ISEngineering

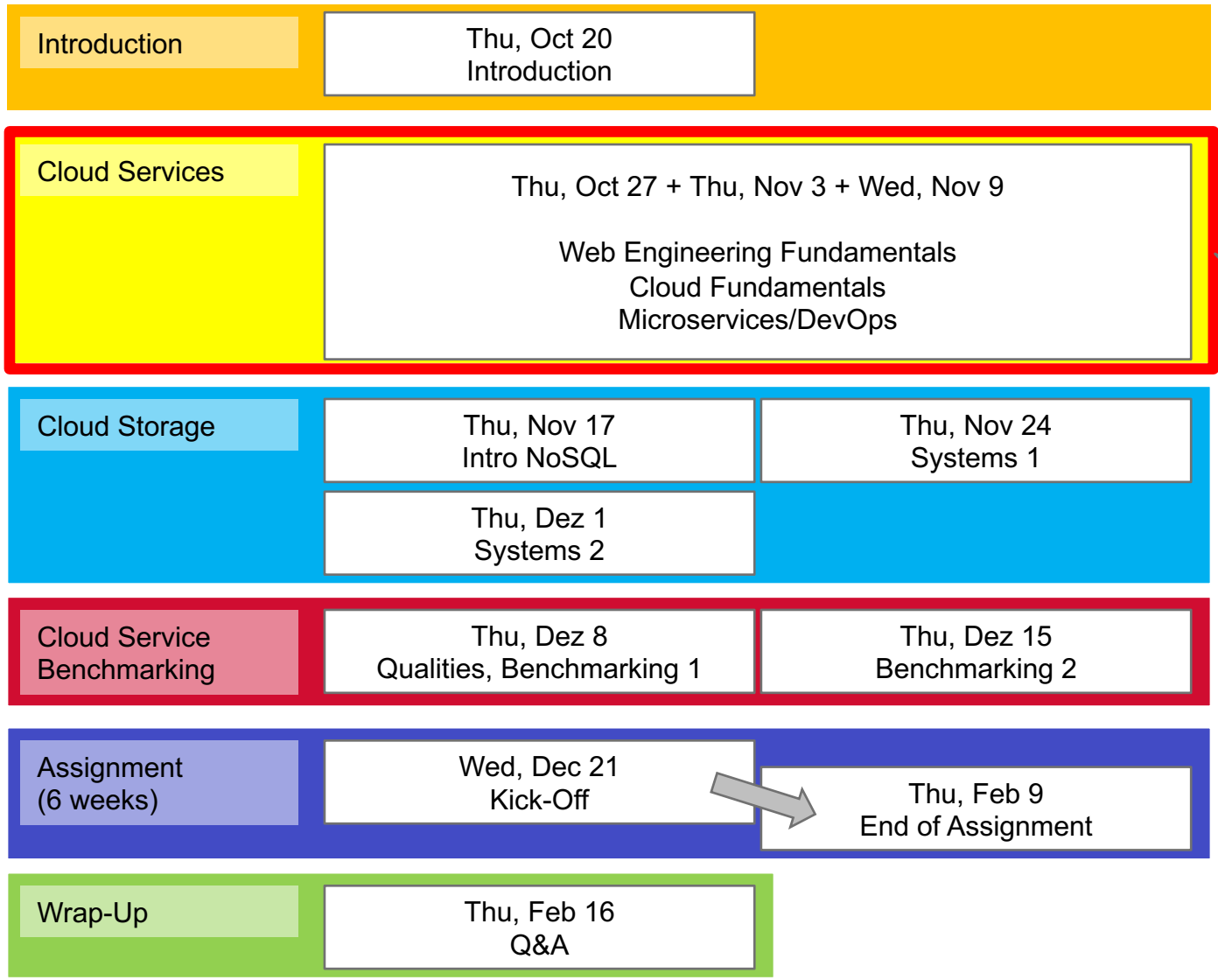
Technische Universität Berlin

Enterprise Computing – Cloud Services

Stefan Tai, Marco Peise

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Today



Revised agenda, based on kahoot results and feedback

ISEngineering
Wirtschaftsinformatik – Information Systems Engineering

Agenda

1. Web Engineering Fundamentals
    a. HTTP
    b. REST
    c. SOAP
    d. WSDL
    e. WS-*
2. Cloud Fundamentals
    a. Sample Cloud Services: AWS
    b. Design Principles
3. Microservices
    a. DevOps, CI/CD
    b. Microservices and Server-less Architectures

This is a recap/summary of material well-known to some, but not all participants this year.

Technische Universität Berlin

Recap: HTTP (Hyper Text Transfer Protocol)

Hypertext Transfer Protocol (HTTP)

HTTP is a request-response, application layer protocol and a standard for client-server communication

In HTTP e.g. a web browser acts as a client submitting HTTP requests to a server;
the server which stores content (or *resources*) or generates content sends messages back to the client in response

HTTP Requests and Responses

Typically, an HTTP client establishes a TCP connection to a particular port on a host (port 80 by default)

An HTTP server *listening* on that port waits for the client to send a request message

Upon receiving the request, the server sends back a *status* line, such as "HTTP/1.1 200 OK", and a message of its own the body of which is perhaps the requested resource, an error message or some other information.

HTTP Status Codes Categories

Technische Universität Berlin

1xx: Informational

Request received, continuing process

2xx: Success

The action was successfully received, understood, and accepted

3xx: Redirection

Further action must be taken in order to complete the request

4xx: Client Error

The request contains bad syntax or cannot be fulfilled

5xx: Server Error

The server failed to fulfill an apparently valid request

Resource Identification

Technische Universität Berlin

Resources to be accessed by HTTP are identified using

- Uniform Resource Identifiers (URIs) — or, more specifically,
- Uniform Resource Locators (URLs) — using the http: or https: URI schemes.

Reminder: URIs

Technische Universität Berlin

URI: A reference identifying an abstract or physical resource

- Form: <scheme>://<authority><path>?<query>
- Can be a URL, URN, or both

URL: The subset of URIs that identifies resources by their primary access mechanism (e.g. network location)

- Form = transport://user:password@host:port/path[?search][#fragmentid]
- Is a physical address of a resource

URN: The subset of URIs that uniquely identify a resource independent of its primary storage location

- Identifier remains stable even is the resource is moved to a different physical location
- Is a logical address of a resource

URI Schemes

URIs use URI schemes for specification

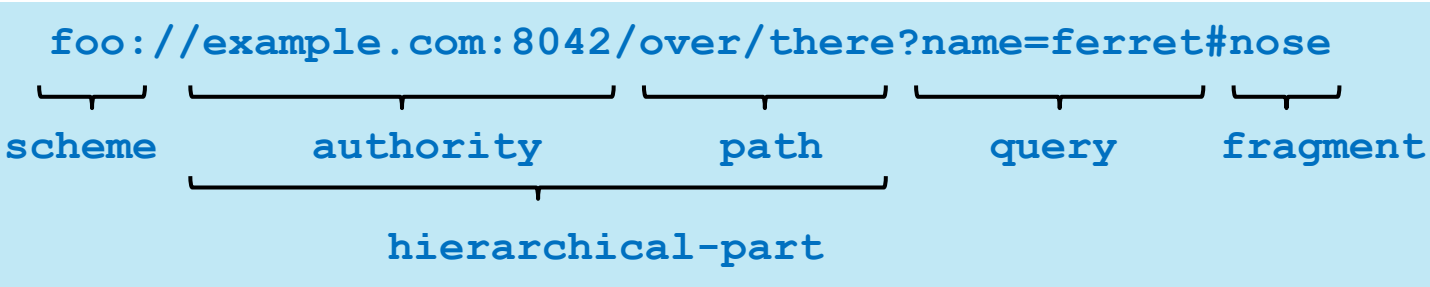
- […] scheme's specification may further restrict the syntax and semantics of identifiers using that scheme

Many URI schemes are hierarchical

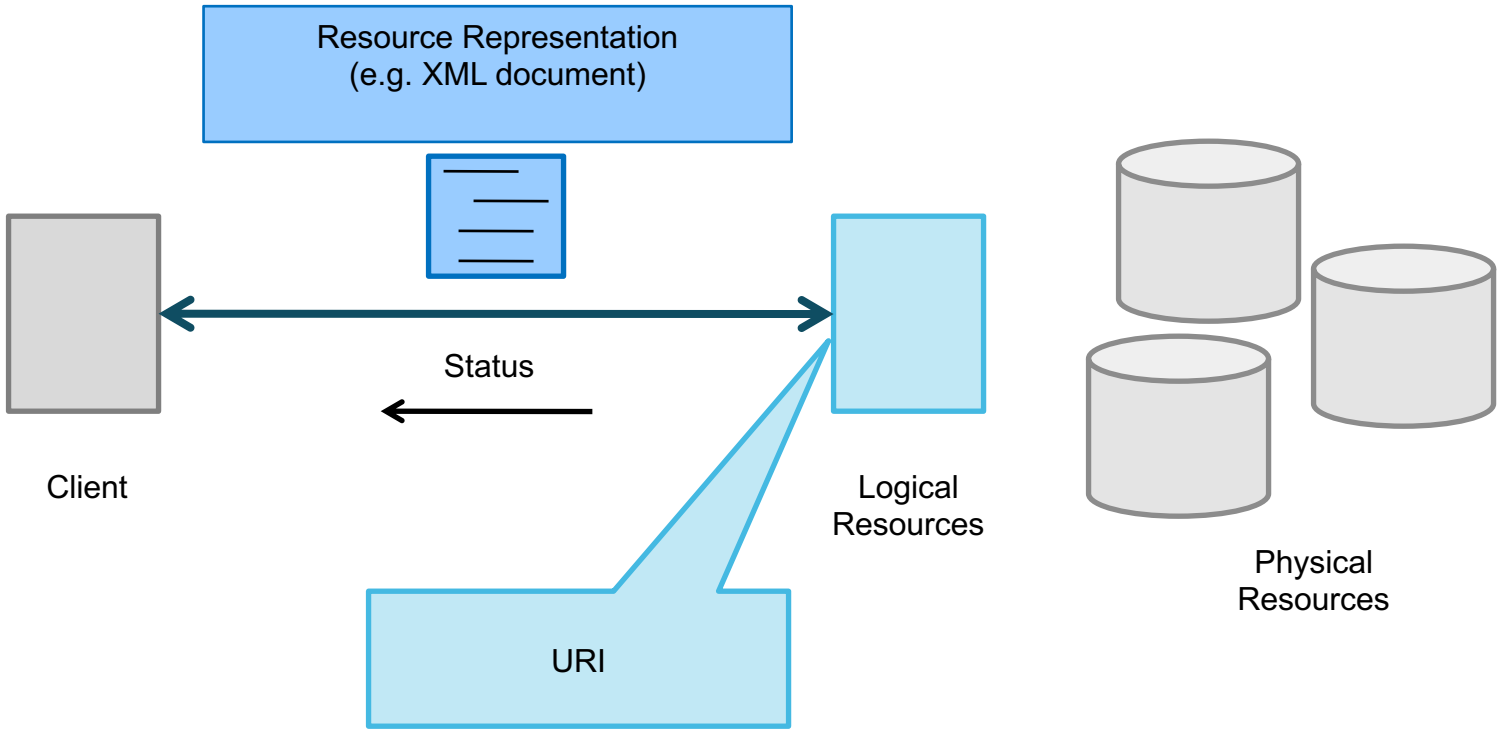
- It is then possible to use relative URIs such as in a href="../"
- The slash character is not just a character, in URIs it has semantics

```
URI = scheme ":" hierarchical-part [ "?" query ] [ "#" fragment ]
```

- Hierarchical part usually contains "//", authority and (optional) path



Resource Architecture



Client Tier

Web Server

HTTP Request Messages

HTTP request messages consists of the following:

Request line, such as

```
GET /images/logo.gif HTTP/1.1
```

which requests a resource called `/images/logo.gif` from the server

Headers, such as `Accept-Language: en`

An empty line

An optional message body

Request Methods

HTTP defines eight methods ("verbs") indicating the desired action to be performed on the identified resource

What the resource represents, whether it is pre-existing data or data that is generated dynamically depends on the implementation of the server. Often, the resource corresponds to a file or the output of an executable residing on the server.

- HEAD
- **GET**
- **POST**
- **PUT**
- **DELETE**
- TRACE
- OPTIONS
- CONNECT

CRUD Resource Lifecycle

A resource is Created with POST

…Read with GET

…Updated via PUT

…and Deleted using DELETE

POST semantics

POST creates a new resource

➔ The server decides on that resource's URI

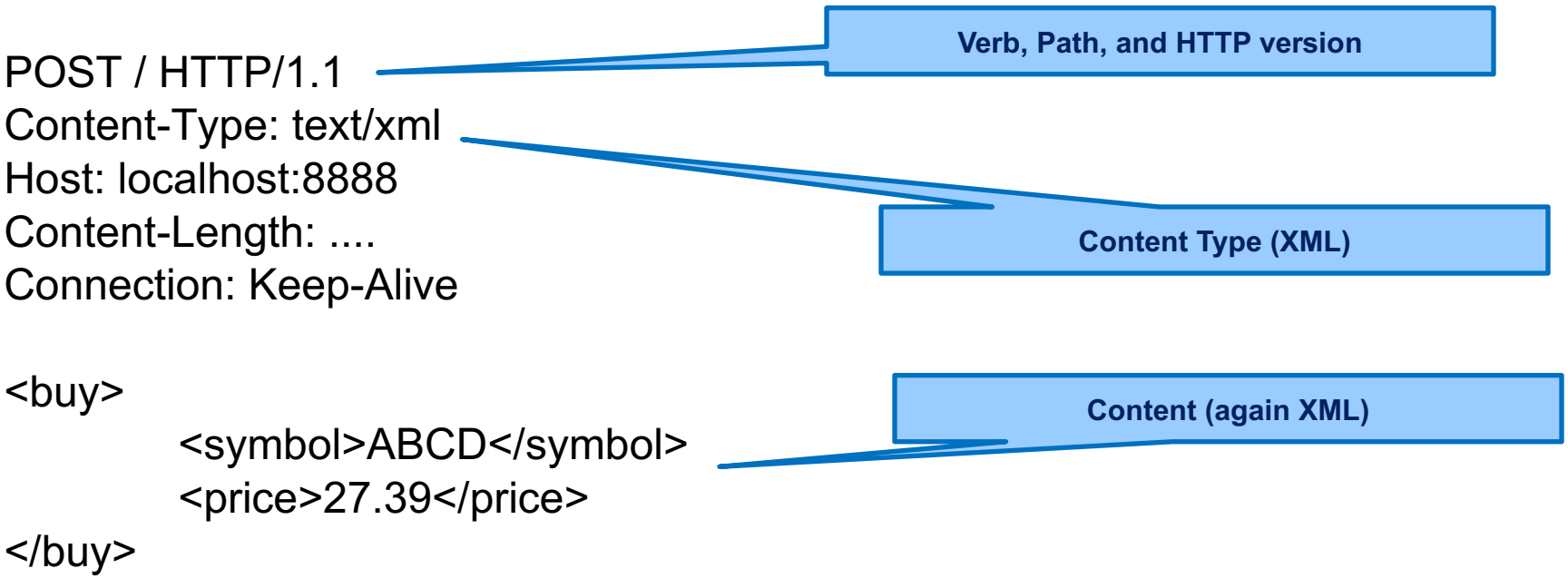
Common human Web example: Posting to Web log

➔ Server decides URI of posting and any comments made on that post

Programmatic Web example:

Creating a new employee record and subsequently adding to it

POST request



POST response

201 CREATED
Location: /orders/halvards/ABCD/2007-07-080-13:50:53

When POST goes wrong

We may get 4xx or 5xx errors

➔ Client versus server problem

We turn to GET!

Find out the resource states first

Then figure out how to make forward or backward progress

Then solve the problem

➔ May involve POSTing again

➔ May involve a PUT to rectify server-side resources in-place

Safety and Idempotency

A safe operation is one which changes no state at all

An idempotent operation is one which updates state in an absolute way, and the result of executing the operation more than once is the same as executing it once
E.g. x = 4 rather than x += 2

Web-friendly systems scale because of safety

➔Caching!

… and idempotent behavior benefits fault tolerance

➔ Just retry in failure cases

GET semantics

GET retrieves the representation of a resource

Should be *safe*

- *Safe* means that the request is intended only for information retrieval and should *not* change the state of the server
- In other words, GET should not have any side effects (beyond relatively harmless effects such as logging, caching, the serving of banner advertisements or incrementing a web counter)

GET example

```
GET /employees?id=1234 HTTP/1.1
Accept: text/xml
Host: crm.example.com
```

When GET goes wrong

Simple!

- Just 404 – the resource is no longer available

Are you sure?

- GET again!

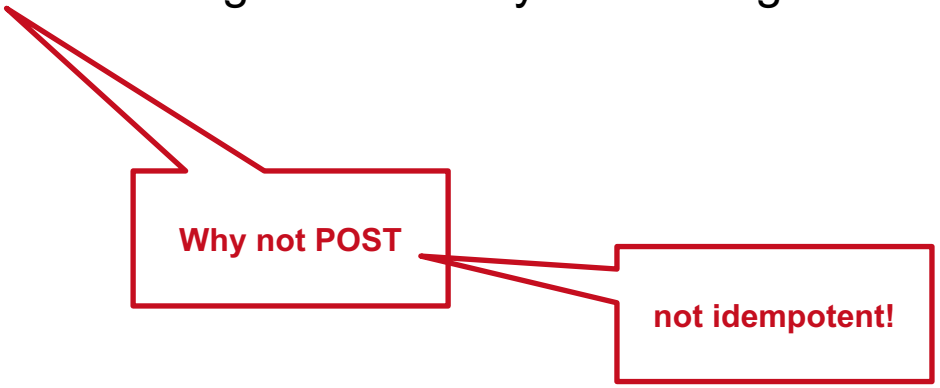
GET is safe (and hence also idempotent)

- Consider crash recovery scenarios

PUT semantics

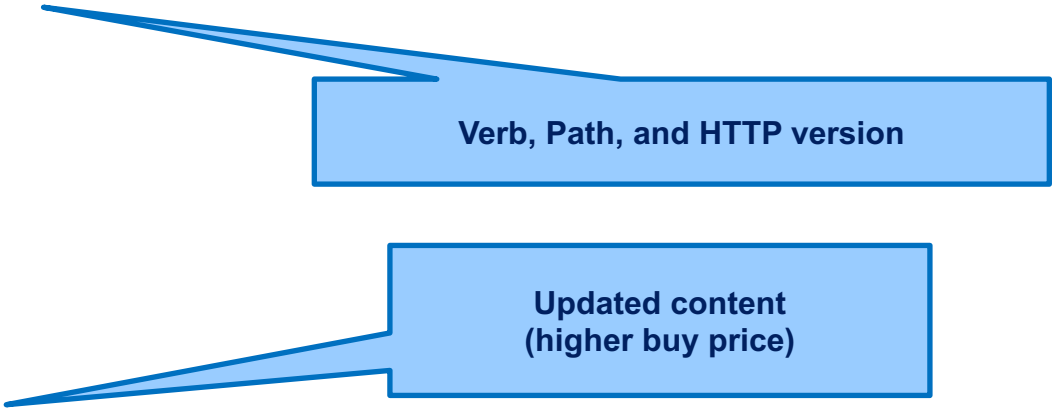
PUT creates a new resource but the *client* decides on the URI, providing the server logic allows it

Also used to update existing resources by overwriting them in-place



PUT request

```
PUT /orders/halvards/ABCD/2007-07-080-13:50:53 HTTP/1.1
Content-Type: text/xml
Host: localhost:8888
Content-Length: ....
Connection: Keep-Alive

<buy>
        <symbol>ABCD</symbol>
        <price>44.42</price>
</buy>
```



PUT response

200 OK
Location: /orders/halvards/ABCD/2007-07-080-13:50:53
Content-Type: text/xml
<nyse:priceUpdated .../>

When PUT goes wrong

Technische Universität Berlin

If we get 5xx error, or some 4xx errors simply PUT again!

This is possible since PUT is idempotent

If we get errors indicating incompatible states (409, 417) then do some forward/backward compensating work

And maybe PUT again

DELETE semantics

Stop the resource from being accessible

➔ Logical delete, not necessarily physical

This is important for decoupling implementation details from resources

Request

```
DELETE /user/halvards HTTP 1.1
Host: example.org
```

Response

```
200 OK
Content-Type: application/xml
<admin:userDeleted>
        halvards
</admin:userDeleted>
```

When DELETE goes wrong

DELETE again!

