Technische Hochschule **Rosenheim**

Modul - IT Systems (IT)

Bachelor Programme AAI

10 - Lecture: REST and HTTP

Prof. Dr. Marcel Tilly

Faculty of Computer Science, Cloud Computing

Vers. 10.1.2022

Agenda

On the menu for today:

- Interaction models
- Distributed systems/ architectures
- REST, HTTP
- WebRequests in Python
- Message-oriented middleware/ MQTT

Bon appetit!

Learning Objectives

Students will be able to ...

- ... name different models of interaction
- ... explain the structure of URI/URLs
- ... can explain HTTP
- ... call a REST endpoint with different tools
- ... develop a REST service

Distributed System

- A **distributed system** (DS) is a system in which
  - hardware and software components,
  - which are located on networked computers,
  - communicate with each other and coordinate their actions
  - by exchanging *messages*.
- A **distributed application** is an application,
  - that uses a distributed system to solve an application problem.
  - It consists of various components that communicate with the components of the DS as well as the users.

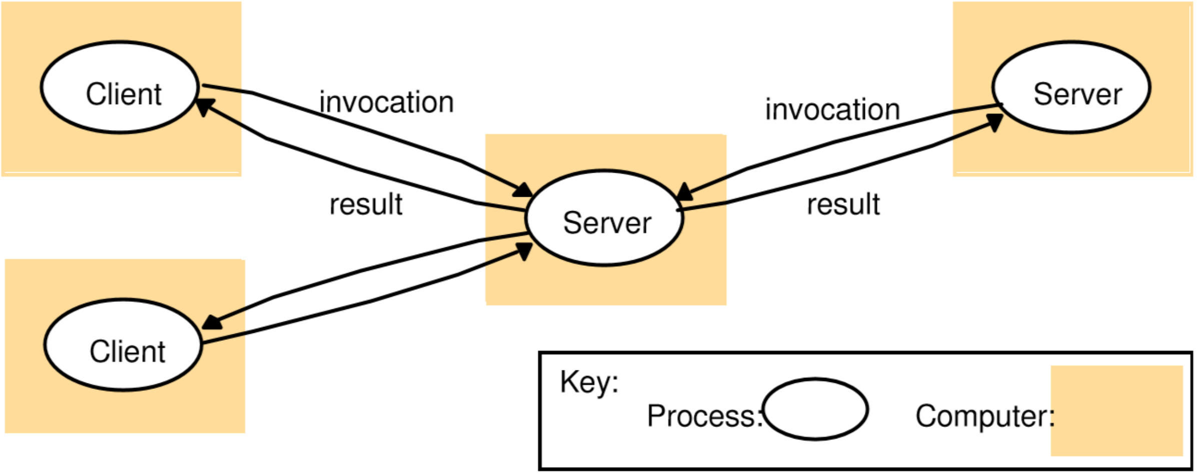
> Important aspect here: transparency, i.e. the distribution is hidden from the user.

Interaction models

- There are most different possibilities, how the software components on the computers interact with each other, in order to solve an application problem.
- In recent years, some basic models (architectures) have emerged for this purpose:
  - Client-Server (also Multi-Server)
  - Message-oriented architectures
  - Service-Oriented Architecture (SOA)
  - Multi-Tiered
  - Peer-to-Peer
  - Grid Computing
  - Cloud Computing
  - Microservices

Client - Server

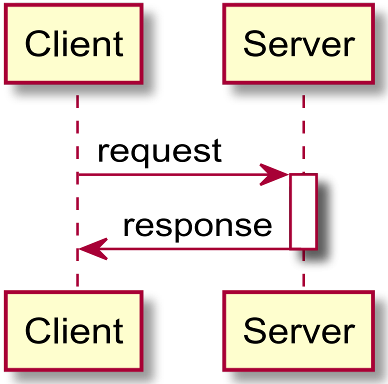
- Client requests a specific service
- Server receives the request, processes it and returns a result



