FHV
Vorarlberg University
of Applied Sciences

Application Integration and Security

Valmir Bekiri

Philipp Scambor

Software Architecture

What is software architecture?

Software architecture may be described as the **structure** of the system combined with the **architectural characteristics**, the **architectural decisions** and **design principles.**

What is software architecture?

Structure of the system

- Also called ***architectural style***
- Microservices or layered architecture, etc.

Architectural characteristics

- Definition of success
- “-ibilities“ (scalability, availability, flexibility, etc.)

Architectural decisions

- Serve as rail for development
- For example: „sensitive data may only be processed in service X“

Design principles

- Rather loose guidelines to follow
- For example: “Asynchronous messaging is used if possible“

What is software architecture?

Everything in software architecture is a compromise

If you think something is not a compromise you most likely just don‘t know it yet.

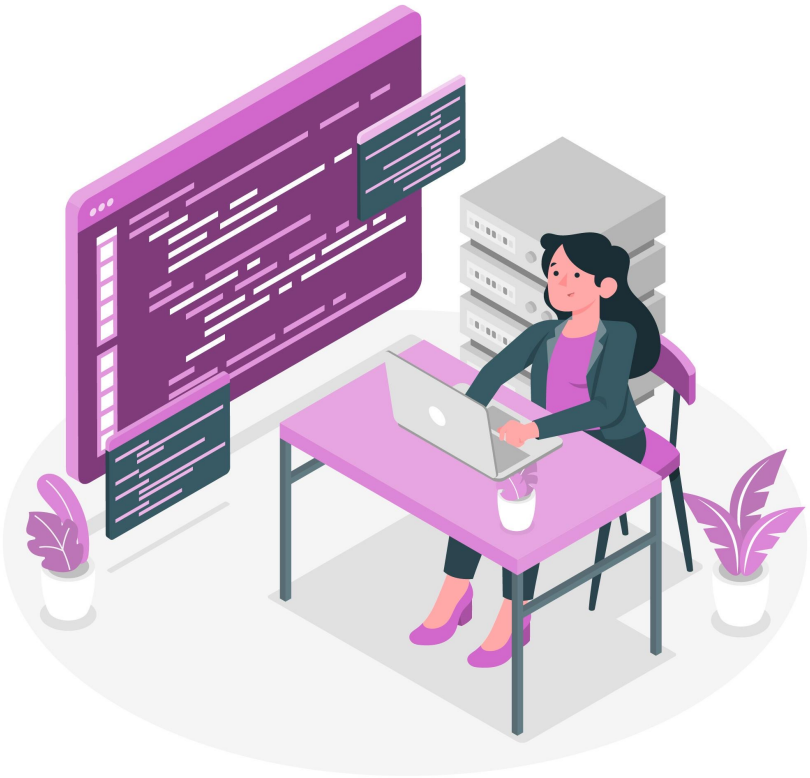
First law of software architecture

The *why* is more important than the *how*.

Architecture is more than just combining structural elements

Second law of software architecture

The Software Architect



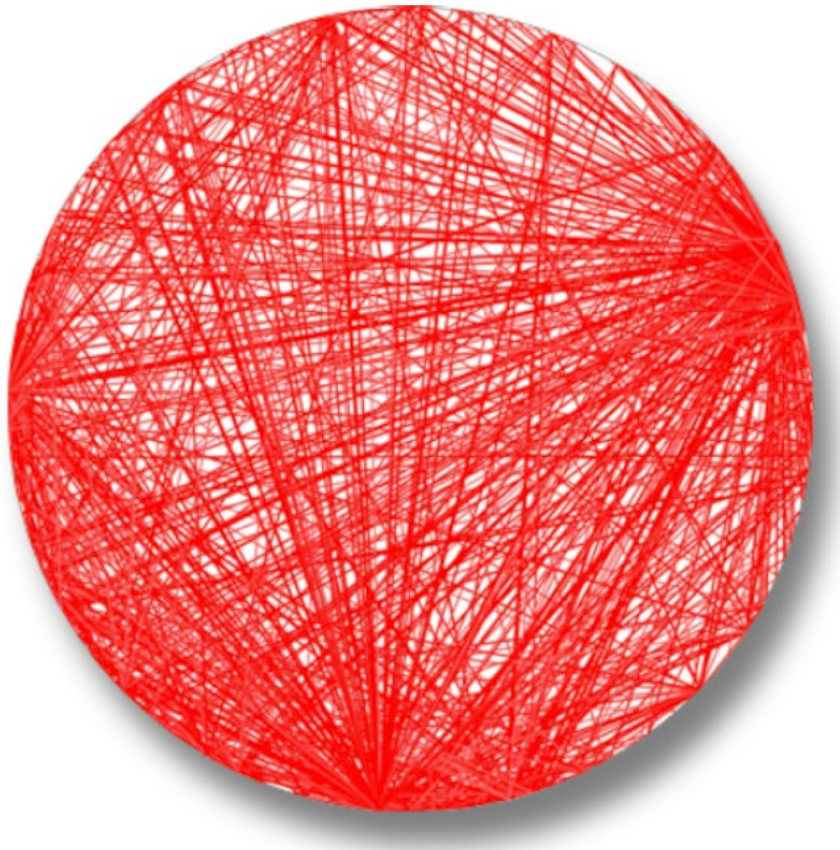
- Makes **architectural decisions**
- Performs **constant analysis** of the architecture
- Keeps up to date with the **latest trends**
- Ensures that decisions are adhered to
- Has a wide range of **knowledge and experience**
- Has experience in the **business environment**
- Has **interpersonal skills**
- **Understands politics** and be able to operate in this sphere

