An Introduction to Microservice Principles and Concepts

Tags	Anti-corruption layer, Bounded Contexts, Docker, Domain-Driven Design, Educative, Microservices
URL	https://www.educative.io/courses/introduction-microservice-principles-concepts

CONTENTS

I. Preface

Microservices require solutions for different challenges:

1. **Integration** - frontend, synchronous & asynchronous microservices
2. **Operation** - monitoring, log analysis, tracing

Micro & Macro

Docker for Microservices

II. Microservices

Microservices - independently deployable modules

- Speeds up deployment
- Reduces the number of necessary tests

Advantages:

1. **Easy scalability of deployment**
    a. Internal structure of Docker container doesn't matter as long as the interface is present and functions correctly.
    b. Teams can make independent decisions, including programming languages.
    c. Individual features can be brought into production on its own.

2. **Replacing legacy systems**
    a. Can replace parts of the old system with integration between old system and new microservices: data replication, REST, messaging, or at level of UI.
    b. New microservices can be like a greenfield project (no old constraints), and devs can employ a new tech stack.

3. **Sustainable development**
    a. Replaceability of microservices - when a microservice can no longer be maintained, it can be rewritten.
    b. In order to do this, the **dependencies between microservices have to be managed appropriately.**

4. **Robustness**
    a. When a memory leak exists, only the specific microservice is affected and crashes. Others compensate for the failure of the crashed microservice (resilience).
    b. To achieve resilience, microservices can cache values and use them in case of a problem, or fallback to a simplified algorithm.

5. **Independent Scaling, Free Technology Choice, Security**

a. Scaling the whole system is not required.

b. Each microservice can be implemented with a different technology; simpler and less risky to gain experience with new technologies.

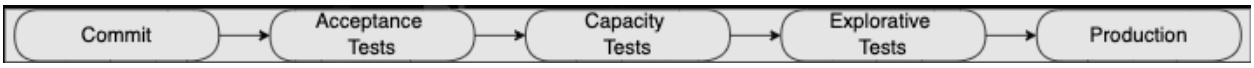
c. Introduce firewalls between microservices and encrypt communication to prevent corruption of additional microservices if hacker takes over one.

d. Stronger isolation enables all of these advantages. Decoupling makes security easier to verify, performance easier to measure, and makes design/development easier.

Managing Dependencies

1. **Classical Architecture** - more dependencies are introduced over time; original design architecture becomes more violated, culminating in completely unstructured system
2. **Microservices Architecture** - clear boundaries due to their interface, so unlikely that architecture violations will occur at the level of dependencies between microservices (interfaces are architecture firewalls).

Continuous Delivery Pipeline



PHASES

1. **Commit Phase** - software compilation, unit tests, static code analysis

2. **Acceptance Tests Phase** - automated tests to assure correctness of software regarding domain logic

3. **Capacity Tests Phase** - check performance at expected load
4. **Explorative Tests Phase** - analyze new functionalities or aspects that are not yet covered by automated tests
5. **Production Phase**

Microservices facilitate continuous delivery:

- Continuous delivery pipeline is faster because deployment units are smaller.
- Tests need to cover fewer functionalities.
- Setting up CD pipeline is faster and takes less powerful hardware to run.
- Deployment of a microservice poses smaller risk than deployment of a monolith.

However, deployment must be automated!

- Integration tests conflict with microservice independence and introduce dependencies between the CD pipelines. So integration tests must be reduced to a minimum.

Two levels of microservices: domain and technical

- Course-grained division by domain
- For technical reasons, some microservices can be further divided

Challenges of Microservice Architecture

1. Operation requires more effort. Feasible when operation is largely automated.
2. Must be independently deployable.
3. Testing must be independent.
4. Changes that affect multiple microservices are more difficult to implement.
5. Compared to local communication, microservices that communicate through a network experience higher latency, and higher likelihood of failure. Microservice system relies on availability of other microservices.

III. Micro & Macro Architecture

- The **micro architecture** comprises all decisions that can be made individually for each microservice.
- The **macro architecture** consists of all decisions that can be made at a global level and apply to all microservices.