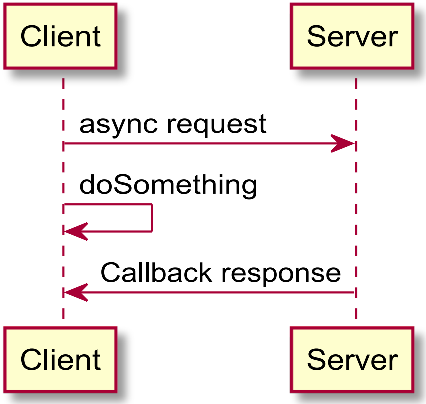
Client - Server

interaction can be either synchronous or asynchronous

synchronous



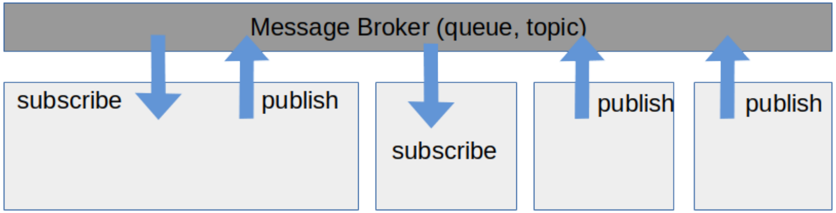
asynchronous



Message-oriented Model

The **MO** model supports three different communication protocols

- Message Passing (Direct communication between applications)
- Message Queuing (Indirect communication via a queue)
- Publish & Subscribe (Publisher provides messages to subscriber)



Advantages

- Asynchronous/synchronous communication
- Server/service does not have to be available immediately
- Loose coupling of server/clients
- Parallel processing of messages possible

Message-Oriented Middleware

- Message Oriented Middleware (MOM) refers to middleware that is based on asynchronous or synchronous communication, i.e. the transmission of messages.
- The format for the messages is not fixed: JSON, XML, ...
- MOM products
  - Websphere MQ from IBM (formerly MQSeries)
  - xmlBlaster from xmlBlaster.org
  - Apache ActiveMQ as JMS broker
  - RabbitMQ as AMQP broker
  - MQTT

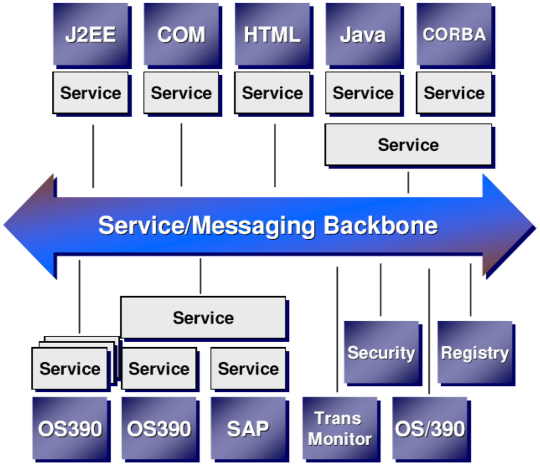
MQTT

- MQTT is a client-server protocol. Clients send messages with a topic to the server ("broker") after establishing a connection.
  - Topic: Classifies messages hierarchically
  - Example: 'Kitchen/Refrigerator/Temperature' or 'Car/Wheel/3/Air pressure'.
  - Clients can subscribe to these topics, and the server forwards the received messages to the appropriate subscribers.
- Messages always consist of a topic and the message content
- Messages are sent with a definable Quality of Service:
  - **at most once**: the message is sent once and may not arrive if the connection is interrupted.
  - **at least once**: the message is sent until reception is confirmed, and may arrive at the recipient multiple times
  - **exactly once**: this ensures that the message arrives exactly once even if the connection is interrupted

SOA

SOA = service-oriented architecture

- Consistent continuation of the client-server principle
- service
    - A software component with a formally described interface
    - Gives access to application logic or components
    - Communicates by request and response (synchronous and asynchronous)
    - Web services are an important basis of SOAs

