VU CLOUD COMPUTING, 2024W
Today: SOA, REST and Microservices

Aral Atakan, Enes Bajrovic, Siegfried Benkner, Martin Köhler, Andrey Nagiyev

Research Group Scientific Computing, University of Vienna

OUTLINE

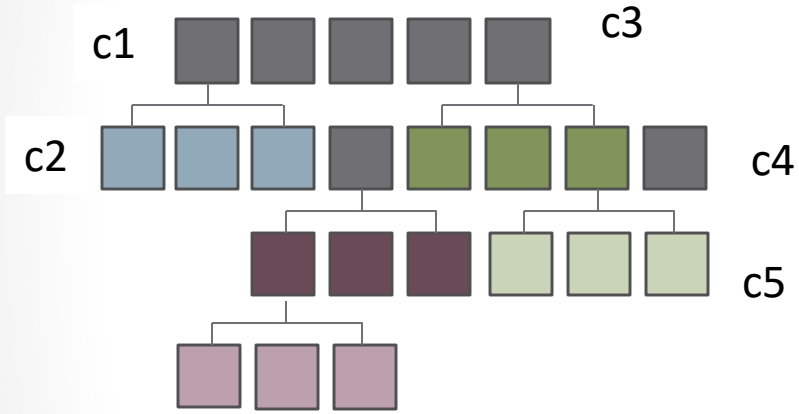
Previously

- Cloud Computing, Service Models
- IaaS

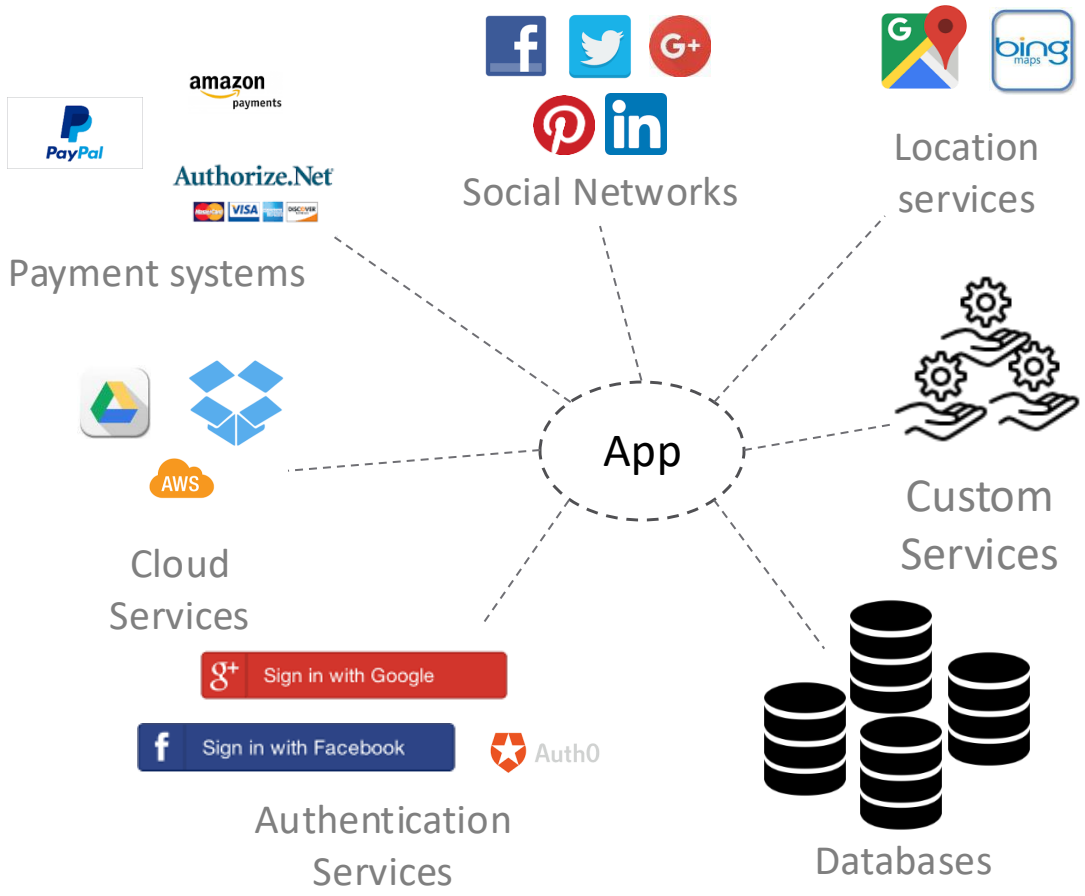
Today:

- A brief overview of Service Oriented Architectures, REST
- **Microservices**

INTRO: COMPLEX SYSTEMS

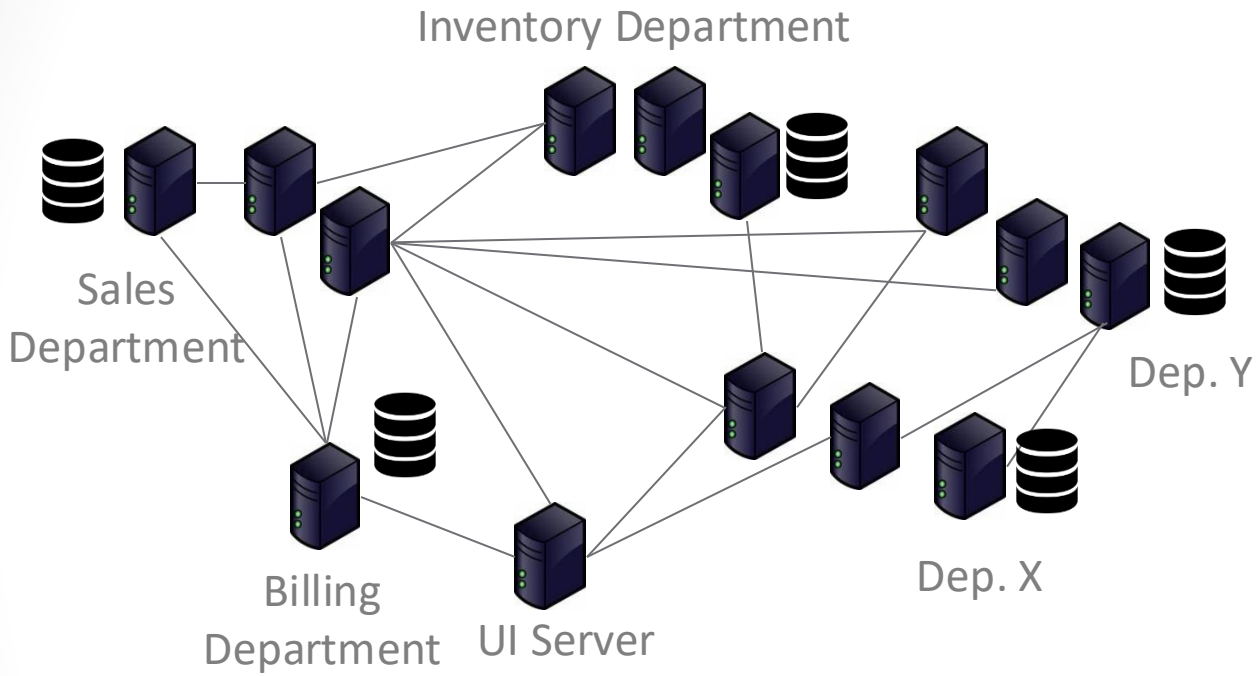


Complex **Monolithic** System



Complex **Modern** System

INTRO: EXAMPLE



Complex System

Clients:

Problems

- **How do different parts of the system communicate?**
- **Could some processes between different parts be automated?**
- **Do different parts of the system use different technologies, and how could they communicate?**

How to make different parts of the system work together?

- **Use a unified approach (e.g.: an architectural style) → SOA, REST**

SOA: SERVICE ORIENTED ARCHITECTURE (OASIS)

- **SOA as a way of designing and implementing enterprise applications that deals with the intercommunication of loosely coupled, coarse grained, reusable artifacts (services).**
- **Service Oriented Architecture is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains.**
- **SOA provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.**

Source: **Reference Model for Service Oriented Architecture 1.0, OASIS Standard**

W3C: SOA DEFINITION

A Service Oriented Architecture is a form of distributed systems architecture that is characterized by the following properties:

Logical view

The service is an abstracted, logical view of actual programs, databases, business processes, etc., defined in terms of what it does, typically carrying out a business-level operation.

Message orientation

The service is formally defined in terms of the messages exchanged between provider agents and requester agents, and not the properties of the agents themselves.

The internal structure of an agent, including features such as its implementation language, process structure, database structure, etc. is abstracted away.

W3C: SOA DEFINITION

Description orientation

A service is described by machine-processable meta data.
Only those details that are exposed to the public and important for the use of the service should be included in the description. The semantics of a service should be documented, either directly or indirectly, by its description.

Granularity

Services tend to use a small number of operations with relatively large and complex messages.