Architectural style

Big ball of mud

The Big Ball of Mud is a haphazardly cobbled-together, sprawling jungle of spaghetti code held together with duct tape and baling wire.

– Brian Foote und Joseph Yoder (1997)



An example of a “big ball of mud” architecture using a real code base (class dependencies)

Source: Richards & Ford 2020, S. 127

Architectural style
Client/Server

2-tier* architecture

- Frontend and single server
- Common for simple websites etc.

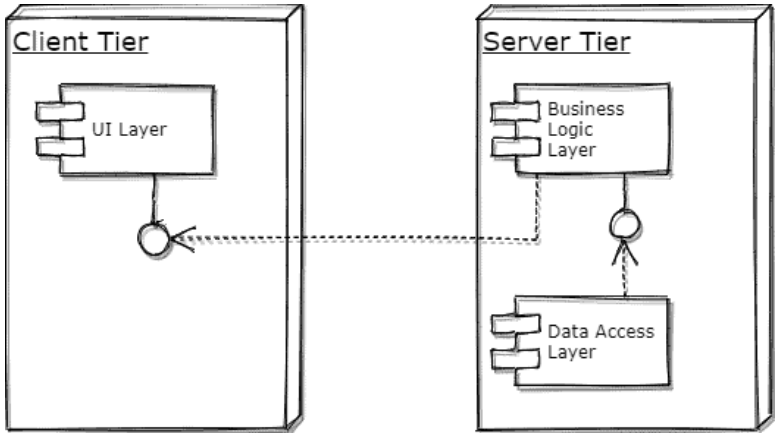
3-tier* architecture

- Frontend, application server and database server
- Common for simple CMS-based websites or web applications

(*) Tier

_refers to physical separation

_scalability is the driving force for dividing an application into tiers



2-tier / client-server architecture

Source: https://www.baeldung.com/cs/layers-vs-tiers

Architectural styles

Monoliths and distributed architectures

Monolithic

all of the code is deployed at once as a single unit

+ simple and cost efficient approach for small projects

- hard to scale in general

- hard to maintain if there are many development teams

Distributed

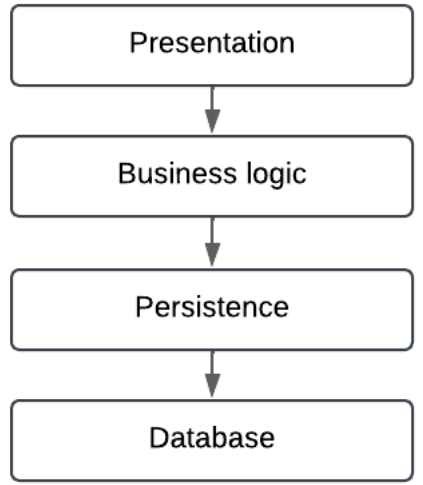
Many connected deployment units

+ Strong in terms of performance, scalability and availability

- Distribution comes with many challenges regarding maintenance, data synchronisation, network related issues, security, etc.