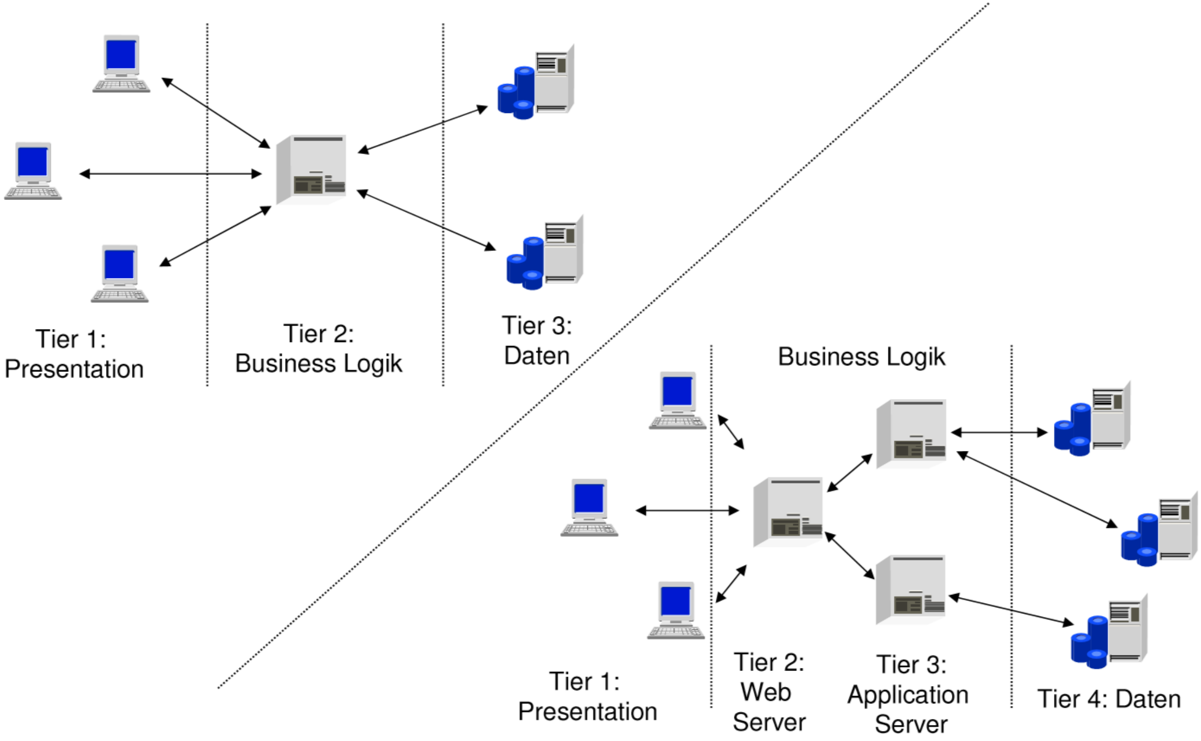


Multi-Tier Architectures

Distributed *(web)* applications today are often developed as **multi-tier applications**.

- Each "tier" (layer, level) has its own task
- Advantages
  - less complexity of the individual parts
  - distribution of implementation tasks
  - flexibility in the distribution of the individual tasks (thinclient)
  - easier maintainability (no client software, exchange of versions)
  - scalability, security
  - In principle a continuation of the client server principle

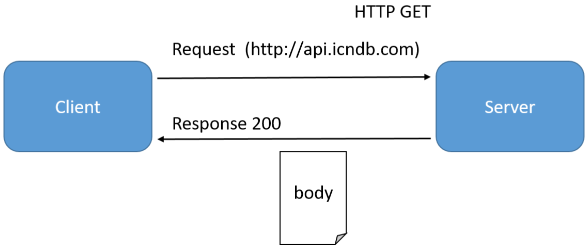
3-tier / 4-tier


Tier 1:
Presentation
Tier 2:
Business Logik
Tier 3:
Daten
Business Logik
Tier 1:
Presentation
Tier 2:
Web
Server
Tier 3:
Application
Server
Tier 4: Daten

