Ca' Foscari University of Venice

Project and written exam (1/2)

- 3 group tasks during the course (two thirds of the grade)
  - 2 small ones at the beginning: DONE!
  - 1 big task at the end implementing the architecture
    - Using **<u>any technology</u>**, not only the ones I'll introduce you
    - Containing **<u>at least one monolithic and one distributed architecture</u>**
    - Deadline: before the written exam (**<u>January 13th</u>**)!
- Written exam at the end of the course (one third of the grade)
  - **<u>Only after submitting and passing</u>** the tasks/project
  - Example from last year in Moodle
  - About **<u>only all</u>** the in presence lectures

Ca' Foscari University of Venice

Project and written exam (2/2)

- You will have to deliver at the end of the project
  - A GitHub repository containing the **code**
  - The **history of the repo** should make clear the contribution of the different students in the group
  - It must contain a **description** (pdf, markdown, …) of the structure of the architecture with references to the code and **how to run it**
    - Docker (file or compose) or Kubernetes (minikube)

Ca' Foscari University of Venice

Schedule

- Schedule for the next lectures:
  - Nov. 14th in Lab. 3
  - (Nov. 16th Google DevFest 24)
  - Tuesday Nov. 19th at 8:45 in Aula C
  - Nov. 28th invited lecture by Unox
    - https://www.unox.com/us_us/
  - Dec. 5th invited lecture by Gianluca Caiazza
    - Zamperla project
    - https://www.zamperla.com/

Ca' Foscari University of Venice

Task 2

Please submit a pdf file where you define **what architecture characteristics** among the ones we identified are **of interest** to your IT system, **relating** each architecture characteristics to the points of the kata that led you to identify that characteristic. For each architecture characteristic you picked up, please **"quantify"** exactly at what level you need it for your IT system.

PS: the pdf can contain also an **updated version of the kata** provided in the previous tasks (e.g., to number the various requirements and refer to them in the new task)

Ca' Foscari University of Venice

Distributed architectures [Concepts]

Software architectures

Pietro Ferrara
pietro.ferrara@unive.it

Ca' Foscari University of Venice

From monolithic to distributed

Monolithic

- Initially, a unique ball of mud
- Then distinct layers or functionalities
  - n-tier
  - pipeline
  - microkernel
- Conceptually, in the same machine
- Practically, might be more than one
  - But the system is seen as a unique block of stone
- Monolithic architectures will be also replicated in distributed systems

Distributed

- Initially, distinct networked computers
  - Communicating through messages
  - E.g., client-server
- Then, several machines communicating to provide a service
  - Service-based/oriented architecture
  - Event-based architecture
  - Microservices
- More complex architectures
- Network communications take time
- Physical distinction of services
- Potentially, many machines for one task

Ca' Foscari University of Venice

API

- Application Programming Interface
  - How sw components communicate
- E.g., interface of Java classes
  - Public methods and fields
- Distributed systems more complex
  - I ping a machine and what I get back?
    - “Hello world!”
    - <xml> … </xml>
    - <html> … </html>
    - { "professor": {“name”: “Pietro”}}
  - Many possible choices
- From an architectural perspective…
  - We don’t care!
  - But we need to exchange information!

Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification

This document is divided into two sections:

Java SE

The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with `java`.

https://docs.oracle.com/en/java/javase/17/docs/api/

Some (very) good material on those topics
(from a course of our bachelor in CS)

REST communication

- A style, not a standard!
- Already found: RestController
  - A controller with a body
  - Plain text, XML, JSON, …
- REST == Representational state transfer
- Stateless protocol
  - No session state (i.e., identifier)
  - All relevant information transferred through the messages
- Advantages: services are independent
  - Do not need to access session information
- Drawbacks: network bandwidth
  - Need to transfer a lot of infos

Representational state transfer (**REST**) is a software architectural style that describes a uniform interface between physically separate components, often across the Internet in a client-server architecture. REST defines four interface constraints:

- Identification of resources
- Manipulation of resources
- Self-descriptive messages and
- Hypermedia as the engine of application state[1]

https://en.wikipedia.org/wiki/Representational_state_transfer

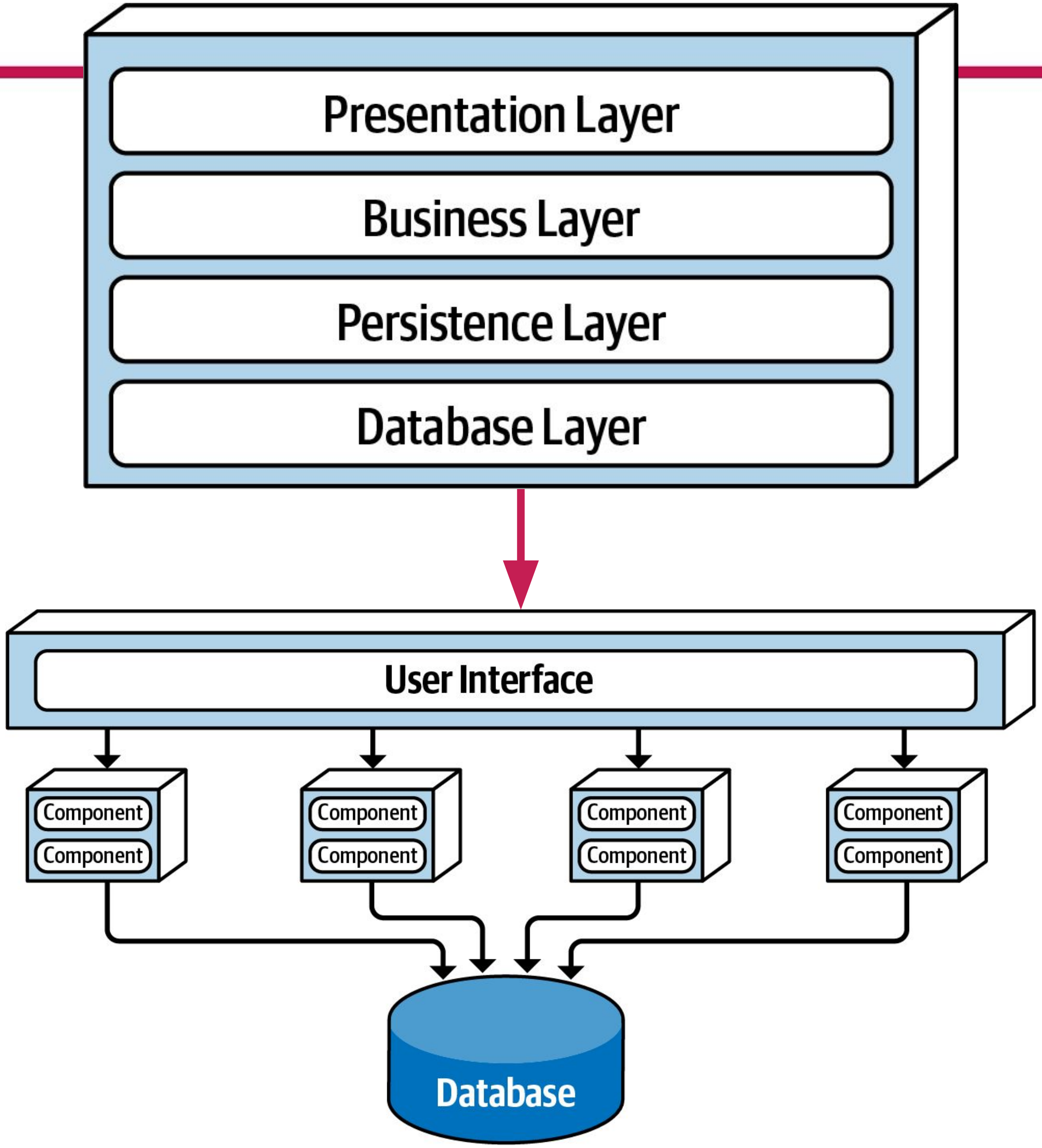
A **stateless protocol** is a communication protocol in which the receiver must not retain session state from previous requests. The sender transfers relevant session state to the receiver in such a way that every request can be understood in isolation, that is without reference to session state from previous requests retained by the receiver.[1]