- Delete is idempotent
- DELETE once, DELETE 10 times has the same effect: one deletion

Some 4xx responses indicate that deletion is not possible

- The state of the resource is not compatible
- Try forward/backward compensation instead

Complete List Of Status Codes

```
  "100" ; Section 10.1.1: Continue
| "101" ; Section 10.1.2: Switching Protocols

| "200" ; Section 10.2.1: OK
| "201" ; Section 10.2.2: Created
| "202" ; Section 10.2.3: Accepted
| "203" ; Section 10.2.4: Non-Authoritative
                          Information
| "204" ; Section 10.2.5: No Content
| "205" ; Section 10.2.6: Reset Content
| "206" ; Section 10.2.7: Partial Content

| "300" ; Section 10.3.1: Multiple Choices
| "301" ; Section 10.3.2: Moved Permanently
| "302" ; Section 10.3.3: Found
| "303" ; Section 10.3.4: See Other
| "304" ; Section 10.3.5: Not Modified
| "305" ; Section 10.3.6: Use Proxy
| "307" ; Section 10.3.8: Temporary Redirect

| "400" ; Section 10.4.1: Bad Request
| "401" ; Section 10.4.2: Unauthorized
| "402" ; Section 10.4.3: Payment Required
| "403" ; Section 10.4.4: Forbidden
| "404" ; Section 10.4.5: Not Found
| "405" ; Section 10.4.6: Method Not Allowed
| "406" ; Section 10.4.7: Not Acceptable
| "407" ; Section 10.4.8: Proxy Authentication
                          Required
| "408" ; Section 10.4.9: Request Time-out
| "409" ; Section 10.4.10: Conflict
| "410" ; Section 10.4.11: Gone
| "411" ; Section 10.4.12: Length Required
| "412" ; Section 10.4.13: Precondition Failed
| "413" ; Section 10.4.14: Request Entity Too Large
| "414" ; Section 10.4.15: Request-URI Too Large
| "415" ; Section 10.4.16: Unsupported Media Type
| "416" ; Section 10.4.17: Requested range not
                           satisfiable
| "417" ; Section 10.4.18: Expectation Failed

| "500" ; Section 10.5.1: Internal Server Error
| "501" ; Section 10.5.2: Not Implemented
| "502" ; Section 10.5.3: Bad Gateway
| "503" ; Section 10.5.4: Service Unavailable
| "504" ; Section 10.5.5: Gateway Time-out
| "505" ; Section 10.5.6: HTTP Version not supported
```

There‘s more to HTTP

Technische Universität Berlin

Especially, runtime aspects, including:

- Persistent Connections
- Chunked Transfer
- Content Negotiation
- Caching
- Cookies

…

Technische Universität Berlin

Recap: REST

The Architectural Principle of the Web

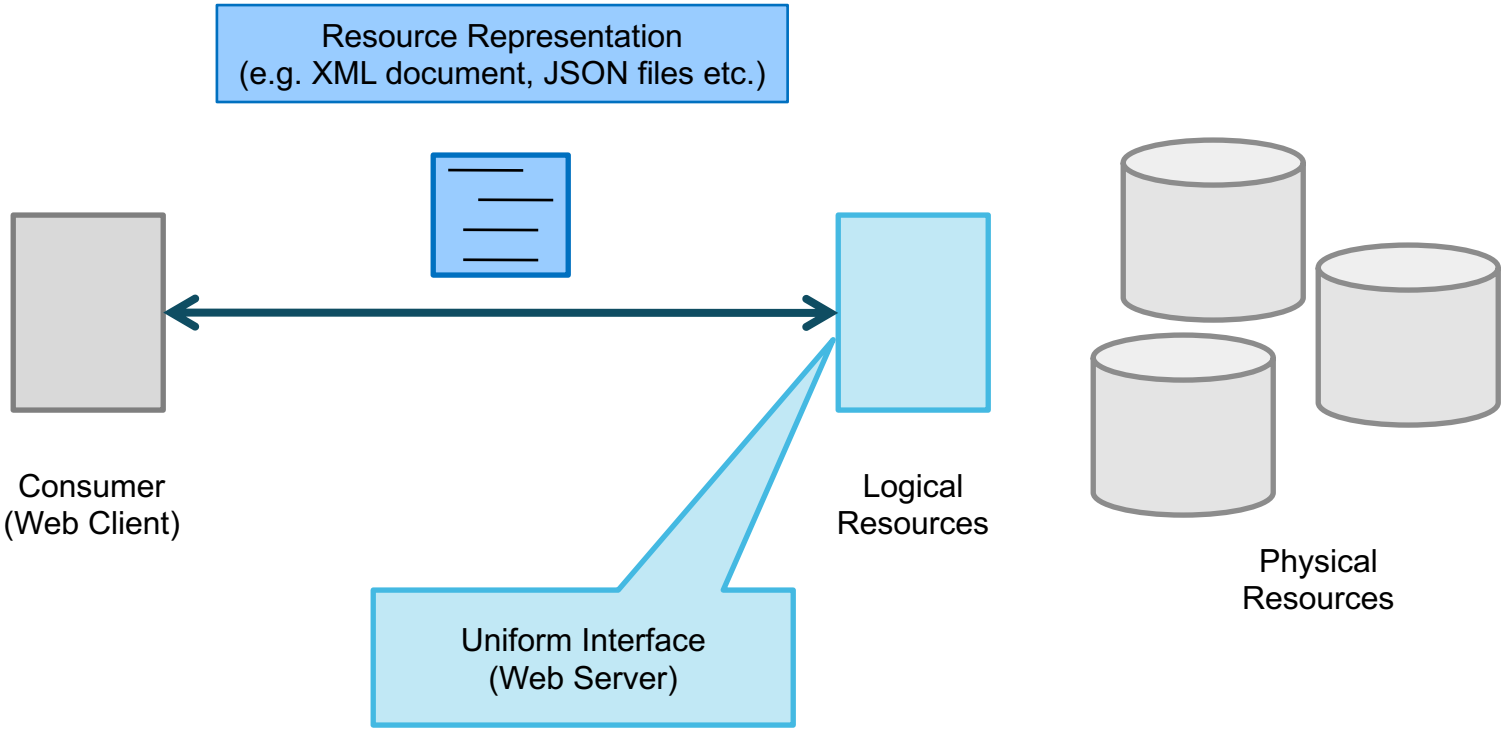
Back to Basics: Web Fundamentals

The Web is a distributed hypermedia model; it does not try to hide the distribution from you

The Web emphasizes scalability:

- Loose coupling: Growth of the Web in one place is not impacted by changes in other places
- Uniform interface: HTTP
  - Standard interface for all actors on the Web
  - Replication and caching is baked into this model

Focus on Resources



The Acronym REpresentational State Transfer (REST)

First articulated in Roy Fielding's PhD

- Co-author of HTTP
- Early work in 1994/95

REST is the "abstract architecture" of the Web

Goals:

- Speed
- Scalability
- Simplicity
- Data independence

"After all the web works."

"The modern Web architecture emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems." [Fielding 2002]

What is Architecture?

Technische Universität Berlin

Architecture is constraint-based design

- Design without constraints probably is art

Constraints can be derived from a wide variety of sources

- Technical infrastructure (current landscape and expected developments)
- Business considerations (current landscape and expected developments)
- Time horizon (short-term vs. long-term requirements)
- Existing architecture
- Scalability
- Performance (based on performance requirements and definitions)
- Cost (development, deployment, maintenance)

Architectural Styles

Technische Universität Berlin

Architectural Style: General principles informing the creation of an architecture
Architecture: Designing a solution to a problem according to given constraints

Architectural styles *inform* and *guide* the creation of architectures

Architecture: Louvre
Architectural Style: Baroque

Architecture: Villa Savoye
Architectural Style: International Style

The REST Architectural Style

A set of constraints that inform an architecture:

1. *Resource Identification*
2. *Uniform Interface*
3. *Self-Describing Messages*
4. *Hypermedia as the Engine of Application State*
5. *Stateless Interactions*

Claims: scalability, mashup-ability, usability, accessibility

Technische Universität Berlin

1. Resource Identification

2. Uniform Interface
3. Self-Describing Messages
4. Hypermedia as the Engine of Application State (HATEOAS)
5. Stateless Interactions

Resource Identification

Name everything that you want to talk about

"Everything" in this case should refer to *anything*, *i.e., a* "resource" can be whatever might be identified

- *Products* in an online shop
- *Categories* that are used for grouping products
- *Customers* that are expected to buy products
- *Shopping carts* where customers collect products
- Not necessarily accessible via Internet, e.g., human beings, corporations
- Abstract concepts

Identifying Resources on the Web

Technische Universität Berlin

Resources are identified by

Uniform Resource Identifiers (URI)

URIs are human-readable universal identifiers

- Many identification schemes are not human-readable (binary or hex strings)
- Many RPC-based systems do not have universally identified objects

Making every “thing” a “*universally unique identified resource*” is important

- It removes the necessity to *scope* non-universal identifiers
- It allows for talking about all things in exactly the same way

Query Information

Technische Universität Berlin

Query components specify additional information

- It is non-hierarchical information further identifying the resource
- In most cases, it can be regarded as input to the resource

Query components often influence caching

- Successful GET/ HEAD requests may be cached
- Only cache query string URIs when explicitly requested (Expires/Cache-Control)

Processing URIs

Technische Universität Berlin

Processing URIs is not as trivial as it may seem

- Escaping and normalization rules are non-trivial
- Many implementations are broken
- Even more complicated when processing an *Internationalized Resource Identifier (IRI)*

URIs are not just strings

- URIs are strings with a considerable set of rules attached to them
- Implementing all these rules is non-trivial
- Implementing all these rules is crucial
- Application development environments provide functions for URI handling

1. Resource Identification

2. Uniform Interface

3. Self-Describing Messages
4. Hypermedia as the Engine of Application State (HATEOAS)
5. Stateless Interactions

Uniform Interface

The same small set of operations (*verbs*) applies to a large set of resources (*nouns*)

Verbs are **universal** and not invented on a per-application base

If many applications need new verbs, the uniform interface can be extended
Natural language works in the same way (new verbs rarely enter language)

How RESTful Applications Talk on the Web

Hypertext Transfer Protocol (HTTP)

- HTTP defines a small set of methods for acting on URI-identified resources

Mis-using HTTP turns applications into non-RESTful applications

- They lose the capability to be used just by adhering to REST principles
- It's a bad sign when you think you need an interface description language

Extending HTTP turns applications into more specialized RESTful applications

- May be appropriate when more operations are required
- Seriously reduces the number of potential clients

Recall: Idempotency and Safety

An ***idempotent*** operation is one which updates state in an absolute way, and the result of executing the operation more than once is the same as executing it once

- Can be repeated without side-effects: if something goes wrong (server down, server internal error), the request can be simply replayed until the server is back up again.
- Arithmetically safe: $41 \times 0 \times 0 \times 0 \times 0 \ldots$

A ***safe*** operation is one which changes no state at all

- Can be ignored or repeated without side-effects
- Arithmetically safe: $41 \times 1 \times 1 \times 1 \times 1 \ldots$
- Can be cached

In practice, without side-effects means without relevant side-effects

Idempotency and Safety in HTTP

GET and HEAD are *safe operations*

```
GET /book
```

PUT and DELETE are *idempotent operations*

```
PUT /order/x
DELETE /order/y
```

POST is the catch-all and can have side-effects

```
POST /order/x/payment
```

Web-friendly systems *scale* because of safety

- Caching!

…support fault tolerance because of idempotent behavior

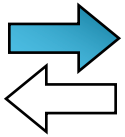
- Just re-try in failure cases

Unsafe and non-idempotent methods should be treated with care

Design Decision: POST vs. PUT

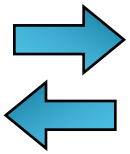
What is the right way of creating resources (initialize their state)?

Problem: How to ensure resource {id} is unique?
(Resources can be created by multiple clients concurrently)



- `PUT /resource/{id}`
- `201 Created`

Solution 1: let the client choose a unique id (e.g., GUID)



- `POST /resource`
- `301 Moved Permanently`
- `Location: /resource/{id}`

Solution 2: let the server compute the unique id

Problem: Duplicate instances may be created if requests are repeated due to unreliable communication

1. Resource Identification
2. Uniform Interface

3. Self-Describing Messages

4. Hypermedia as the Engine of Application State (HATEOAS)
5. Stateless Interactions

Self-describing Messages

Resources are abstract entities – resources are never exchanged or otherwise processed directly

- Resource identification guarantees that they are clearly identified
- They are accessed through a uniform interface

All interactions use resource representations

- To capture current or intended state of the resource
- To transfer the representation between components

Resource representations are structured documents

- Also should provide support for Hypermedia Driving Application State (see later)

Resource Representation

Technische Universität Berlin

Consists of data, metadata describing the data, and, on occasion, metadata to describe the metadata (usually for verifying message integrity)

Resource representations are sufficient to represent a resource

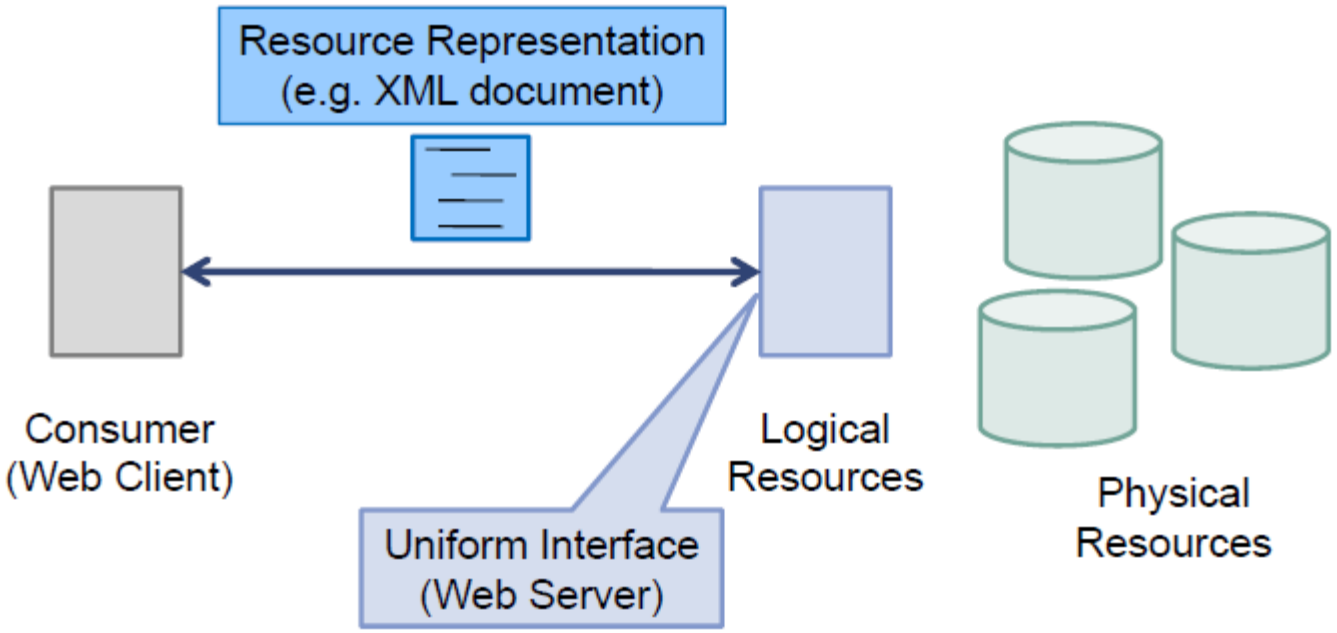
A resource has one or more representations

- If multiple representations of the resource exist at the time it is accessed, a content selection algorithm can be used to dynamically select a representation that best fits the capabilities of the client
- It is communicated which kind of representation is used

Data format of a representation is known as a media type

- Media types can be negotiated between peers
- XML and JSON can represent the same model for different users

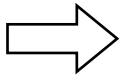
Resource Architecture



Content Negotiation (Conneg)

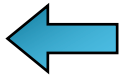
Technische Universität Berlin

Negotiating the data format does not require to send more messages (the added flexibility comes for free)



```
GET /resource
Accept: text/html, application/xml, application/json
```

1. The client lists the set of understood formats (MIME types)



```
200 OK
Content-Type: application/json
```

2. The server chooses the most appropriate one for the reply (status 406 if none can be found)

Extensible Markup Language (XML)

The language that started it all

- Created as a streamlined version of SGML
- Took over as the first universal language for structured data

XML is a metalanguage (a language for representing languages)

- Many domain-specific languages are defined as XML vocabularies
- Some metalanguages use XML syntax (RDF is a popular example)

XML is only syntax and has almost zero semantics

- Very minimal built-in semantics (language identification, IDs, relative URIs)
- Semantics are entirely left to the XML vocabularies

XML is built around a tree model

- Each XML document is a tree and thus limited in structure
- RESTful XML introduces hypermedia to turn XML data into a graph

JavaScript Object Notation (JSON)

The XMLHttpRequest API has been built for requesting XML via HTTP

- All requested data has to be processed by using XML access methods in JavaScript

JavaScript does not have XML as its internal data model

- The XML received via XMLHttpRequest has to be parsed into a DOM tree
- DOM access in JavaScript is inconvenient for complex operations
- Alternatively, the XML can be mapped to JavaScript objects (also requires parsing)

JavaScript Object Notation (JSON) encodes data as JavaScript objects

- Because the consumer is written in JavaScript, this is more efficient for the consumer
- With Node.js, also the server (and even the DB) is written in JavaScript → JSON can be used efficiently throughout the stack
- Turns the generally usable XML service into a JavaScript-oriented service
- For large-scale applications, it might make sense to provide XML and JSON
- This can be negotiated with HTTP content negotiation

XML and JSON Examples

```
<?xml version="1.0"?>
<menu id="file" value="File">
   <popup>
      <menuitem value="New" onclick="CreateNewDoc()"/>
      <menuitem value="Open" onclick="OpenDoc()"/>
      <menuitem value="Close" onclick="CloseDoc()"/>
   </popup>
</menu>
```

menu.xml

```
{ "menu" : {
    "id" : "file",
    "value" : "File",
    "popup" : {
      "menuitem" : [
        { "value" : "New", "onclick" : "CreateNewDoc()" },
        { "value" : "Open", "onclick" : "OpenDoc()" },
        { "value" : "Close", "onclick" : "CloseDoc()" }
      ]
    }
  }
}
```

menu.json

Resource Description Framework (RDF)

Technische Universität Berlin

Developed around the same time as XML was developed

- Based on the idea of machine-readable/understandable semantics
- Builds the *Semantic Web* as a parallel universe on top of the Web

RDF uses URIs for *naming things*

- RDF's data model is based on (URI, property, value) triples
- Triples are combined and inference is used to produce a graph

RDF is a metalanguage built on the triple-based data model

- RDF has a number of syntaxes (one of them is XML-based)
- RDF introduces a number of schema languages (often referred to as *ontology languages*)

1. Resource Identification
2. Uniform Interface
3. Self-Describing Messages

4. Hypermedia as the Engine of Application State (HATEOAS)

5. Stateless Interactions

Hypermedia Driving Application State

Technische Universität Berlin

Resource representations contain ***links*** to identified resources (***linked documents***)

- Links make interconnected resources navigable
- Resources and state can be used by navigating
- Without navigation, identifying new resources is service-specific

RESTful applications ***navigate*** instead of *calling*

- Representations contain information about possible traversals
- The application navigates to the next resource depending on link semantics
- Navigation can be delegated since all links use identifiers

Application state also is represented as a resource

- E.g. *next* links on multi-page submission processes
- *Paged results* with URIs identifying following pages

“Calling” versus “Navigation”

RPC-oriented systems need to expose the available functions

- Functions are essential for interacting with a service
- Introspection or interface descriptions make functions discoverable

RESTful systems use a Uniform Interface

➢ No need to learn about functions

But how to find resources?

- Find them by following links from other resources
- Learn about them by using URI Templates
- Understand them by recognizing representations

Technische Universität Berlin

1. Resource Identification
2. Uniform Interface
3. Self-Describing Messages
4. Hypermedia as the Engine of Application State (HATEOAS)

5. Stateless Interactions

Stateless Interactions

This constraint *does not* say Stateless Applications!

- For many RESTful applications, state is an essential part
- The idea of REST is to avoid long-lasting transactions in applications
- The very name of “Representational State Transfer” is centered around how to deal with state in a distributed system.

- Every request happens in complete isolation, all information required for the server to fulfil the request are included in the request

- Server never relies on information from previous requests

- Statelessness means to move state to clients or resources
  - The most important consequence: avoid state in server-side applications

Client and Resource State

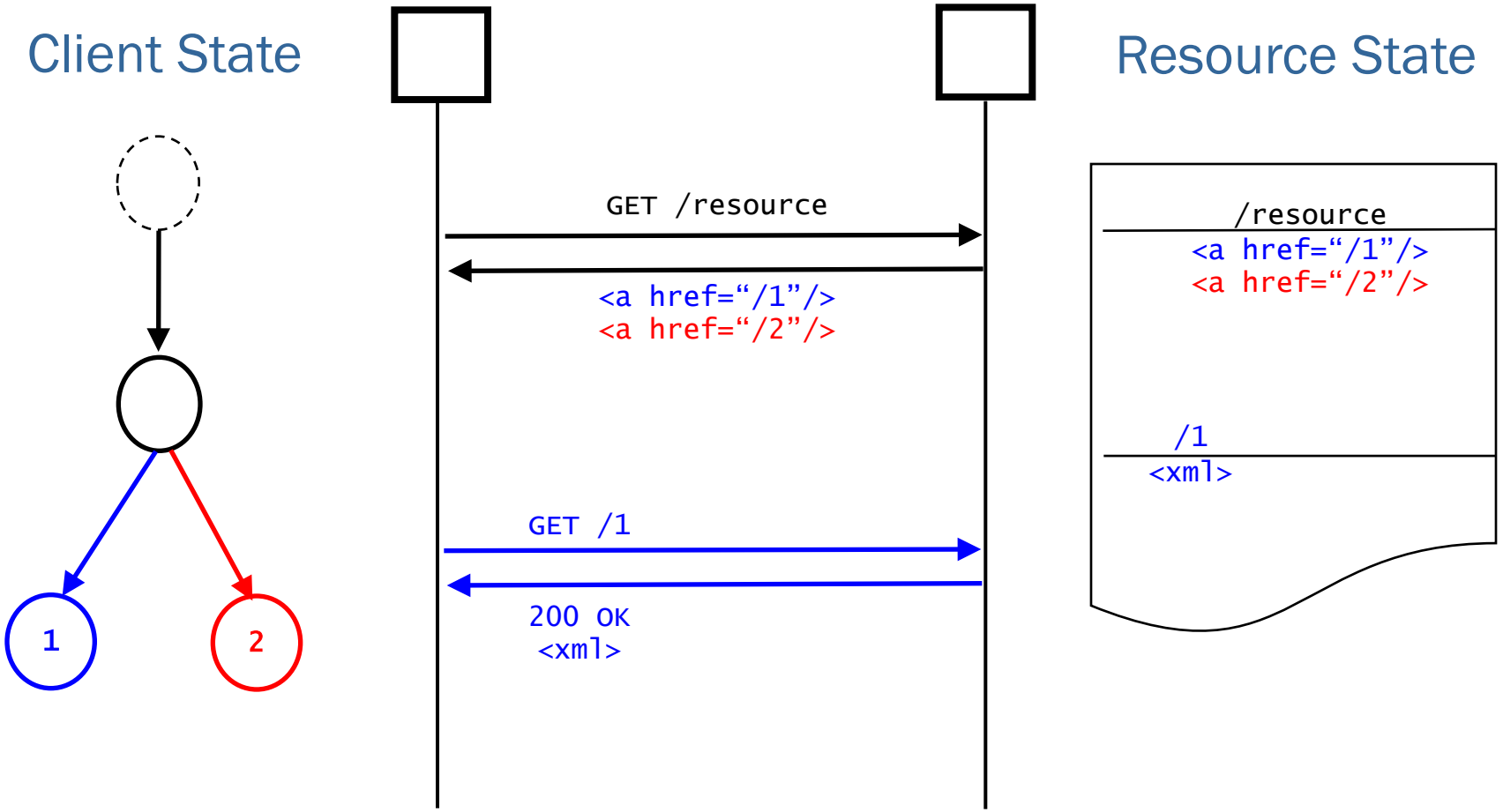
Client state is managed on the client

- It is specific for a client and thus has to be maintained by each client
- It may affect access to server resources, but not the resources themselves
- The client interacts with resources by “navigating hyperlinks” and its state captures the current position in the hypertext.
- The server may influence the state transitions of the client by sending different representations (containing hyperlinks to be followed) in response to GET requests

Resource state is managed on the server

- The state of resources captures the persistent state of the service
- This state can be accessed by clients under different representations
- The client manipulates the state of resources using the uniform interface CRUD-like semantics (PUT, DELETE, POST)
  - It is the same for every client working with the service
  - When a client changes resource state other clients see this change as well

Stateless or Stateful?