A word about REST

REST = REpresentational State Transfer

- REST stands for REpresentational State Transfer, API for Application Programming Interface.
- Programming interface based on the paradigms and behavior of the World Wide Web (WWW).
- Defines an approach for communication between client and server in networks
- Roy Fielding introduced the concept in 2000 in his dissertation, "Architectural Styles and the Design of Network-based Software Architectures"
- However, implementations of the architecture characterized as "RESTful" use standardized methods, such as HTTP/S, URI, JSON or XML



REST

REST does not specify in detail how services are implemented. Rather, the approach assumes the following six architectural principles ("constraints"):

- **Client-Server Model**: REST requires a client-server model, i.e., wants to see the user interface separated from the data store.
  - On the one hand, this should make it easier to port clients to different platforms; on the other hand, simplified server components should scale particularly well.
- **Statelessness**: Client and server must communicate with each other statelessly.
  - This means that every request from a client contains all the information that a server needs; servers themselves cannot access any stored context. This constraint thus improves visibility, reliability and scalability. For this, however, REST accepts disadvantages in terms of network performance; moreover, servers lose control over consistent client app behavior.

REST cont'd

- **Caching**: To improve network efficiency, clients can also store responses sent by the server and reuse them later for similar requests.
  - The information must be marked as "cacheable" or "non-cacheable" accordingly. The advantages of more responsive applications with higher efficiency and scalability are bought with the risk that clients fall back on outdated data from the cache.
- **Unified Interface**: The components of REST-compliant services use a unified, common interface that is decoupled from the implemented service.
  - The goal of all this is simplified architecture and increased visibility of interactions. In exchange, one accepts poorer efficiency when information is put into a standardized format - and not customized for the needs of specific applications.

REST cont'D

- **Layered System**: REST relies on multi-layered, hierarchical systems ("Layered System") - each component can only see directly adjacent layers.
    - Thus, for example, legacy applications can be encapsulated. Intermediaries acting as load balancers can also improve scalability. The disadvantages of this constraint are additional overhead and increased latencies.
- **Code-On-Demand**: This constraint requires that the functions of clients can be extended via reloadable and executable program parts
    - for example in the form of applets or scripts. However, as an optional constraint, this condition can be disabled in certain contexts.

Implementation via HTTP

- In practice, the *REST* paradigm is preferably implemented using HTTP/S.
- Services are addressed via **URL/URI**.
- The HTTP methods (GET, POST, PUT,...) specify which operation a service should perform.

Since the World Wide Web already provides the necessary infrastructure for REST, one can already make first attempts with appropriate interfaces via browser. A possible starting point for this is the Fake Online REST API for Testing and Prototyping available at http://jsonplaceholder.typicode.com/.

URL/URI

- **URL = Uniform Resource Identifier** (RFC: 3986): A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource.
- **URL = Uniform Resource Locator**: The term "Uniform Resource Locator" (URL) refers to the subset of URIs that, in addition to identifying a resource, provide a means of locating the resource by describing its primary access mechanism (e.g., its network "location")

```
         foo://example.com:8042/over/there?name=ferret#nose
         \__/ \_______________/\_________/ \_________/ \__/
          |         |              |             |       |
       scheme   authority         path         query  fragment
          |   _____________________|__
         / \ /                        \
         urn:example:animal:ferret:nose

authority = [ userinfo "@" ] host [ ":" port ]
```

URI Examples

Examples:

```
ftp://ftp.is.co.za/rfc/rfc1808.txt
http://www.ietf.org/rfc/rfc2396.txt
http://localhost:8000/assets/10-vorlesung/slides.html#19

mailto:John.Doe@example.com
telnet://192.0.2.16:80/
```

Exotics

```
news:comp.infosystems.www.servers.unix
tel:+1-816-555-1212
urn:oasis:names:specification:docbook:dtd:xml:4.1.2
```

- URN : The term "Uniform Resource Name" (URN) has been used historically to refer to both URIs under the "urn" scheme [RFC2141], which are required to remain globally unique and and persistent even when the resource ceases to exist or becomes unavailable, and to any other URI with the properties of a name.