Architectural style
Layered Architecture

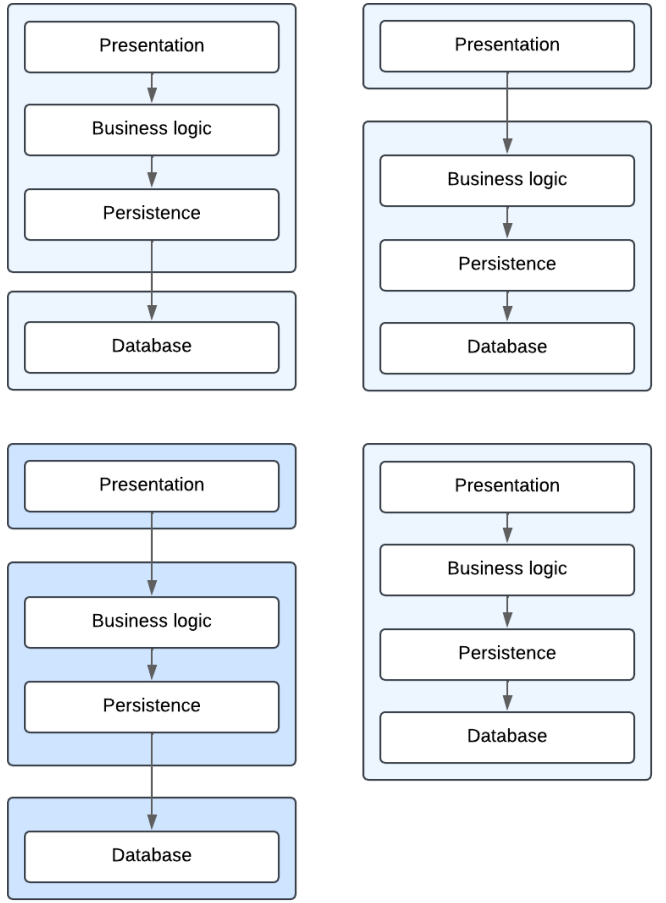
- Probably most **common architecture** in the real world due to simplicity and straight forward approach
- Components of the software get divided into **horizontal layers of a given purpose**
  - Every **layer** defines an interface that **is consumed by the layer that is above**.
  - A layer **never gets called from below**.
  - A layer may only call the layer that is directly beneath it
- **Layers serve as abstraction**: Presentation layer does not need to know how data is stored or what the logic behind certain operations is



Layered architecture example

Architectural style
Layered Architecture

- Also called **n-tiered** architectural style because its layers may be deployed on different physical units
- Example 3-tier layered architecture:
  - Presentation layer is a React frontend hosted on the edge
  - Business logic and persistence layers are deployed as a single unit on a Windows server (implemented with C# .NET)
  - Database layer is running on its own server (some MySQL-Server)