State Management on the Web

Mechanisms to provide support for longer ***sessions*** in RESTful enterprise architectures

- A session is an ongoing exchange of information, where later parts of the exchange depend on earlier parts
- The current status of a session is referred to as its state
- E.g. a user logging in and working on a Web site as an identified user

HTTP is a stateless protocol

- Each request/response interaction is a separate interaction
- There is no protocol support for longer sessions

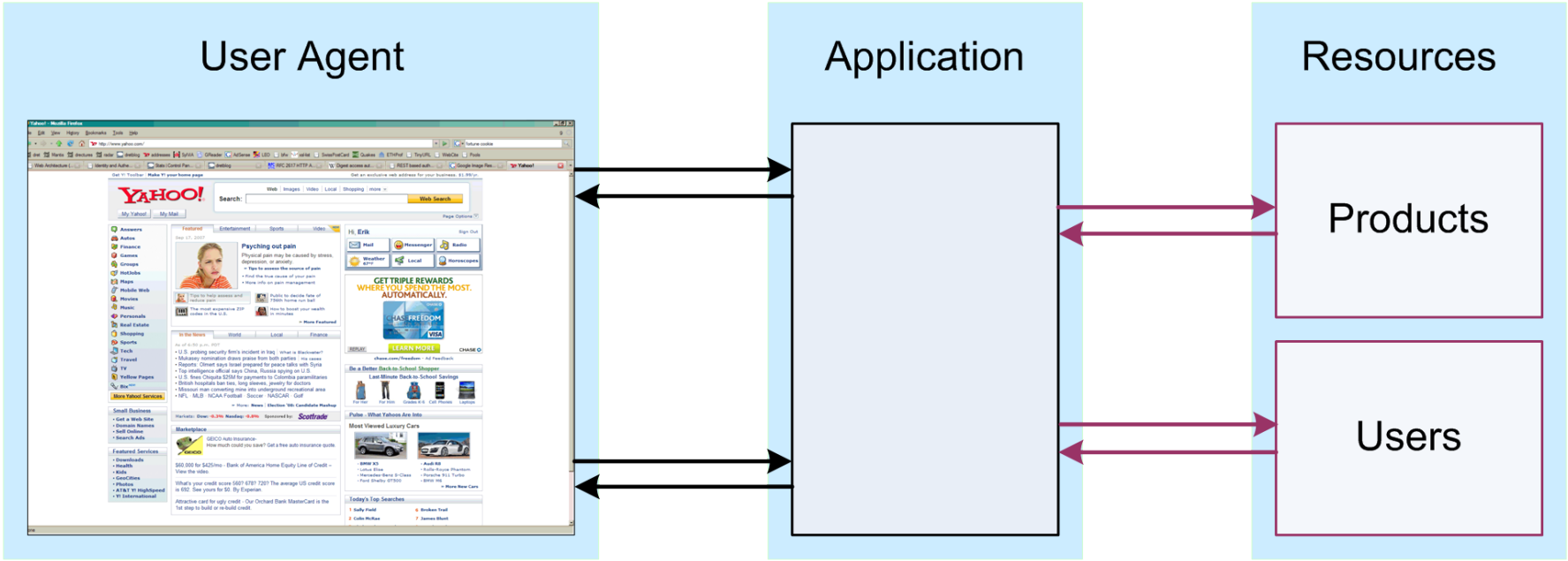
Manage session state through

- Sending back and forth state as part of every request and response
- Store state in the server
- State as a resource

Source: http://courses.ischool.berkeley.edu/i153/su11/cookies

State in HTML or HTTP

Sending back and forth state as part of every request and response



State in HTML or HTTP Authentication Info

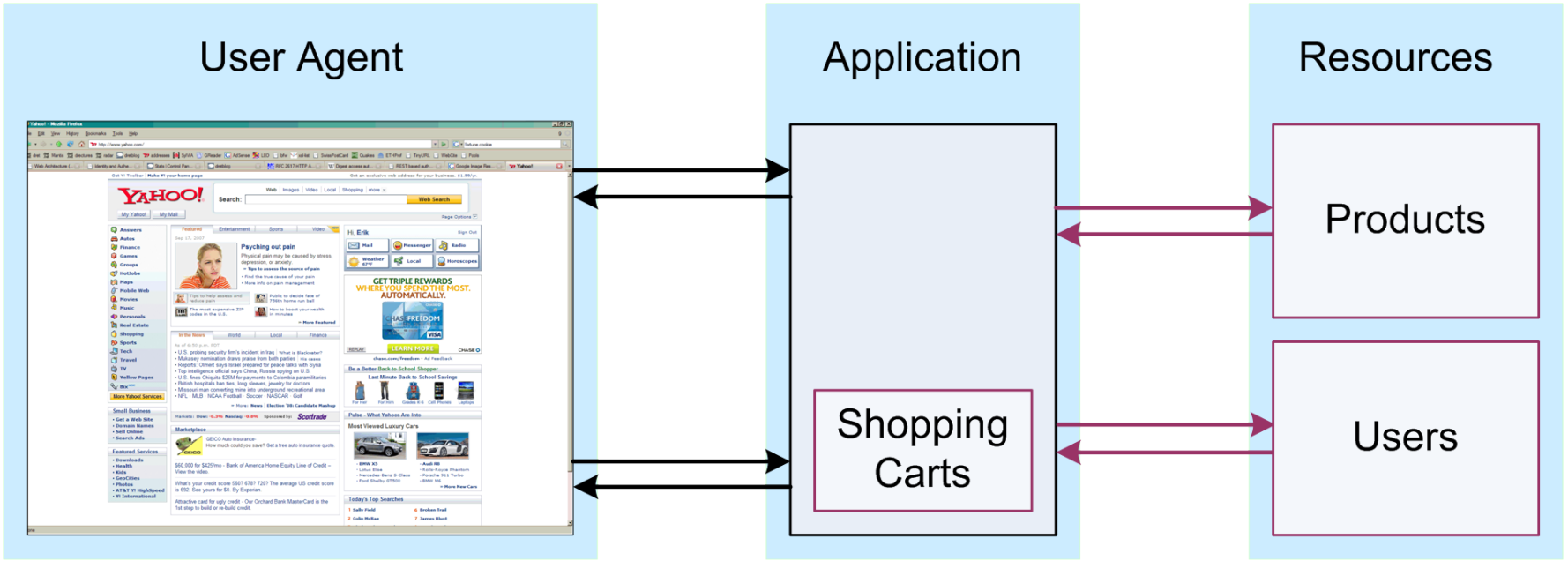
Source: http://courses.ischool.berkeley.edu/i153/su11/cookies

State in the Server Application

Store state in the server and refer to it from the client

Tradeoffs

- costs of maintaining client state, scalability vs. client experience, cost of extra messages
- Still RESTful?

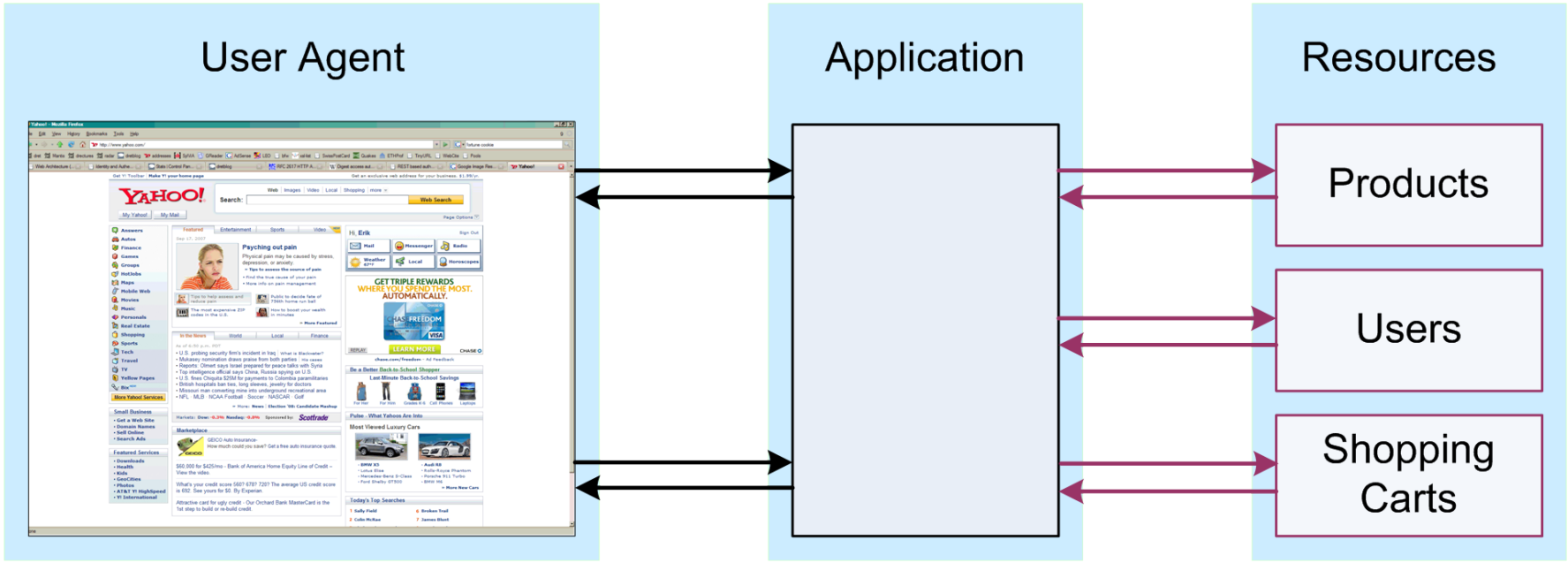


Session-Identified State

Source: http://courses.ischool.berkeley.edu/i153/su11/cookies

Treat State as a Resource

Store state at a URI and use the URI to refer to that state



Application State (User Session)

State as a Resource

Source: http://courses.ischool.berkeley.edu/i153/su11/cookies

Stateless Shopping

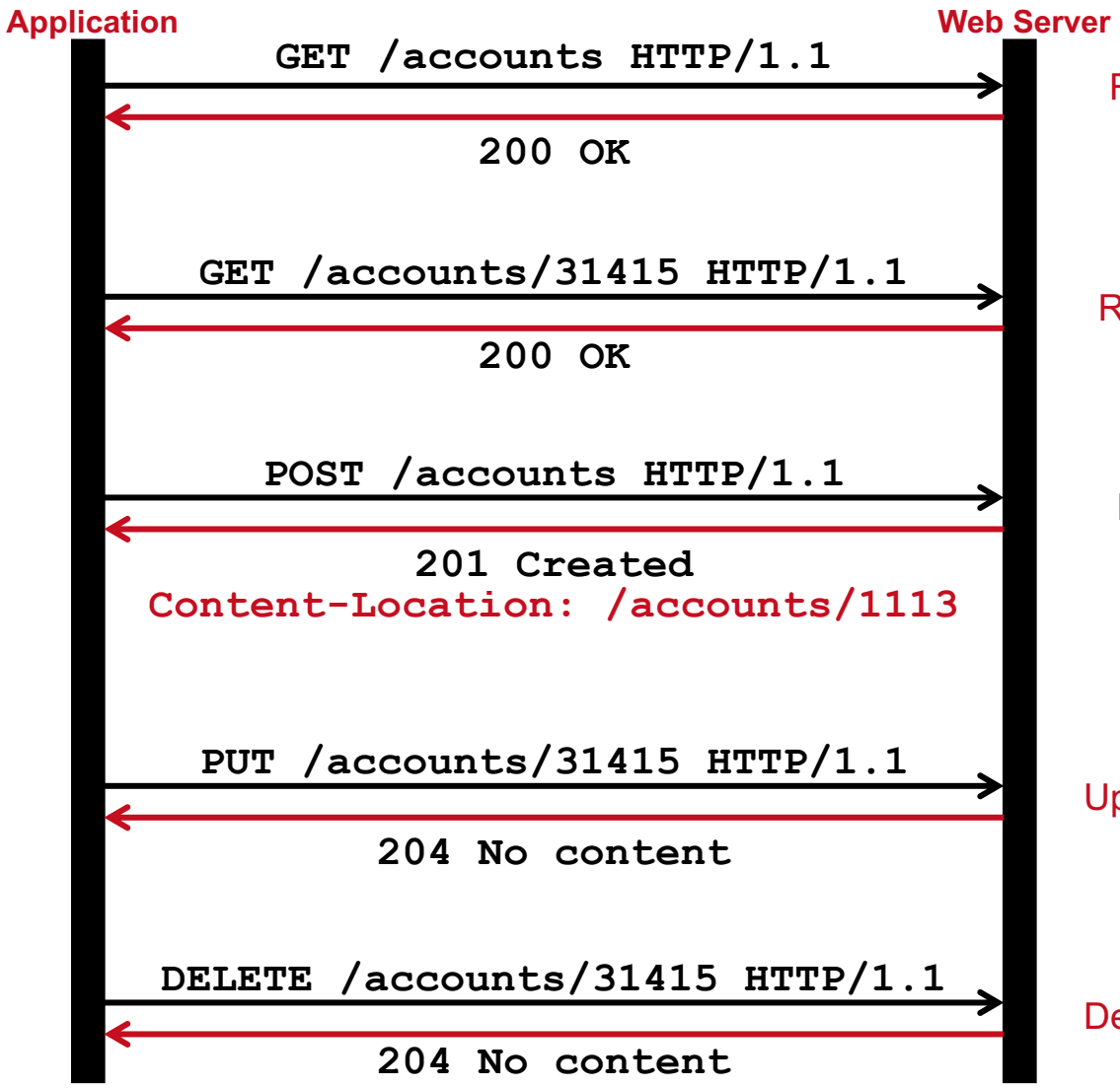
Typical session scenarios can be mapped to resources

- Client: Show me your products
- Server: Here's a list of all the products
- Client: I'd like to buy 1 of http://ex.org/product/X, I am "John"/"Password"
- Server: I've added 1 of http://ex.org/product/X to http://ex.org/users/john/basket
- Client: I'd like to buy 1 of http://ex.org/product/Y, I am "John"/"Password"
- Server: I've added 1 of http://ex.org/product/Y to http://ex.org/users/john/basket
- Client: I don't want http://ex.org/product/X, remove it, I am "John"/"Password"
- Server: I've removed http://ex.org/product/X to http://ex.org/users/john/basket
- Client: Okay I'm done, username/password is "John"/"Password"
- Server: Here is the total cost of the items in http://ex.org/users/john/basket

This is more than just renaming session to resource

- all relevant data is stored persistently on the server
- the shopping cart's URI can be used by other services for working with its contents
- instead of *hiding the cart in the session*, it is *exposed as a resource*

RESTful Application: Example



Retrieve a list of all current accounts

Retrieve a particular account

Create a new account.
Retrieve new account URI
from HTTP Headers

Update an account

Delete an account

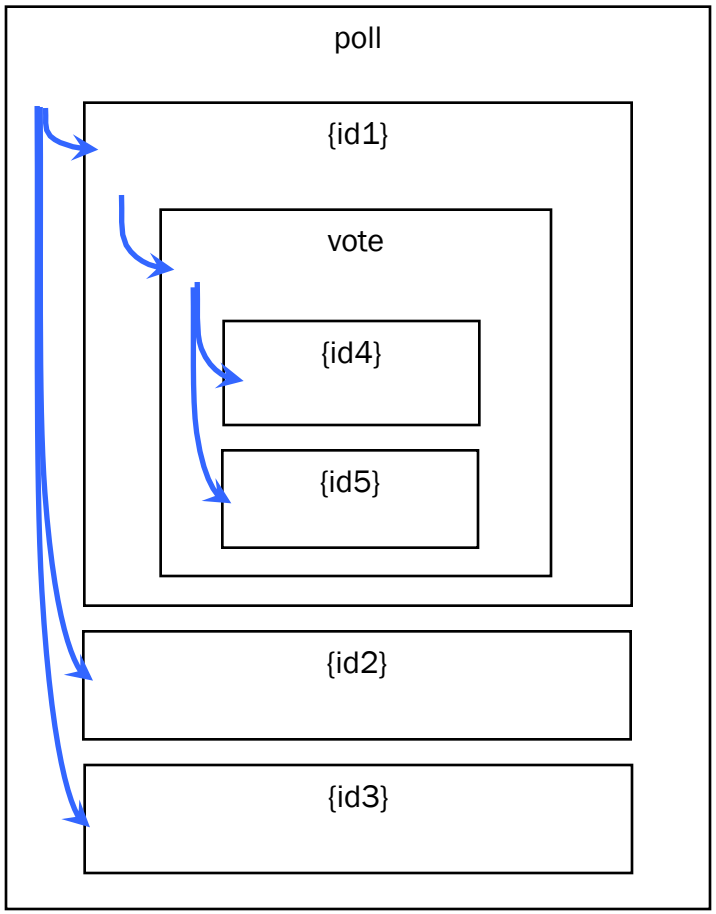
Create
Read
Update
Delete

Technische Universität Berlin

Example

Simple Doodle API Example Design

1. Resources: polls and votes
2. Containment Relationship:



	GET	PUT	POST	DELETE
/poll	✓	✗	✓	✗
/poll/*{id}*	✓	✓	✗	✓
/poll/*{id}*/vote	✓	✗	✓	✗
/poll/*{id}*/vote/*{id}*	✓	✓	✗	?

3. URIs embed IDs of “child” instance resources
4. POST on the container is used to create child resources
5. PUT/DELETE for updating and removing child resources

Simple Doodle API Example

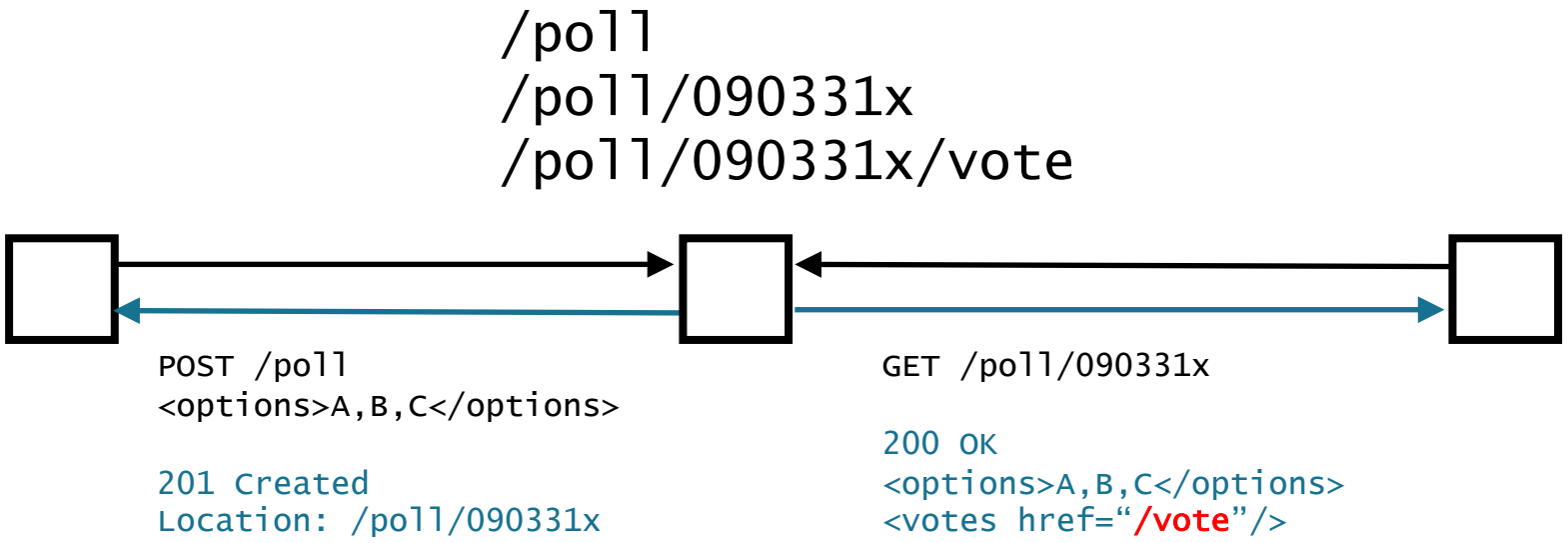
Technische Universität Berlin

Creating a poll (transfer the state of a new poll on the Doodle service)
Reading a poll (transfer the state of the poll from the Doodle service)

```
/poll
/poll/090331x
/poll/090331x/vote
```

Simple Doodle API Example

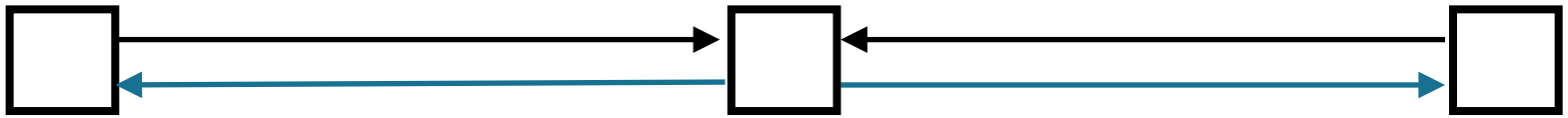
Creating a poll (transfer the state of a new poll on the Doodle service)
Reading a poll (transfer the state of the poll from the Doodle service)



Simple Doodle API Example

Participating in a poll by creating a new vote sub-resource

```
/poll
/poll/090331x
/poll/090331x/vote
/poll/090331x/vote/1
```