HTTP(S)

RFC

- The **Hypertext Transfer Protocol** (HTTP) is what users of a web browser come into contact with whenever they load the web pages of a remote server.
- Published by the Internet Engineering Task Force (IETF) in 2014 (RFC 7231), HTTP1.1.
- Characterizes a stateless protocol that resides at the application level and is suitable for distributed, collaborative hypertext information systems

HTTP(s)

Client Request

```
GET /docs/index.html HTTP/1.1
Host: www.nowhere123.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
(blank line)
body
```

- Line 1: *Request Line* `request-method-name request-URI HTTP-version`
- line 2-5: *Request Header* `request-header-name: request-header-value1, request-header-value2, ...`
- Line 7: *Body*

HTTP(s)

Server response

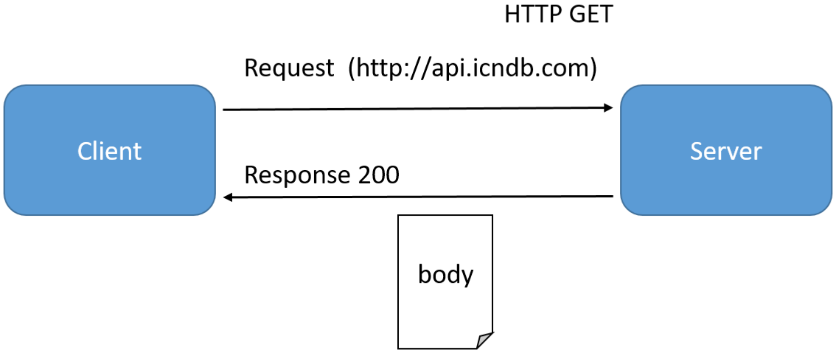
```
HTTP/1.1 200 OK
Date: Sun, 18 Oct 2009 08:56:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Sat, 20 Nov 2004 07:16:26 GMT
ETag: "10000000565a5-2c-3e94b66c2e680"
Accept-Ranges: bytes
Content-Length: 44
Connection: close
Content-Type: text/html
X-Pad: avoid browser bug

<html><body><h1>It works!</h1></body></html>
```

- Line 1: *status line* `HTTP-version status-code reason-phrase`
- Lines 2-10: *Response Headers* `response-header-name: response-header-value1, response-header-value2, ...`
- Line 12: *Body*

Making requests

REST requires the client to submit a request so that the server can respond with a response. A *Request* usually consists of:

- **HTTP verb** (standard operation): Defines the semantic of the *Request*.
- **Header** : Allows the client to send additional information (format etc.)
- **Path**: Path to the resource (URL/URI)
- **Body** (optional): A message to be sent



HTTP Verb

There are several HTTP verbs (=commands) in the RFC.

The 4 most important ones are (compare CRUD):

- **GET** - Requests a resource (by URI).
- **POST** - Updates a resource
- **PUT** - Creates a resource
- **DELETE** - Deletes a resource

How to execute this now?

Telnet and more...

- Our *Swiss Knife* Telnet can do that of course:

```
$ telnet 127.0.0.1 8000
Connecting To 127.0.0.1...
GET /index.html HTTP/1.0
```

- Telerik Fiddler: Web Debuggin Tool
- Postman: API Development Environment

But it can be better!

cURL = Client for URLs

… is a program library and a command line program for transferring files in computer networks

- cURL is available on almost all OS
- supports the HTTP commands: Option -X [GET|POST|PUT|DELETE] -d : Transfer data in HTTP body --data "@path_of_file": transfer the contents of a file

GET

```
curl -X GET https://jsonplaceholder.typicode.com/todos/1
```

POST

```
curl -d '{"key1": "value1", "key2": "value2"}'
     -H "Content-Type: application/json"
     -X POST http://localhost:3000/data
```

HTTP Verb

There are several HTTP verbs (=commands) (aka CRUD):

- **GET** - Requests a resource (by URI)
- **POST** - Updates a resource
- **PUT** - Creates a resource
- **DELETE** - Deletes a resource

Test the REST

```
curl -X GET http://jsonplaceholder.typicode.com/posts/1
```

```
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident",
  "body": "quia et suscipit\nsuscipit recusandae"
}
```

What is JSON?