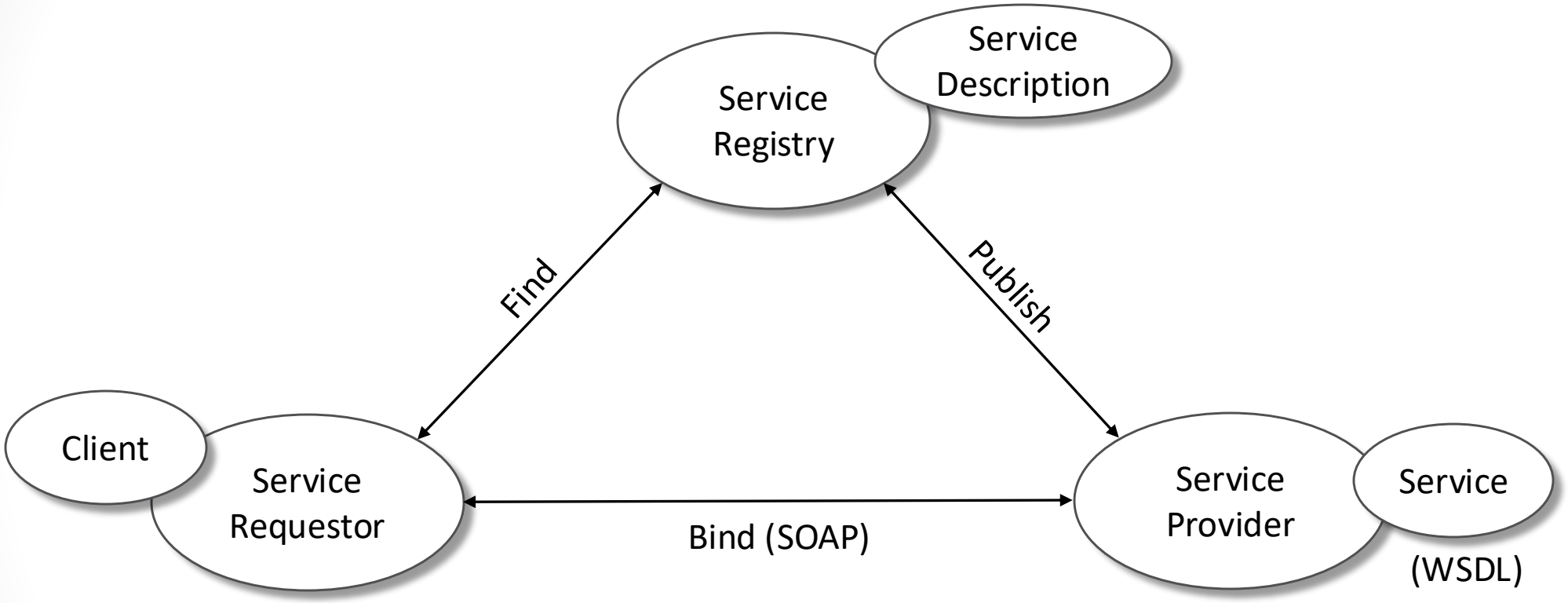
Network orientation → Services tend to be oriented toward use over a network

Platform neutral

Messages are sent in a platform-neutral, standardized format delivered through the interfaces.

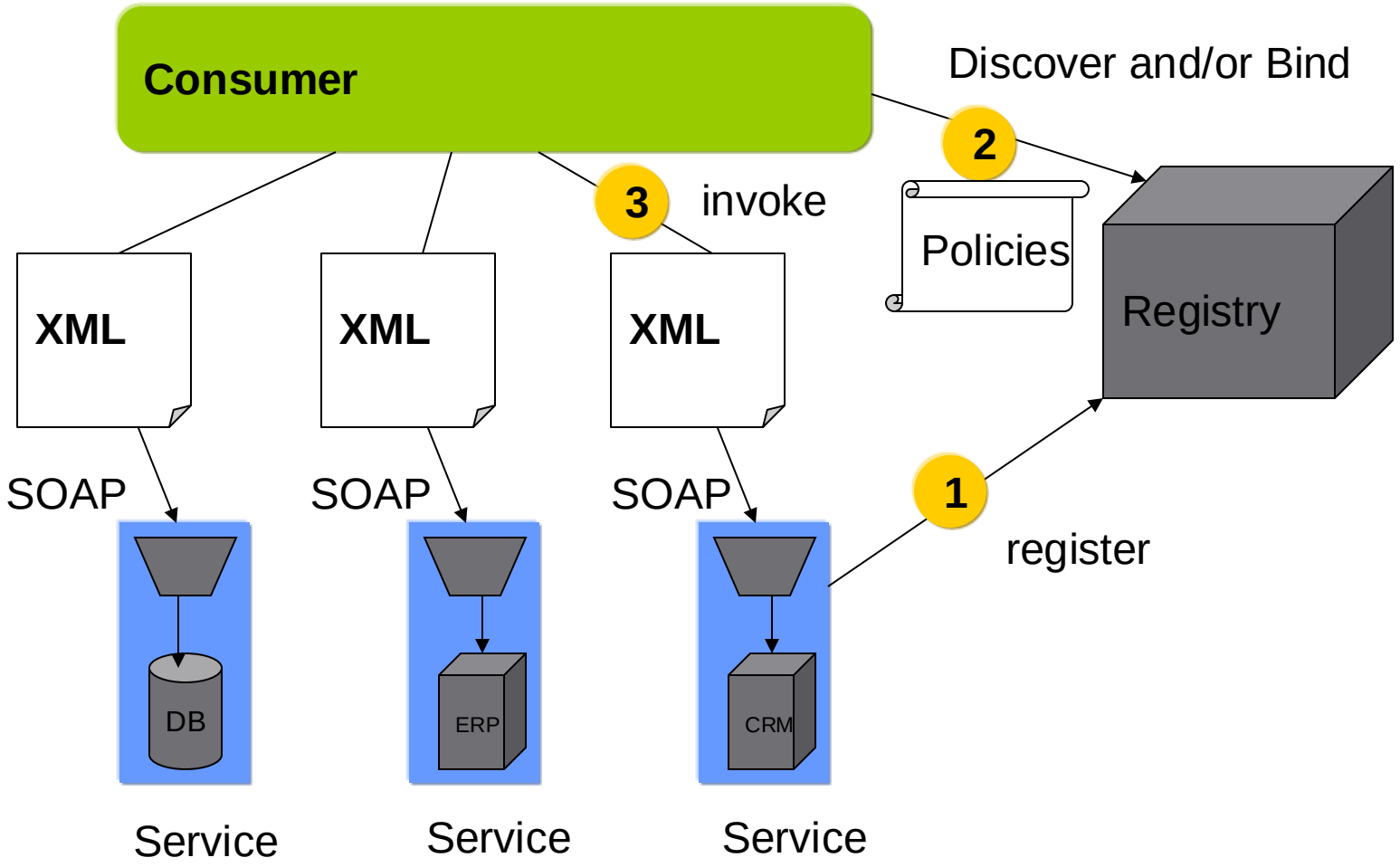
XML is the most obvious format that meets this constraint.

SOA CONCEPT


Service Description
Service Registry
Find
Publish
Client
Service Requestor
Bind (SOAP)
Service Provider
Service
(WSDL)

SOA VERSUS DISTRIBUTED OBJECT SYSTEMS

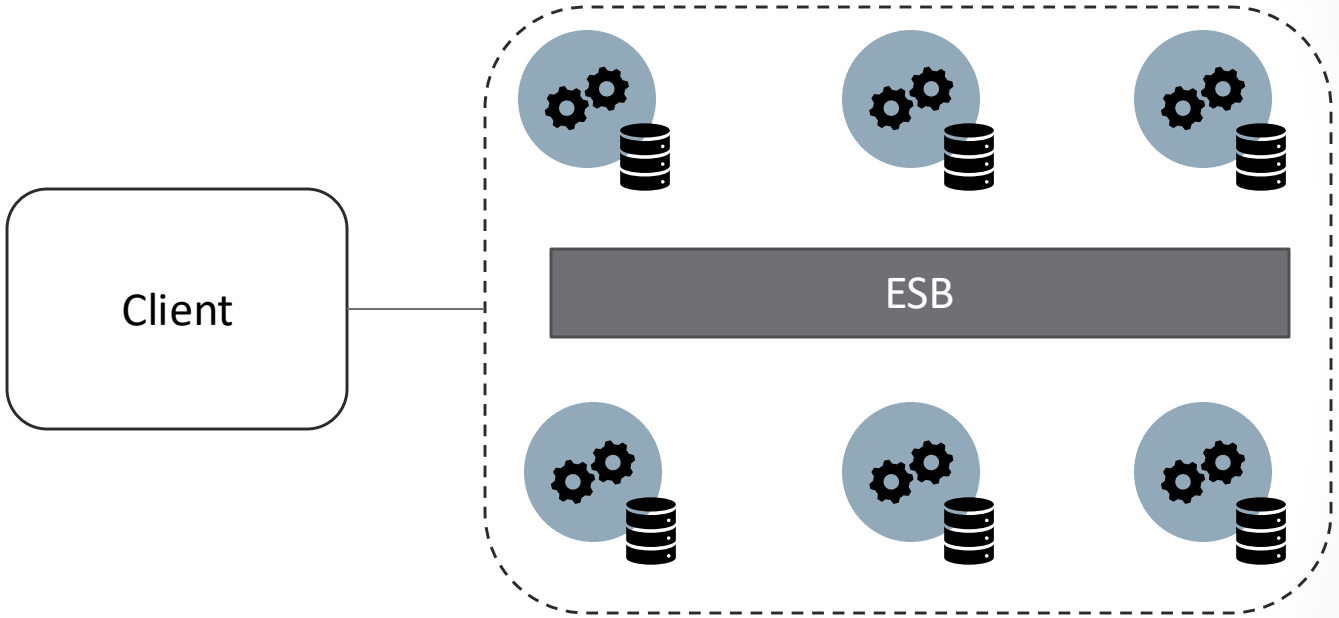
SOA based systems



SOA :: ENTERPRISE SERVICE BUS (ESB)

Architecture / Middleware

- Connectivity
- Routing
- Security
- Service management
- Data transformation
- Monitoring
- Conversion of communication protocols
- Resolving contention
- …



WEB SERVICES: DEFINITIONS

Web Services are applications accessible to other applications over the Web [2]

Web Services are self-contained, modular business applications that have open, Internet-oriented, standards-based interfaces [3]

A Web Service is a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web Services supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols [4]

WEB SERVICES TECHNOLOGIES

Web Services

- Interoperability, Loose Coupling, Scalability
- Location transparency (through registration and discovery)
- Implementation independence, Platform independence
- Key technology for implementing Service-Oriented Architectures