https://en.wikipedia.org/wiki/Stateless_protocol

Ca' Foscari University of Venice

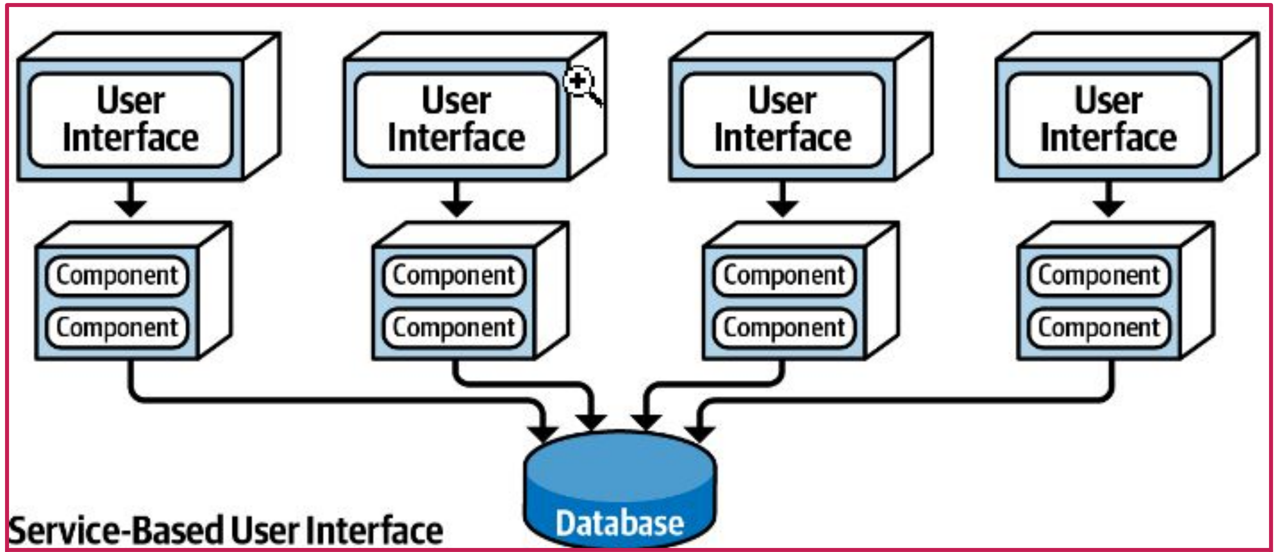
Service-based architecture style

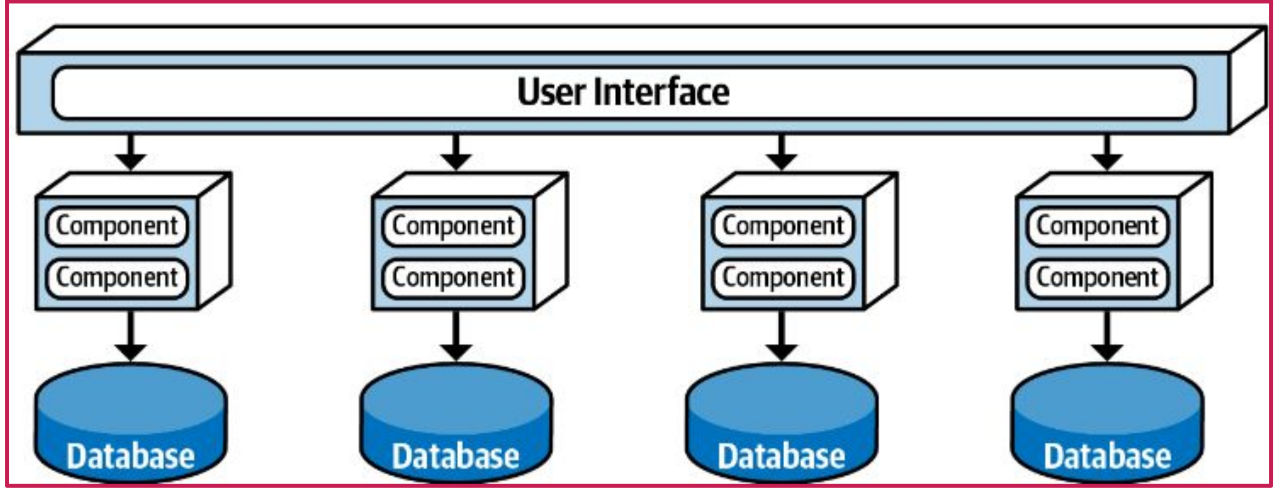
- The back-end comprises all the complex logic of the application
- It's usually the bottleneck
- Simple idea: split it into many services
  - Domain-based splitting
  - Still relying on a n-tier architecture
- A service is a portion of the application
  - Independent deployable unit
- The user interface collects data from some/all the different services
- Data is stored in a unique database



Variations

- Service-based user-interface
  - Each UI interacts with only one service
  - Require to access each service in isolation from the others
- Service-based database
  - Tables belong to one service
  - No relationship between different DBs
- An API layer can be added between user interface and services
  - Proxy knowing who provides what
- More components, more complexity
  - More scalability!





Your wooclap poll will be displayed here

1

Install the **Chrome** or **Firefox extension**

2

Make sure you are in **presentation mode**

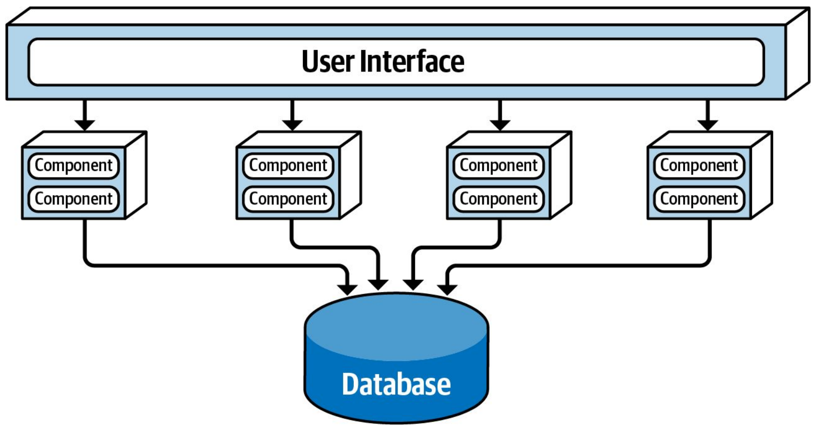
wooclap

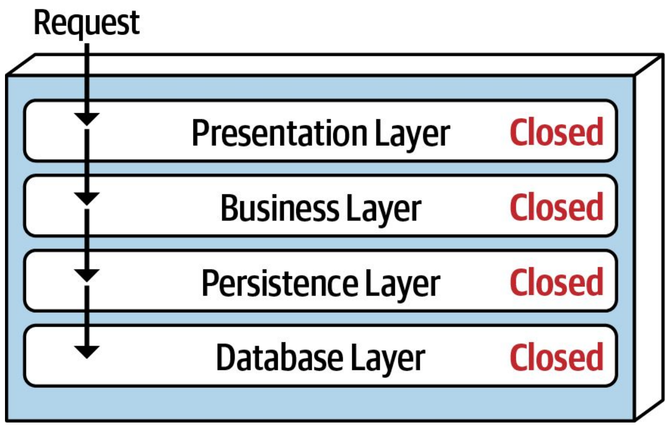
Service-based architecture style Textbook solution

Deployability	★★★★	**Performance**	★★★
Elasticity	★★	**Reliability**	★★★★
Evolutionary	★★★	**Scalability**	★★★
Fault tolerance	★★★★	**Simplicity**	★★★
Modularity	★★★★	**Testability**	★★★★
Overall cost	★★★★		

Synchronous vs asynchronous

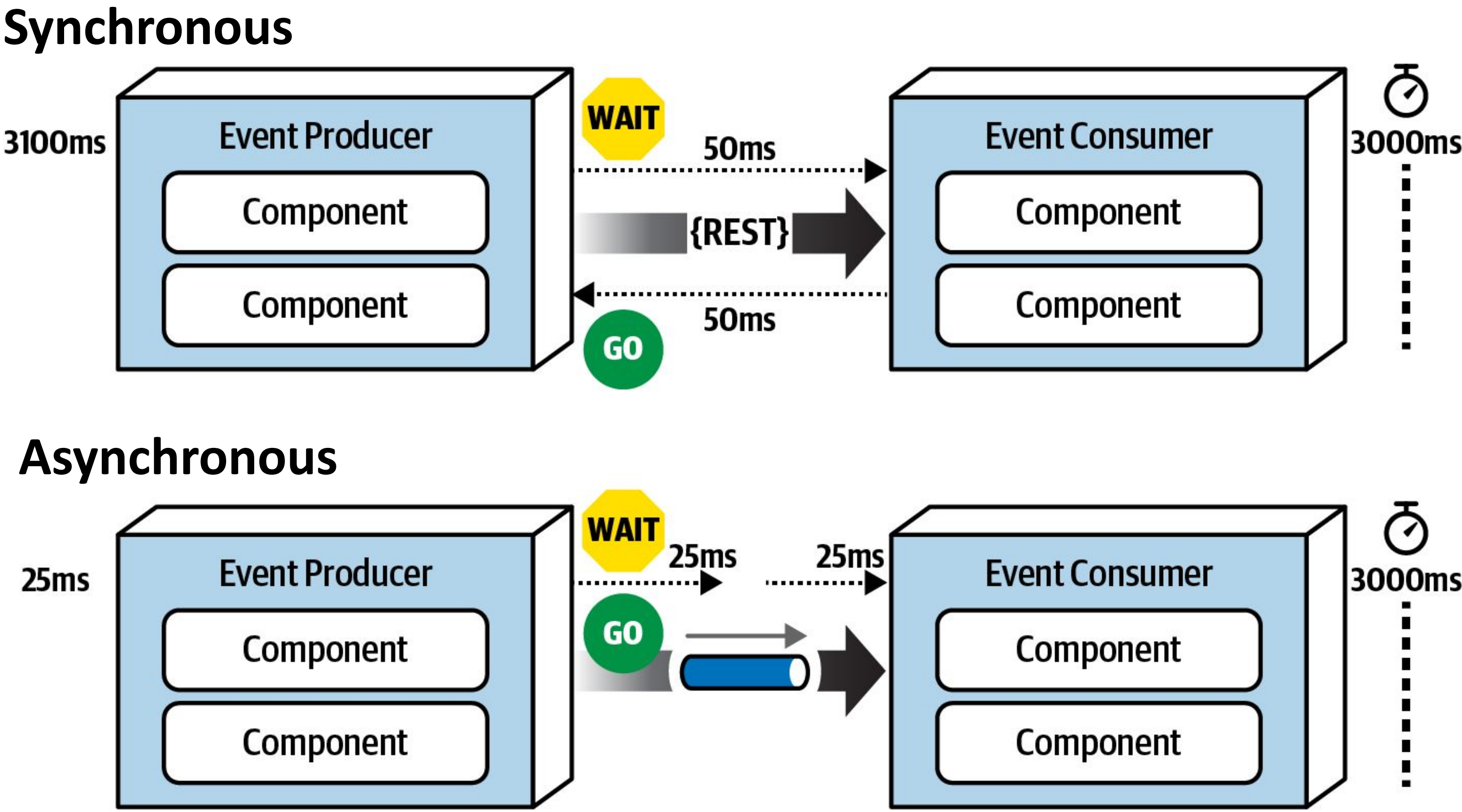
- So far, only synchronous components
  - I call a method: I wait for the returned value
  - I run a pipeline: each step waits for the results of the previous step
  - I rely on a distributed service: I make a request, I wait until I get a reply
- Obviously, this limits our architecture
  - If a component fails, everything fails
  - If a component is slow, everything is slow
  - If a component does not scale up, the system is not scalable
- We need to move to asynchronous
  - More complexity coming…