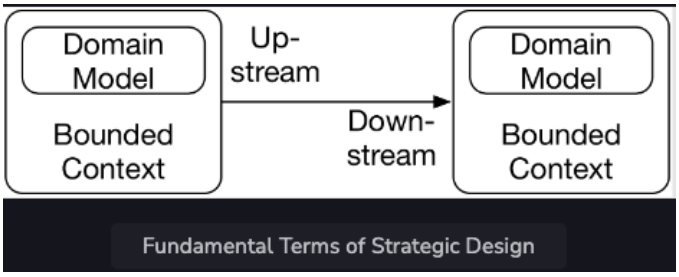
Domain-Driven Design & Bounded Contexts

Domain-driven design (DDD) offers a collection of **patterns** for the domain model of a system.

Each domain model is valid only in a **bounded context**.

For the communication between bounded contexts, we can use **domain events**.

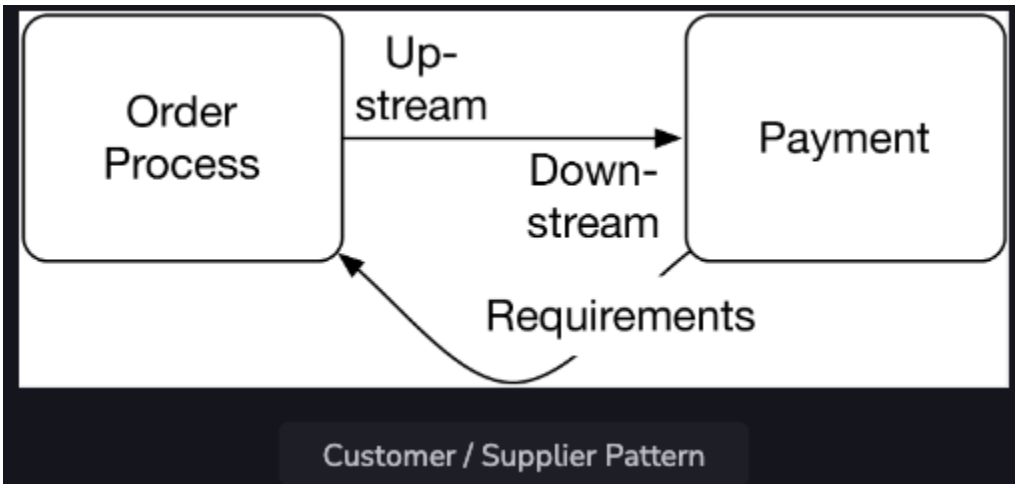
Strategic Design - the integration of bounded contexts.



The upstream team can influence the success of the downstream team, but not the other way around. For example, the success of the team responsible for payment depends on the order process team. If data such as prices or credit card numbers are not part of the order, it is impossible to do the payment. However, the order process does not depend on the payment to be successful. Therefore, **order processing is upstream**. It can make payment fail. **Payment is downstream** since it cannot make the order process fail.

1. **The customer/supplier pattern -** the supplier is upstream and the customer is downstream.

.

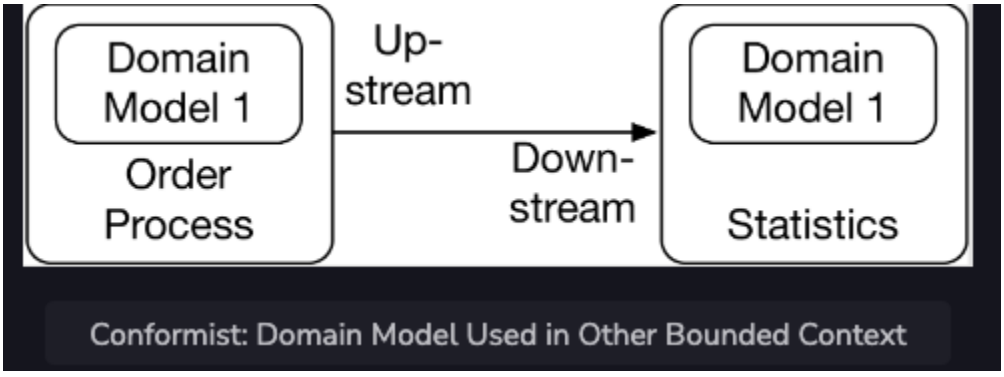


In the drawing above, for example, payment uses the model of the order process. However, payment defines requirements for the order process. Payment can only be done successfully if the order process provides the required data.