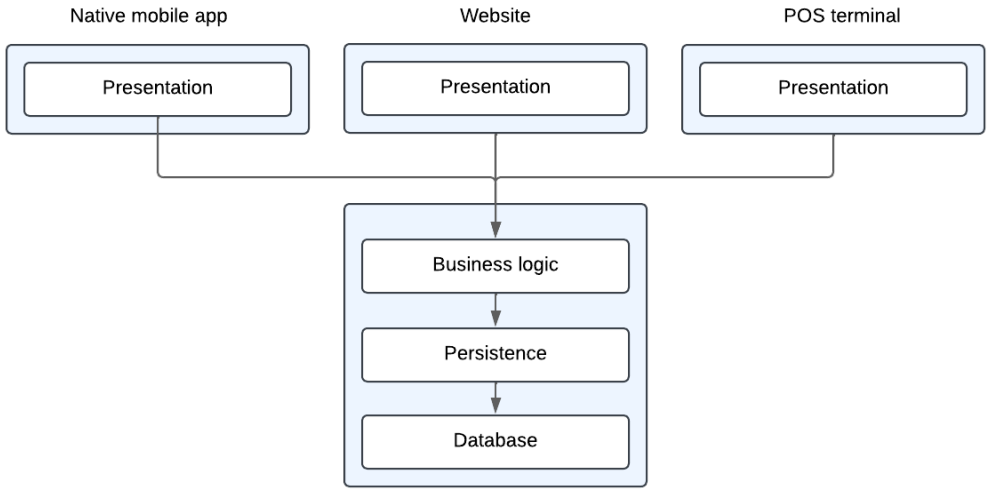


Layered architecture n-tier variant examples

Richards & Ford 2020, S. 139-148

Architectural style
Layered Architecture

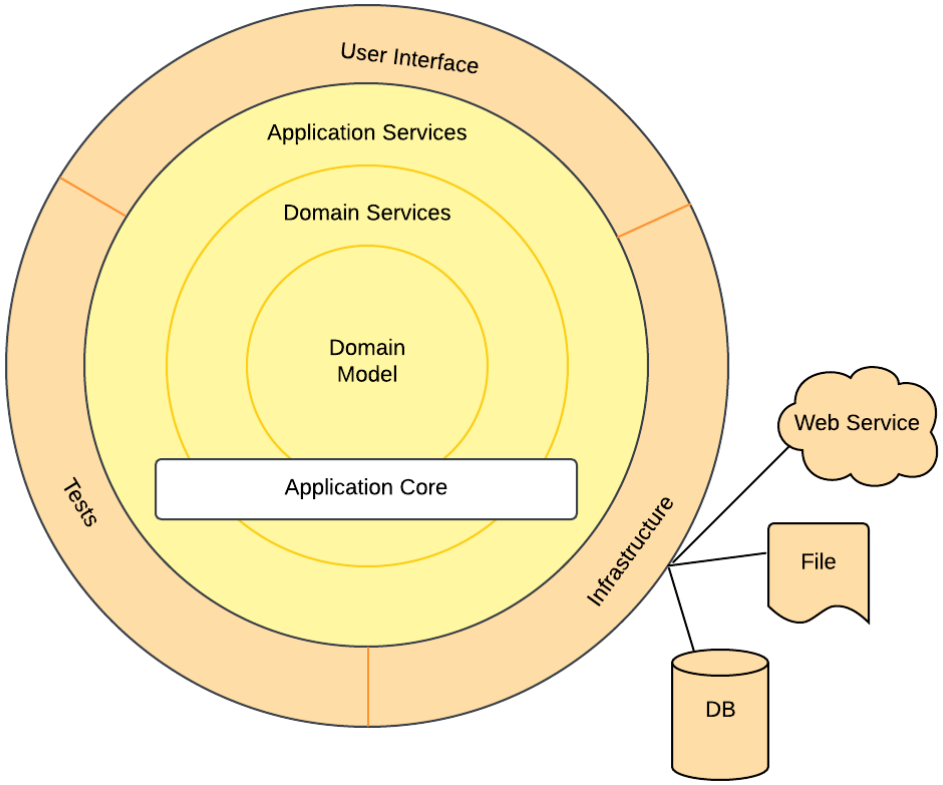
- Whenever you here about **headless** architecture (e.g. headless CMS, headless Commerce, etc.) you can just think of an at least 2-tiered layered architecture where the presentation is an independent tier
  - The term “headless” is widely known and used by non technical folks in business
  - The purpose of a “headless” approach is to allow multiple frontends to consume the same backend or just to divide backend from frontend to be able to use modern technologies for the frontend regardless of the backend technology stack



“headless” architecture that contains of several independent presentation tiers that use the same business logic layer

Architectural style
Onion Architecture

- Introduced in 2008 by Jeffrey Palermo (Original Blog Post)
- Key tenets of Onion Architecture:
  - The application is built around an independent object model
  - Inner layers define interfaces. Outer layers implement interfaces
  - Direction of coupling is toward the center
  - All application core code can be compiled and run separate from infrastructure
- Based on the **dependency inversion principle** which promotes decoupling modules from each other by making them independent from details and dependent on abstractions (often done with interfaces).

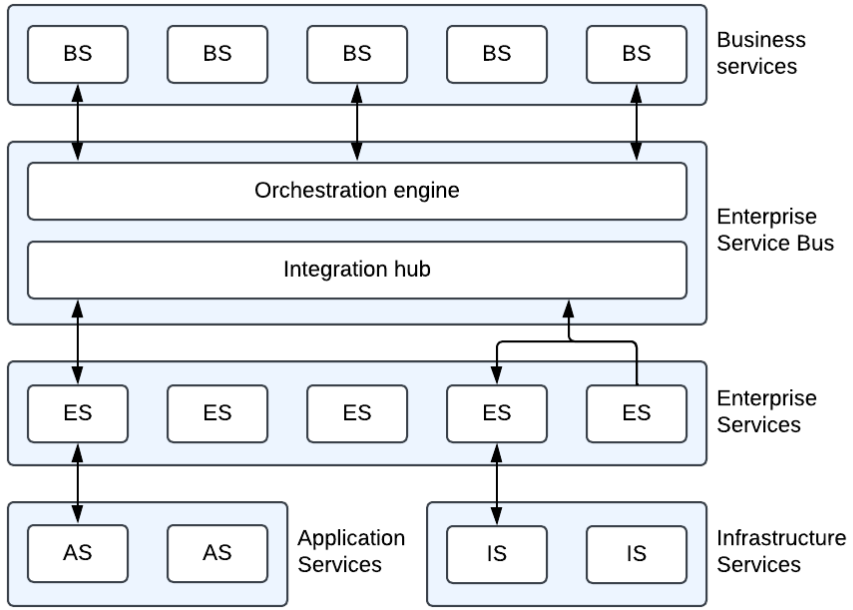


Onion Architecture

https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

Architectural Style
Service Oriented Architecture (SOA)