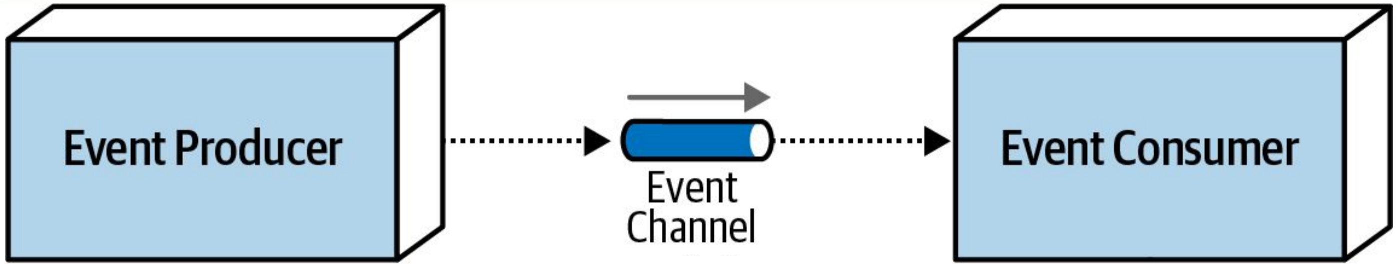
Asynchronous capabilities

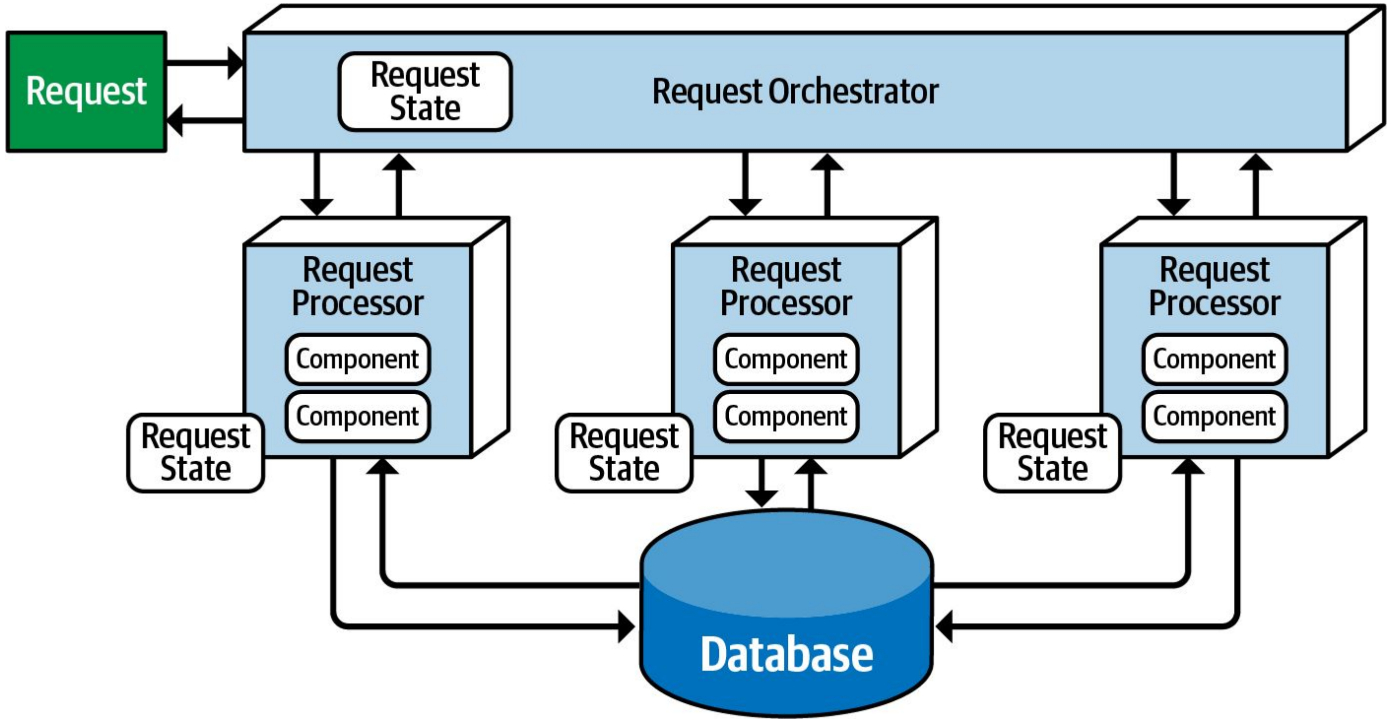
If we are posting a comment, no need to wait everything is finalized!



Event-Driven Architecture style

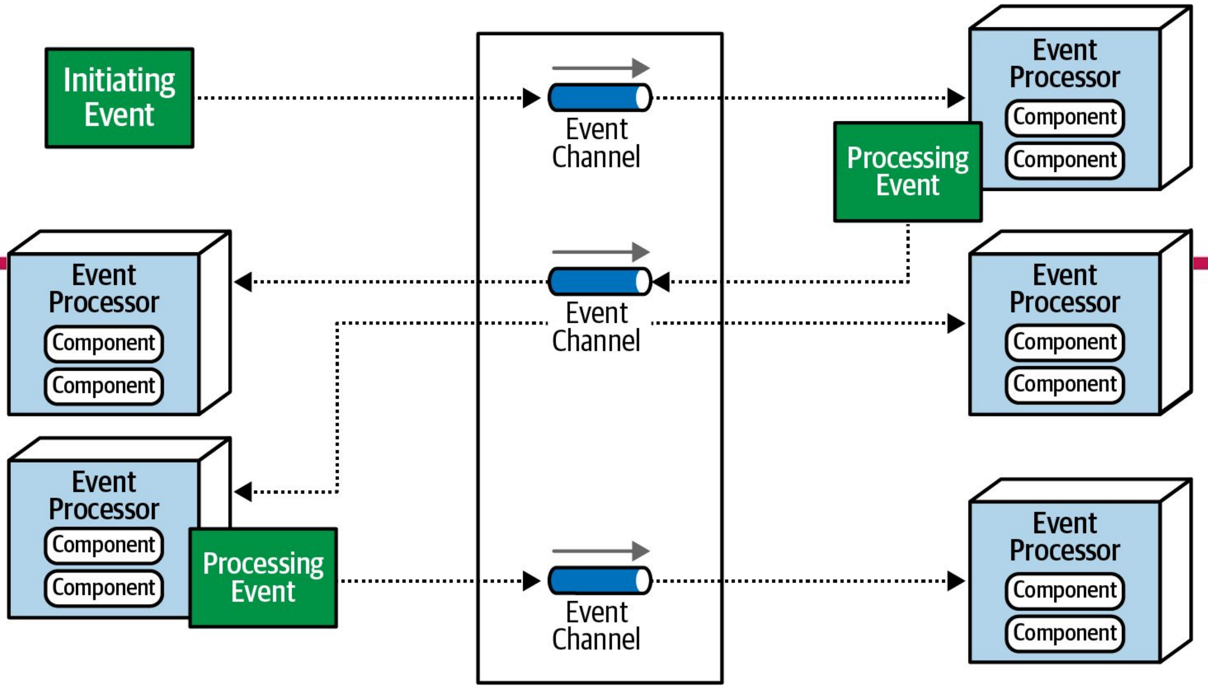
- Request-response (APIs):
  - I ask for something (request)
  - I idle until a get a reply (response)
  - Method calls, RPCs, RESTs, …
- Event-driven:
  - I ask for something
  - I do something else until
  - I get pinged: I have a reply!
- Three major components:
  - A producer of events
  - A communication (event) channel
  - A consumer of events
- Just the basic block…