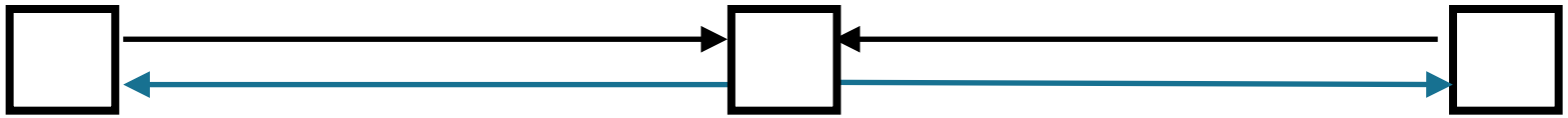
```
POST /poll/090331x/vote
<name>Paul</name>
<choice>B</choice>

201 Created
Location:
/poll/090331x/vote/1
```

```
GET /poll/090331x

200 OK
<options>A,B,C</options>
<votes><vote id="1">
<name>Paul</name>
<choice>B</choice>
</vote></votes>
```

Simple Doodle API Example

Existing votes can be updated (access control headers not shown)

```
/poll
/poll/090331x
/poll/090331x/vote
/poll/090331x/vote/1
```



```
PUT /poll/090331x/vote/1
<name>Paul</name>
<choice>C</choice>

200 OK
```

```
GET /poll/090331x

200 OK
<options>A,B,C</options>
<votes><vote id="/1">
<name>Paul</name>
<choice>C</choice>
</vote></votes>
```

Simple Doodle API Example

Polls can be deleted once a decision has been made

```
/poll
/poll/090331x
/poll/090331x/vote
/poll/090331x/vote/1
```

`DELETE /poll/090331x`

`200 OK`

`GET /poll/090331x`

`404 Not Found`

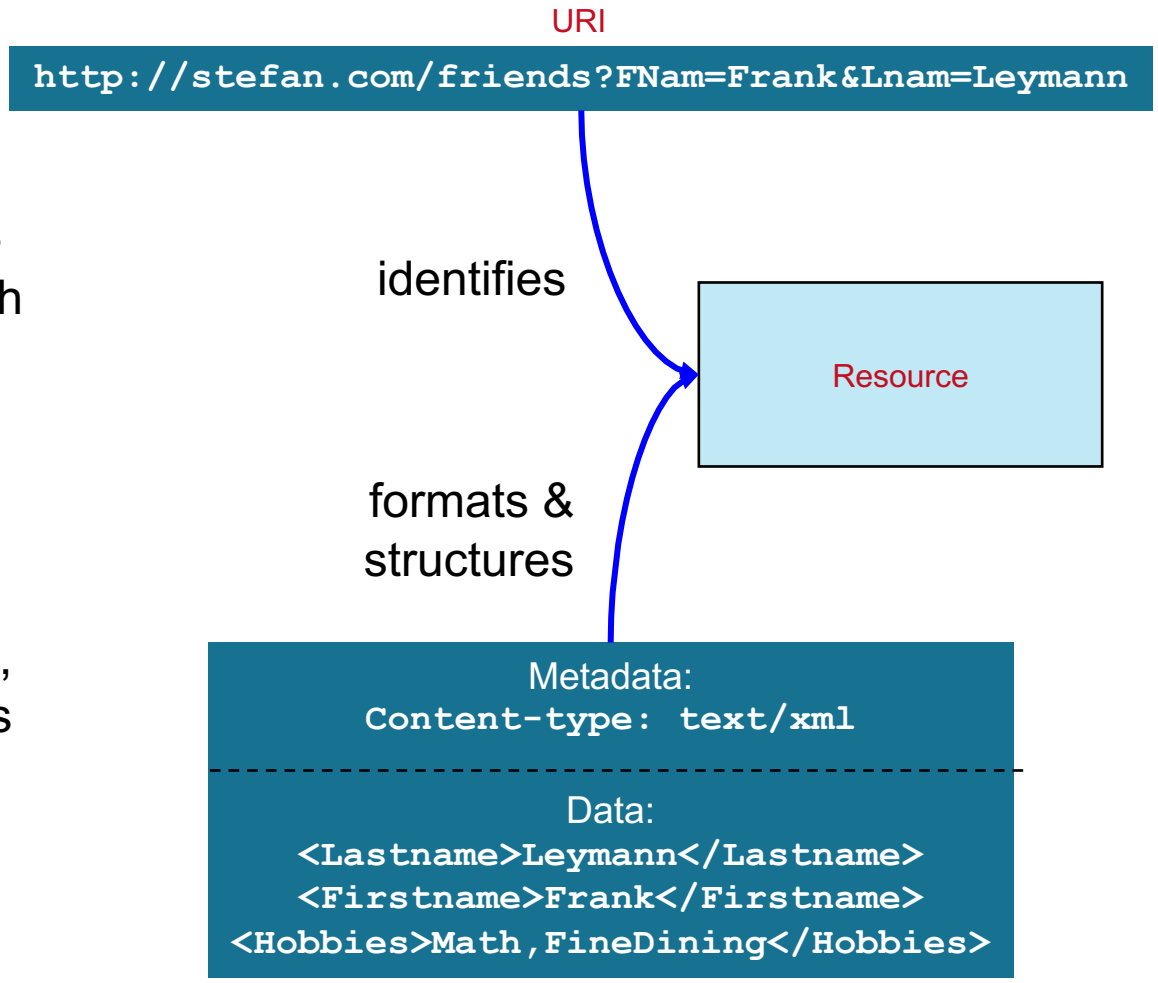
REST Summary: Main ingredients

Technische Universität Berlin

Resources identified by URI
- Use parameters if needed

Representations in an open-ended set of formats such as XML, JSON, RDF,…
- Content negotiation

Stateless interactions via uniform interface (HTTP), self-describing messages and HATEOAS



ISEngineering
Wirtschaftsinformatik – Information Systems Engineering

REST Strengths

- Simplicity
  - Uniform interface is immutable (no problem of breaking clients)
- HTTP is ubiquitous (goes through firewalls)
- Stateless interactions
- Proven scalability
  - “After all the Web works”, caching, clustered server farms for QoS
- Perceived ease of adoption (light infrastructure)
- „Grassroots“ approach
- Leveraged by all major Web 2.0 applications and cloud services

  Amazon AWS usage: >90% clients prefer RESTful API over SOAP/WSDL

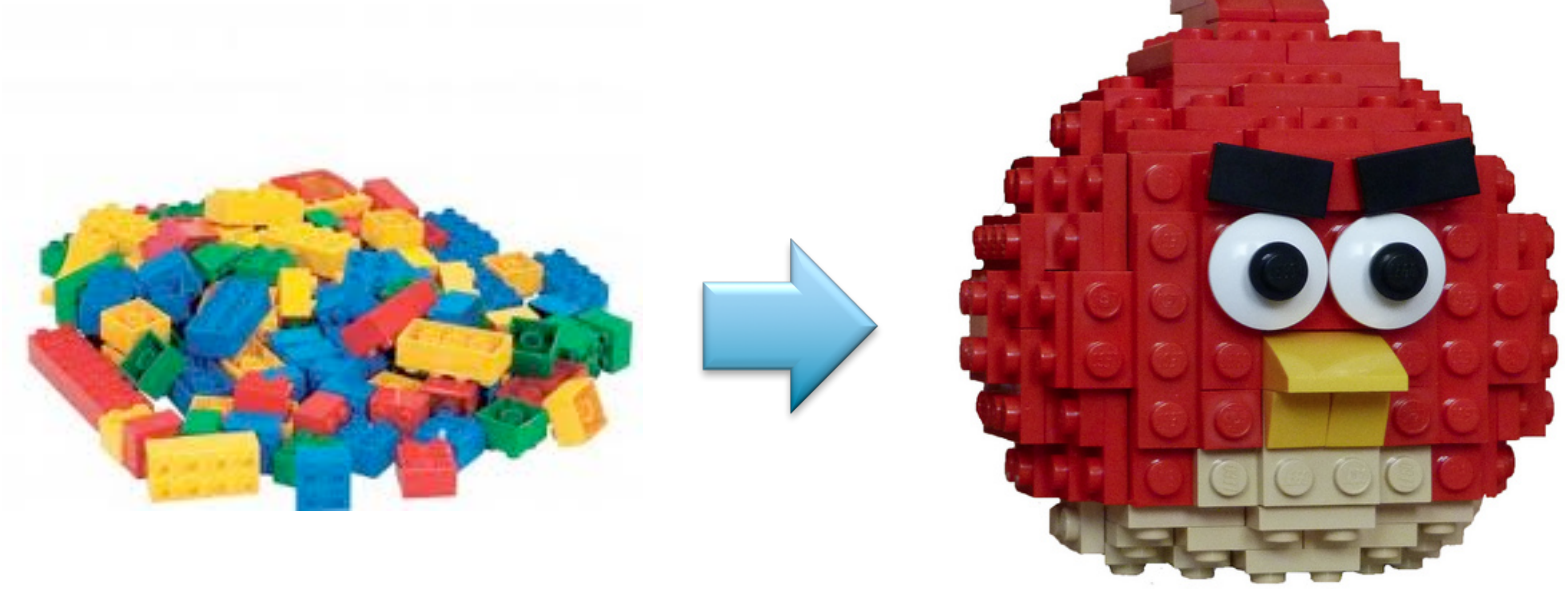
REST Weaknesses

- Mapping REST-style synchronous semantics on top of back end systems creates design mismatches (when they are based on asynchronous messaging or event driven interaction)
- Cannot deliver enterprise-style “-ilities” beyond HTTP/SSL
- Challenging to identify and locate resources appropriately in all applications
- Apparent lack of standards (other than URI, HTTP, XML, MIME, HTML)
- Semantics/Syntax description very informal (user/human oriented)

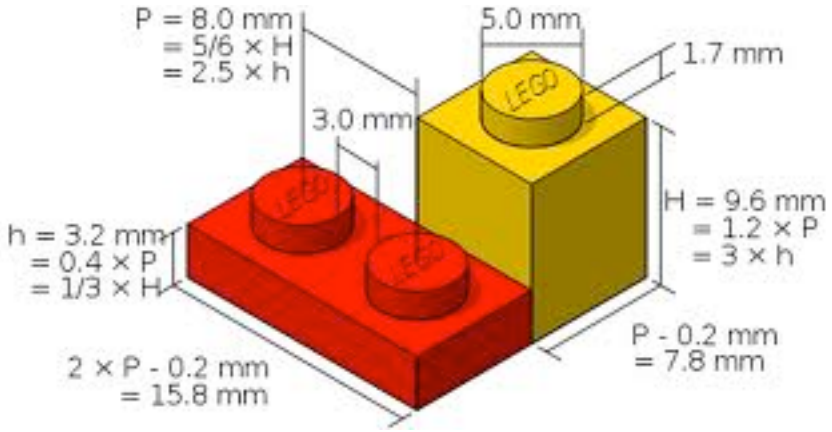
Technische Universität Berlin

Recap: SOAP/WSDL-based Web Services

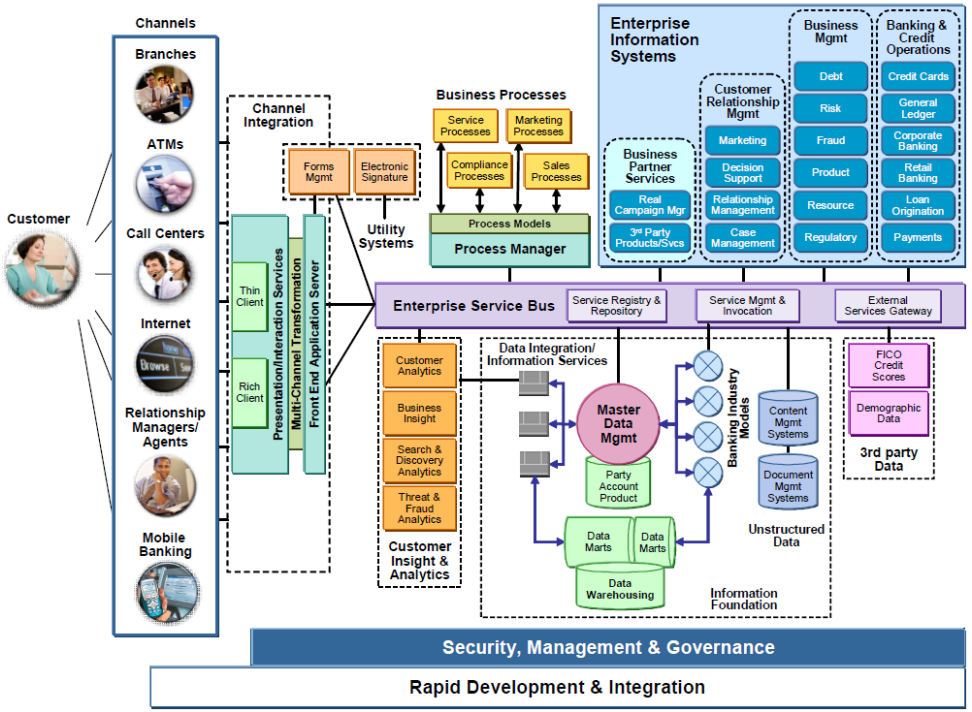
Services as composable building blocks



Norms and standards play a role here



Service interfaces vs. service implementations



what the client sees ()

…and what may be inside*

Source: IBM Banking Reference Architecture, available at http://www.redbooks.ibm.com/redpapers/pdfs/redp4467.pdf

Example:
An online marketplace for nice things…

Home > Home And Tech

HOME AND TECH

HOME AND TECH

ALL HOME AND TECH
- AUDIO
- BOOKS
- CANDLES
- CASES AND COVERS
- GIFTS AND HOME
- GROOMING
- LUGGAGE AND TRAVEL

COLOUR
- All
- Black
- Blue
- Brown
- Gold
- Gray
- Green
- Metallic
- Neutrals
- Orange
- Red
- Silver
- White
- Yellow

DESIGNER
- All

SORT BY PRICE: HIGH | LOW PRODUCT VIEW | OUTFIT VIEW PAGE 1 OF 5 | NEXT | VIEW ALL

LE LABO
Vetiver 46 Body Lotion
€50
NEW

BAXTER OF CALIFORNIA
After Shave Balm 120ml
€20
NEW

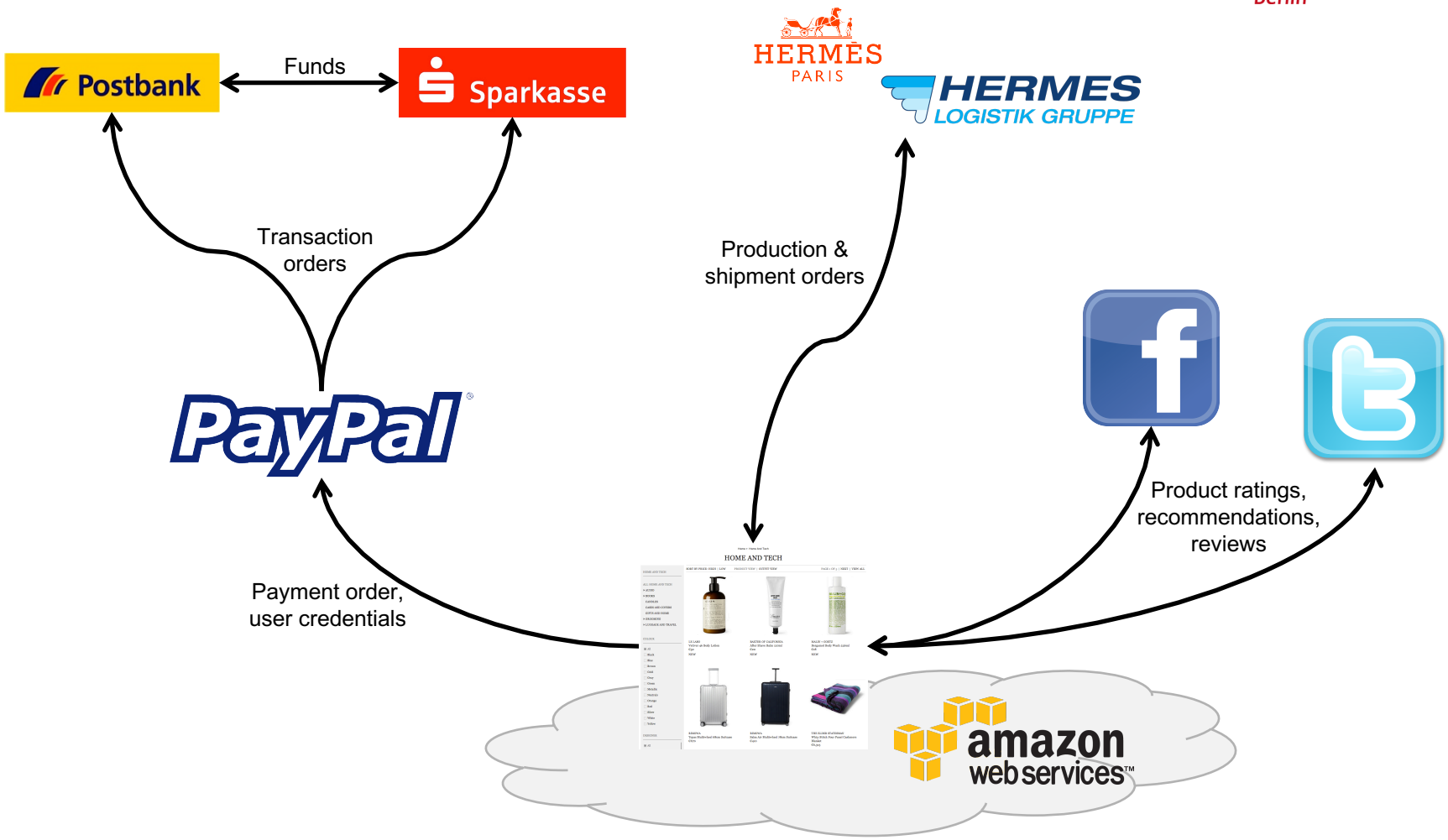
MALIN + GOETZ
Bergamot Body Wash 236ml
€18
NEW

RIMOWA
Topas Multiwheel 68cm Suitcase
€670

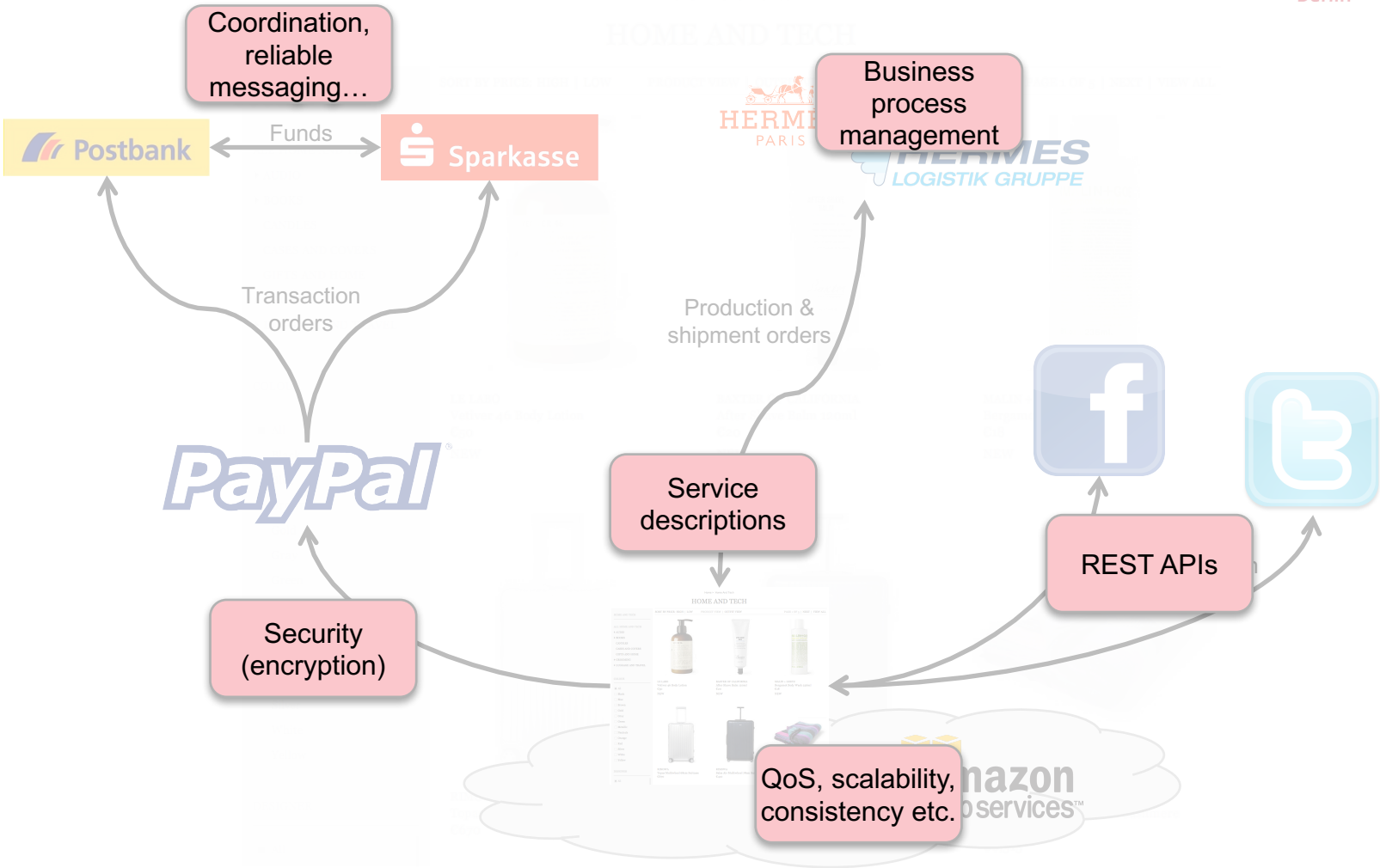
RIMOWA
Salsa Air Multiwheel 78cm Suitcase
€410