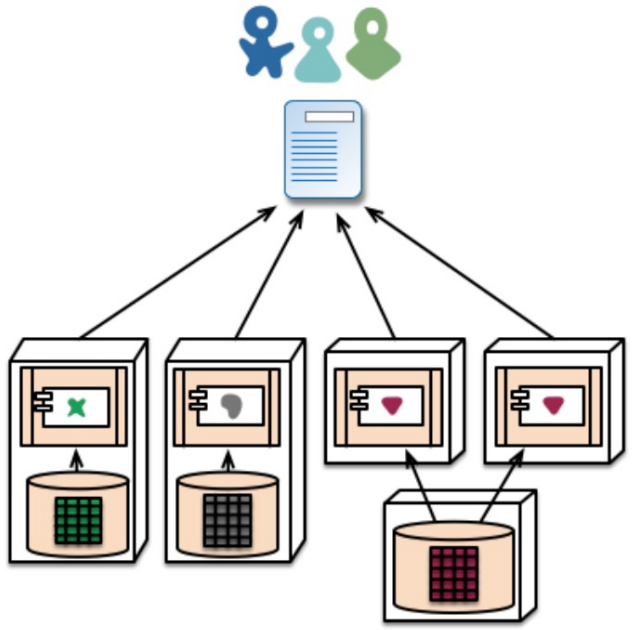
- SOA emerged in the **late 90s** because of the need to scale the IT systems in companies.
  - database servers were expensive commercial products. Therefore, SOAs were often based on a **single shared database** (accessed via infrastructure service)
  - Computational resources were expensive. Therefore, architects tried extensively to make things reusable.
- Breaking the system into **services**
  - that are **developed and maintained by independent teams** and departments
  - that can **be reused and rearranged** to solve future business needs
- Business services serve as entry point (not including any logic) that call the **orchestration engine** (Enterprise Service Bus) that is the heart of the architecture. This engine then orchestrates calls to certain enterprise services (=**bottle neck**).



Topology of a orchestrated service oriented architecture

Architectural Style
Microservices

- Emerged in 2012 as an optimised approach to SOA
  - **Smart endpoints and dump pipes:** moving away from a heavy orchestration engine to rich APIs and direct communication
  - As **decoupled** and **cohesive** as possible.
  - **Decentralised data management:** each microservice may have its own database and domain model
- Microservices communicate via lightweight **RESTish protocols** or over a **message bus**
- Microservices often use mechanisms like **service discovery** to dynamically resolve services before calling them. This allows dynamically adding or removing service instances and balancing loads between them.



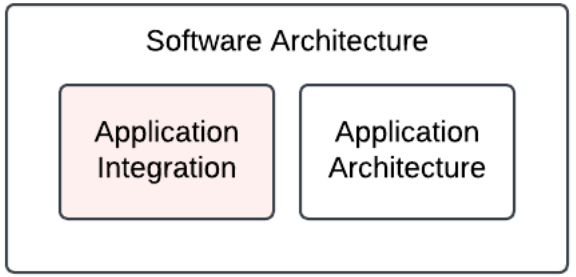
Microservices – application databases

https://martinfowler.com/articles/microservices.html

Application Integration

What is Application Integration?

Application Integration is the process of enabling distinct applications, systems, or services to **communicate and exchange data**.

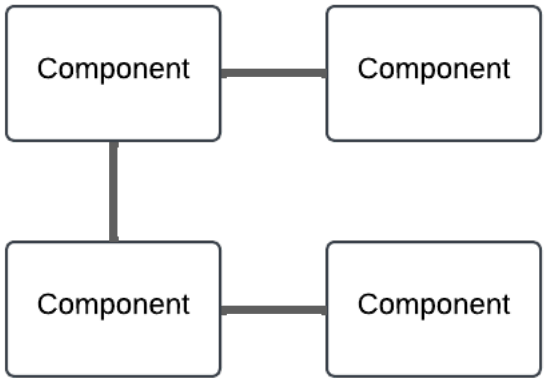


Application Integration in the context of Software Architecture

- How do applications A and B communicate? (Data flow and protocols)
- How do applications A and B know each other? Do they even know each other? (Coupling)
- What happens if application A or B is unavailable? (Availability)
- What happens to application A if application B's interface changes? (Versioning)

What is Application Integration?