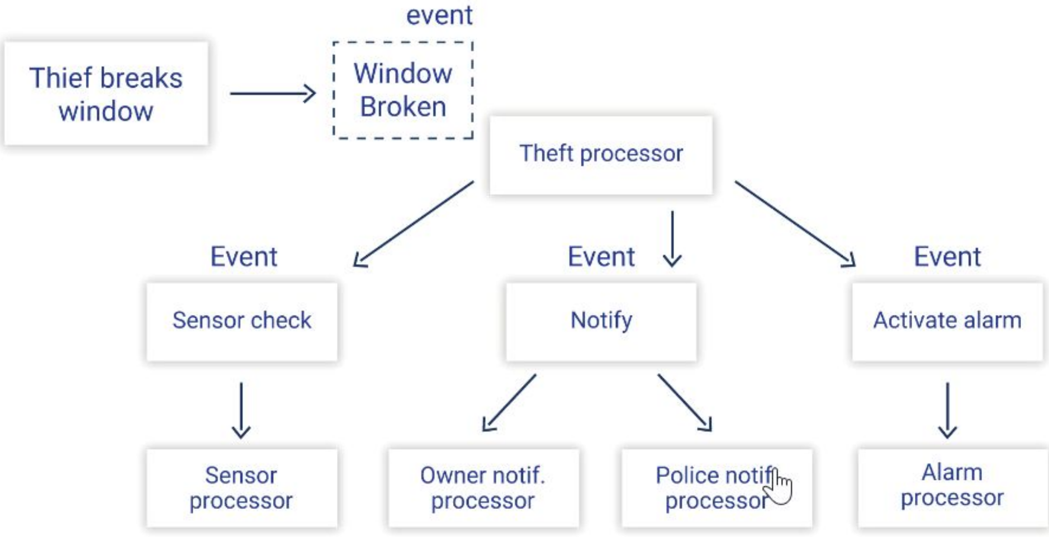




Broker topology

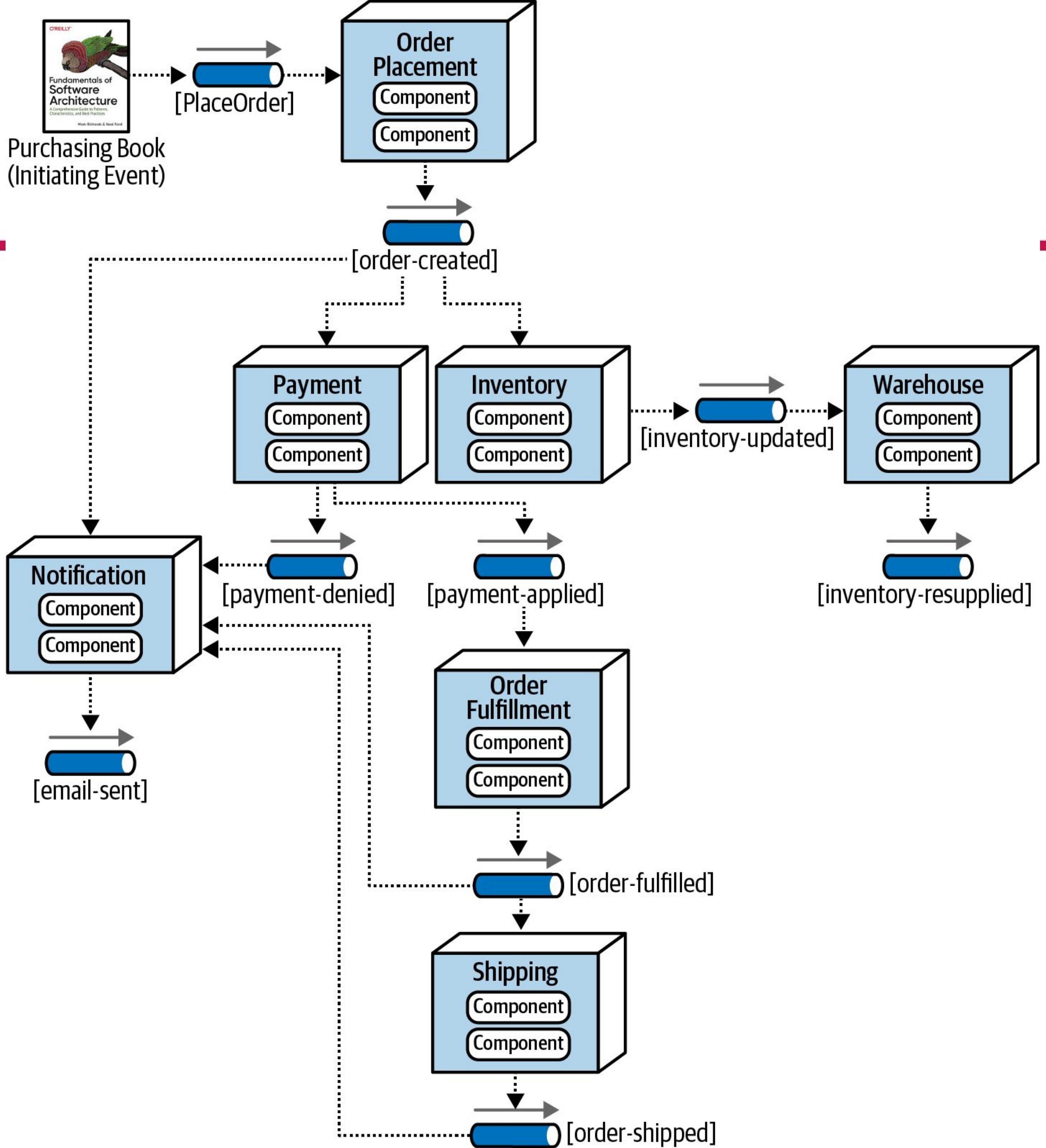
- Messages flow among several processors
  - … pipeline architecture?
- Initiating event: the begin of everything
  - Entrypoint!
- Event broker: communication channel
- Event processor: receives the event
- Processing event: the output of the event processor at the end of the processing
  - Another processor will take care of it
- Usually, publish-and-subscribe communication model through topics
  - RabbitMQ, Kafka, etc…





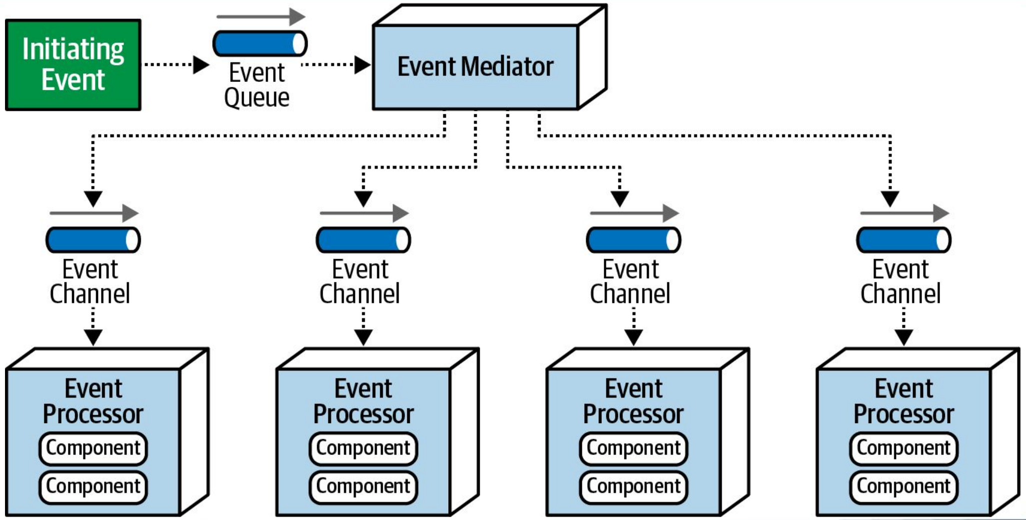
https://www.3pillarglobal.com/insights/event-driven-architecture-topologies-broker-and-mediator

