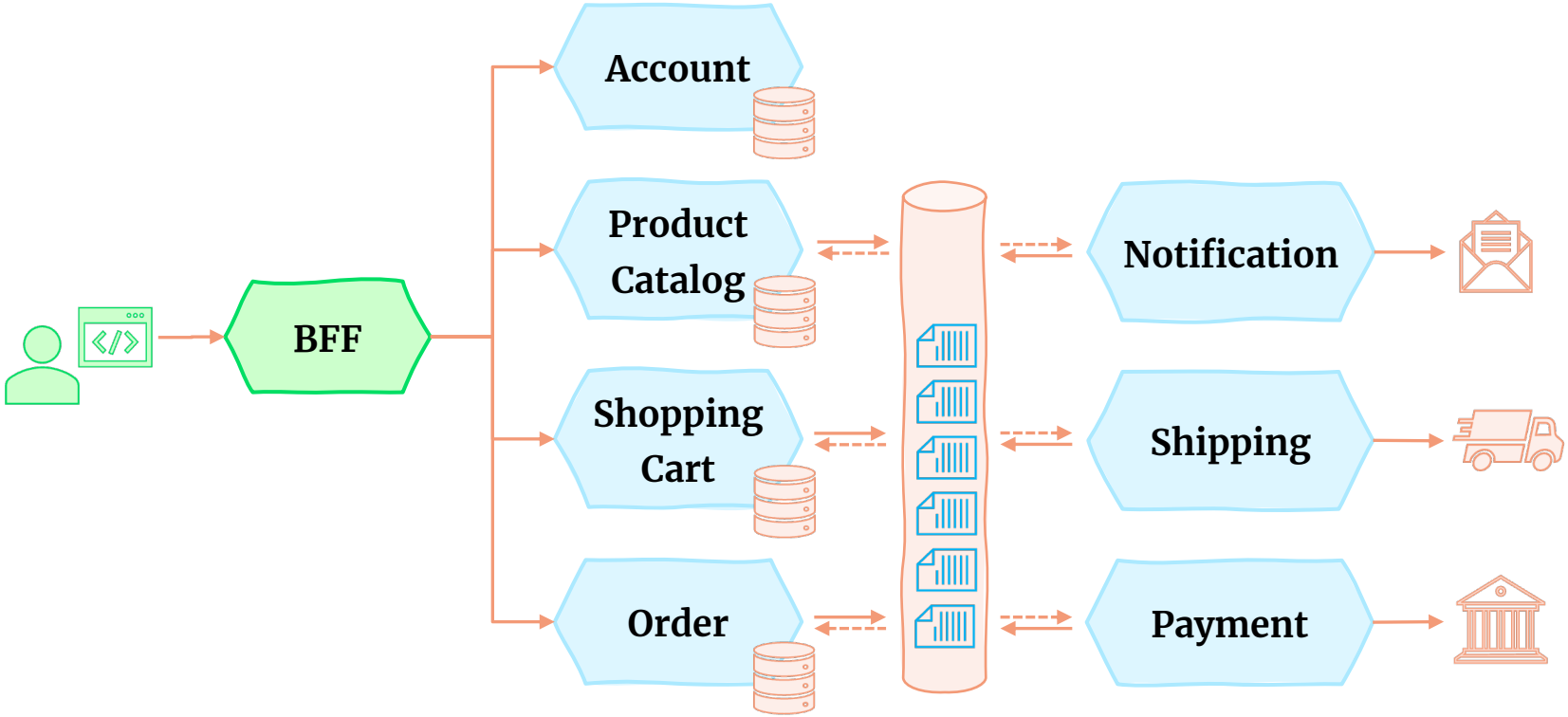
The eCommerce platform is now one of the preferred online shopping platforms in the country. However, during peak times, especially with thousands of concurrent users, other problems started to emerge.

Some services, like Order, Payment, and Shipping, are becoming unresponsive on high load

Some (shipping, payment) orders were lost, leading to frustration among users

The analysis revealed that even horizontal scalability of these services did not handle the high number of requests adequately.

Event Driven Architecture (EDA)



The Key Benefits of Event Driven Architecture

☺ Event-driven architectures (leveraging on message queues) **enables asynchronous communication** between components.