THE ELDER STATESMAN
Whip Stitch Four Panel Cashmere Blanket
€6,315

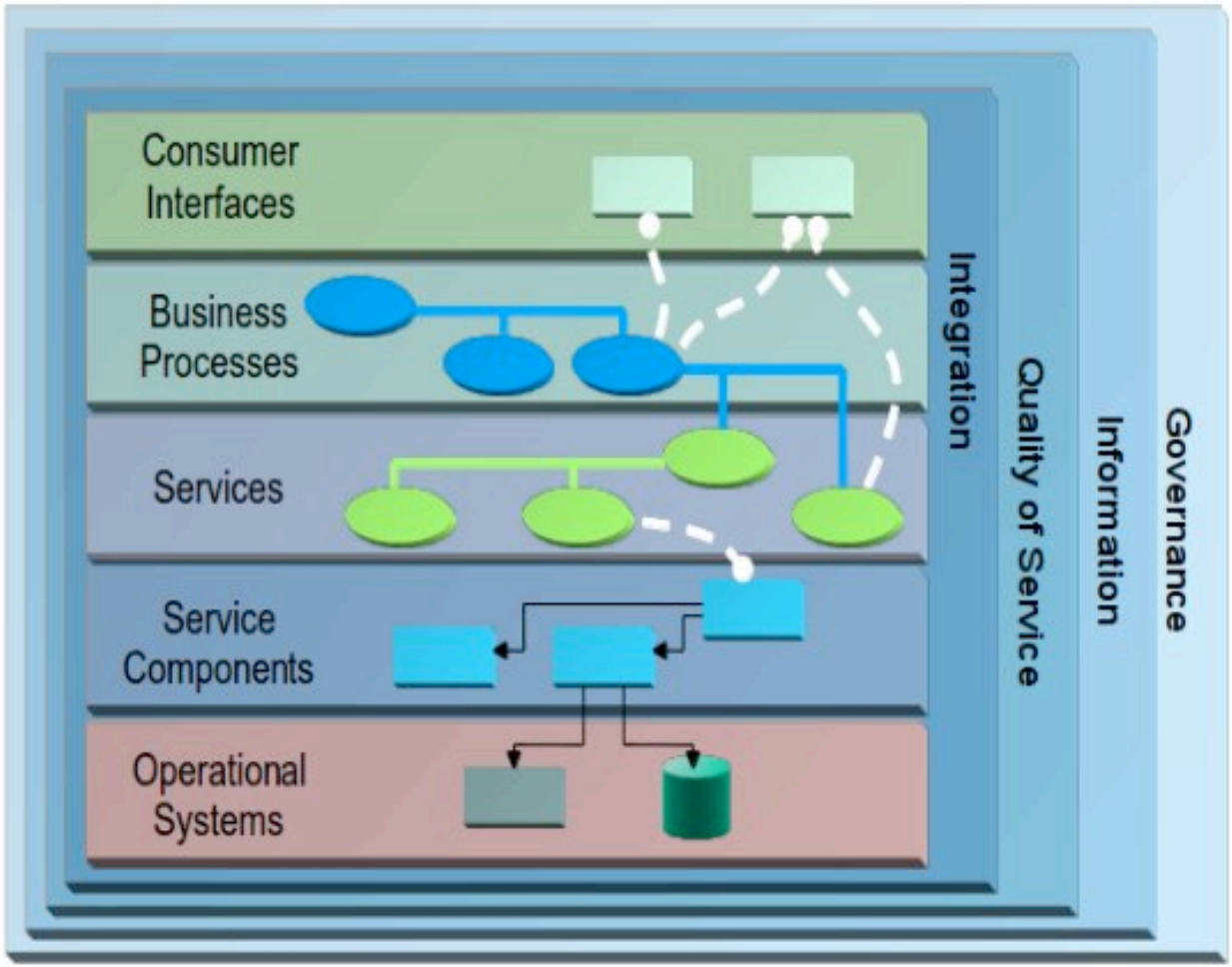
…the involved services…



…and resulting enterprise computing challenges:



Service-oriented architectures



Services as Building Blocks

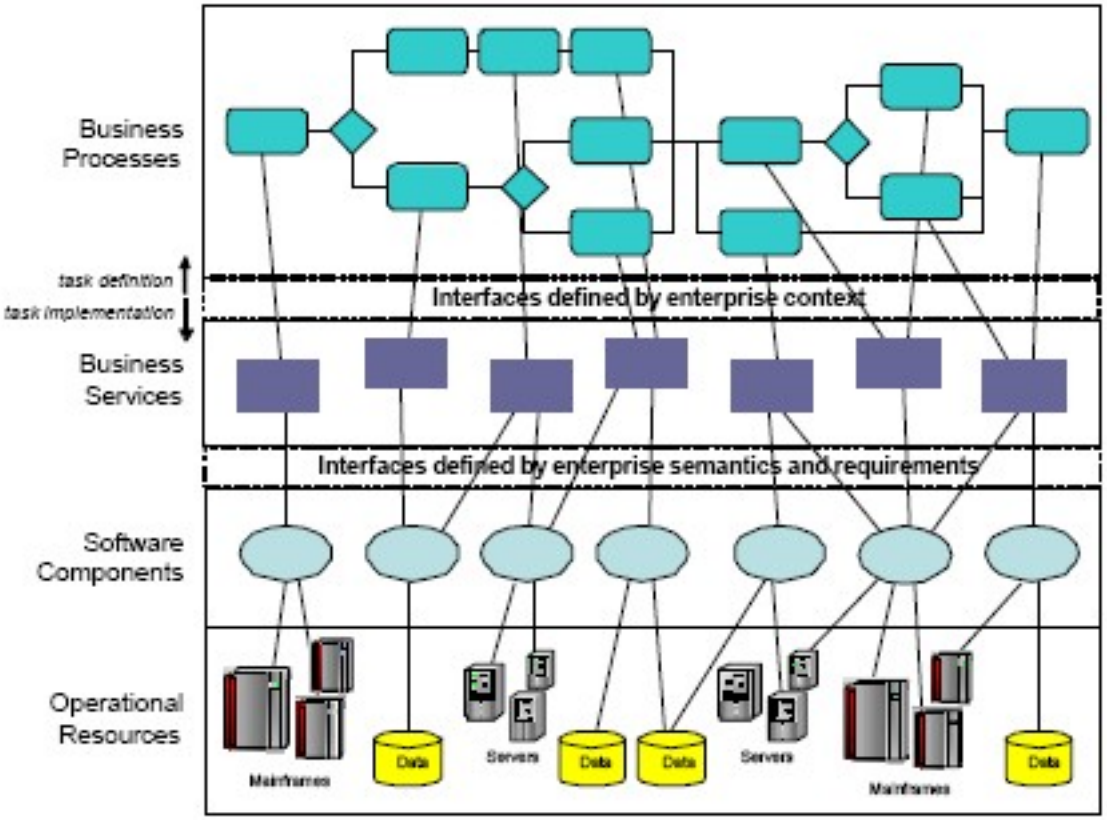
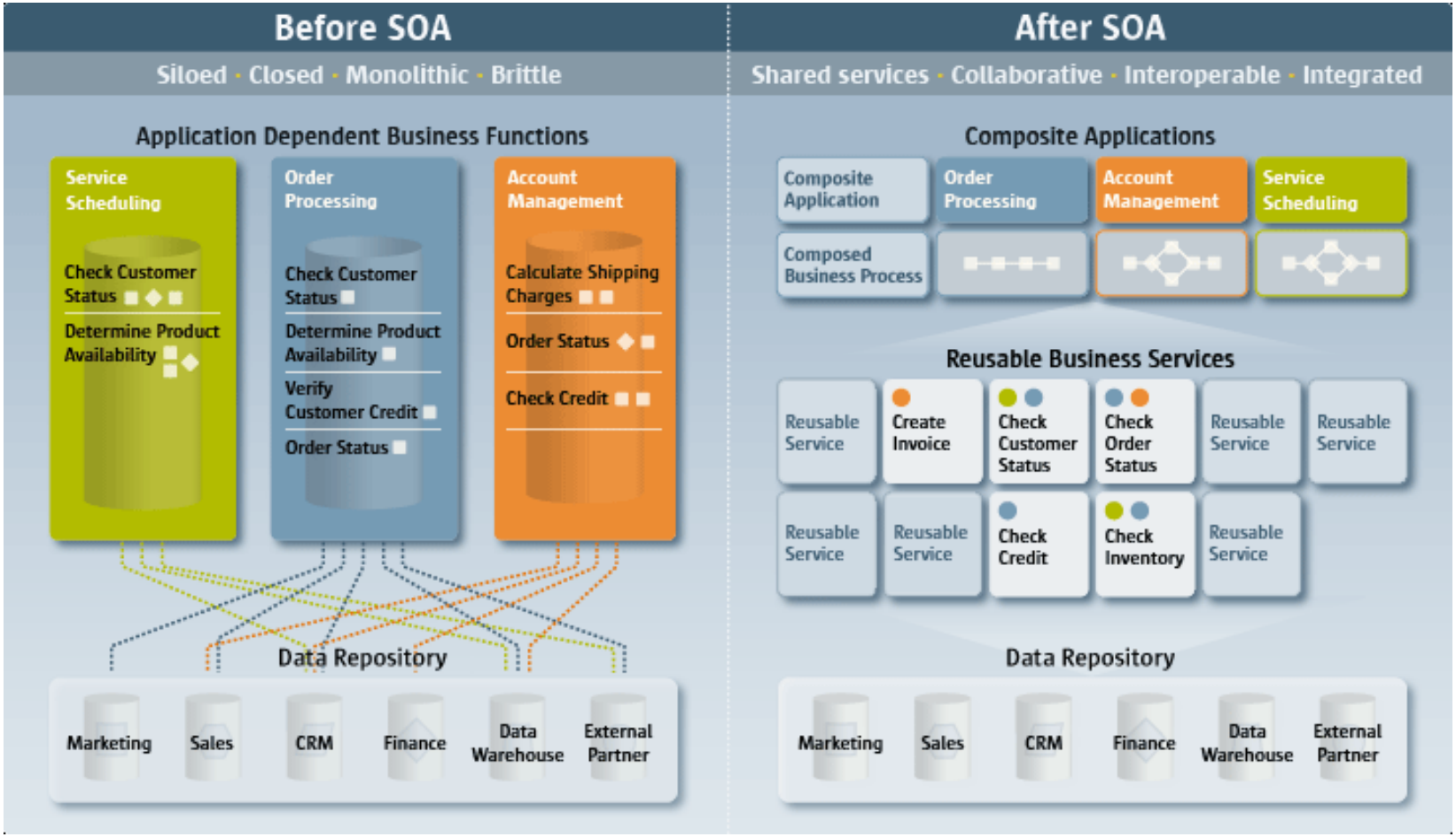


Figure source: OMG

Why are we interested in SOA?



The promises of SOA and (Web) Services

Technische Universität Berlin

Within an enterprise (Enterprise Application Integration)

- Accelerate and reduce the cost of integration
- Save on infrastructure deployment & management costs
- Reduce skill requirements
- Improve reuse

Between enterprises (e-Business integration)

- Provide online services to a company's customers
- Consume services from a company's partners and suppliers
- Standards and common Web-based infrastructure reduce barriers
- Simplicity accelerating deployment
- Dynamics opening new business opportunities

What are Web Services then?

…can be seen as a natural evolution of conventional middleware, exposing the functionality of an information system and making it available through standard Web technologies

Web services perform encapsulated business functions such as:

- a self-contained business task, such as a funds withdrawal or funds deposit service
- a full-fledged business process, such as the automated purchasing of office supplies
- an application, such as a life insurance application or demand forecasts and stock replenishment
- a service-enabled resource, such as a particular back-end database containing patient medical records

Web Services Definitions and Kinds

Technische Universität Berlin

"Web services" tend to fall into one of the following 'camps':

- **SOAP/WSDL-based Web Services (aka WS-* Web Services)**

  The W3C defines a Web Service as *"a software system designed to support interoperable, machine-to-machine interaction over a network"*

  …composable into executable business processes (BPEL processes), following a business process management (BPM) approach

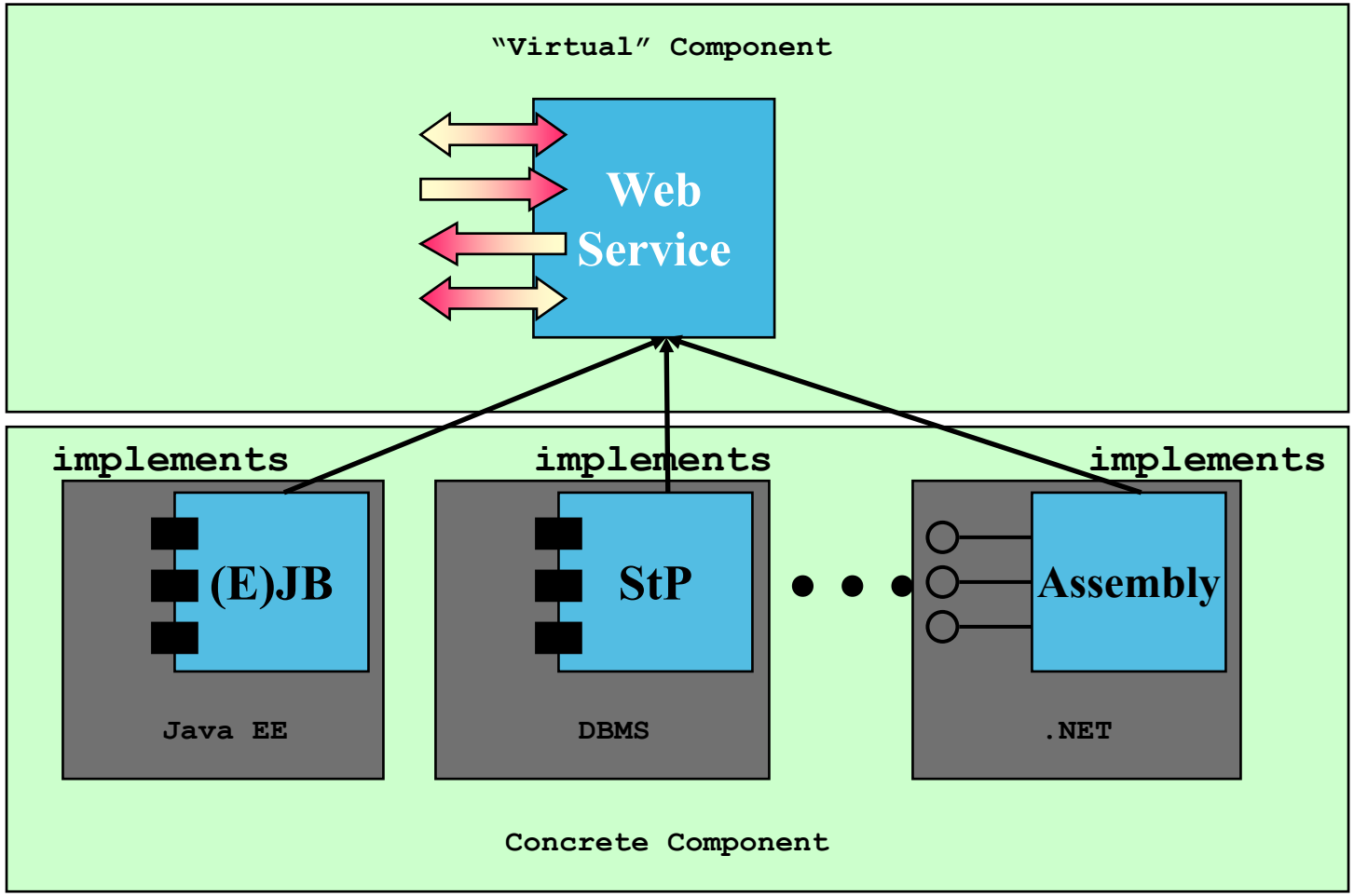
- **RESTful Services**

  In common usage, RESTful services refer to stateless, client-server architectures over the Web, typically using HTTP.

- **Microservices**

  Small, single business function focused, following a DevOps-based organization and supporting continuous delivery of applications

WS-* Web Services: Interoperable software components with diverse implementations



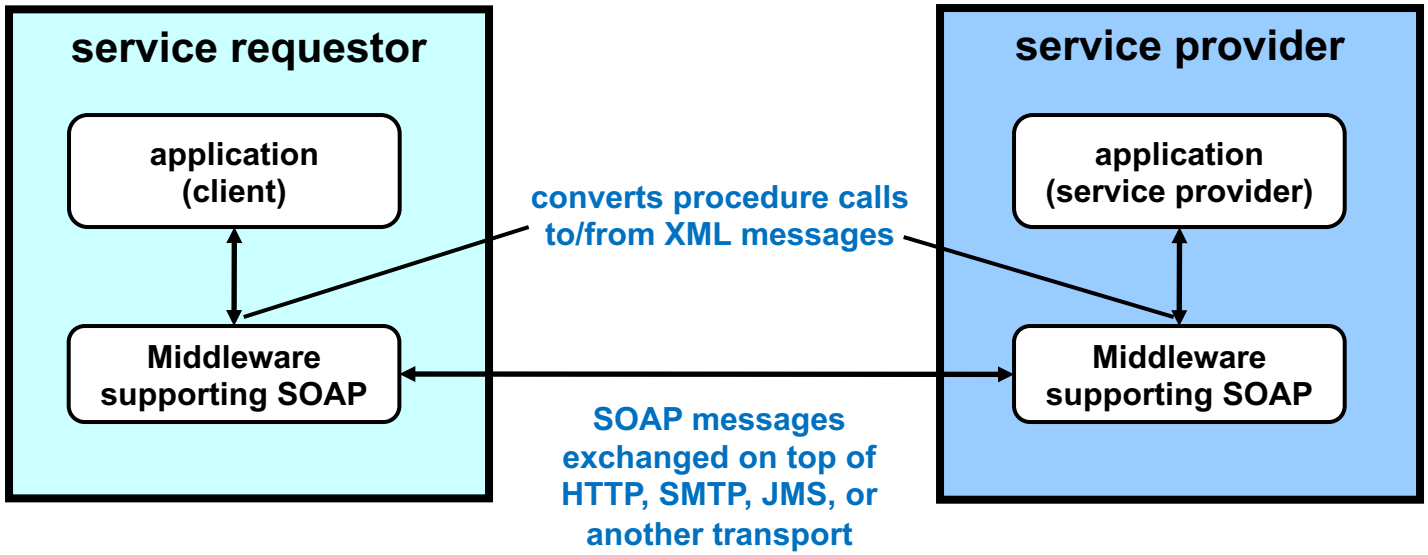
Source: Frank Leymann

Technische Universität Berlin

Recap: SOAP

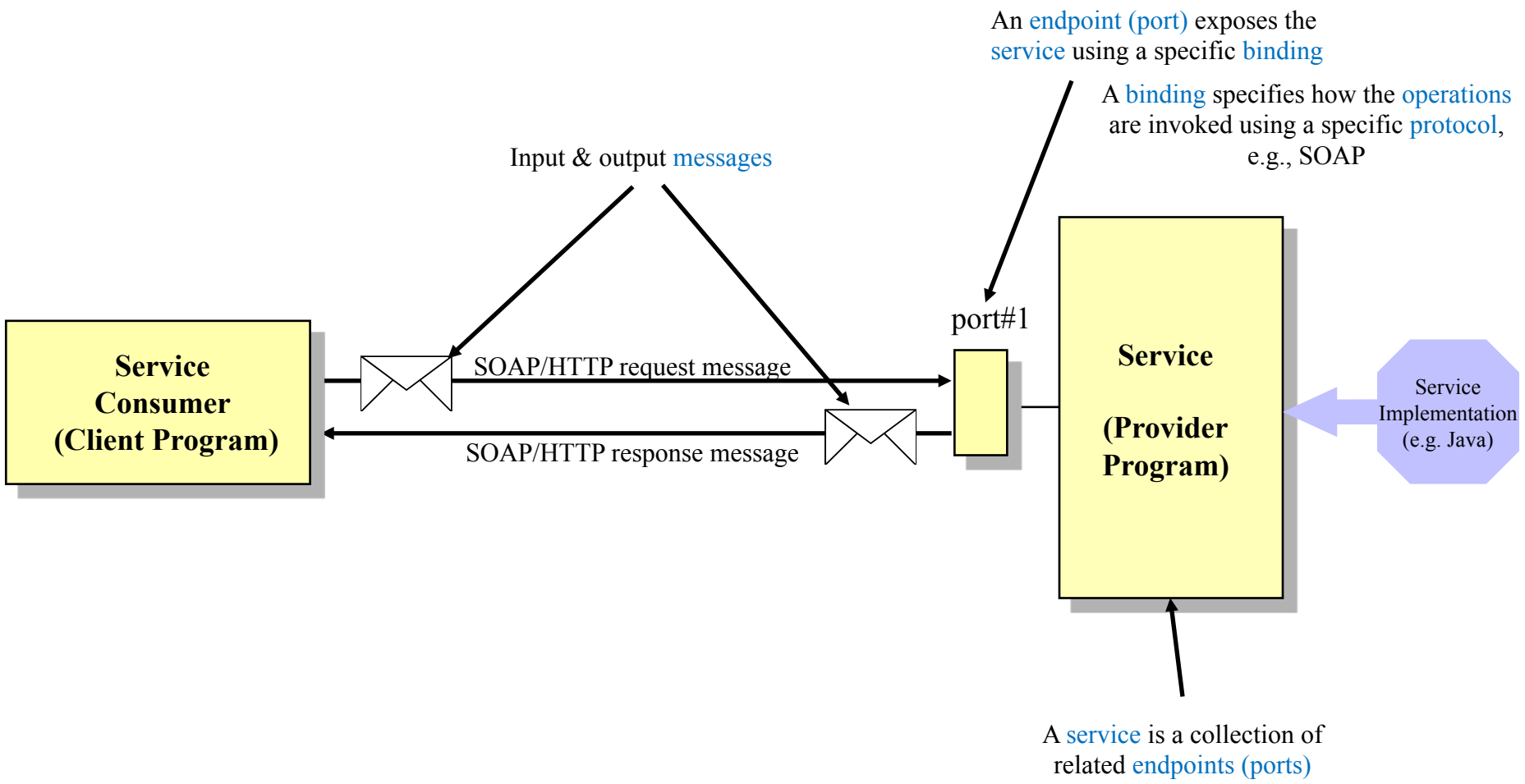
ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

How SOAP works



SOAP was originally created in 1999 (as “XML-RPC”)
Since 2001 the W3C has been working on SOAP (Simple Object Access Protocol); the current W3C recommendation is Version 1.2 (2007)
Since 2003 SOAP no longer is an acronym

A closer look



What is SOAP?