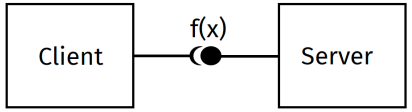
- Connectors make it possible to **assemble components** together, even if components are very **different from each other**.
- Depending on the choices that were made on how to connect the architecture together, it will **affect** the **reliability**, the **performance**, the **security** and also **how easy it is to evolve** the system.
  - **Strong coupling**: makes components highly dependent on each other
  - **Loose coupling**: components might not be affected by changes of another



Application Integration is about the edges connecting the components

Application Integration
Connectors and Transparency

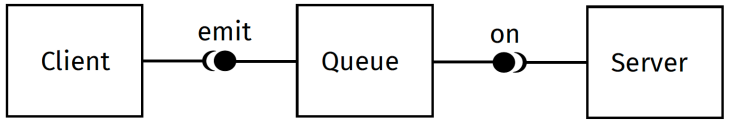
- **Direct**: Components are directly connected and aware of the other component
  - E.g. a client is calling a function *f(x)* of a server
  - Strong coupling as the components depend on each other. If the server changes the function name or parameters the client needs to be aware and updated along.



Direct connection

Source: Pautasso 2021, S. 359

- **Indirect**: Components are connected to the others via the connector and remain unaware
  - E.g. a client sends a message of a defined format into some queue or message broker. The server consumes the messages.
  - Loose coupling as client and server do only need to know the message broker and message format



Indirect connection

Source: Pautasso 2021, S. 359

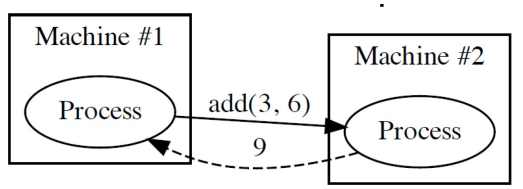
Application Integration
Connectors and Availability

- **Synchronous**: Both components need to be available **at the same time**
  - If the client calls the server the server needs to be available otherwise the client fails
  - Remote Procedure Calls (RPC), TCP/HTTP based connections (gRPC, SOAP, REST, etc.)
- **Asynchronous**: The connector makes the communication possible between components even if they are not available at the same time
  - E.g. a client sends a message into a queue and a server later consumes the queue
  - Does not require the client and server to be available at the same time.
- **Data based integration:** components access the same data
  - E.g. a shared database

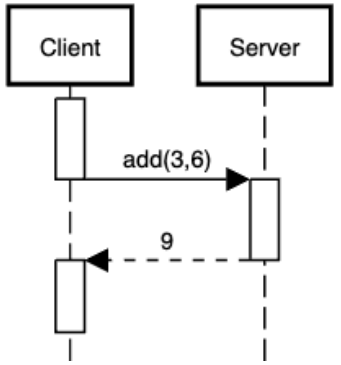
Application Integration

Remote Procedure Call (RPC)

- **Synchronous, direct** and **blocking** connection between client and server
  - You can think of RPC as calling a normal method/function in your code with the only difference that it is executed on another server.
  - The program stops during the RPC execution and continues after the function returns
  - Interface definition language (IDL) necessary for the client to know the function



Abstract view on RPC

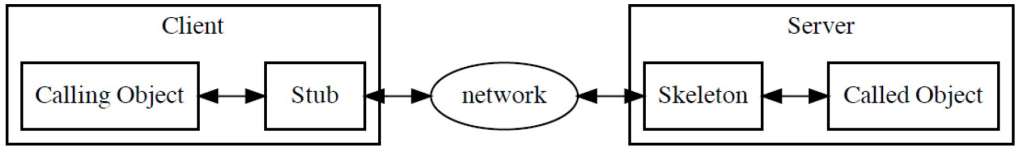


Client/Server view on RPC

Application Integration

Remote Procedure Call (RPC)

- **RPC implementations** differ between technologies, are **often incompatible** but follow a similar architecture.
  - A **calling object** on the client invokes a remote function by using the stub.
  - **The stub** acts as proxy for the function of the server and handles the network connection.
  - On the server side similar to the stub there is a **skeleton** that receives information from the network and calls the function implemented in the **called object** on the server.