☺ Message queues facilitate **effective workload handling**, improving system responsiveness during load fluctuations.

☺ **Loosely coupled** relationships between **producers and consumers** services (especially in communication with external systems, establish an effective asynchronous decoupling model).

Cons of Event Driven Architecture

☹ Given its **increased complexity** (e.g., managing event schemas, event correlation, event ordering, event reprocessing, error handling, testing, debugging), **use EDA only in the critical parts of the system** where it is really needed.

☹ Understanding (and extending) the **overall flow of events** in larger systems and their relationships **requires more time**.

eCommerce Application Demo

Event Driven Architecture in Practice

Reference: [https://github.com/ionutbalosin/ecommerce-app]

Serverless Architecture

05

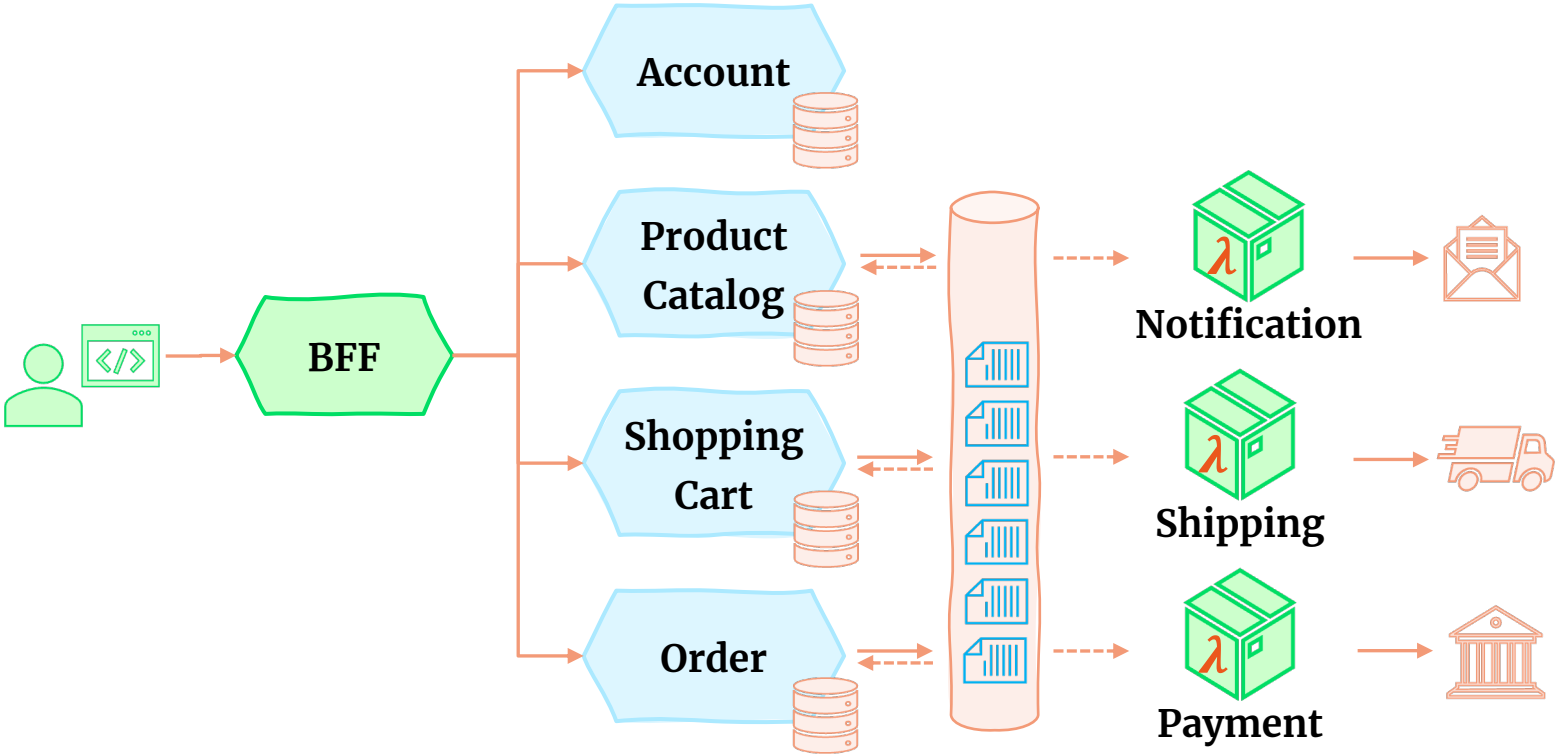
Use Case VI: eCommerce Platform

The eCommerce platform generates substantial income; however, management has decided to reduce operational costs due to their significant increase.