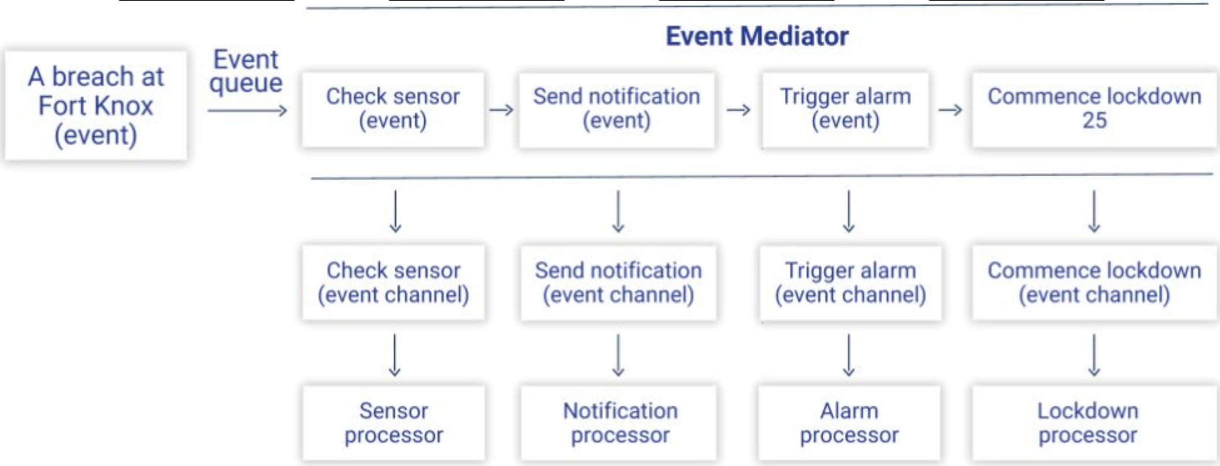
An example of the broker topology



Mediator topology

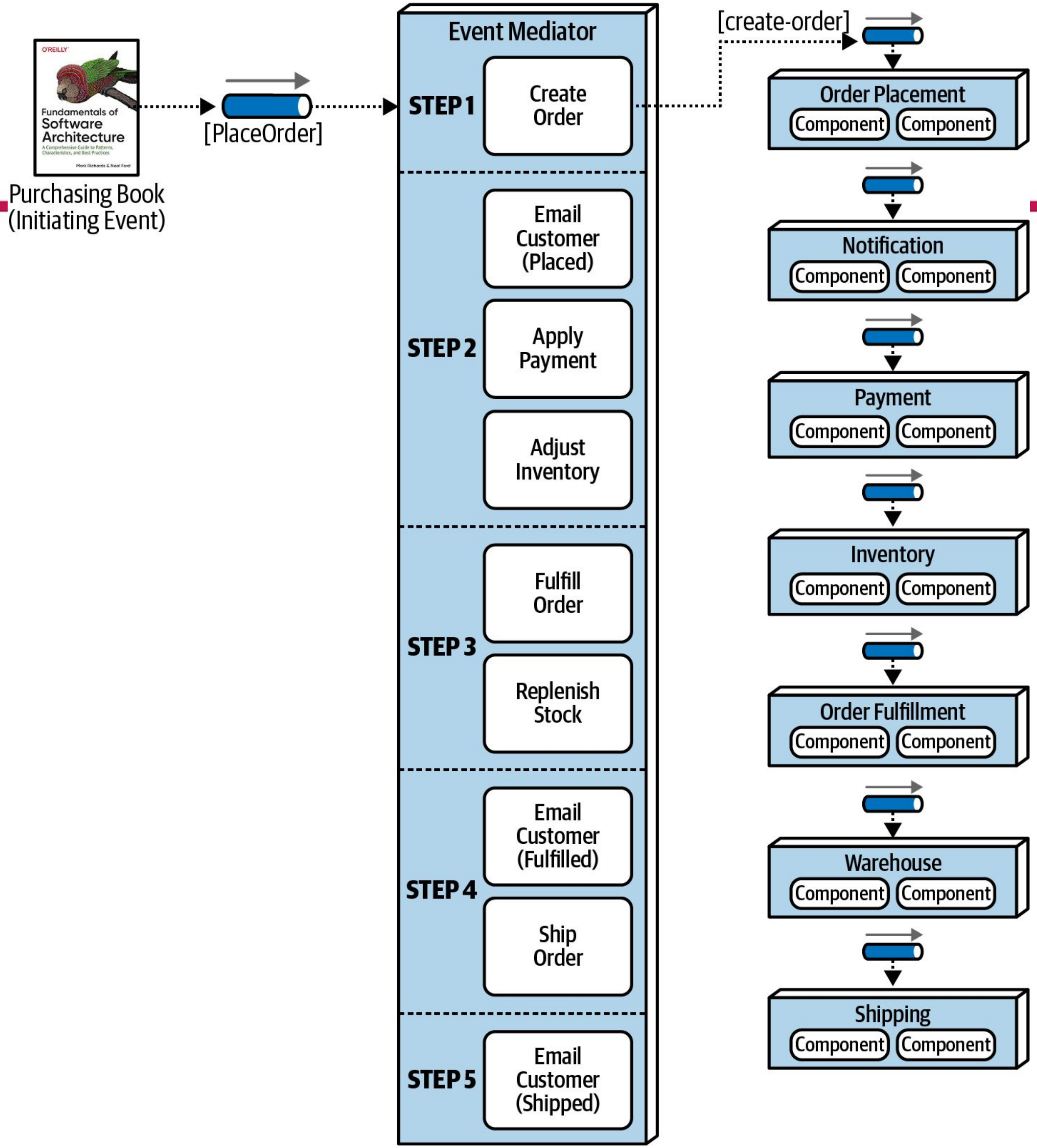
- Central mediator
  - Receive the original event
    - Aka request
  - Send events to all the other services involved
  - Collect the results of the process
  - Eventually gives back some info
- Multiple mediators
  - Communications by a queue with many mediators listening
- In some ways, it recalls request-response pattern
  - But in an asynchronous way!





https://www.3pillarglobal.com/insights/event-driven-architecture-topologies-broker-and-mediator

An example of the mediator topology



Ca' Foscari University of Venice

Broker vs mediator

Broker

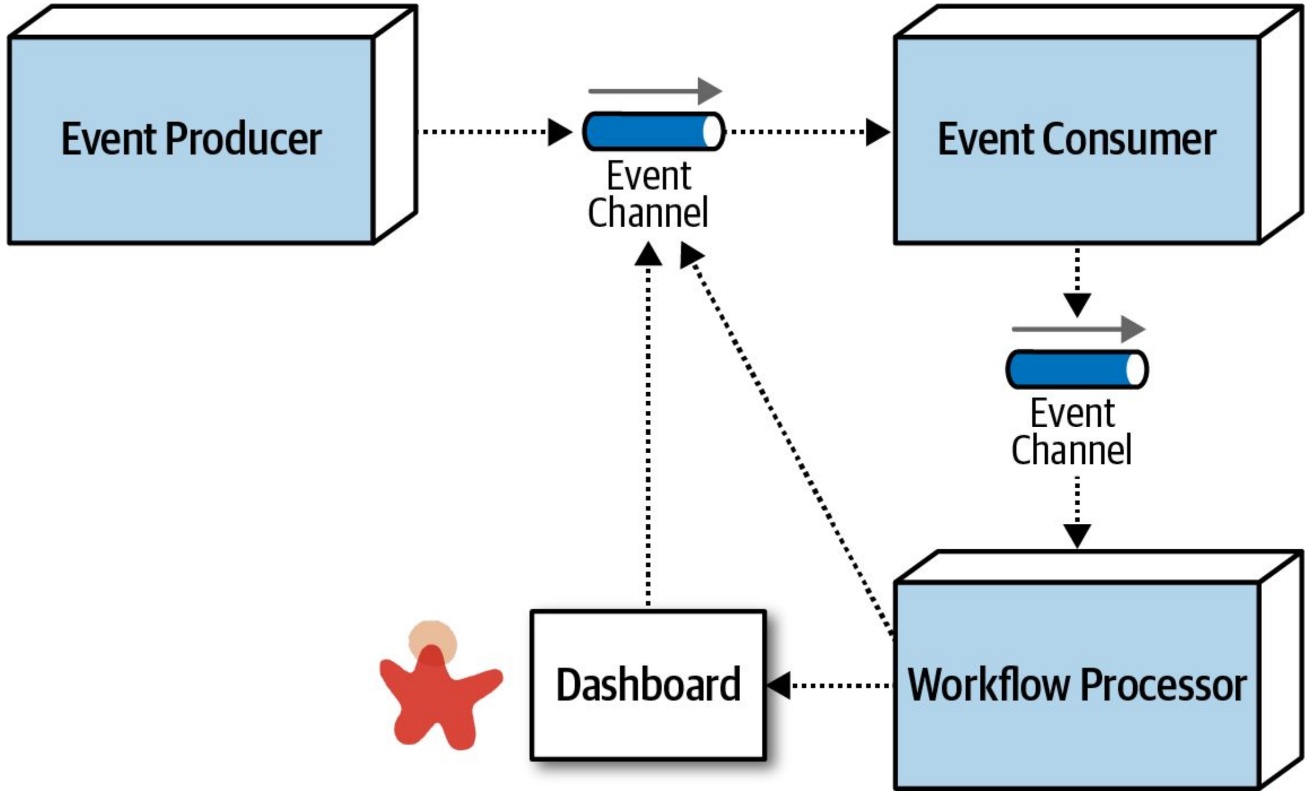
Advantages	Disadvantages
Highly decoupled event processors	Workflow control
High scalability	Error handling
High responsiveness	Recoverability
High performance	Restart capabilities
High fault tolerance	Data inconsistency

Mediator

Advantages	Disadvantages
Workflow control	More coupling of event processors
Error handling	Lower scalability
Recoverability	Lower performance
Restart capabilities	Lower fault tolerance
Better data consistency	Modeling complex workflows

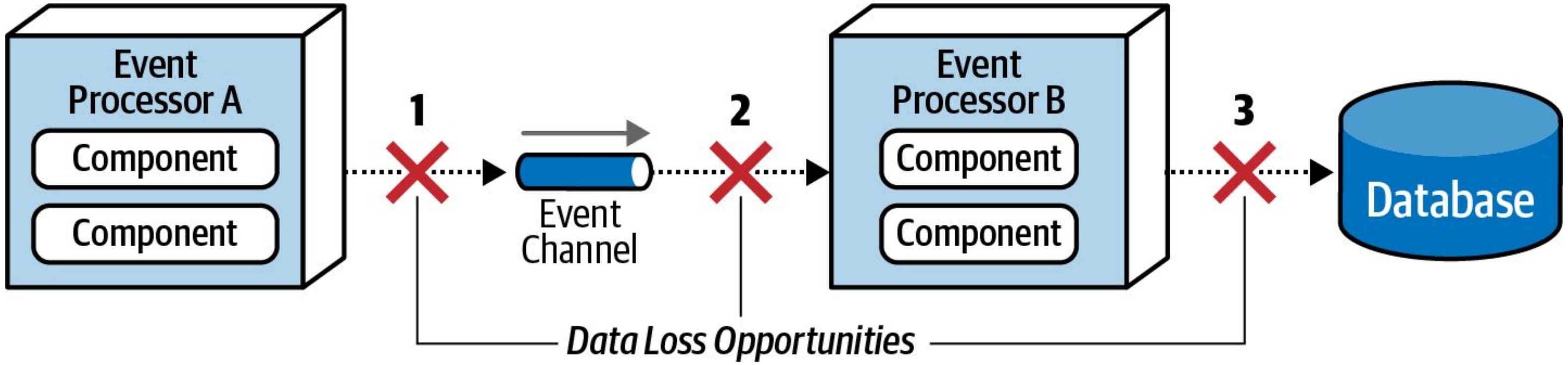
Error handling (aka fault tolerance)

- Workflow processor
  - Check if a processor produced an error
  - If so, it potentially fixes the message and puts it back to the event channel
- The workflow processor automatically changes the data
  - Trying to repair it in some ways
  - Without any human intervention
- If it is not able to fix the message
  - Ask a user to give a look to it
  - Through some dashboard
    - Kind of email client