Web Services are based on standard XML technologies

- Service Description - **WSDL**
- Simple Object Access Protocol - **SOAP** (XML-RPC over HTTP,…)
- Universal Discovery, Description and Integration - **UDDI**

WEB SERVICES TECHNOLOGIES (2)

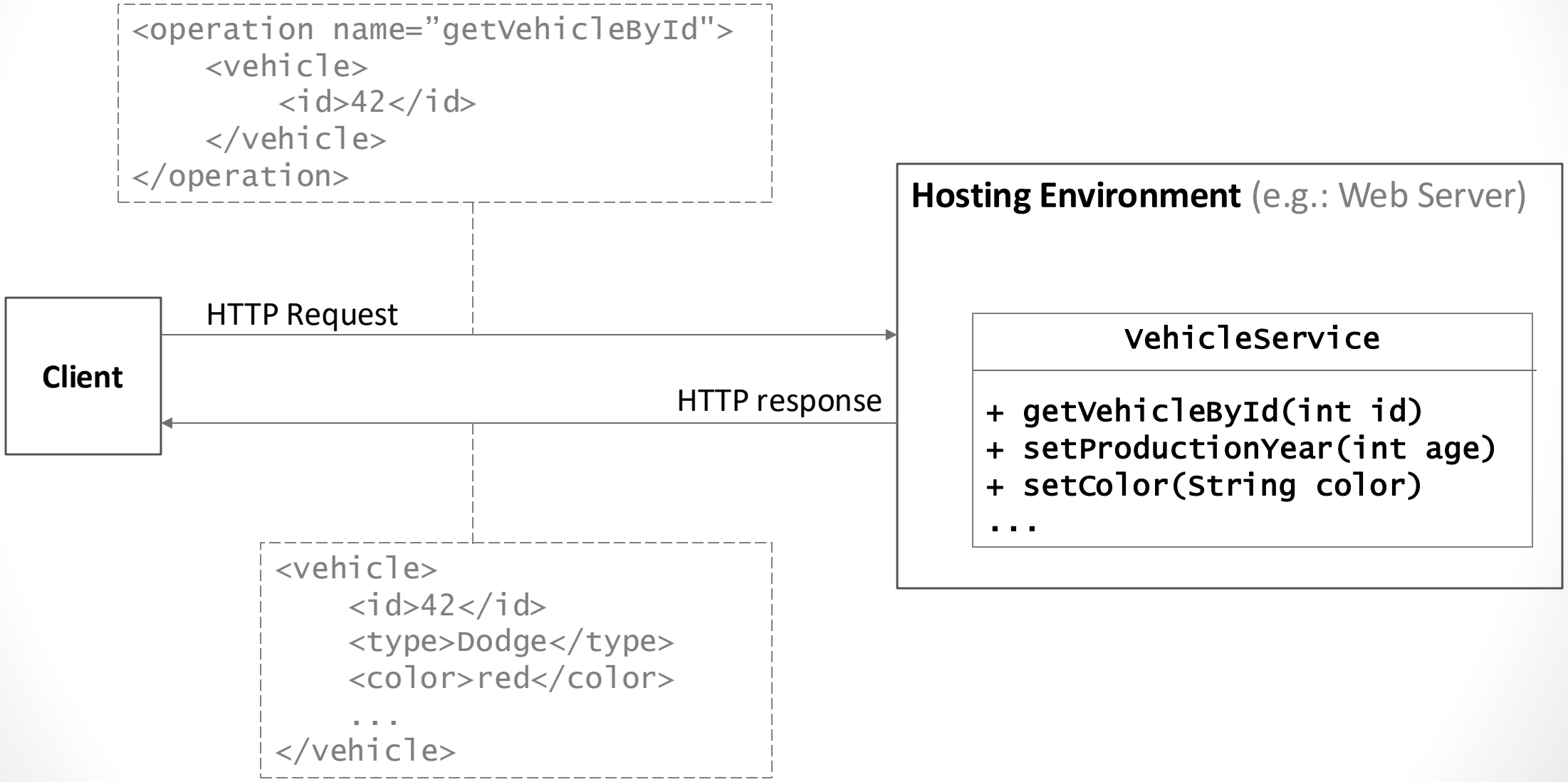
SOAP (Simple Object Access Protocol): Standard to exchange XML-based messages to and from Web Services independent from the transport protocol [5]

- SOAP RPC: Blocking invocation of service operations
- SOAP Messaging: Non-Blocking messages to services

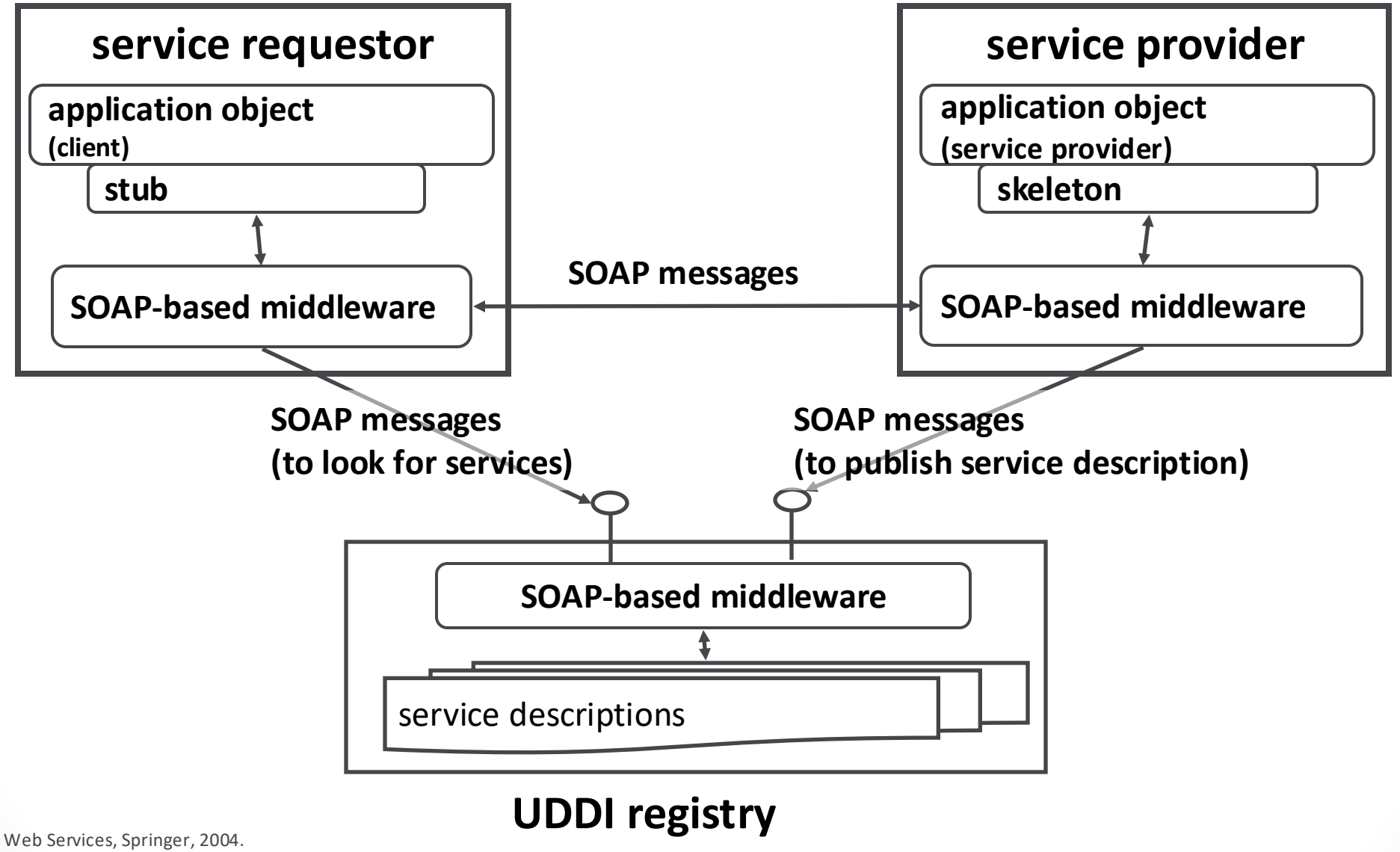
WSDL (Web Services Definition Language): Standard to specify operations (in an interface), transport and addressing of a Web Service in XML-based document.

UDDI (Universal Description, Discovery and Integration): Standard to specify service registries, service descriptions and how to publish and discover services in registries.

WEB SERVICE :: SIMPLIFIED



WEB SERVICES, SOAP & UDDI



Source: [1] G. Alonso et al. Web Services, Springer, 2004.

REPRESENTATIONAL STATE TRANSFER

REST is an architectural style for distributed hypermedia systems

It is about

- roles of different components
- the constraints upon their interaction
- interpretation of significant data elements

It is not about component implementation and protocol syntax

Some terms: Resources, Representations, Components, Connectors

Roy Fielding's dissertation [5]: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

REST RESOURCES

A resource in REST is the key abstraction of information. Any information can be named can be a resource (a document, an image, etc..)…

Resource identifiers are used to identify a particular resource involved in an interaction between components.

Resource representations capture current or intended state of a resource. Bytes, metadata that describes the data.

Resources are decoupled from their representation so that their content can be accessed in different formats (e.g. HTML, XML, …)

REST: CONSTRAINTS (PART 1)

Client-Server

Separation of concerns

e.g., servers do not take care of user interface or state, while clients do not take care about data storage, and data representation on the server

Stateless

Requests from client must contain all information necessary to understand the request.

"… and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client."