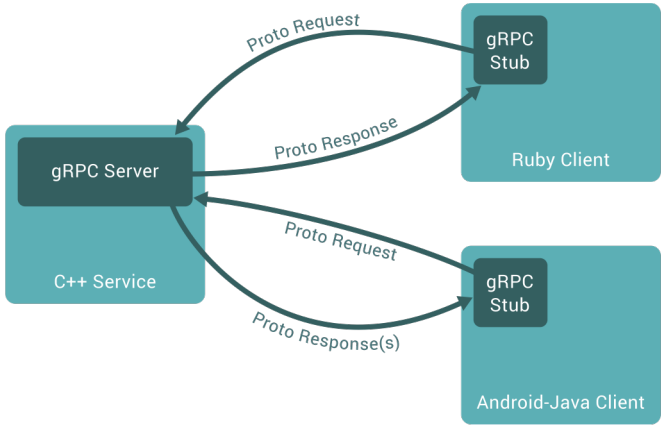


Common RPC implementation architecture

Application Integration

General-purpose RPC (gRPC)

- Originally invented by Google (**g**oogle **RPC**) to improve RPC
- Open sourced in 2015 (**g**eneral-purpose/**g**eneric **RPC**): https://grpc.io/
- Based on HTTP/2 and Protocol Buffers: easy definition in a .proto file
- Different service methods possible:
  - **unary RPCs:** single request / single response
  - **server streaming RPCs:** single request / streamed response of multiple messages
  - **client streaming RPCs:** streamed request of multiple messages / single response
  - **bidirectional streaming RPCs:** streamed request of multiple messages / streamed response of multiple messages



gRPC architecture visualisation: connecting heterogenous components

Source: https://grpc.io/docs/what-is-grpc/introduction/

https://grpc.io/
https://grpc.io/docs/what-is-grpc/core-concepts/

Application Integration

Simple Object Access Protocol (SOAP)

- First published in 1999 as an evolution of XML-RPC (1998), current version is 1.2 published in 2007 (https://www.w3.org/TR/soap12)
- SOAP is based on exchanging XML between client and server
  - Request contains **XML** (**SOAP envelope**) that describes which operation is to be called, contains parameters, etc.
  - Response contains XML (SOAP envelope) with the result of the operation
- SOAP is not explicitly bound to HTTP but it is most common
  - Does not make use of HTTP methods or status codes. E.g. all requests are sent as HTTP POST and return status code 200 even if the operation failed or did not return anything
- WSDL (web services description language) is used to describe SOAP services and helps tools to generate code

```
POST /calculator.asmx HTTP/1.1
Host: www.dneonline.com
Content-Type: text/xml; charset=utf-8

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-ir
 <soap:Body>
  <Add xmlns="http://tempuri.org/">
   <intA>6</intA>
   <intB>3</intB>
  </Add>
 </soap:Body>
</soap:Envelope>

HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/env
  <soap:Body>
    <AddResponse xmlns="http://tempuri.org/">
      <AddResult>9</AddResult>
    </AddResponse>
  </soap:Body>
</soap:Envelope>
```

Exemplary SOAP service request and response

Application Integration
Representational State Transfer (REST)

- REST is a resource based interface model
- Introduced in 2000 by Roy Fielding (Ph.D. dissertation) with original constrains:
  - **All important resources are identified by one resource identifier mechanism** (induces simple, visible, and reusable)
  - **Access methods have the same semantics for all resources** (induces visible, scalable, and available by enabling application of layered system, cacheable, and shared caches styles)
  - **Resources are manipulated through the exchange of representations** (induces simple, visible, reusable, cacheable, and evolvable via information hiding)
  - **Representations are exchanged via self-descriptive messages** (induces visible, scalable, and available by enabling application of layered system, cacheable, and shared caches styles, and evolvable via extensible communication);
  - **Hypertext as the engine of application state** (HATEOAS; induces simple, visible, reusable, and cacheable through data-oriented integration, evolvable via loose coupling, and adaptable though late binding of application transitions

Application Integration
Representational State Transfer (REST)

- The **Richardson-Maturity-Model (RMM)** describes a step by step approximation to REST-design where level 3 is a pre-condition of REST.
  - **Level 0**: Using HTTP as transport system for remote interactions
  - **Level 1**: Rather than making all requests to a singular service endpoint, we now start talking to individual resources (URIs)
  - **Level 2**: Using HTTP Verbs closely. E.g. using DELETE to delete a resource or GET to retrieve some data, etc. Relying on HTTP status codes to communicate (e.g. 404 Not Found).
  - **Level 3**: Hypermedia Controls (HATEOAS) for service discoverability

```
GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
Host: royalhope.nhs.uk


HTTP/1.1 200 OK
[various headers]

<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450">
     <link rel = "/linkrels/slot/book"
           uri = "/slots/1234"/>
  </slot>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650">
     <link rel = "/linkrels/slot/book"
           uri = "/slots/5678"/>
  </slot>
</openSlotList>
```

Martin Fowlers example for a RMM compliant RESTful API

https://martinfowler.com/articles/richardsonMaturityModel.html

Application Integration

Representational State Transfer (REST)

RMM Level 1: URIs

/doctors

/doctors/<doctorId>

/doctors/<doctorId>/slots

/doctors/<doctorId>/slots/<slotId>

….