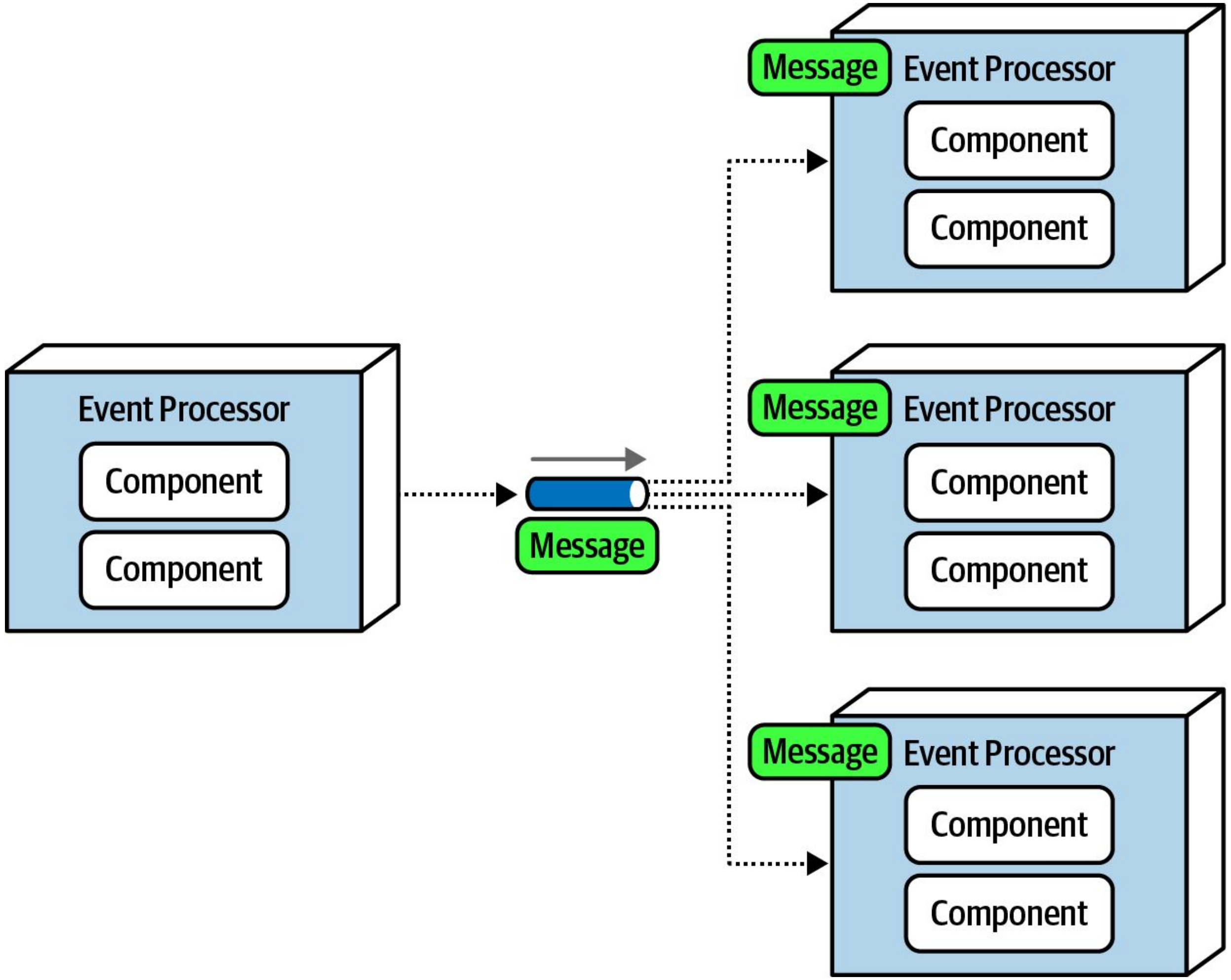
Losing data

- Messages might be lost because
  - communication fails
  - an error of the event channel
- Rely on standard messaging techniques
  - Case 1 (a message never arrives to the queue): persistent message queues + synchronous send (guaranteed delivery)
  - Case 2 (a message is dequeued but the processor fails before processing it): client acknowledge mode (keep the message with the client ID, if fail then re-queue it)
  - Case 3 (unable to persist the message): ACID transactions via a database commit



Broadcast capabilities

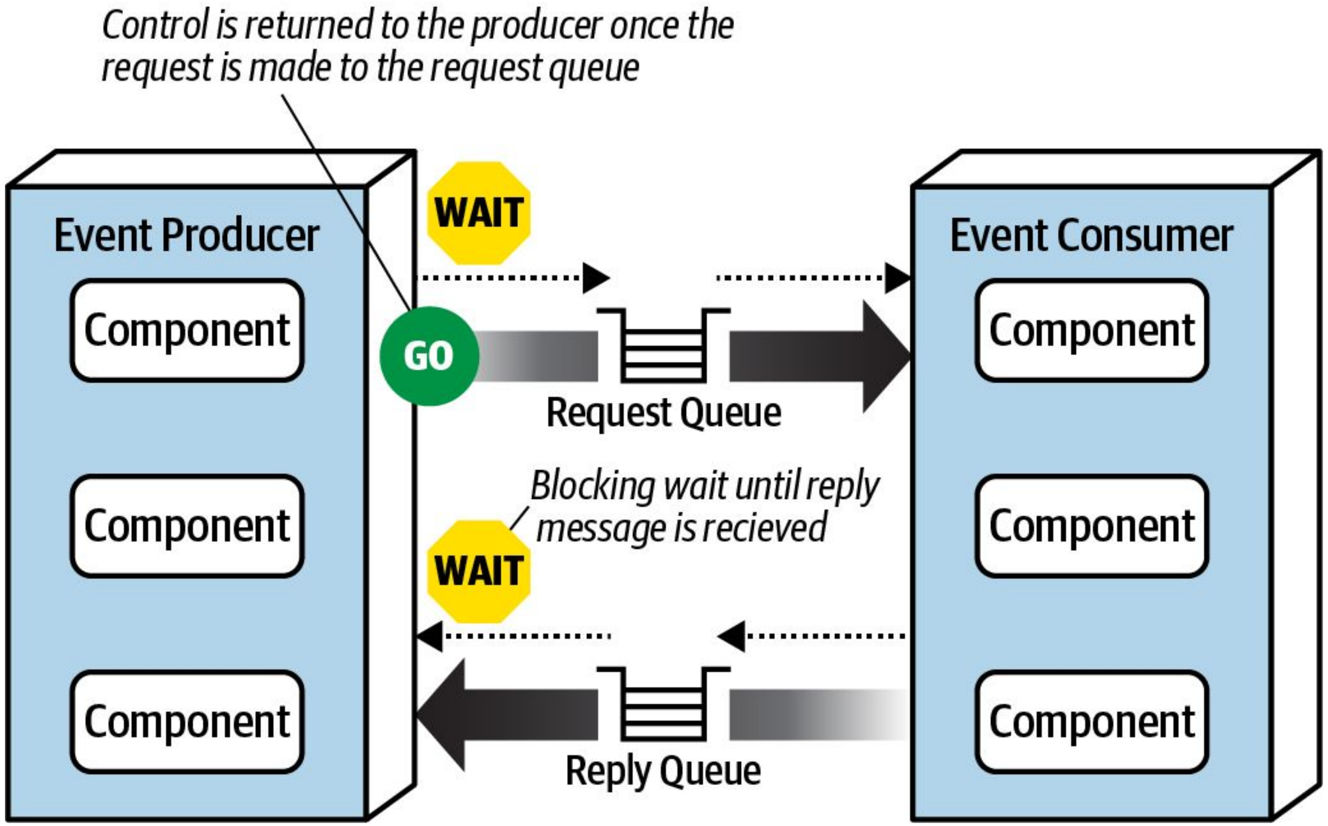
- Event-based architecture does not know who process what!
  - Many subscribers might process the same message
- Highest level of decoupling between producer and consumer
- This enhances a lot
  - Scalability
  - Elasticity



Ca' Foscari University of Venice

Request-Reply

- Everything asynchronous so far
- But sometimes we need to end up an action before moving on
- Adopt request-reply messaging
  - A queue receives the requests
  - After their processing, a message is sent to the reply queue
  - Then the produces will know that the processor finalized the task
- Quite more complex than standard REST request-response
- But we can do something else while waiting for the reply…