Cacheable

Resources implicitly or explicitly labeled as cachable or non-cacheable

REST: CONSTRAINTS (PART 2)

Uniform Interface → uniform interface between clients and servers and decoupling of interface from implementation, with following constraints:

1. **Identification of resources**
   Resources are identified with resource identifiers (URI)
2. **Manipulation of resources through representations**
   A Web page is a representation of a resource (human readable)
3. **Self-descriptive messages**
   contain all necessary information required to process the message
4. **Hypermedia as the engine of Application State (HATEOAS)**

REST: CONSTRAINTS (PART 3)

Layered System

The system may be composed of hierarchical layers. Therefore client is unaware if he is connected directly to the server, or some intermediary.

Code-On-Demand (Optional)

Client's functionality can be extended by downloading and executing code in the form of applets or scripts.

RESTFUL WEB SERVICES

REST applied to Web Services, HTTP and its special role in Web

Resources are identified by URIs

Resources are decoupled from their representation and represented by various formats such as HTML, XML, JSON, and other media types

Explicit use of HTTP methods (GET to retrieve resources from a server, and POST, PUT and DELETE to create change and retrieve a resource on the server).

Interactions are stateless, and request messages are self-contained. A RS does not hold information about the application state/session

REST AND HTTP (1)

Nouns

- URIs are the equivalent of a noun
- Resources are represented by Nouns

Verbs

- Loosely describe actions that are applicable to nouns (resources)
- Using different verbs for every noun would make widespread communication impossible

When applied to web, in REST we use following verbs (~HTTP Actions)

- POST initialize the state of a new resource (create)
- GET to retrieve the current state of the resource (read)
- PUT modify the state of a resource (update)
- DELETE clear a resource (delete)

```
HTTP GET /customers
HTTP POST /customers
HTTP PUT /customers/{id}
HTTP DELETE  /customers/{id}

HTTP GET /customers?age=26
```

REST AND HTTP (2)

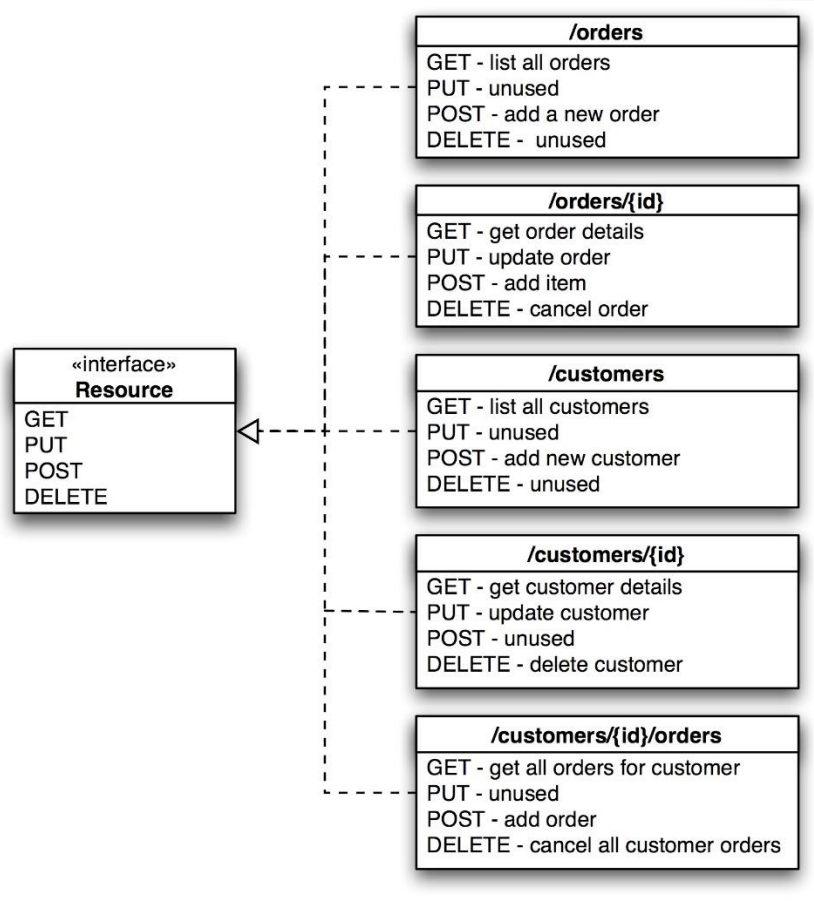
URI is identification of a resource
e.g., entity, files, location, place,…

- Good URI scheme can be reused, regardless of backend architecture.
- It does not matter what the actual resource is url interface to the server describing them, so that the clients can interact

Use of HTTP actions to describe actions on resources

Future Proofing?

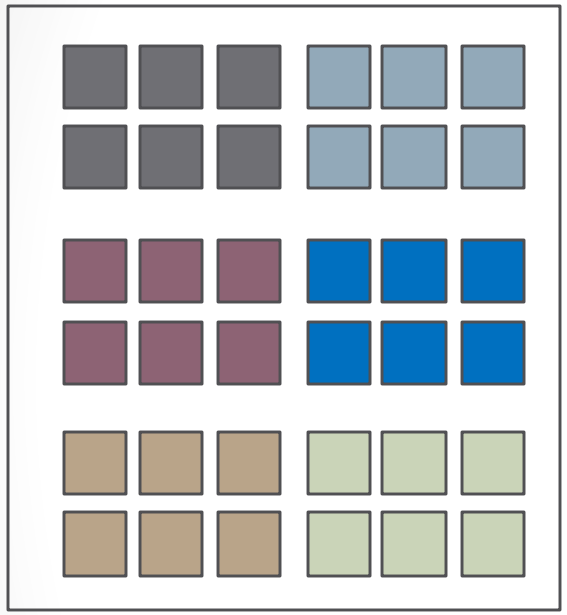
- Breaking vs. Non-breaking change
  also how to update API so that it does not break your app?



src: https://www.infoq.com/articles/rest-introduction/

Microservices

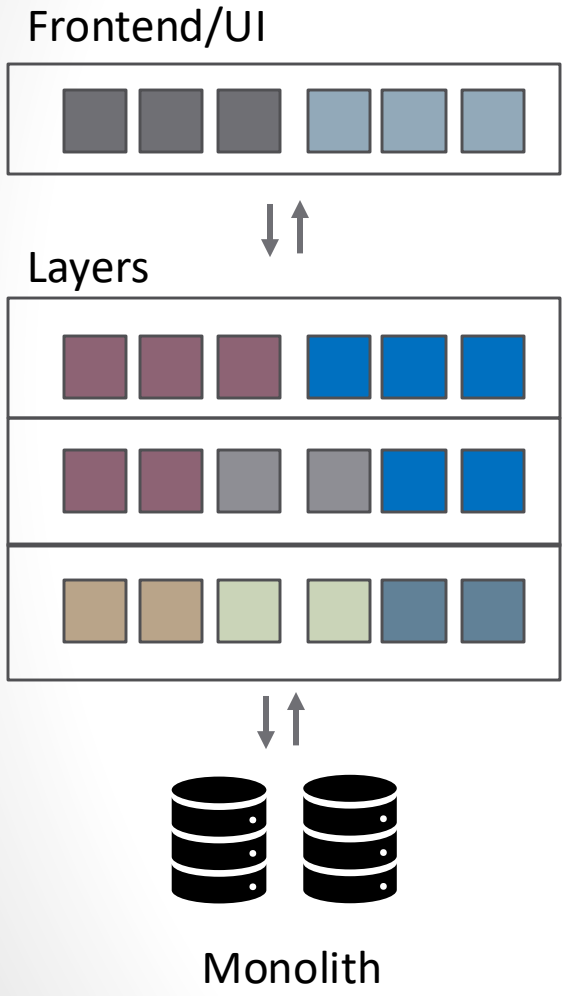
SCALING (A MONOLITH)



Large, complex **monolith** comprised of many modules and components

- Comprised of components and modules, but still builds and **deploys as a single application**

SCALING (A MONOLITH)



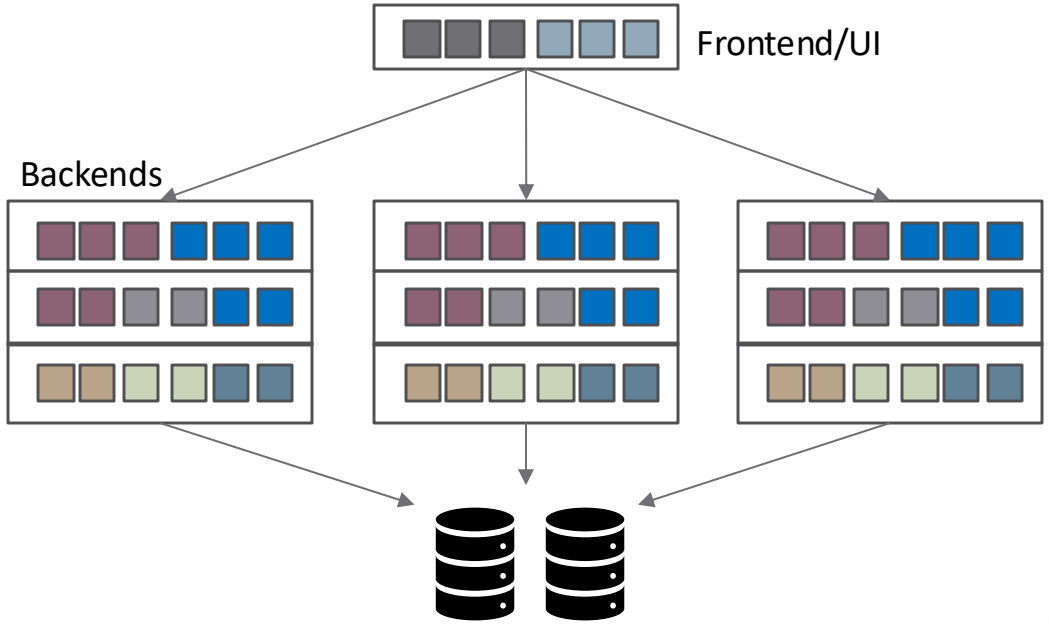
- Comprised of components and modules, but still builds and **deploys as a single application**
- Complex (online) systems increase **development** complexity
- A single deployable application may not be enough (or viable) for every usage scenario
- Complexity needs also to be handled at runtime
- The whole application needs to be tested before **deployment**
- **Failure of single module**, may induce downtime
- **Scalability** -> how to handle traffic spikes?