Higher operational costs that need to be made more efficient

The analysis revealed that some services are not always busy; however, even in idle mode, they still incur costs.

Function as a Service (FaaS)



The Key Benefits of Serverless Architecture

☺ FaaS (e.g., AWS Lambda, Google Cloud Functions, Microsoft Azure Functions) is a **cost optimization model** (‘**pay as you go**’ **model**).

☺ With FaaS, **managing the underlying infrastructure** (e.g., provisioning, scaling, and maintenance) is entirely the **responsibility of the cloud provider**.

Cons of Serverless Architecture

☹ FaaS has **execution time and memory limits** (e.g., AWS Lambda is constraint to 15 minutes maximum and 10,240 MB memory limit).
For longer tasks, complementary solutions are required (e.g., AWS Step functions).

☹ Serverless decreases the compute cost but **increases the networking communication costs**.

☹ The **cold start-up time** of FaaS may introduce slight execution delays compared to an already initialized long-running service.

When to Use a Serverless Architecture

✅ FaaS is suitable for **services with infrequent or irregular usage** patterns and/or for executing **periodical (short-running) scheduled tasks**.

Note: Although possible, implementing a **fully-fledged FaaS system**, especially for complex applications, may lead to **increased complexity** (e.g., state and event management, invocation overhead).

When to NOT Use a Serverless Architecture

⊗ For **long-running tasks**, **complex processing**, and **extensive computation**.

⊗ For **consistently high traffic and predictable workloads**, FaaS cold starts may negatively impact performance and scalability.

⊗ **Longer I/O wait times** can lead to **suboptimal cost efficiency** due to the billing model, where execution time is measured in extremely granular increments, often rounded up to the nearest 100 milliseconds.

Understand the strengths and weaknesses of each architectural style and combine the best from each.

In many cases, reaching an optimal solution involves a combination of multiple approaches.

Thank You

Additional Resources

Code Source: [https://github.com/ionutbalosin/ecommerce-app]

Skill Up Now: [https://ionutbalosin.com/training]