Your wooclap poll will be displayed here

1

Install the **Chrome** or **Firefox extension**

2

Make sure you are in **presentation mode**

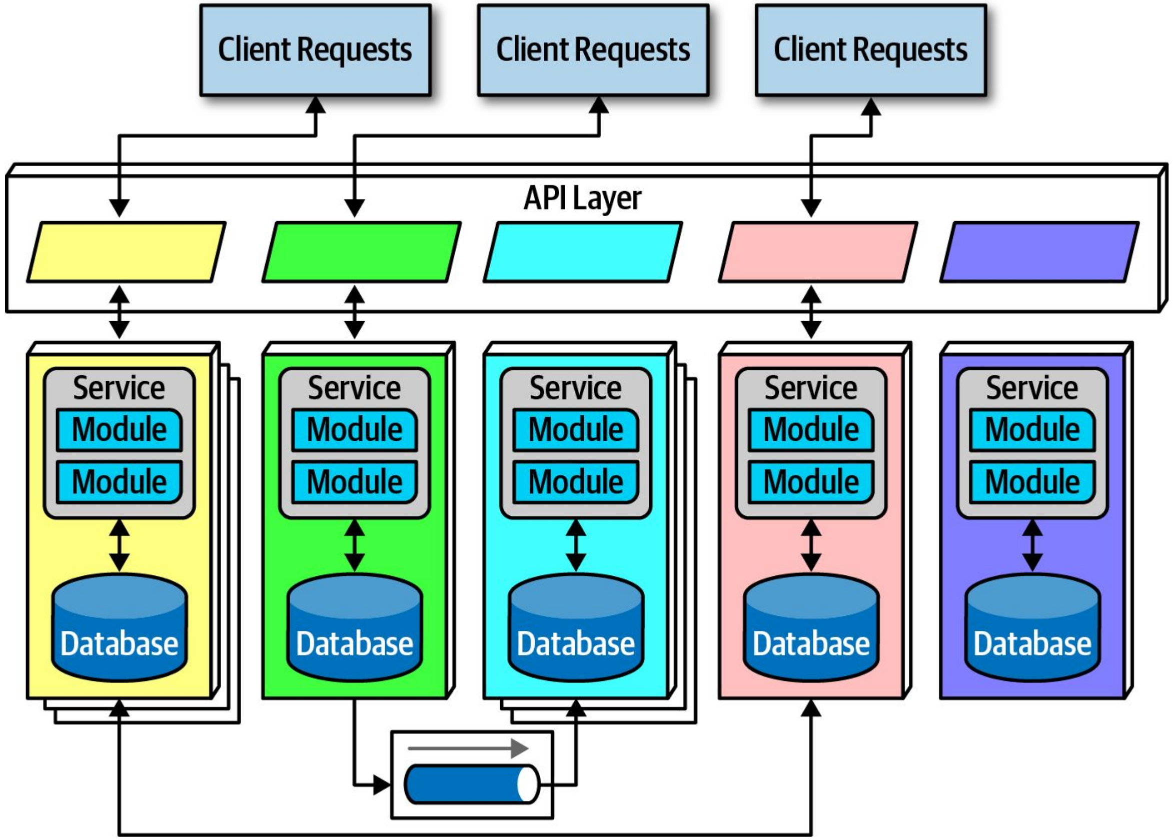
wooclap

Event-based architecture style
Textbook solution

Deployability	★★★	**Performance**	★★★★★
Elasticity	★★★	**Reliability**	★★★
Evolutionary	★★★★★	**Scalability**	★★★★★
Fault tolerance	★★★★★	**Simplicity**	★
Modularity	★★★★	**Testability**	★★
Overall cost	★★★		

Microservices

- "Recent" big hype (for how long?)
  - Name appeared in 2014
- Derived from a common pattern
- Domain-driven design, bounded context
  - Each service has knowledge only about the part of its interest
    - No reusable classes/linked DBs in different services
- High level of decoupling, less code reuse
- Services are single-purpose (aka, micro)
- Each service includes all the necessary parts to operate independently

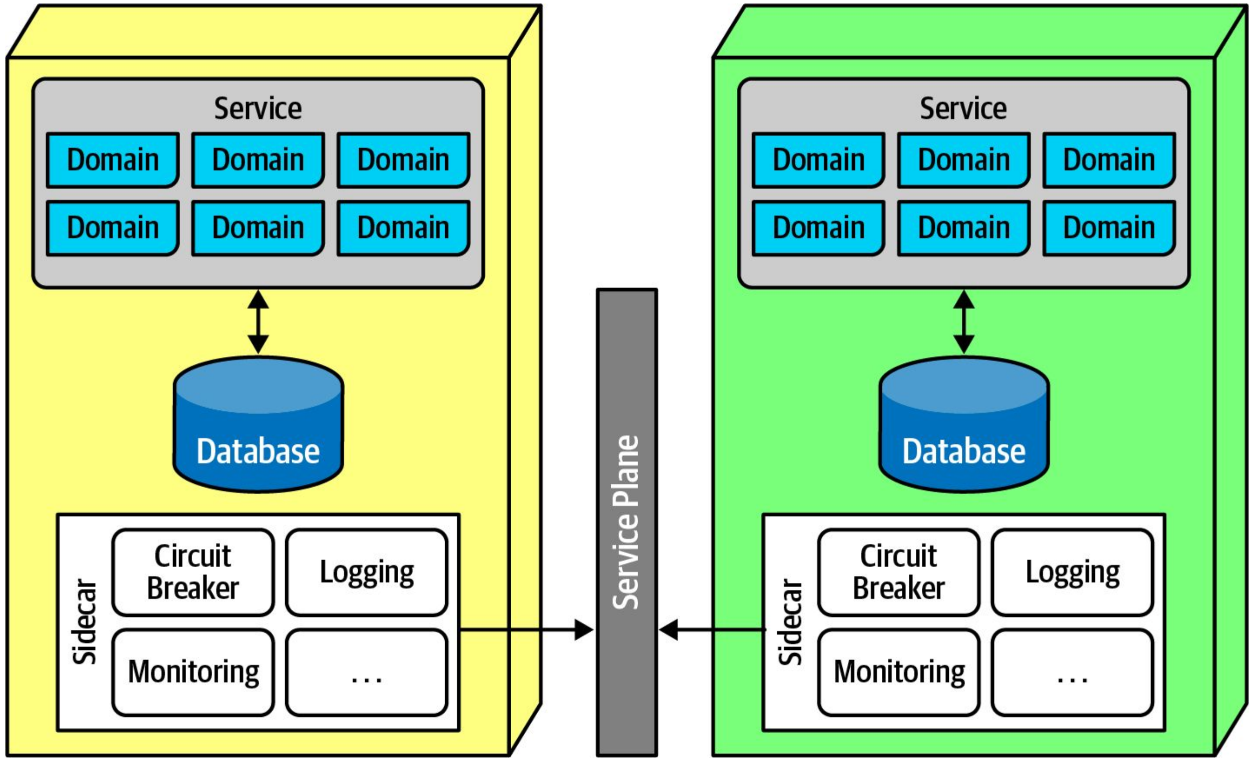


Boundaries

- Micro is just a label, not a description!
  - In contrast with large services, quite common in 2014!
- How to choose the “right” service granularity?
  - Purpose: domain driven, provides one significant behavior
  - Transactions: not across service boundaries!
  - Choreography: avoid extensive communications among services
- Data isolation is a must, each microservice has its own database
  - There is no single source of truth (old relational DBs)
- The API layer is optional but extremely common
  - Be careful: it is not a mediator or orchestrator!
  - Just a proxy redirecting the request to the “right” microservice

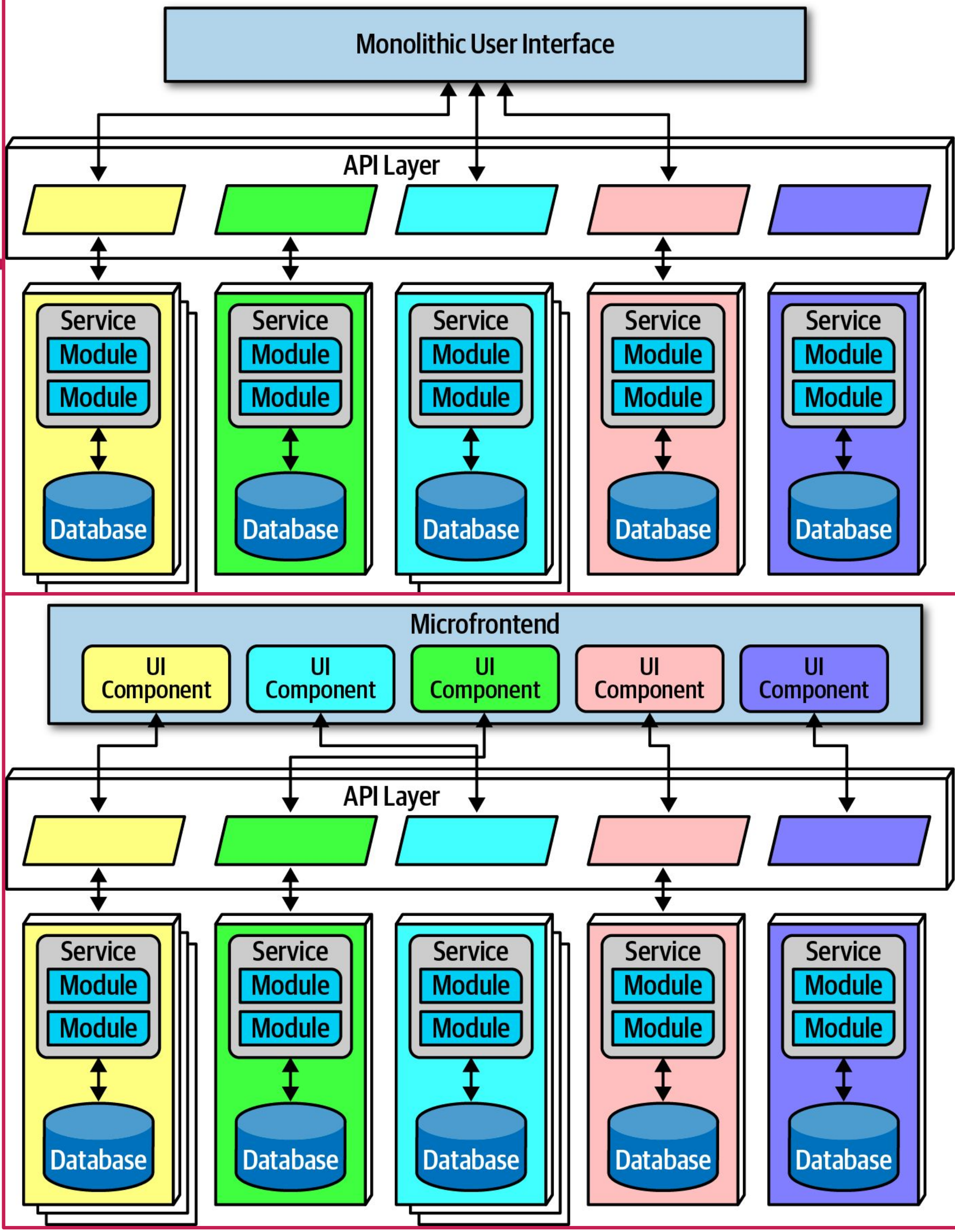
Operational reuse

- Still shared operational components
  - Logging, monitoring, etc etc…
  - This is unavoidable!
- Some services must be uniform
  - Otherwise the overall project cannot be managed (monitor, deploy, upgrade…)
- A sidecar component is added
  - Allow to build up a service mesh
- Each service is a node in the mesh
  - Console that allows to monitor services
- Service discovery is key for elasticity
  - No direct request, go through API layer
  - Monitor requests, spin up new instances