SCALING (A MONOLITH)

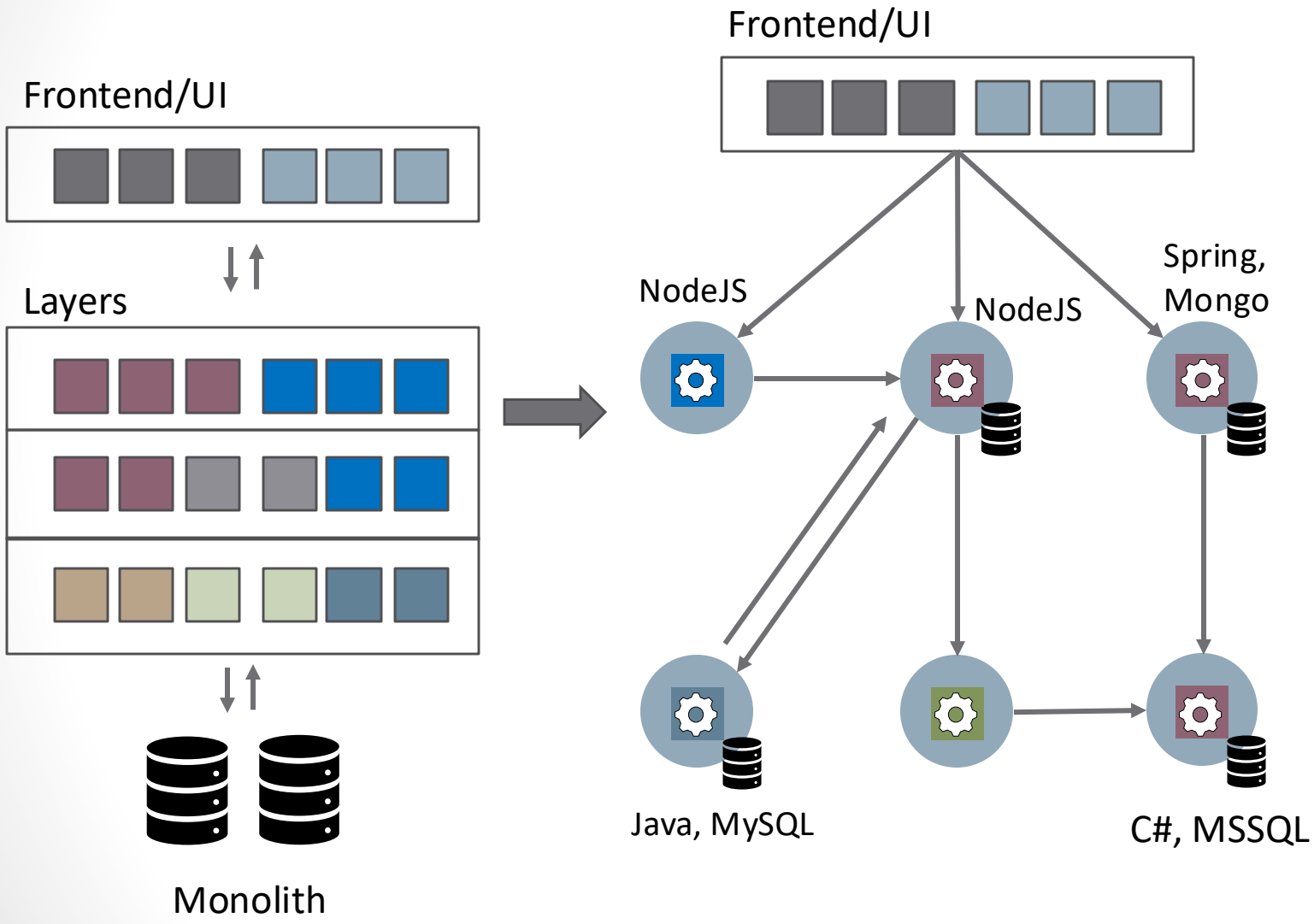
A web app suddenly needs to serve 100x more users due to increase in popularity?
you have sudden short increases in traffic with your app? How to scale a monolith?

Maybe in some scenarios (e.g. EC2)?

Due to the lack of control
the whole application is
always deployed, even though only
some subset of features
needs to be scaled up



SCALING, OTHER IDEAS?



Small, independent standalone applications

Each running on its own server

Services communication over the network over by calling REST APIs

Running different technologies

Scaling?
Fault tolerance?
Development?
Deployment?
How do services call each other?

C#, MSSQL

MICROSERVICE ARCHITECTURAL STYLE

***…Microservice architectural style is an approach to developing a single application as a suite of small services**, each **running in its own process** and communicating with lightweight mechanisms, often an HTTP resource API.*

*These services are **built around business capabilities** and **independently deployable** by fully automated deployment machinery. There is a **bare minimum of centralized management** of these services, which may be written in different programming languages and use different data storage technologies.*

James Lewis and Martin Fowler (2014): https://www.martinfowler.com/microservices/

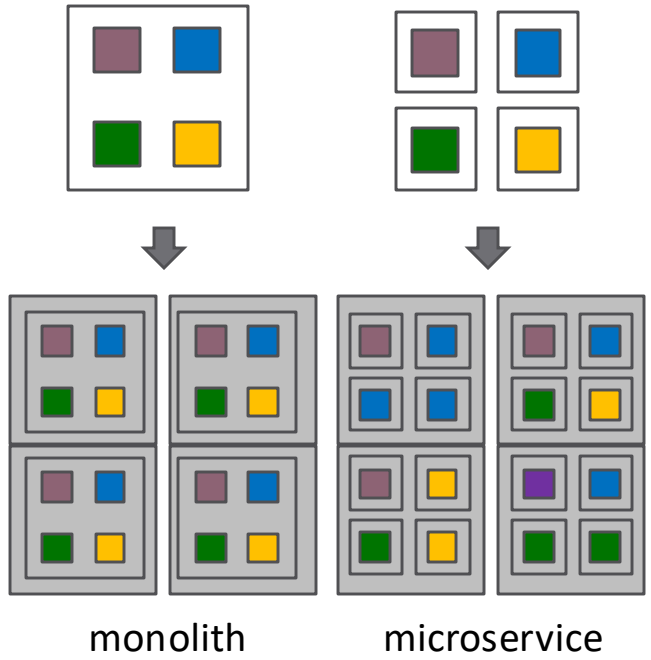
MICROSERVICES

Small autonomous services that work together around a business domain

Self-contained, reusable, independent

This style gives more options and more directions

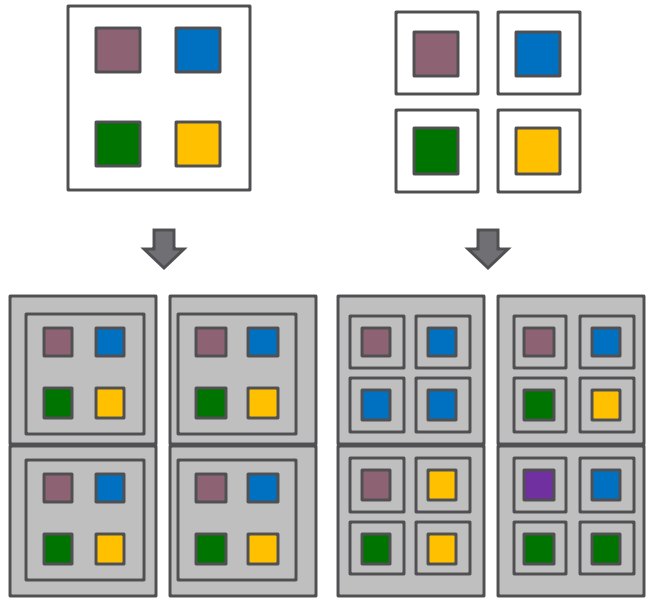
They can change, evolve and be deployed independent of each other



COMMON CHARACTERISTICS

Componentization via services

- Individual pieces of software in their own processes, which can communicate between each other
- Well bounded
- More diversity, more options and more design directions



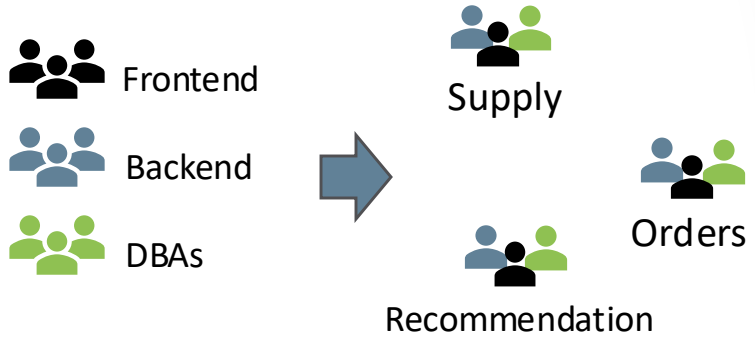
Not defined around development, but about upgradeability and replaceability

- Services are out-of-process components that communicate through mechanism such as WS requests or RPCs
- Forces more explicit component interface
- Not relying on integration of libraries but rather on executables exchanging messages

COMMON CHARACTERISTICS

Organization around business capabilities

- Teams organize around concrete business capability (not around infrastructure)
  e.g.: Orders, Shipping, …
- Unlike the traditional division across layers
  e.g., around UI Middleware Dba… (around infrastructure)
- Teams have full range of skills for development, from UI,
  to backend and project management
- Each team has direct connection to end-user
- As much about team organization, as it about technology
- Software architecture and team organization are always heavily coupled together