1. A message format for one-way communication describing how a message can be packed into an XML document

2. A description of how the message (the XML document) should be transported using HTTP (for Web based interaction), SMTP (for e-mail based interaction), or other transport protocols

3. A set of rules that must be followed when processing a SOAP message and a simple classification of the entities involved in processing a SOAP message. It also specifies what parts of the messages should be read by whom and how to react in case the content is not understood.

4. A set of conventions on how to turn an RPC call into a SOAP message and back, as well as how to implement the RPC style of interaction

Source: Gustavo Alonso

SOAP messages

SOAP messages are seen as envelopes where the application encloses the data to be sent

A message has two main parts:

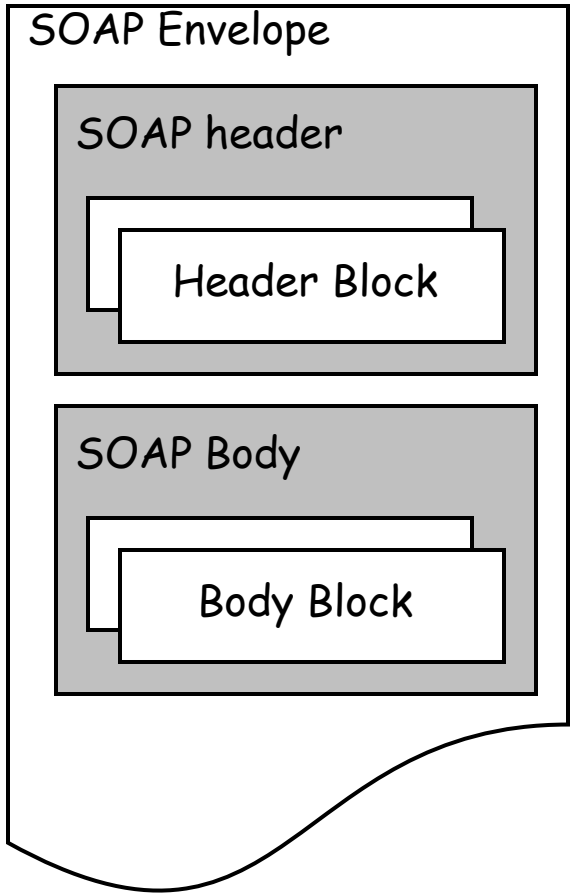
- Header (divided into blocks)
- Body (also divided into blocks)

SOAP does not say what to do with these parts, it only states that

- the header is optional
- the body is mandatory

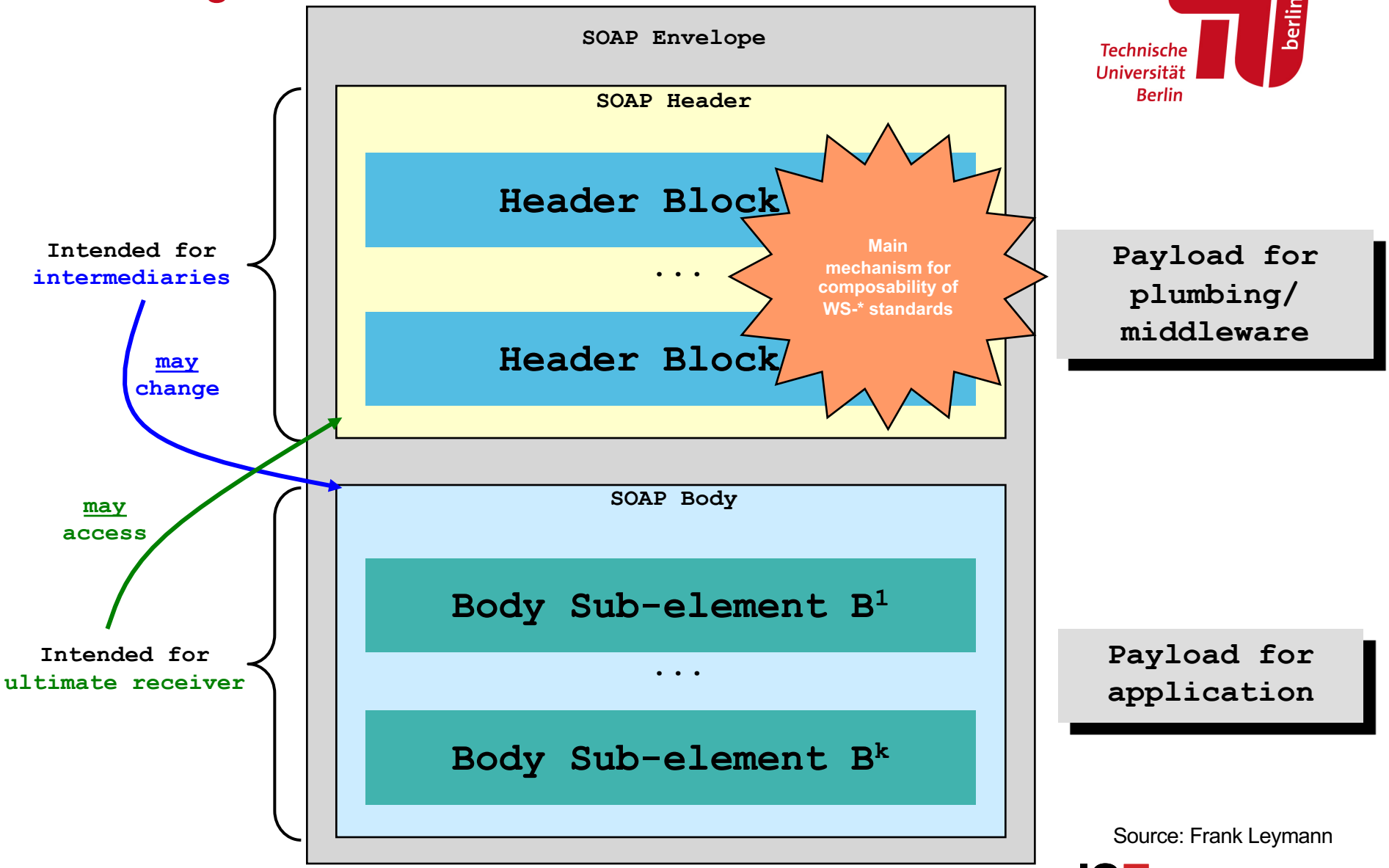
The use of header and body, however, is implicit:

- Body: for application-level data
- Header: for infrastructure-level data



Source: Gustavo Alonso

Message structure



Source: Frank Leymann

The SOAP header

Contains application-independent information used at the Middleware level

- Typical uses: coordination information, identifiers (e.g. transactions), security information (e.g. certificates)

SOAP provides means to specify how to process header blocks:

- `role` attribute: who shall process the header block. Values can be:
  - none: additional information which must be ignored
  - next: any receiver may process the block
  - ultimateReceiver: final receiver may process the header block
  - application-specific roles
- `mustUnderstand` attribute: with values true or false, indicating whether it is mandatory to process the header.
- `relay` attribute: forward header if not processed (SOAP 1.2)

Source: Gustavo Alonso

The SOAP body

The body is intended for the application-specific data

Unlike for headers, SOAP does specify the contents of some body entries:

- mapping of RPC to a collection of SOAP body entries
- The Fault entry (for reporting errors in processing a SOAP message), comprising five elements:
  - **code:** indicating the kind of error
  - **reason:** human readable explanation of the fault (not intended for automated processing)
  - **detail:** application specific information about the nature of the fault
  - **node:** URI of the node who created the fault (opt.)
  - **role:** role of the node who created the fault (opt.)

Source: Gustavo Alonso

Document-style vs. RPC-style message exchange

Technische Universität Berlin

Document-style

Payload does not indicate what to do with the message

Typically used for „conversation“-kind of interactions between sender and receiver using application-specific correlation ids in the header

RPC-style

Payload specifies exactly what to do with the message:
it contains the “method name” to be invoked

SOAP defines a convention for naming the root elements in the bodies of associated request and a reply messages:

- *Request body root element = „Request name“*
- *Response body root element = Request name + string „Response“*

SOAP example (data), header

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <m:transfer xmlns:m="http://bank.example.com/transfer"
                env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
                env:mustUnderstand="true">
      <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
      <m:dateAndTime>2001-11-29T13:20:00.000-05:00</m:dateAndTime>
    </m:transfer>
    <c:credentials xmlns:n="http://bank.example.com/accountmgmt/user"
                   env:role=http://www.w3.org/2003/05/soap-envelope/role/next
                   env:mustUnderstand="true">
      <n:username>richyrich</n:username>
      <n:password>b1gm$ney</n:password>
    </n:credentials>
  </env:Header>
  <env:Body>
    <p:participants xmlns:p="http://bank.example.com/transfer/role">
      <p:sender>
       ...
    </p:participants>
    <q:type xmlns:q="http://bank.example.com/transfer/type">
     ...
  </env:Body>
</env:Envelope>
```

SOAP example (data), body only

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Body>
                <p:participants xmlns:p="http://bank.example.com/transfer/role">
      <p:sender>
        <p:fullname>Jonathan Rich</p:fullname>
                              <p:iban>DE89370400440532013000</p:iban>
        <p:swift>BNKTBD99</p:swift>
        <p:location>Karlsruhe, Germany</p:location>
        <p:accountType>savings</p:accountType>
      </p:sender>
      <p:receiver>
                              <p:fullname>Herbert Needy</p:fullname>
                              <p:iban>LI2108810000232401 3AA</p:iban>
                                   <p:swift>FINIST00</p:swift>
                              <p:location>Liechtenstein</p:location>
                                        <p:accountType/>
      </p:receiver>
                                     </p:participants>
                  <q:type xmlns:q="http://bank.example.com/transfer/type">
      <q:priority>high</q:priority>
                                       </q:type>
  </env:Body>
</env:Envelope>
```

SOAP example (RPC)

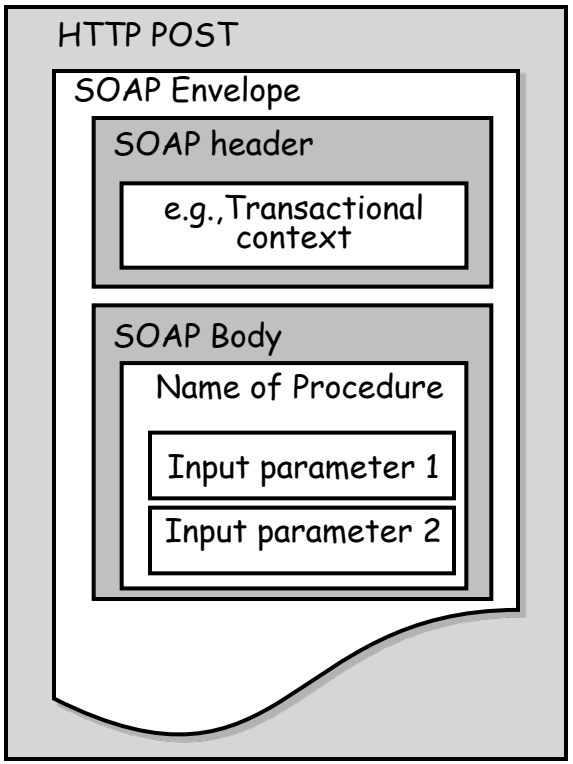
```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope”>
  <env:Header>
    <t:transaction xmlns:t="http://thirpartybank.example.com/transaction"
                   env:encodingStyle="http://bank.example.com/encoding"
                   env:mustUnderstand="true”>5</t:transaction>
  </env:Header>
  <env:Body>
    <m:performTransfer env:encodingStyle="http://www.w3.org/2003/05/soap-encoding"
                       xmlns:m="http://bank.example.com/">
      <m:transfer xmlns:m="http://bank.example.com/transfer">
        <m:code>FDS40FJL9230</m:code>
      </m:transfer>
      <o:account xmlns:o="http://bank.example.com/accountmgmt">
        <n:accountowner xmlns:n="http://bank.example.com/account">
                  Åke Jógvan Øyvind
        </n:accountowner>
        <o:accountid>0012030940</o:accountid>
      </o:account>
    </m:chargeReservation>
  </env:Body>
</env:Envelope>
```

SOAP and HTTP

A binding describes how a SOAP message is to be sent via a transport protocol

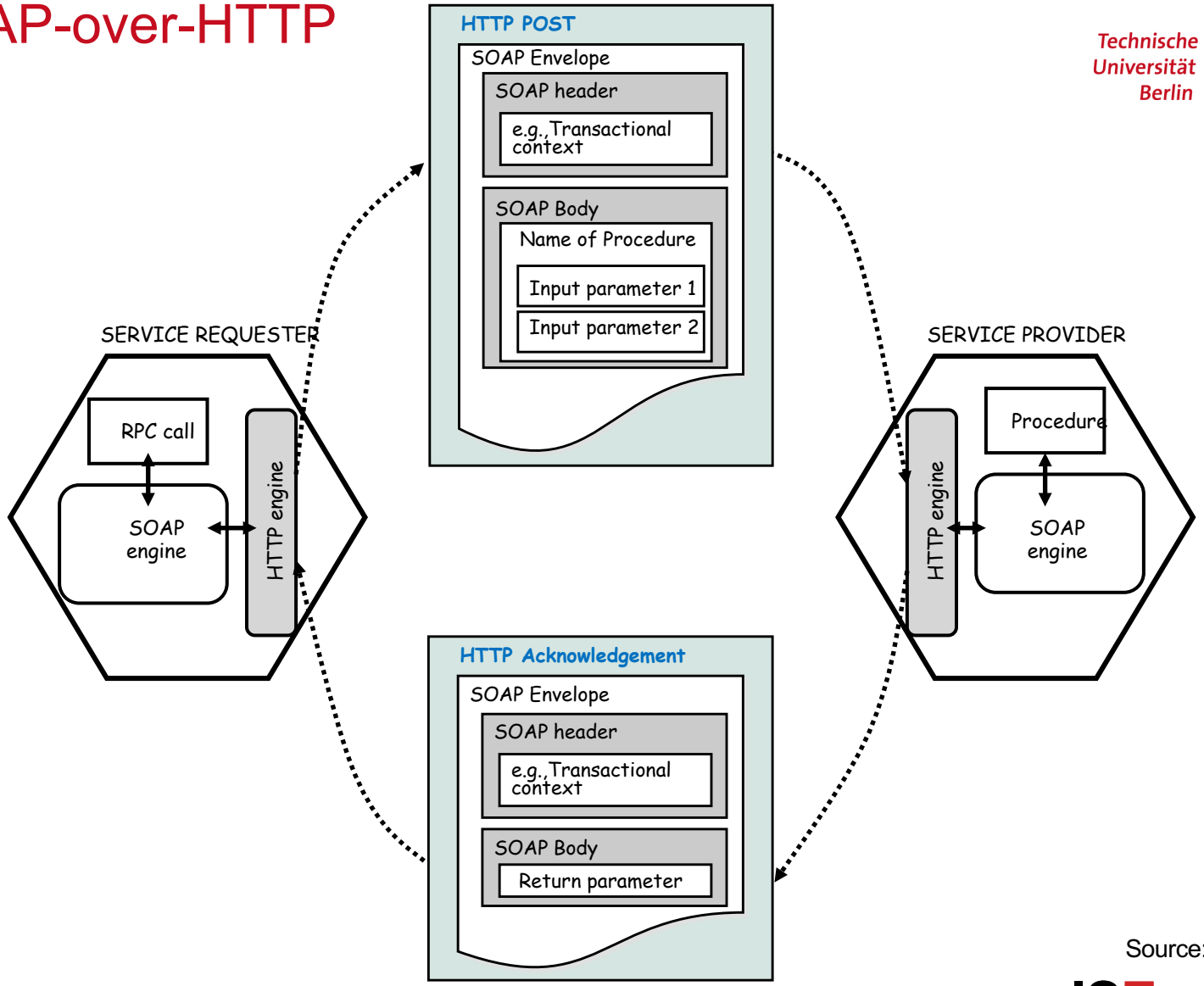
Common case: SOAP over HTTP

- SOAP can use GET or POST.
  - GET: only the response is a SOAP message
  - POST: both request and response are SOAP messages
- SOAP uses the same error and status codes as HTTP
  => HTTP responses can be directly interpreted by a SOAP module



Source: Gustavo Alonso

SOAP-over-HTTP



Source: Gustavo Alonso

Message processing: Implementing the Chain-of-Responsibility pattern

SOAP specifies in detail how messages must be processed (in particular, how header entries must be processed)

- Each SOAP node along the message path looks at the role associated with each part of the message
- There are three standard roles: none, next, or ultimateReceiver
- Applications can define their own roles and use them in the message
- The role determines who is responsible for each part of a message

If a block does not have a role associated to it, it defaults to ultimateReceiver

If a mustUnderstand flag is included, a node that matches the role specified must process that part of the message, otherwise it must generate a fault and do not forward the message any further

SOAP 1.2 includes a relay attribute. If present, a node that does not process that part of the message must forward it (i.e., it cannot remove the part)

The use of the relay attribute, combined with the role next, is useful for establishing persistence information along the message path (like session information)

Source: Gustavo Alonso

Recap: "Chain-of-responsibility" OO design pattern

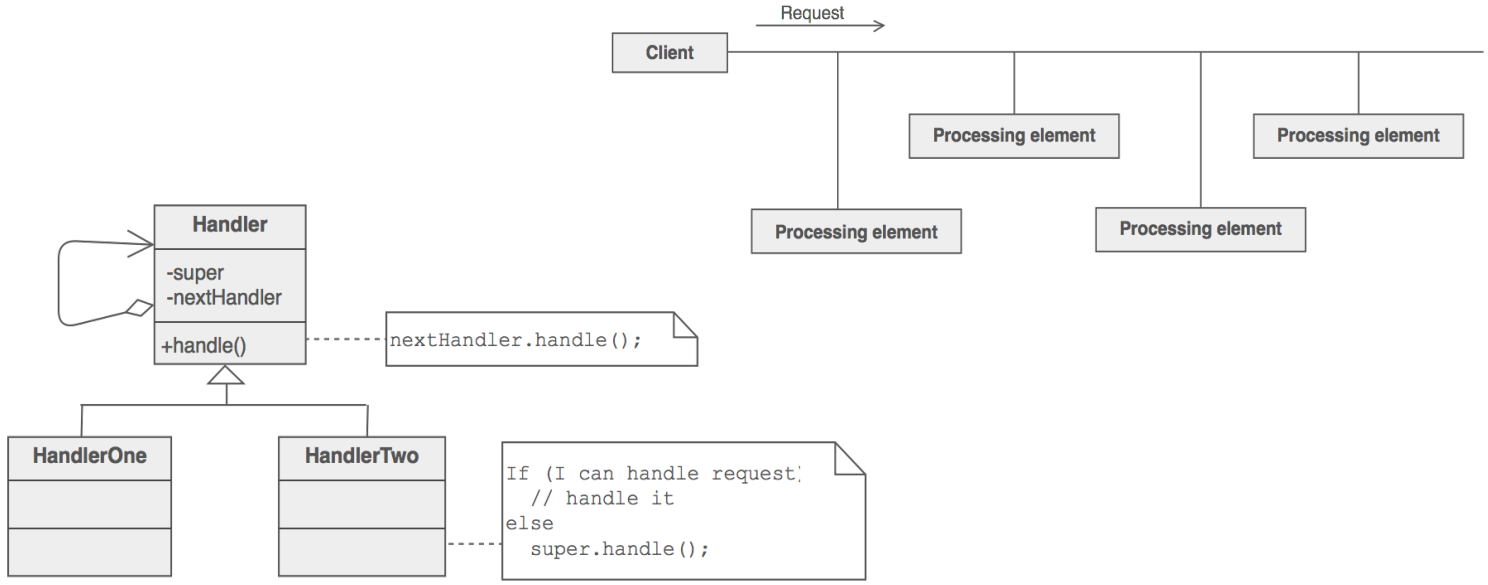
Intent:

- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Launch-and-leave requests with a single processing pipeline that contains many possible handlers.
- An object-oriented linked list with recursive traversal.

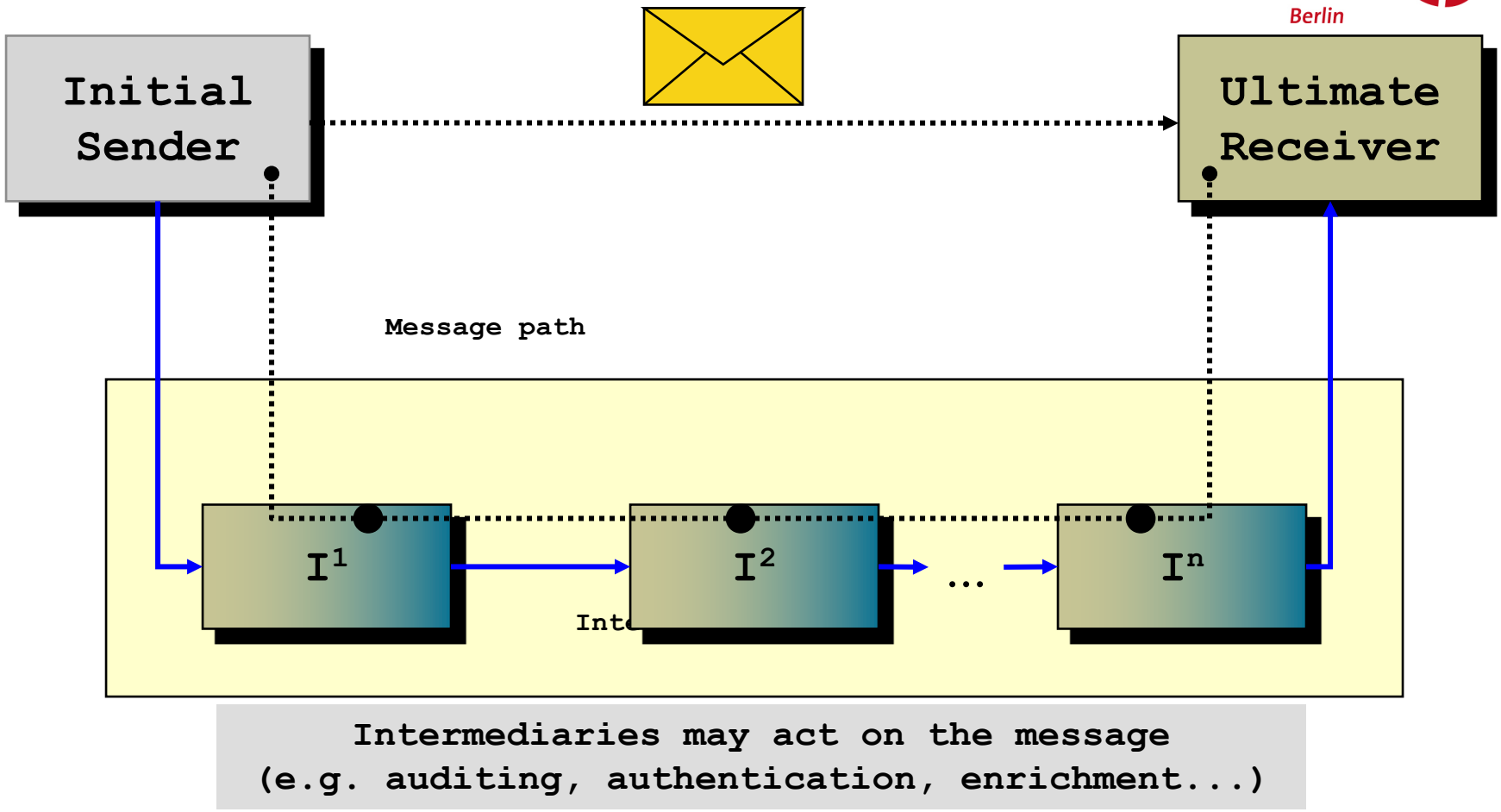
Problem:

- There is a potentially variable number of "handler" or "processing element" or "node" objects, and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.