Frontends

- UI communicates with many services
  - As service-based architecture
- This is called monolithic frontend
  - Single UI interfaced with the API
- Originally, UIs should be bounded
  - Following domain-driven principles
  - Isolate UIs together with back-end
- Microfrontends are a hot trend
  - Unify entire domain in a single team

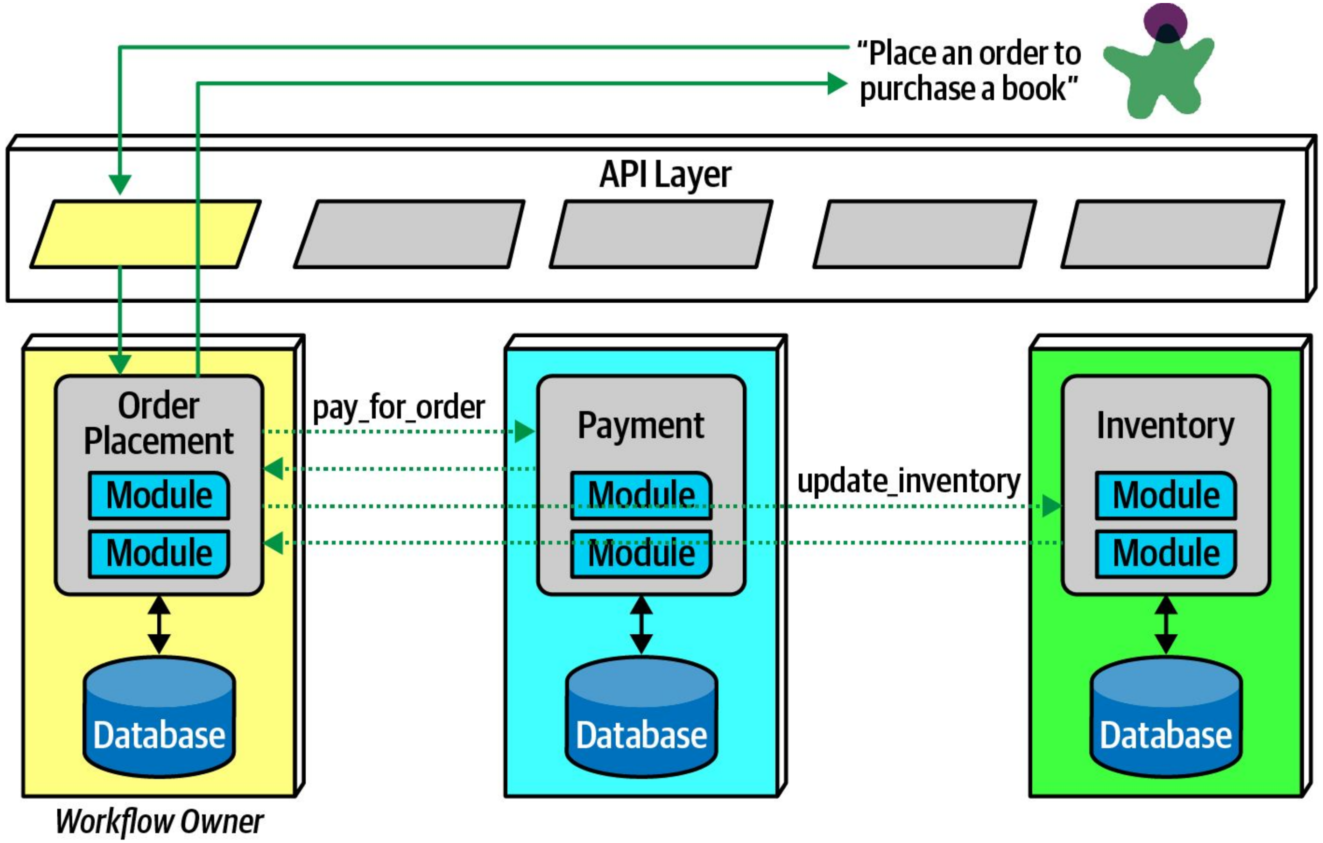


Synchronous or asynchronous?

- But… what type of communications we have in microservices?
  - In particular, should it be synchronous or asynchronous?
- Each microservice is independent also in this regard
- Protocol-aware heterogeneous interoperability:
  - Protocol-aware: no centralized hub, each service should now how to use other services (e.g., REST APIs, queues, …)
  - Heterogeneous: each service might rely on different technologies
  - Interoperability: services call one another
- Synchronous communications usually through REST APIs
- Asynchronous communication usually through events/messages

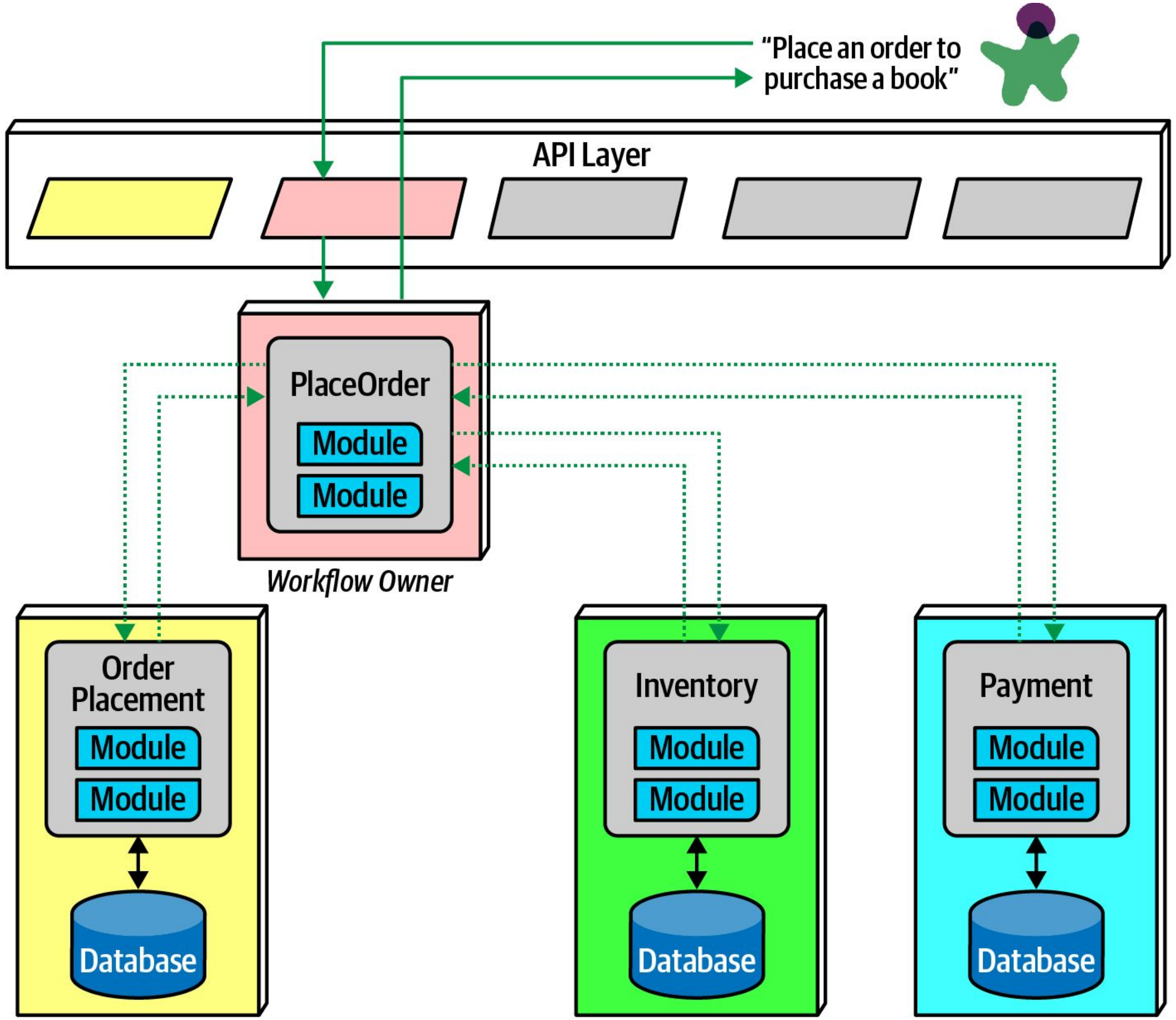
Choreography

- No central coordinator
  - Broker event-driven architecture
  - Bounded context philosophy
- Each service calls other services as needed
  - This might go pretty deep
- Natural solution aligned with microservice philosophy
- But what happens if failure happens in the middle?
  - See saga



Orchestration

- We cannot have a global mediator
- Create a service as local mediator
  - Invokes several microservices
  - Coordinate their invocation
  - Aggregate and return the results
- Create coupling between services
  - Mediator depends on all others
  - Sometimes unavoidable
- Single services do not need to coordinate with other services



Your wooclap poll will be displayed here

1

Install the **Chrome** or **Firefox extension**

2

Make sure you are in **presentation mode**

wooclap

Ca' Foscari University of Venice

Microservices architecture style
Textbook solution

Deployability	★★★★	**Performance**	★★
Elasticity	★★★★★	**Reliability**	★★★★
Evolutionary	★★★★★	**Scalability**	★★★★★
Fault tolerance	★★★★	**Simplicity**	★
Modularity	★★★★★	**Testability**	★★★★
Overall cost	★		

References

- Textbook, Part II
  - Service-based architecture: chapter 13
  - Event-driven architecture: chapter 14
  - Microservices: chapter 17