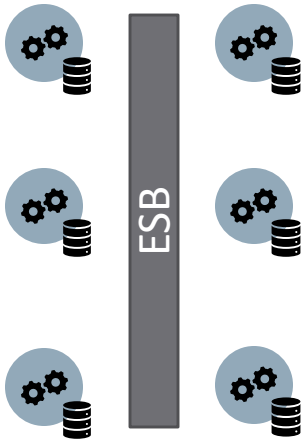


Traditional vs. Teams in microservices

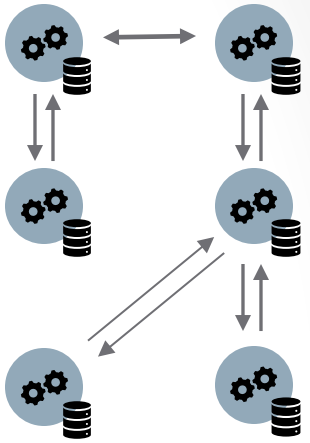
COMMON CHARACTERISTICS

Smart endpoints and dumb pipes

- As decoupled a cohesive as possible
- Communication choreographed/orchestrated through REST protocols, rather than complex (unlike SOA, there is no complex middleware)
- Mainly focuses on usage of principle of world wide web and Unix
- One of the bigger challenges when switching from monolith is changing communication pattern
- Example: ESB, SOAP Middleware
- Possible side effects of having a middleware: draining smarts from services



SOA ESB approach

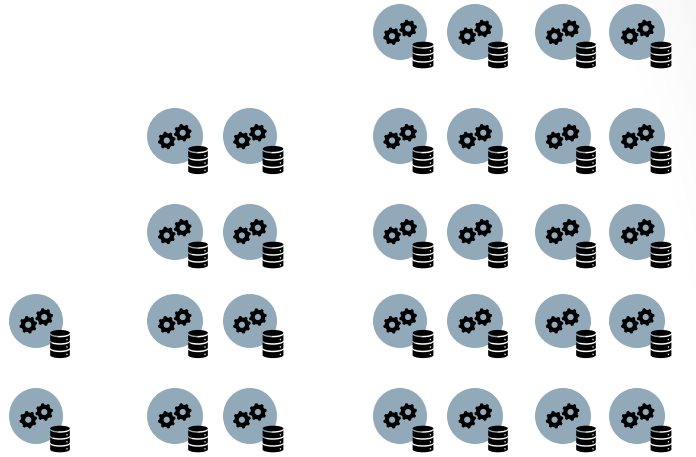


Microservices

COMMON CHARACTERISTICS

Infrastructure Automation

- Continuous Integration
- Automated Testing, Continuous Delivery
- Management of services in production



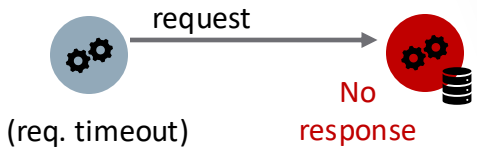
Example: Increase in # of services in time

Design for Failure

- Application need to be designed to tolerate the failure of services
- Detection of failures very important.
- Automatic restoration of services if possible

COMMON CHARACTERISTICS

Design for Failure (2)



- Assume that failure will happen, especially as you scale
- RPC calls (vs distributed objects) → local functions do not fail (usually)
- Real-time monitoring (architectural elements and business relevant metrics)
  Netflix → Chaos Monkey (https://github.com/Netflix/chaosmonkey)

Evolutionary Design

- Frequent, fast and well-controlled changes to software
- Component independent replacement and upgradeability
- More granular release planning

COMMON CHARACTERISTICS

Decentralized Data Management

- Each service gets its own data storage, and does not share it directly with others
- Sharing has to go through services that wrap the data
  → each service chooses its own flavor of database (SQL, NOSQL, filesystem, …)
- Conceptual model may be different for different parts of the system
  e.g.: sales view of customer differs from the support view
- Preference towards decentralized storage
  → each server manages its own database
  → different DB technologies may be more suitable for types of problems
- Transactionless coordination between services, **eventual consistency**

COMMON CHARACTERISTICS

Decentralized Governance

- Development teams take responsibility for software in production as well
- Autonomy: give people as much as freedom as possible to do the job at hand
- Smaller granularity of services makes it easier to focus it on supporting business capabilities
- Products not projects
- Read more: (https://gilt-tech.tumblr.com/post/102628539834/making-architecture-work-in-microservice)

ADVANTAGES

Dynamic scalability at runtime

Fault Isolation

- If one service fails it does not mean that the whole system will crash

Language/Platform Independence

- Services can be written in different languages, frameworks, different technologies

Teams

- Team can be small
- Developers have freedom to pick the technology, and strategies that work best for them
- New team members can become productive more quickly
- Fast Iteration cycles

ADVANTAGES

Deployment flexibility

- When a change is required only those services that require change need to be redeployed
- Easy integration and automatic deployment (using various available tools)

Technology flexibility

- Components can be **developed**, **deployed** and **tested** individually, **without** affecting the whole system
- Different component may **evolve** at different paces, **without affecting** two overall functionality of the system

Well defined boundaries (forces modularity)

DISADVANTAGES

Complexity

- Can be **difficult to breakdown** a single application
- Working with **distributed systems is hard**, asynchrony is essential, but adds a lot of complexity
- Complex networking and complex communication
- Difficult to debug, tracing requests can be difficult
- Architecture complexity and deployment complexity
- Data organization and handling, consistency
- Difficult to rearrange and to make **changes** across the components

DISADVANTAGES

Discovery

- How do services know where to find other services?

Challenges

- In terms of knowledge, organization of architecture, deployment
- Containerization of services
- Dealing with many applications

Resilience does not come for free

- Fault tolerance needs to be addressed

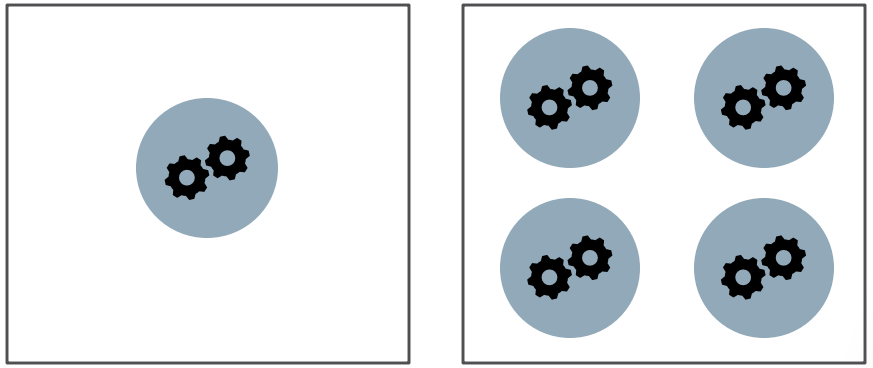
SOME CHALLENGES: INDEPENDENT DEPLOYMENT

When a services is changed, it should be deployable to production without requiring to change anything else

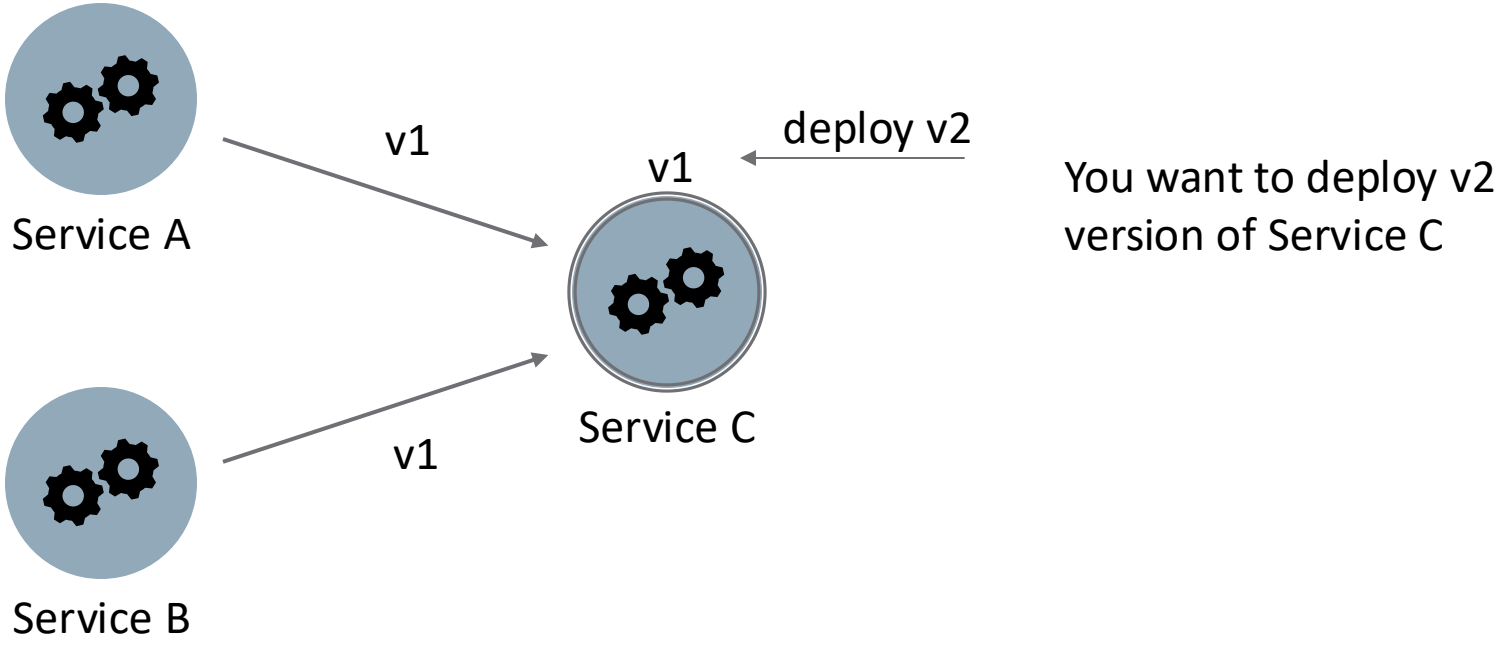
Having 1 host per service can be very costly! (containers)