So, payment can become a customer of the order process. That way the customer's requirements can be included in the planning of the order process.

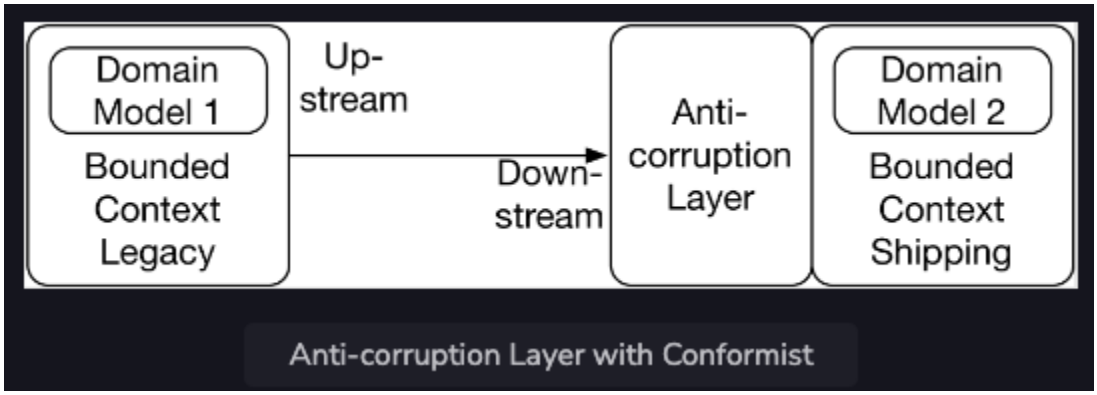
A pattern like *customer/supplier* may not be desirable as it **requires a lot of coordination.**

2. **The conformist pattern -** a bounded context simply uses a domain model from another bounded context.

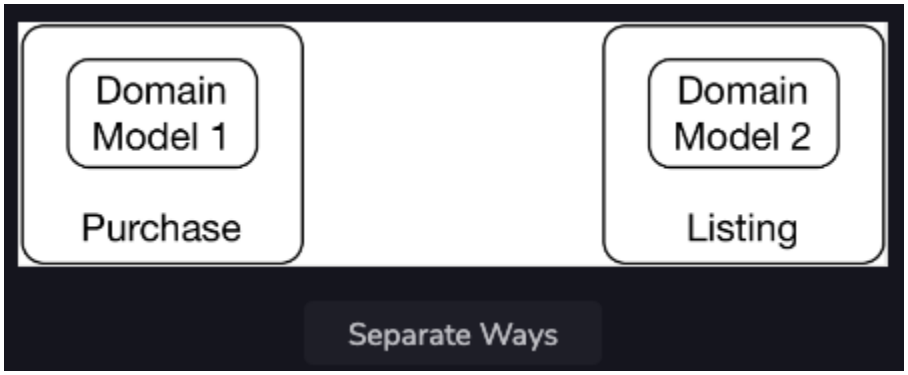


The data warehouse team could not demand additional information from the other bounded context. However, it is still possible that they would receive additional information out of altruism. Essentially, the data warehouse team is not deemed important enough to get a more powerful role.

3. **The anti-corruption layer** pattern - the bounded context does not directly use the domain model of the other bounded context, but contains a layer for decoupling its own domain model from the model of the bounded context. This is useful in conjunction with the conformist pattern to generate a separate model decoupled from the first.