- JSON = *JavaScript Object Notation*.
- Specification is available here: JSON
- JSON is a lightweight format for storing and transmitting data
- JSON is an open "quasi-standard" (used very often with RESTful services)
- JSON is *almost* "self-writing" and easy to understand
    - No schema validation, (see XML and XMLSchema).
    - but there is JSON Schema

```
{
    "employees":[
        {"firstName": "John", "lastName": "Doe"},
        {"firstName": "Anna", "lastName": "Smith"},
        {"firstName": "Peter", "lastName": "Jones"}
    ]
}
```

Task#1: Formats

Do you know alternative formats for data transmission?

Let's collect!

XML

XML stands for eXtensible Markup Language

- XML is a text-based data format (https://www.w3.org/XML/)
- XML files can be opened and edited in an editor
- XML consists like HTML of so-called tags, which stand between angle brackets '<' '>'.
- With XML you can define your own tags via XML Schema (which is also XML!).
    - Only how a tag must look like is defined, but not what it means.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
    <note>
        <to>Blubb</to>
        <of>Bla</of>
        <headline>Thing</headline>
    </note>
```

> A good summary: https://www.w3.org/XML/1999/XML-in-10-points

Task#2: XML

Which XML languages do you know?

HTML

- HyperText Markup Language (https://www.w3.org/TR/html52/)
- HTML is a language for describing web pages
  - meta information
  - headers, texts, table, pages, images, ...
  - Links (references) to other URLs
- XHTML is the extension of HTML, which is XML *compliant*.
- All about HTML at: https://wiki.selfhtml.org/

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Page description (appears in the browser title bar)</title>.
  </head>
  <body>
    <p>This text will be displayed in the browser window.</p>
  </body>
</html>
```

JSON in Python

- *JSON is just super for storing and transmitting.*

JSON is used to serialization and deserialization.

```
{
    "age": 75,
    "company_name" : "Bell System",
    "emails" : [
        {
        "email" : "alex@bell.com",
        "type" : "work"
        }
    ],
    "job_title" : "CEO",
    "name" : "Alexander Graham Bell"
}
```

Now how is this converted to a JSON format in Python?

There are tools...

... for each framework: Newtonsoft (.NET), GSON (Java)

So, how does it look like with Python:

```python
import json

my_dictionary = '{"name": "Alexander Graham Bell", "job_title": "CEO",
                  "company_name": "Bell System"}'

txt = json.loads(my_dictionary)

out = json.dumps(txt)

print(out)
```

Task: ICNDB

The Internet Chuck Norris Database provides Chuck Norris jokes.

Can you formulate a `curl` expression that retrieves a random joke?

1. use the `GET` command to do this
2. use the `/jokes/random` endpoint

Solution: ICNDB

The Internet Chuck Norris Database provides Chuck Norris jokes.

Can you formulate a `curl` expression that retrieves a random joke?

1. use the `GET` command to do this
2. use the `/jokes/random` endpoint

```
$curl -X GET http://api.icndb.com/jokes/random
```

```
{ "type": "success", "value":
 { "id": 316,
   "joke": "In the medical community, death is referred to
            as Chuck Norris Disease",
   { "categories": []
 }
}
```

WebRequest in Python

How to implement an HTTP request in Python?

- Use `requests` module: `pip3 install requests`
- Use `requests.get(<url>)` function to call a server
- Use result to read the response (*request*)

HTTPRequest in Python

Get a joke from ICNDB:

```python
import requests

r = requests.get('https://api.icndb.com/jokes/random')

print(r)
print(r.text)

data = json.loads(r.text)

print(data)
```

Summary

Lessons learned today:

- Distributed Systems
- architectures
- HTTP, REST, JSON
- WebRequests with Java
- MQTT