Solution:



Fundamental message exchange mechanism



=> Chain of responsibility pattern

Source: Frank Leymann

SOAP summary

SOAP provides a basic mechanism for:

- encapsulating messages into an XML document
- mapping the XML document / SOAP message to an HTTP request
- transforming RPC calls into SOAP messages
- processing a SOAP message

SOAP takes advantage of the standardization of XML to resolve problems of data representation and serialization (it uses XML Schema to represent data and data structures, and it also relies on XML for serializing the data for transmission).

SOAP is a relatively simple protocol intended for transferring data from one middleware platform to another. In spite of its claims to be open (which are true), current specifications are very tied to RPC and HTTP.

Source: Gustavo Alonso

Technische Universität Berlin
ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

SOAP as a simple protocol

SOAP does not include anything about:

Reliability, complex message exchanges, transactions, security …

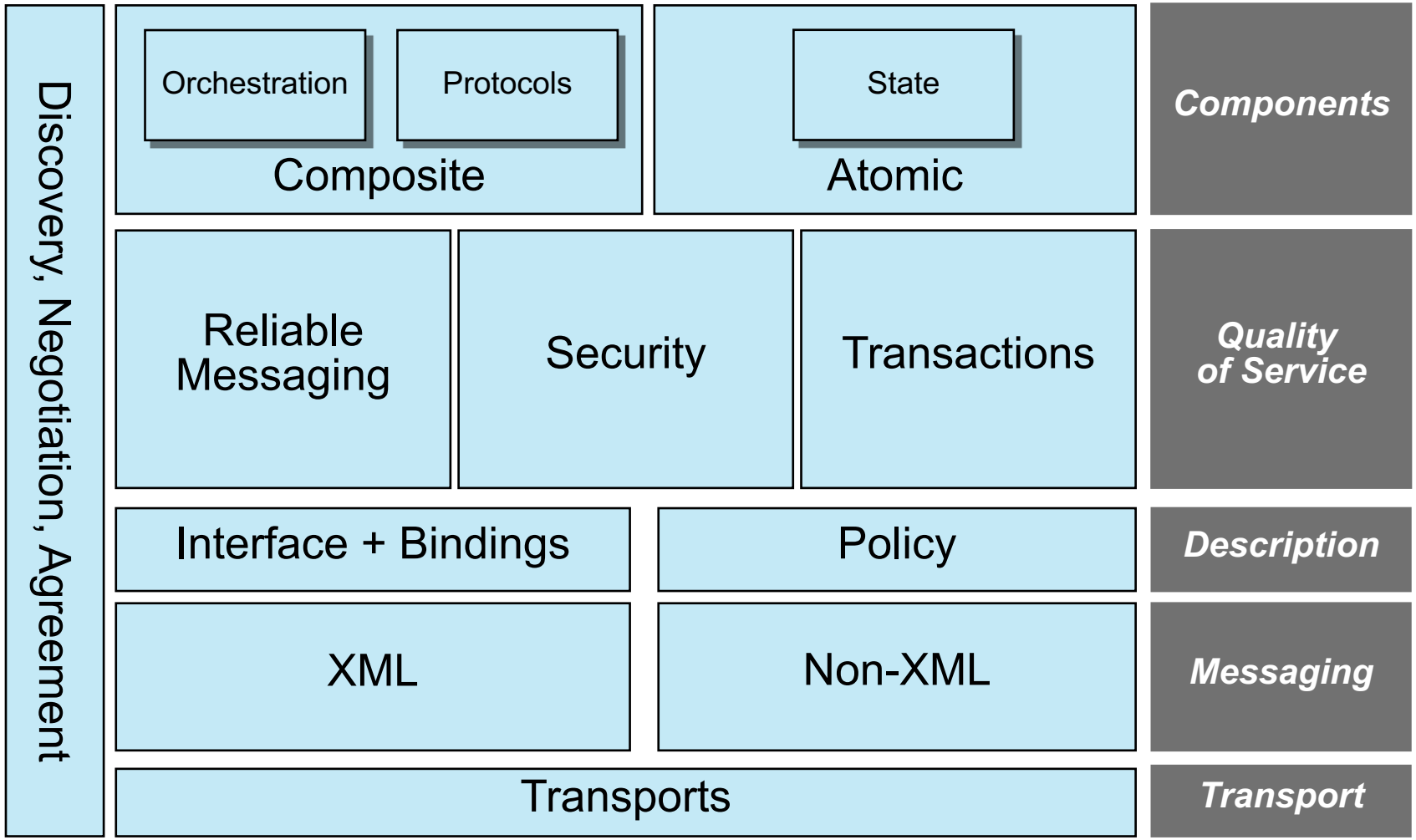
As such, it is not adequate by itself to implement industrial strength applications that incorporate typical middleware features like transactions or reliable delivery of messages

SOAP does not prevent such features but they need to be standardized to be useful in practice:

- WS-Security, WS-Coordination, WS-Transactions …
- A wealth of additional standards are being proposed to add the missing functionality usually referred to as "WS-*" specifications

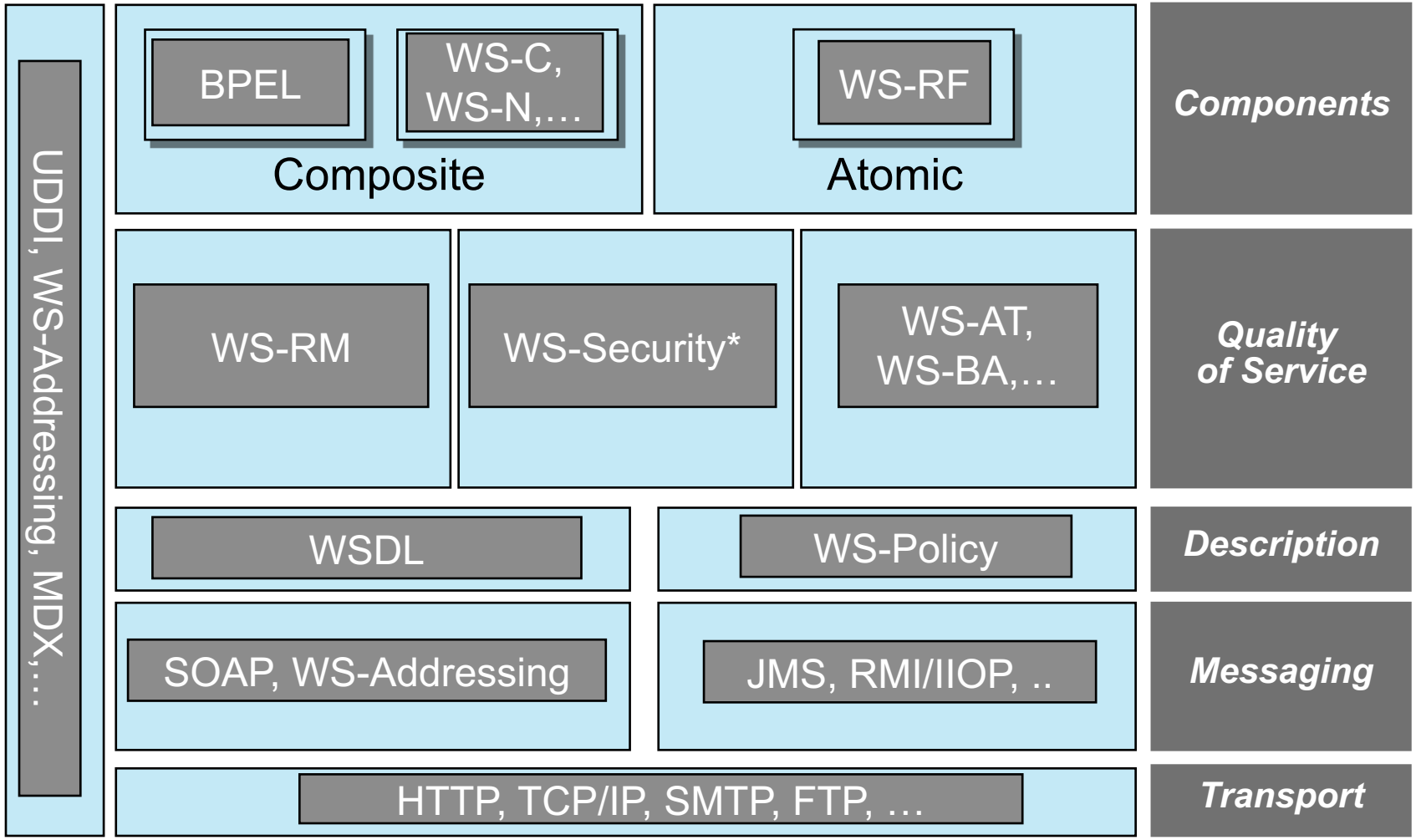
Source: Gustavo Alonso

The Web Services Platform Architecture (WS-*)



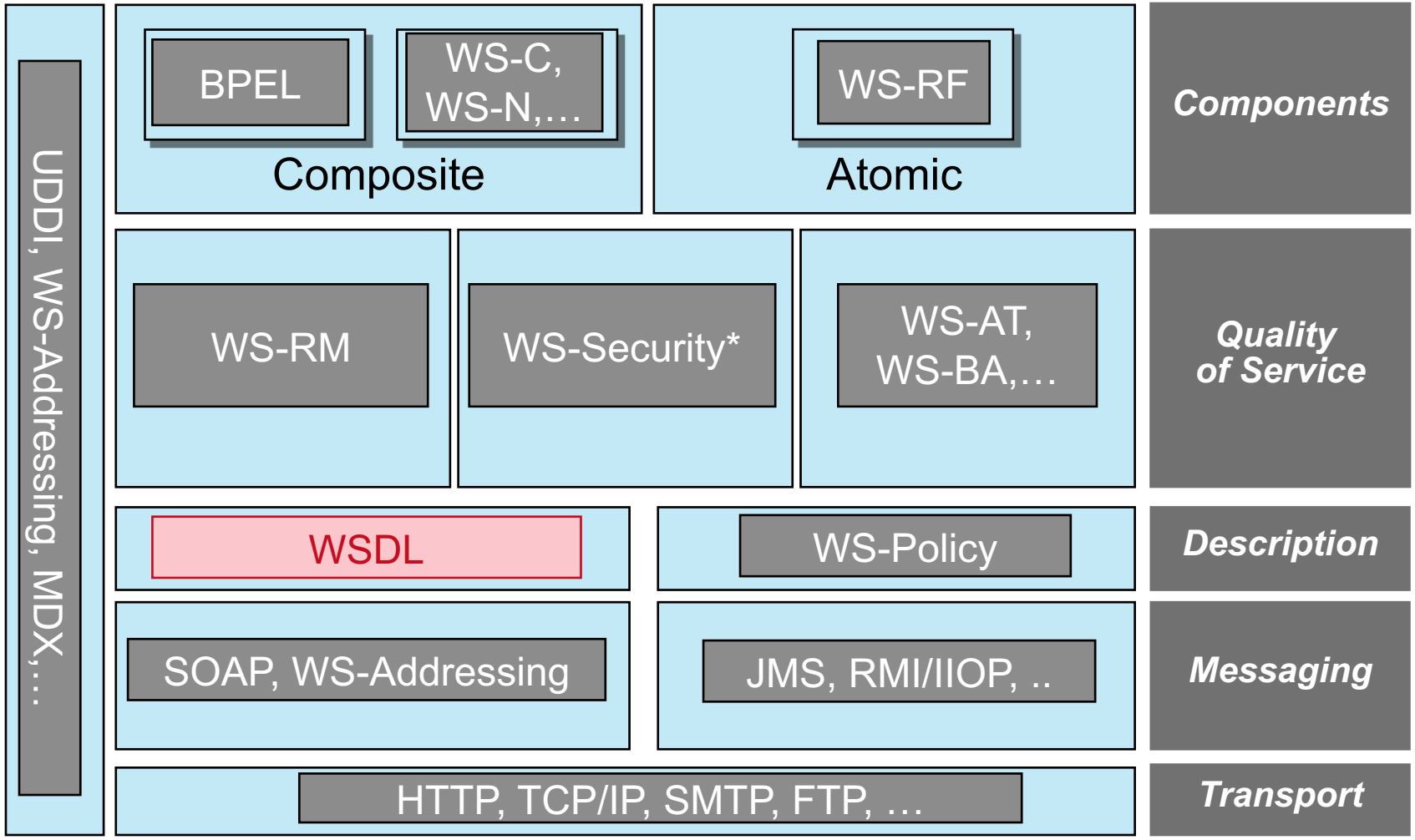
Source: Frank Leymann

The Web Services Platform Architecture (WS-*)



Source: Frank Leymann

The Web Services Platform Architecture (WS-*)



Source: Frank Leymann

Technische Universität Berlin

Recap: WSDL

Web Service Description Language (WSDL)

- XML-based
- Platform- and language-independent
- Describes a “contract“ between service provider and requester
- It contains information on
  - What a service does (the operations)
  - Where a service resides (an address)
  - How to invoke a service (data formats, protocols)

Structure of WSDL documents

- Two distinct sections (can be defined and reused separately:
  - Service interface definition: describes an abstract web service interface (operations, operation parameters)
  - Service implementation part: describes where and how the interface can be reached

- Using WSDL, a client knows how and where to invoke a web service

More on WSDL interface and implementation parts

Technische Universität Berlin

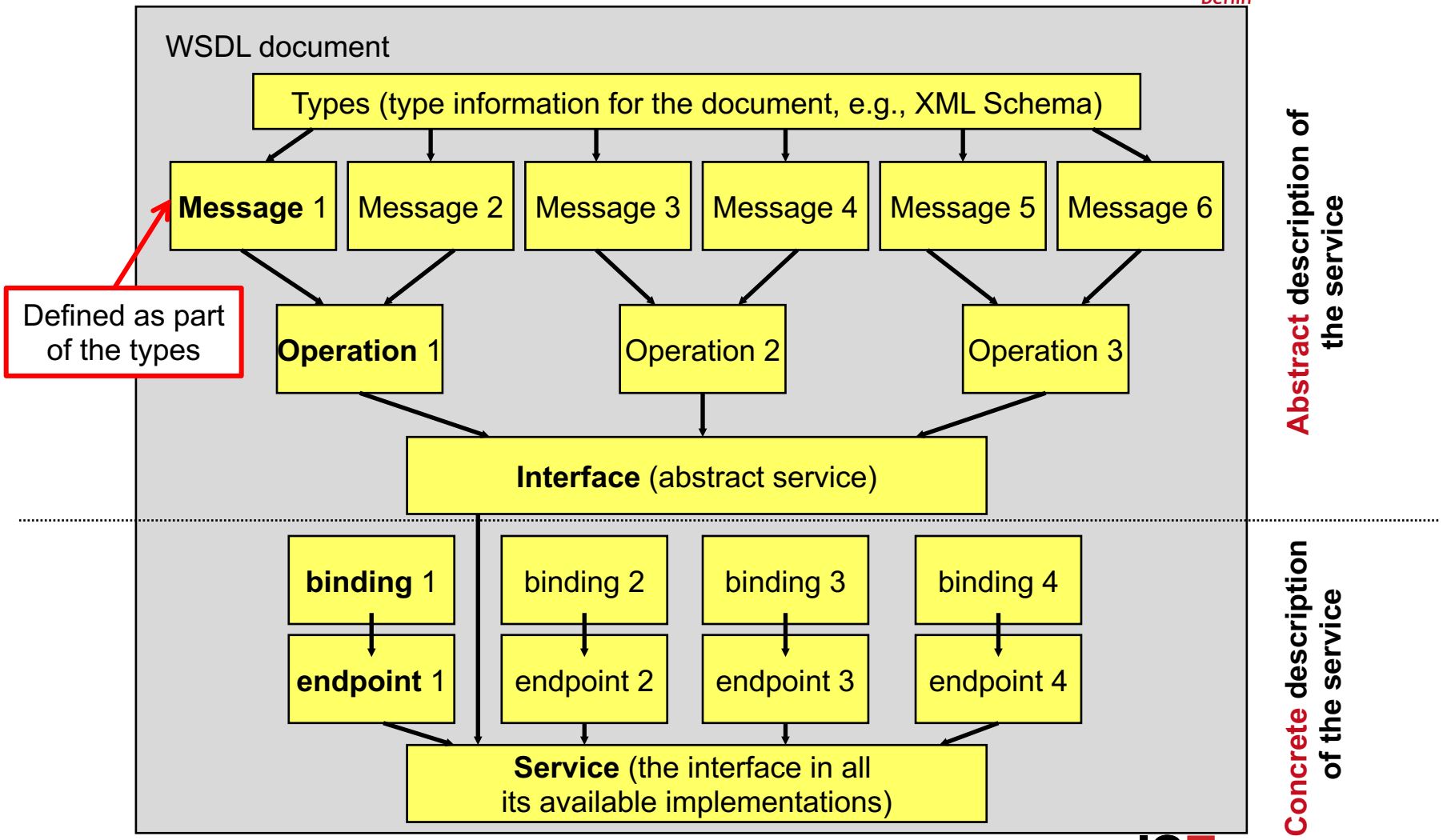
Abstract description

- types: describes messages and their payloads (based on XSD)
- operation: groups opt. input and output messages (~ Java method)
- interface: combines one or more operations that constitute an abstract service (~Java interface)

Concrete description

- binding: maps an interface to a transport protocol
- endpoint: network address of the binding
- service: collection of endpoints

Elements of WSDL



Example of WSDL Interface definition

```
<wsdl:definitions name="PurchaseOrderService"
    targetNamespace="http://supply.com/PurchaseService/wsdl"
    xmlns:tns="http://supply.com/ PurchaseService/wsdl"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

  <wsdl:types>
    <xsd:schema
      targetNamespace="http://supply.com/PurchaseService/wsdl"
       <xsd:complexType name="POType">
           <xsd:sequence>
               <xsd:element name="PONumber" type="integer"/>
               <xsd:element name="PODate" type="string"/>
                <xsd:element name="CusName" type="xsd:string"/>
               <xsd:element name="CusAddress" type="xsd:string"/>
           </xsd:sequence>
       </xsd:complexType>
       <xsd:complexType name="InvoiceType">
           <xsd:all>
               <xsd:element name="InvPrice" type="float"/>
               <xsd:element name="InvDate" type="string"/>
           </xsd:all>
       </xsd:complexType>

       <xs:element name="POMessage" type="POType"/>
       <xs:element name="InvMessage" type="InvoiceType"/>

    </xsd:schema>
  </wsdl:types>

<wsdl:interface name="PurchaseOrderInterface">
       <wsdl:operation name="SendPurchase">
           <wsdl:input message="tns:POMessage"/>
           <wsdl:output message="tns:InvMessage"/>
       </wsdl:operation>
  </wsdl:interface>

</wsdl:definitions>
```

Abstract data type and message definitions

Data that is sent

Data that is returned

interface with one operation

An operation with request (input) & response (output) message

Example of WSDL Implementation

```
<wsdl:definitions> . …
   <import namespace="http://supply.com/PurchaseService/wsdl"
    location="http://supply.com PurchaseService/wsdl/PurchaseOrder-interface.wsdl"/>

   <wsdl:binding name="PurchaseOrderSOAPBinding"  interface="tns:PurchaseOrderInterface">
     <soap:binding style="rpc“ protocol="http://schemas.xmlsoap.org/soap/http/"/>

     <wsdl:operation name="SendPurchase">
       <soap:operation soapAction="http://supply.com/ PurchaseService/wsdl/ SendPurchase" style="rpc"/>
         <wsdl:input>
           <soapbind:body  use="literal" namespace="http://supply.com/PurchaseService/wsdl"/>
         </wsdl:input>
         <wsdl:output>
           <soapbind:body  use=“literal"  namespace="http://supply.com/ PurchaseService/wsdl"/>
         </wsdl:output>
     </wsdl:operation>

 </wsdl:binding>

 <wsdl:service name=“PurchaseOrderService">

    <wsdl:endpoint name=“PurchaseOrderEndpoint" binding="tns:PurchaseOrderSOAPBinding">
      <soap:address location="http://supply.com/POS"/>
   </wsdl:endpoint>

 </wsdl:service>

</wsdl:definitions>
```