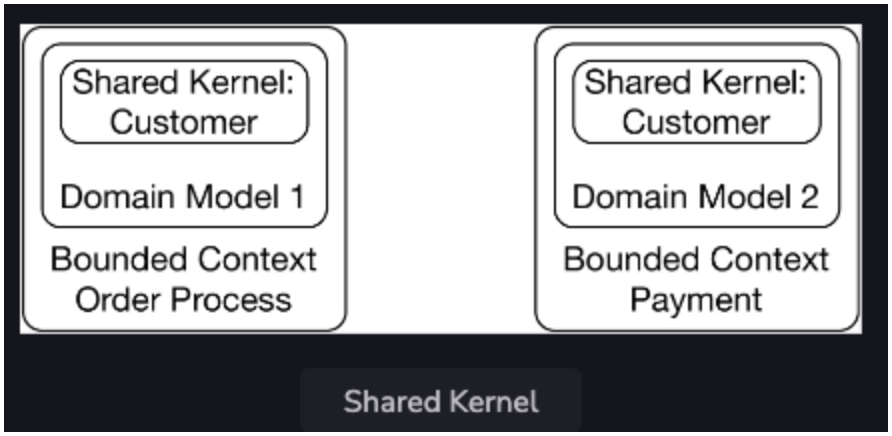


4. **The separate ways** pattern - the bounded contexts are not related at the software level; the systems are independent and can be evolved completely independently.

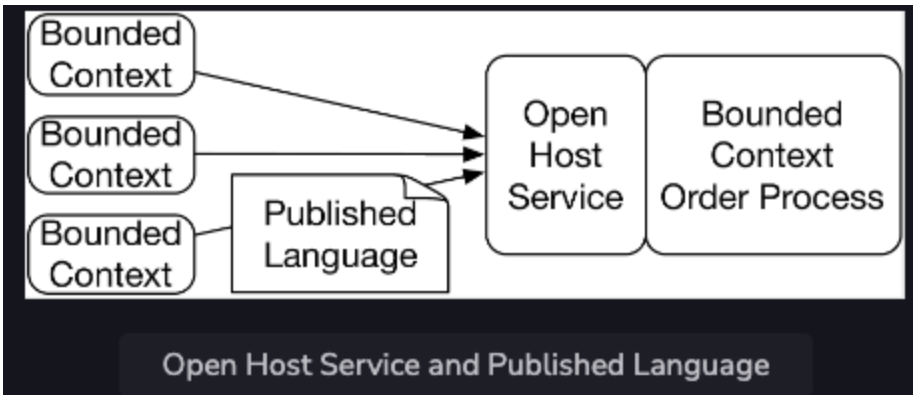


5. The **shared kernel** pattern - a common core shared by multiple bounded contexts. i.e. the data of a customer. The shared kernel comprises shared

business logic and shared database schema, therefore it is an anti-pattern for microservice environments, but can be used in deployment monoliths that follow DDD.



6. The **open host service** pattern - the bounded context offers a generic interface with several services. Other bounded contexts can implement their own integration with these services. Frequently found at public API's on the internet.
7. The **published language** model - domain model accessible by all bounded contexts. Such as the EDIFACT for transactions between companies.



The choice of patterns has to be in line with:

1. The domain
2. The power structures
3. The communication relationships between the teams.

Architecture Decisions

Programming languages, frameworks, and infrastructure can be defined for each **microservice** individually at the **micro architecture** or uniformly across the **macro architecture.**

DATABASES

Micro: Each microservice can also have its own instance of the database. If databases were defined at the **micro architecture**. A crash of one database will cause only one microservice to fail which makes the entire app **more robust**.

Macro: To avoid needing many different databases, the database can be defined as part of the **macro architecture** for all microservices. Even if the database is defined in the macro architecture, **multiple microservices must not share a database schema**. That would contradict the bounded contexts.

USER INTERFACE

Sometimes different UI's for different users of a subsystem is fine. Other times, a uniform style guide is needed across parts of a system.

DOCUMENTATION can also be micro or macro.

Typical Macro Architecture Decisions

1. **Communication Protocol** - ie REST interface or messaging interface; data format: JSON or XML

2. **Authentication**
3. **Integration Testing**

Typical Micro Architecture Decisions

1. **Authorization**
2. **Testing** (besides integration testing)

The following table shows the typical micro and macro architecture decisions:

Micro or Macro	Micro Architecture	Macro Architecture
Programming Language	Continuous Delivery Pipeline	Communication Protocol
Database	Authorization	Authentication
Look and Feel	Tests of the Microservice in Isolation	Integration Tests
Documentation		