Appendix

Database Scalability Patterns

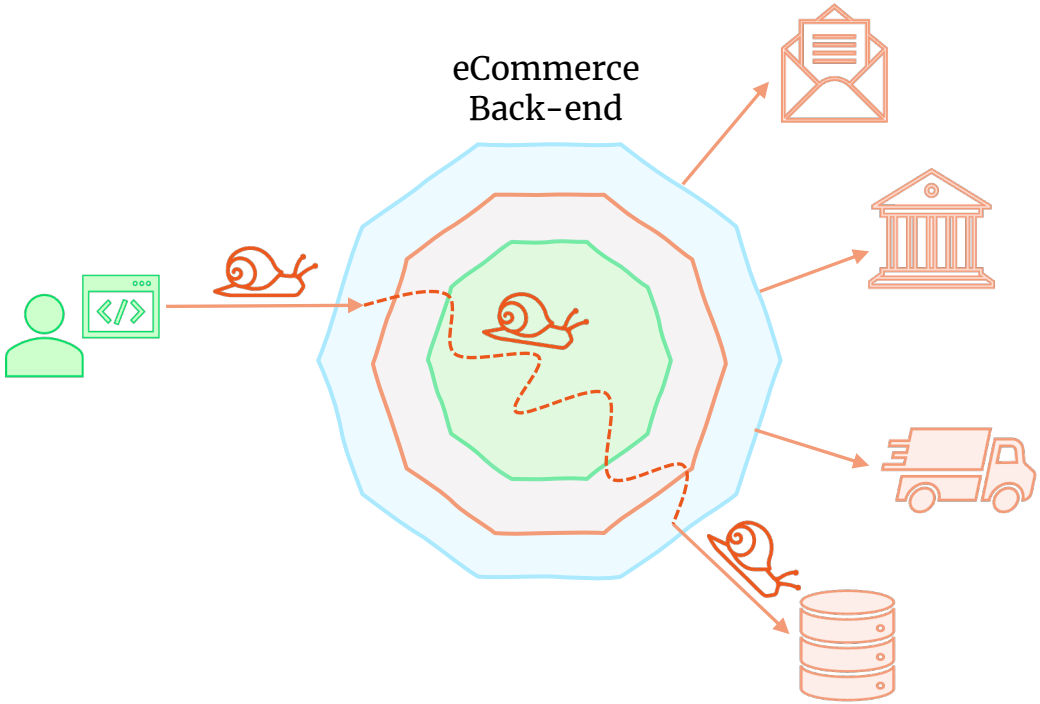
Use Case III: eCommerce Platform

The platform gained popularity and attracted more users. The product catalogue expanded significantly from its initial volume. However, some problems have emerged.

Slower core operations: product search, order placement, etc.

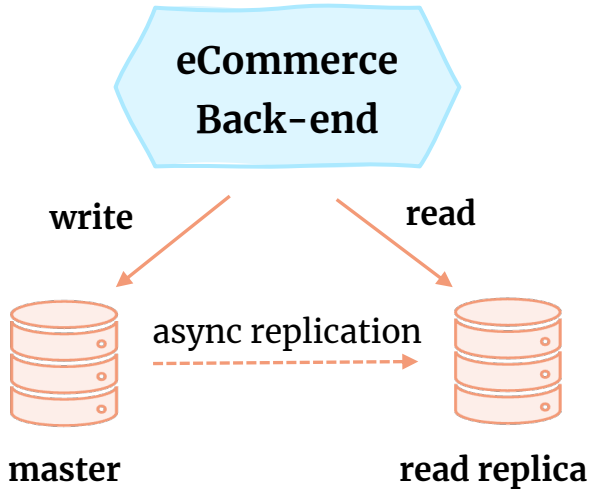
The analysis revealed that increased traffic leads to **database bottlenecks**.
Tuning the database queries/connection pools did not provide much improvement.

Database Bottlenecks



Option I: Database Read Replicas

Read replicas are copies of a primary database that are used to offload read operations from the primary database server (useful for use cases with more reads than writes).

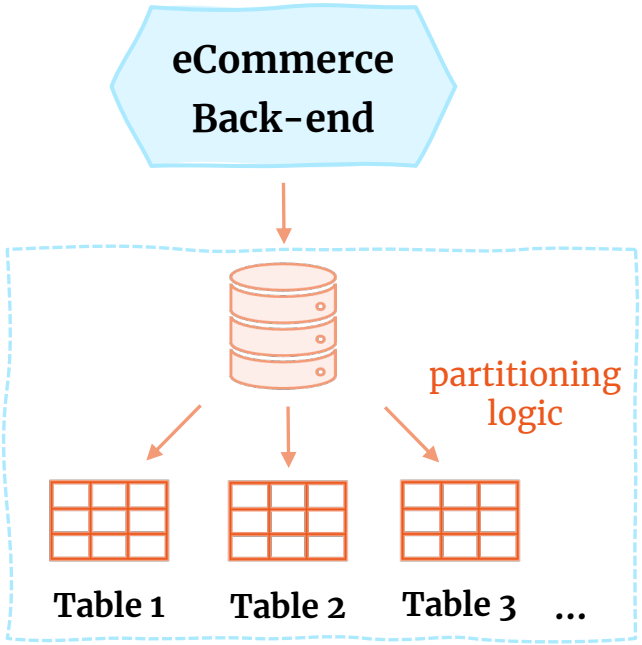


Replication setup combines manual configuration steps with built-in database mechanisms or tools.

Option II: Database Partitioning

Partitioning (i.e., **vertical partitioning**) divides tables within a database instance into smaller sub-tables or partitions, thereby splitting up the database.

Partitioning is efficient in the case of **large datasets** that can be **queried in parallel** across multiple partitions.



Option III: Database Sharding

Sharding (i.e., **horizontal partitioning**) is a database partitioning technique that divides a large database into smaller, more manageable parts called shards.

Sharding is efficient when the **data could be logically separable** based on specific partitioning criteria (e.g., key, hash value, geo, etc.).