Consumer-driven Contracts

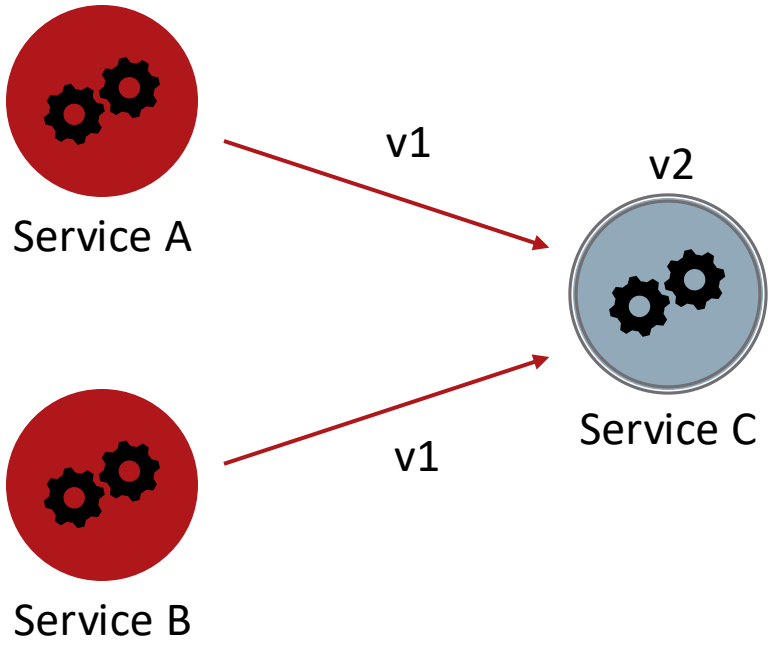
Lockstep releases need to be avoided



SOME CHALLENGES: LOCKSTEP RELEASE



SOME CHALLENGES: LOCKSTEP RELEASE

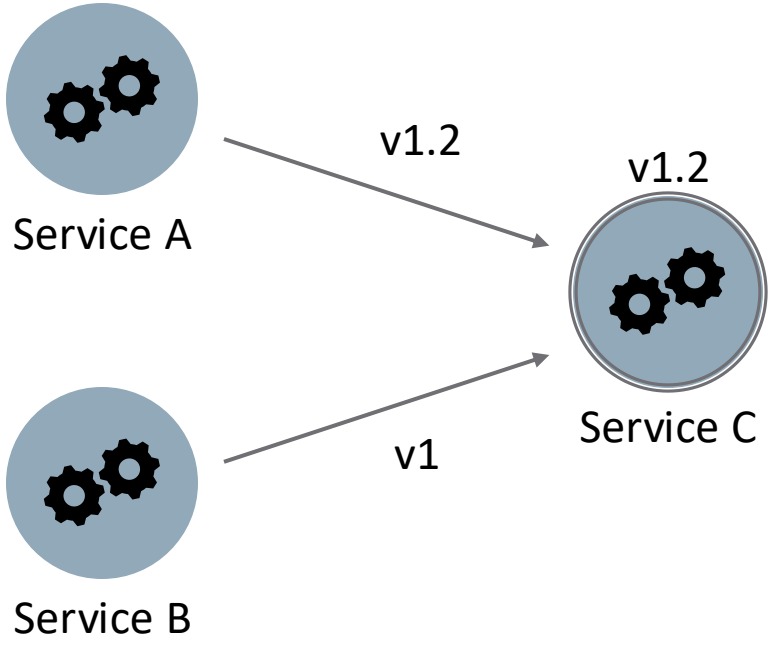


Services A and B cannot communicate with Service C

Deploy v2 to Service A and B?
Result: a lockstep deployment

CHALLENGES: VERSIONING

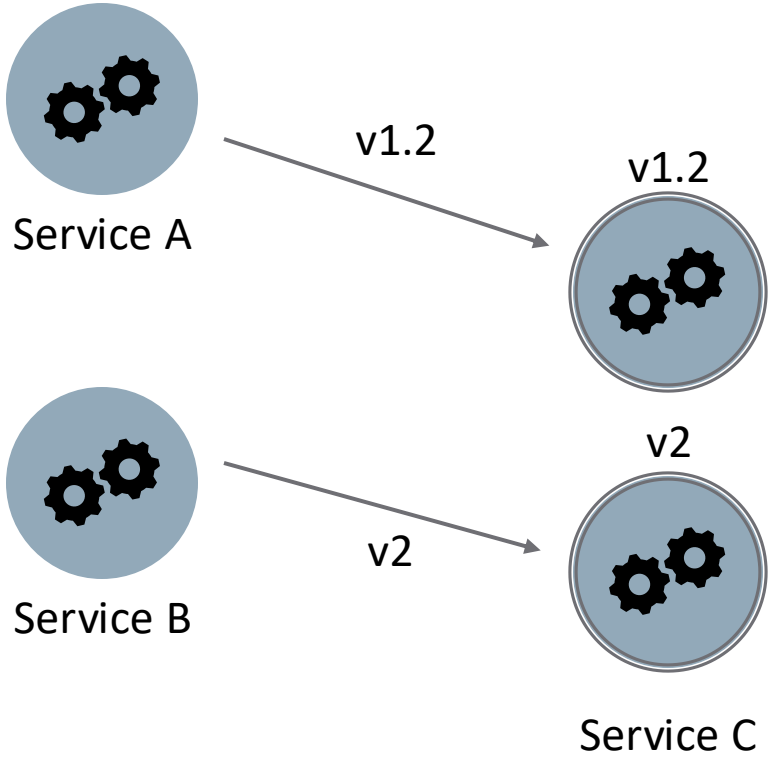
Semantic versioning: major, minor and patch versions



Major versions: backwards incompatible changes
Minor versions: backwards compatible changes
Patches: whenever

CHALLENGES: VERSIONING

Semantic versioning: Major, minor and patch versions



Major versions: backwards incompatible changes
Minor versions: backwards compatible changes
Patches: whenever

Expanding change over a breaking change

Avoiding lockstep

COMMUNICATION

Request/Response

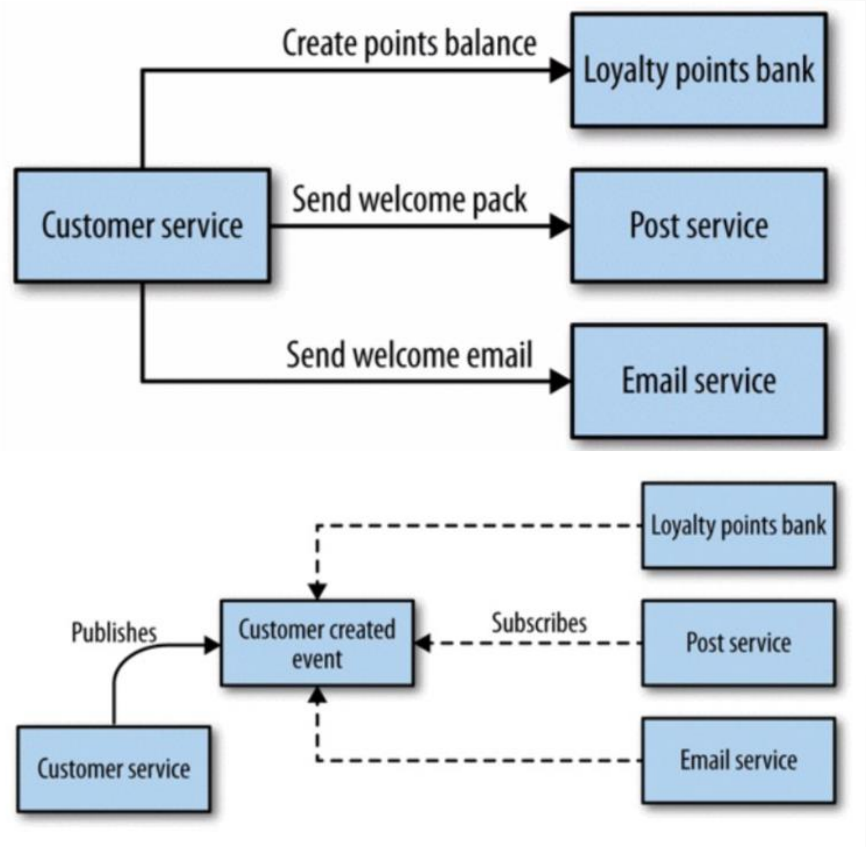
- Often synchronous, but can be asynchronous too by registering a callback

Event-Based

- Asynchronous by nature
- Decoupled collaboration
- Client does not know who will respond to an event

Asynchronous vs synchronous

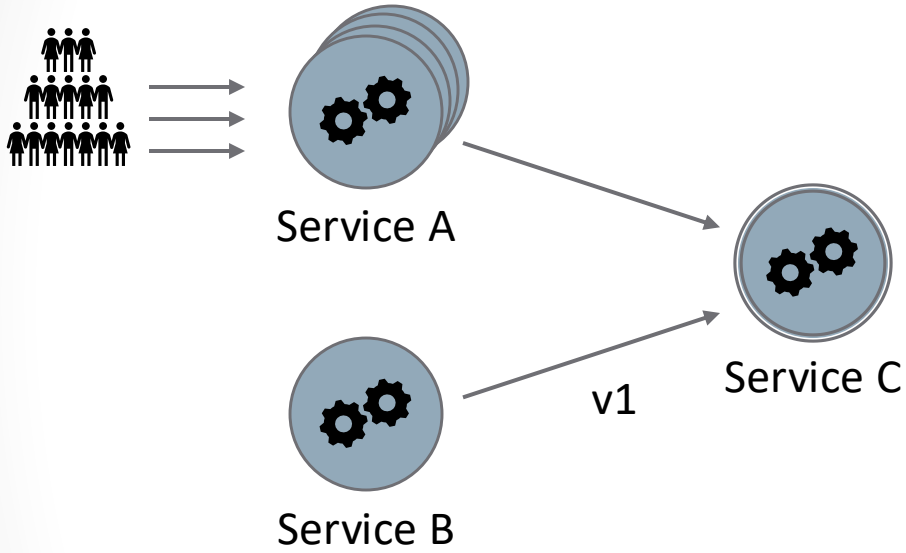
- How to handle long-running tasks
- What about simplicity, and results that we want straightaway?