RMM Level 2: (common) HTTP status codes

- Successful responses (200-299)
  - 200 OK
  - 201 Created
- Client error responses (400-499)
  - 400 Bad Request
  - 401 Unauthorized
  - 403 Forbidden
  - 404 Not Found
  - 405 Method Not allowed
- Server error responses (500-599)
  - 500 Internal Server Error

RMM Level 2: (common) HTTP Verbs

- GET (retrieve resources)
- POST (create resources)
- PUT (update resources)
- DELETE (delete resources)

Application Integration
Representational State Transfer (REST)

- Many APIs in the real world are called REST-APIs even though they do not meet the RMM criteria nor all of the original constraints. In the most cases those are more like JSON-RPC-APIs instead of REST-APIs.
  - Personal take: Level 0 to level 2 in combination with JSON payloads turned out to be a great approach in practice. It evolved from REST in the first place which is the reason it is often referred to as REST. However, “real” REST as defined by Fielding in 2000 and Richardson just did not fit for most use cases because of Level 3 (which was suitable at the time of invention but is not today).

```
GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
Host: royalhope.nhs.uk

HTTP/1.1 200 OK
[various headers]

{
  "openSlotList": [
    {
      "id": "1234",
      "doctor": "mjones",
      "start": "1400",
      "end": "1450",
    },
    {
      "id": "5678",
      "doctor": "mjones",
      "start": "1600",
      "end": "1650",
    }
  ]
}
```

Example of an API that might be referred to as RESTful but isn’t (not meeting Level 3 of RMM)

Application Integration
Graph Query Language (GraphQL)

- Introduced around 2012 at Facebook and open sourced in 2015 (https://spec.graphql.org/)
- Query language designed to build **client applications** by providing an intuitive and flexible syntax and system for **describing their data requirements and interactions**.
- Main design principles
  - **Product-centric**: driven by the requirements and thinking of frontend engineers
  - **Hierarchical:** queries are are structured hierarchically. The request is shaped just like its response
  - **Strong-typing:** a GraphQL service defines a type system. Tools can ensure an operation is syntactically correct and valid within that type system before execution (at build time)
  - **Client-specified response: a** GraphQL service defines available capabilities, but clients specify the exact data they need at field-level, ensuring precise, minimal responses
  - **Introspective:** A GraphQL service's type system can be queryable by the GraphQL language itself

Application Integration
Graph Query Language (GraphQL)

- GraphQL specification does not mention a transport protocol. In practice HTTP is common but:
  - the GraphQL query is sent via HTTP POST
  - GraphQL does not make use of HTTP methods or status codes like REST (by RMM). E.g. if a query does not return anything the HTTP status code is still 200 OK.
  - All requests are sent to a single service endpoint

```
POST /graphql HTTP/1.1
Host: royalhope.nhs.uk

{
  hero {
    name
    friends {
      name
    }
  }
}
```

```
HTTP/1.1 200 OK
[various headers]

{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
        {
          "name": "Leia Organa"
        }
      ]
    }
  }
}
```

https://graphql.org/learn/serving-over-http/

That‘s it ☺
…for now

Literature

- Richards, M., & Ford, N. (2020). Fundamentals of software architecture: an engineering approach. O'Reilly Media.
- Cesare Pautasso (2021). Software Architecture: visual lecture notes. Cesare Pautasso.
- Fielding, R. T., Taylor, R. N., Erenkrantz, J. R., Gorlick, M. M., Whitehead, J., Khare, R., & Oreizy, P. (2017, August). Reflections on the REST architectural style and" principled design of the modern web architecture"(impact paper award). In Proceedings of the 2017 11th joint meeting on foundations of software engineering (pp. 4-14).

Weblinks

accessed 01/2025

- https://www.baeldung.com/cs/layers-vs-tiers
- https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
- https://martinfowler.com/articles/richardsonMaturityModel.html
- https://grpc.io
- https://grpc.io/docs/what-is-grpc/introduction/
- https://grpc.io/docs/what-is-grpc/core-concepts/
- https://spec.graphql.org/
- https://graphql.org/learn/serving-over-http/