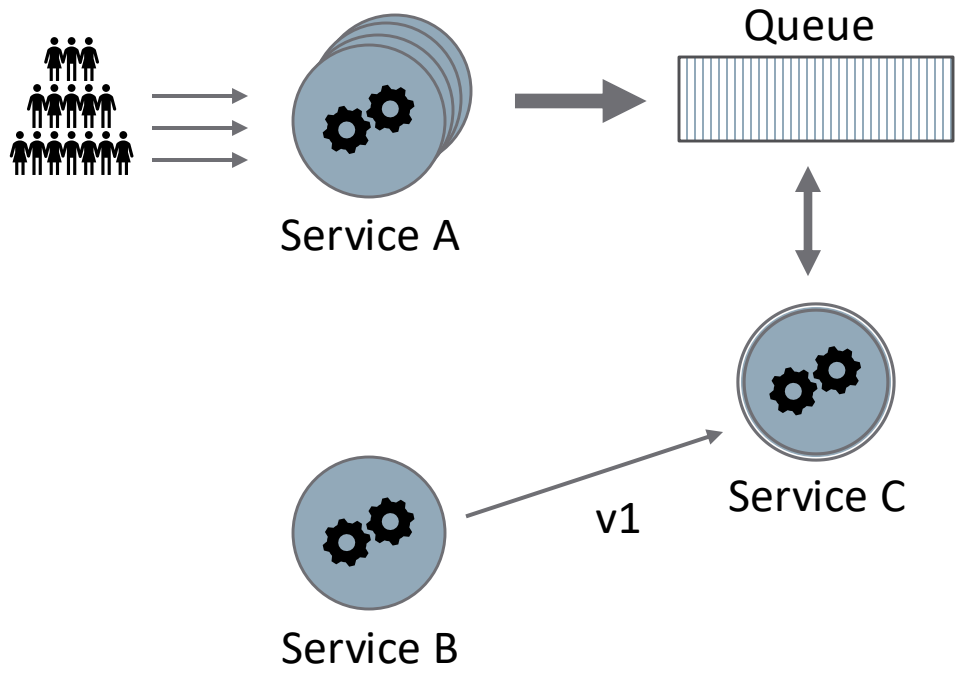


Source: Building Microservices: Designing Fine-Grained Systems

Asynchronous calls lead to services that are more decoupled and scalable

HANDLING TRAFFIC SPIKES

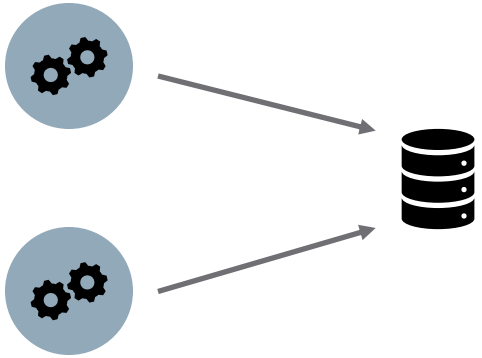




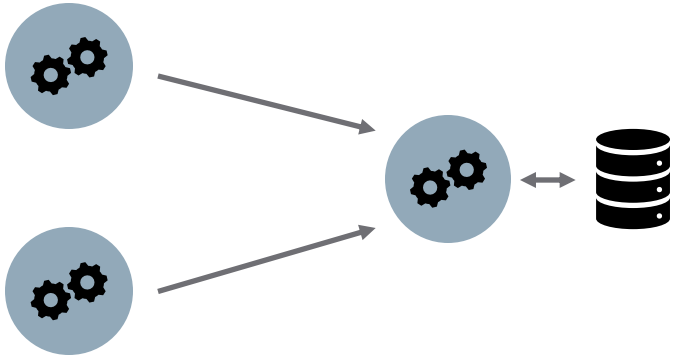
Enqueue messages for later retrieval by 1 or more consumers

HIDING IMPLEMENTATION DETAILS

No 2 Services should be reading/writing to the same database at the same time



- Changes to DB can break all consumers
- Consumers are tied to 1 DB technology
- Coupling is not so loose anymore
- Difficult to avoid breaking changes

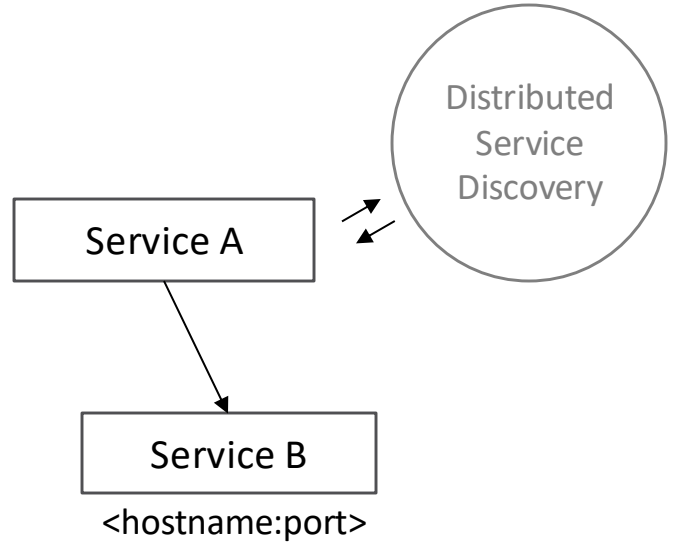


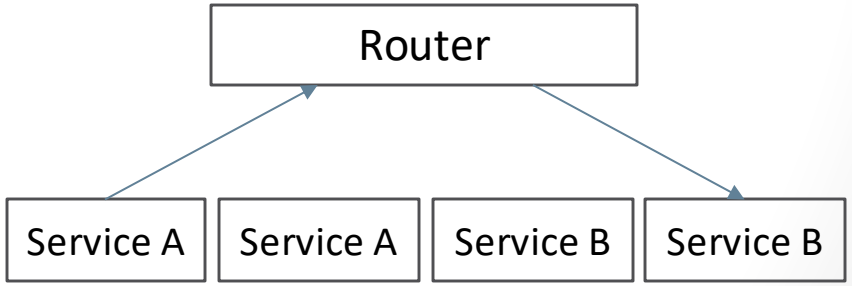
Hides implementation details
Allows DB technologies to be switched

SERVICE LOCATION

Where do services live?

- Hard coding? What about changing URLs?
- Service Discovery solutions (e.g.: consul, etcd)
- Distributed lookup services
- Support registration and deregistration
- Router: Proxying all traffic
- Both require scalability
- Router is transparent, can be exposed externally, but harder to make scalable
- Discovery is usually simpler, and it does not handle data





SOA, MICROSERVICES

SOA uses rich middleware for communication between services

SOA does not provide guidance about service granularity,

whereas Microservices put more constraints on how to split things into components and ensure that they are not overly coupled

Microservice Architecture can be viewed as a subset of SOA, as it adds more constraints to it

SOA based application use Enterprise Service Bus messaging for communication between services, whereas Microservices are loosely connected services providing simple functionalities, and favor light-weight communication protocols

MICROSERVICES & CONTAINERS

How do I deploy my service or microservice on different servers and environments?

You need to find a way to package each application and all of its dependencies so it can be moved between different environments and run without changes?

→ wait, a container actually does exactly that!

Containers are packages of your software that include everything that they need to run (binaries, libraries, resources, system tools, etc..) isolated on a shared operating system

Microservices can be packed into a singular packed with all their assets, and dependencies and deployed

- **Isolated workload environment**
  a uniform deployment of containerized services regardless of technology used by service
- **A service in a container can be independently deployed**
- **Orchestration through container orchestration platforms (Kubernetes, Docker Swarm, Helios, etc.)**

REFERENCES, RESOURCES, TUTORIALS

1. **Sam Newman. 2015. Building Microservices (1st ed.). O'Reilly Media, Inc..**
2. https://www.martinfowler.com/microservices/
3. https://dzone.com/articles/notes-microservices-martin
4. **Docker -** https://docs.docker.com/get-started/
5. **Docker -** https://github.com/docker/labs/tree/master/beginner/

Soa, Web Services, REST

1. **Jersey documentation -** https://jersey.java.net/documentation/latest/jaxrs-resources.html
2. **IBM RESTful Web Services: The basics -** http://www.ibm.com/developerworks/library/ws-restful/
3. **REST Web Service Tutorial, with Setup -** https://www.tutorialspoint.com/restful/restful_quick_guide.htm
4. **REST Web Service Tutorial with Jersey -** http://www.vogella.com/tutorials/REST/article.html
5. **JAX-RS Endpoint Lifecycle Explained -** https://github.com/stoicflame/enunciate/wiki/JAX-RS-Lifecycle
6. **JAXB Tutorial -** http://www.vogella.com/tutorials/JAXB/article.html