Specifies RPC style and HTTP as protocol

Maps the operation

Service name

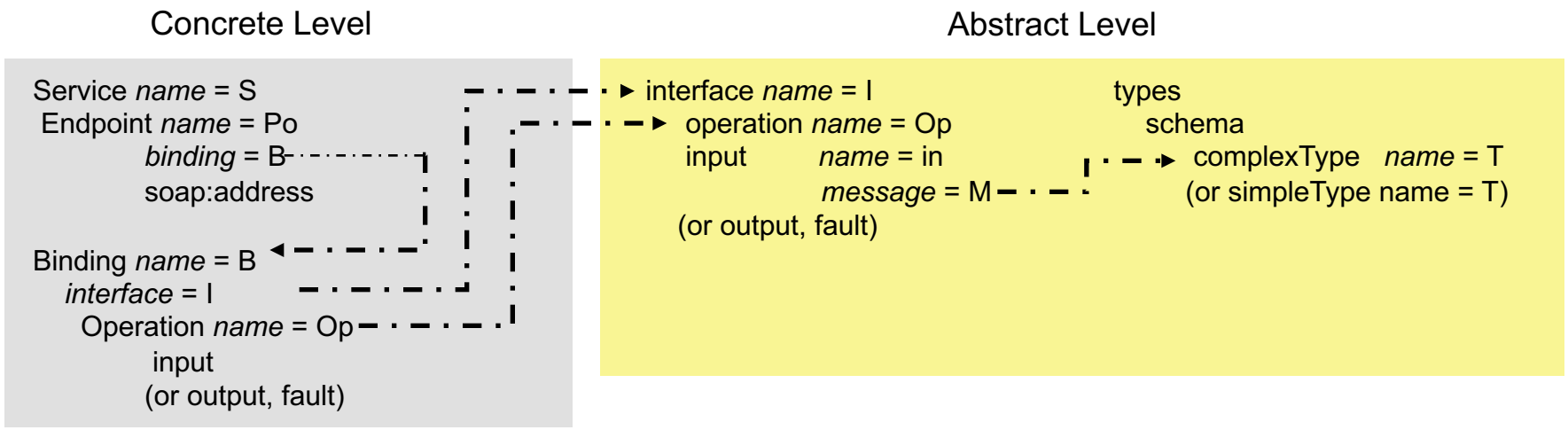
Network address of service

An RPC-style SendPurchase SOAP message

```
<?xml version= "1.0"  encoding= "UTF-8" ?>
<soap:Envelope
xmlns:soapbind="http://schemas.xmlsoap.org/soap/envelope"
   xmlns:tns="http://supply.com/PurchaseService/wsdl ">
      <soap:Body>
          <tns:SendPurchase>
            <POtype>
              <PONumber>223451</PONumber>
              <PODate>10/28/2004</PODate>
              <CusName>John Doe</CusName>
              <CusAddress>Far, far away</CusAddress>
            </POtype>
          </tns:SendPurchase>
      </soap:Body>
</soap:Envelope>
```

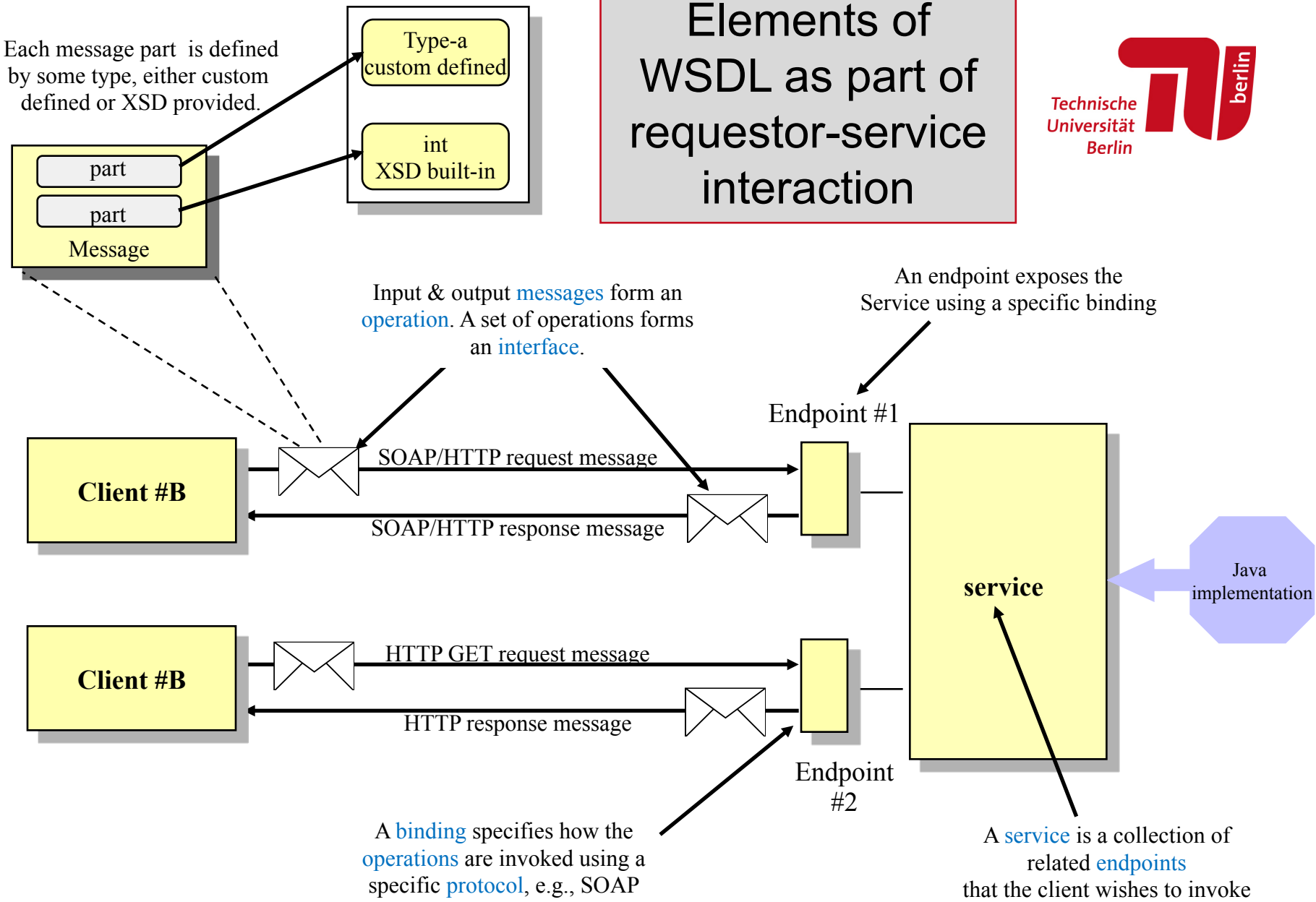
Source: M. Papazoglou

Connecting the abstract & implementation (concrete) levels of a Web service



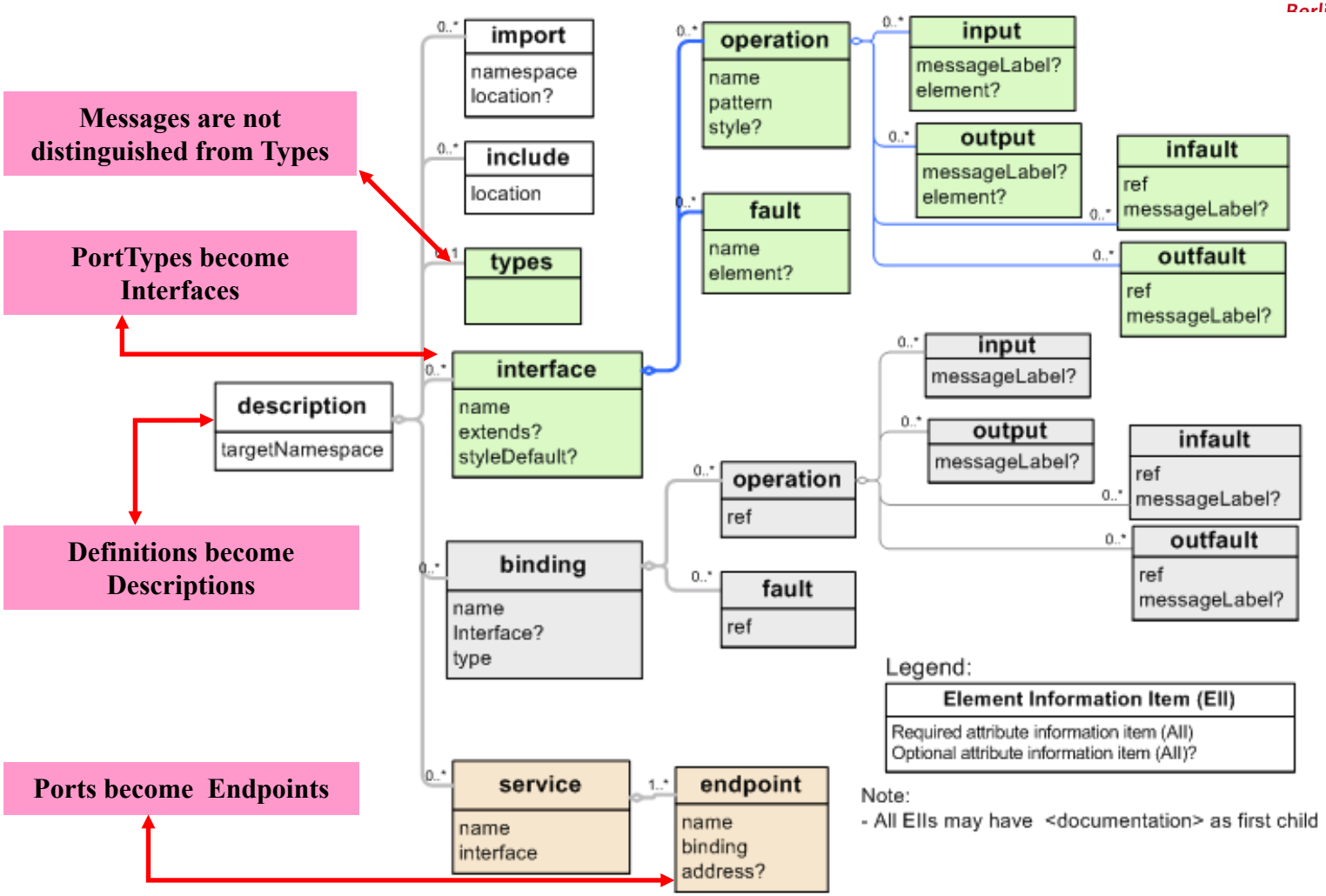
Source: M. Papazoglou

Elements of WSDL as part of requestor-service interaction



Source: M. Papazoglou

WSDL 2.0 data model



http://www.w3.org/TR/wsdl20-primer/#wsdl-infoset-diagram

Recap: WS-*

Example: WS-RM

The WS-* set of specifications

The variety of specifications associated with SOAP/WSDL Web services are collectively referred to as "WS-*"

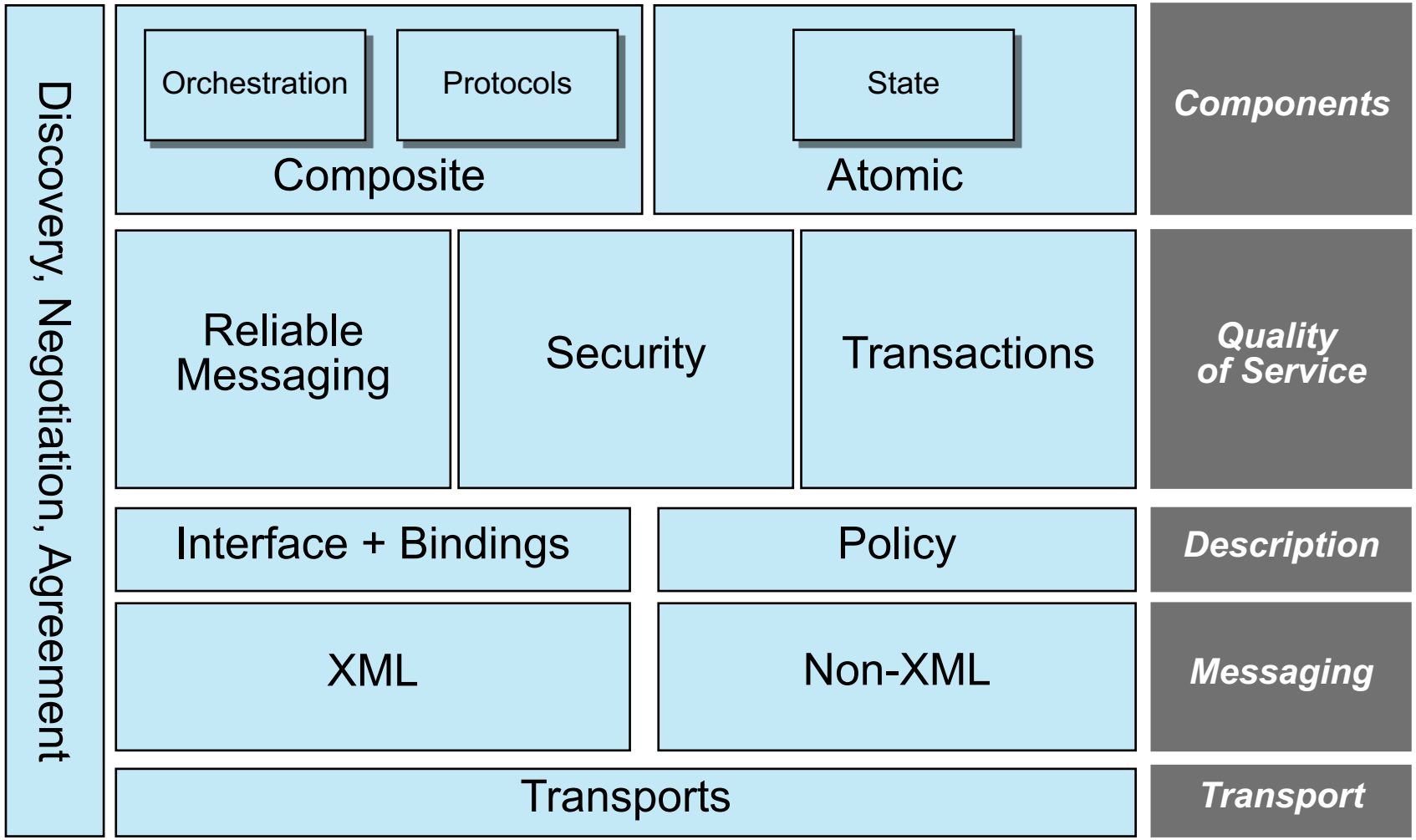
"WS-"is a prefix used to indicate specifications associated with Web Services and there exist many such specifications, authored and managed by different standardization bodies (such as W3C, OASIS, WS-I, and other)

WS-* specifications include WS-Addressing, WS-Policy, WS-ReliableMessaging, WS-Coordination, and many other

Specifications authored by different bodies may complement, overlap, and compete with each other.

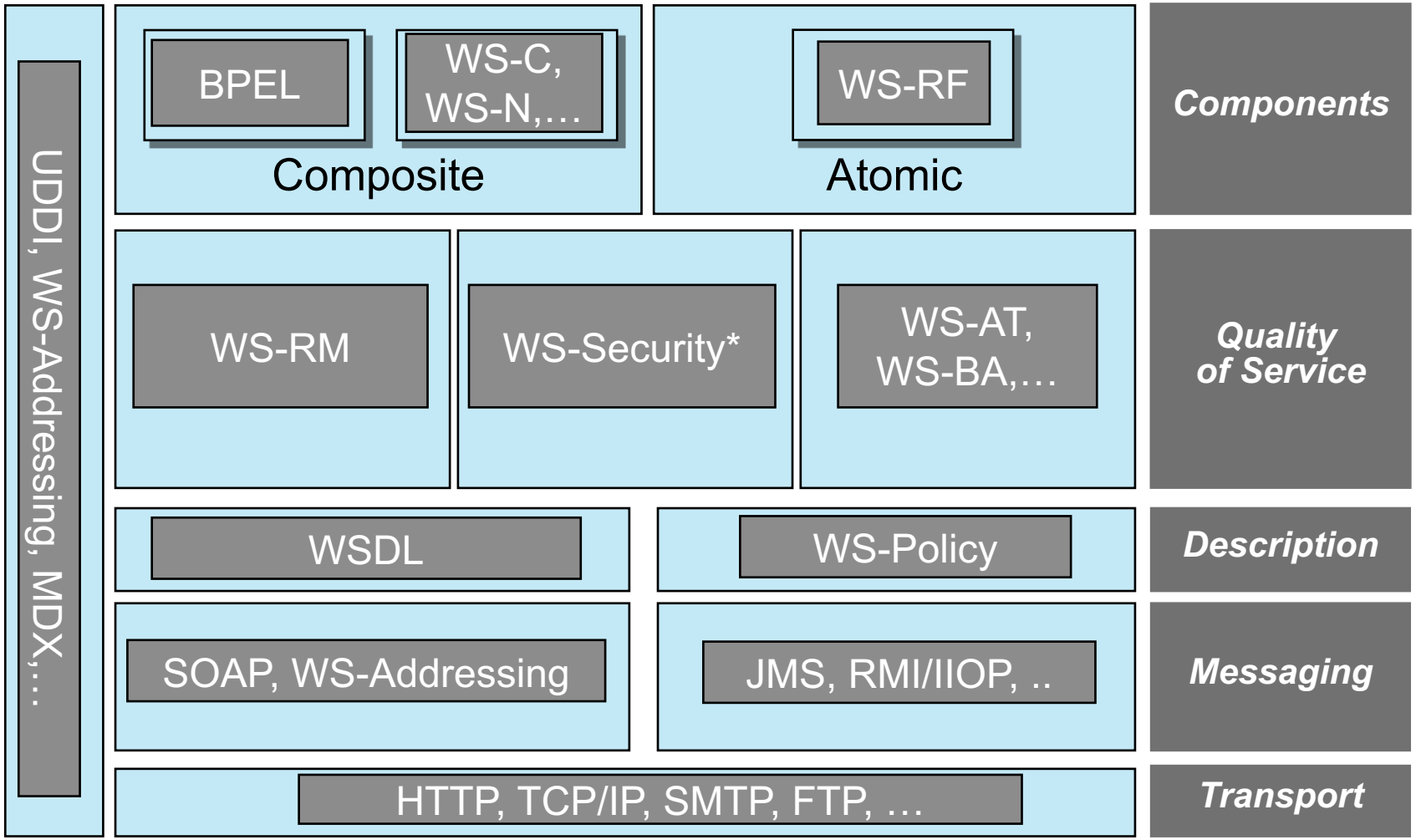
WS-* specifications authored by one author group (IBM/Microsoft in the early 2000s) were intended to be composable without specification overlap

The Web Services Platform Architecture (WS-*)



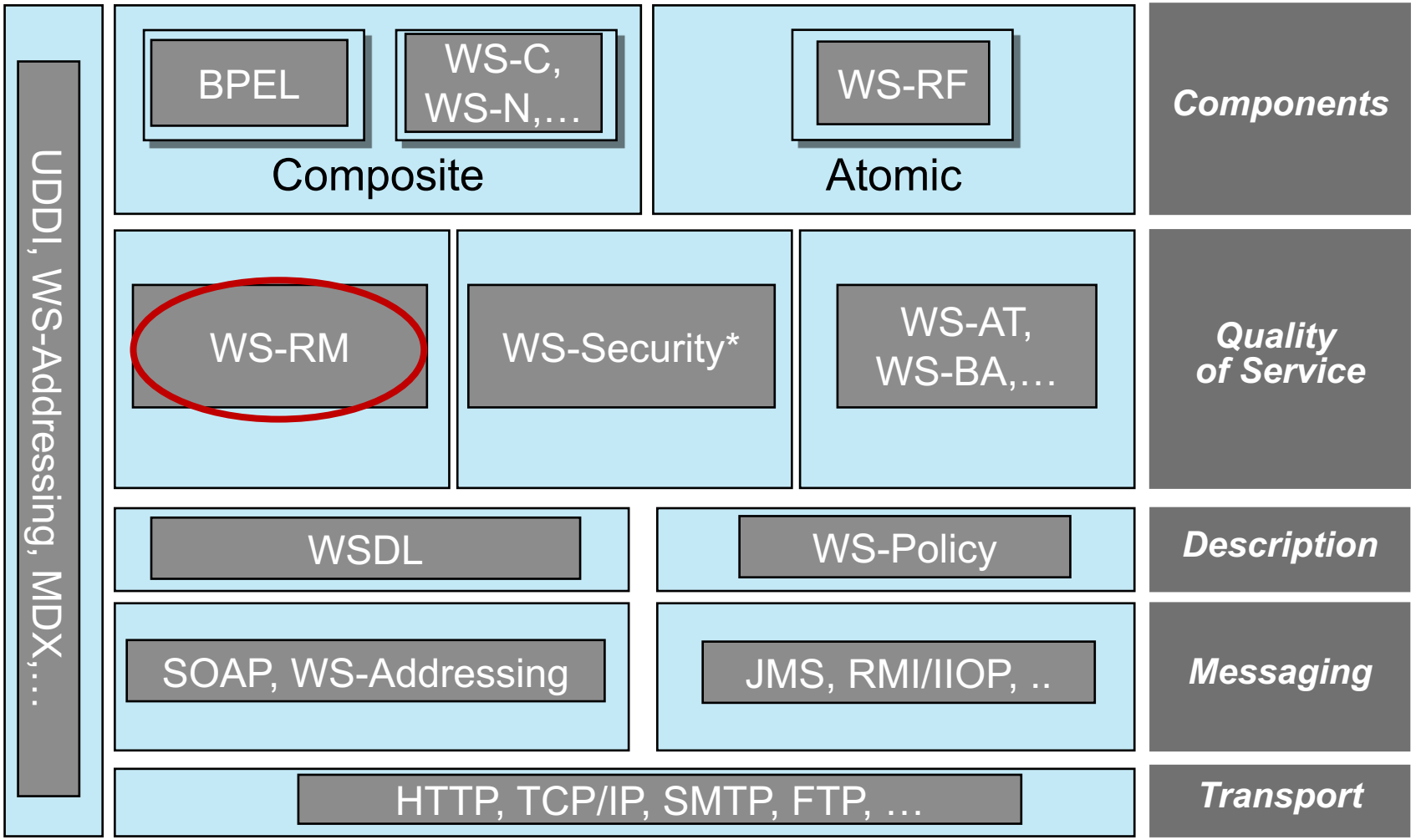
Source: Frank Leymann

The Web Services Platform Architecture (WS-*)



Source: Frank Leymann

The Web Services Platform Architecture (WS-*)



Message reliability

Technische Universität Berlin

Reliable messaging describes a protocol that allows SOAP messages to be delivered reliably between distributed applications in the presence of software component, system, or network failures

Typical reliable messaging guarantees include:

1. Both the sender and recipient of a message must know whether or not a message was actually sent and received, and that the message received is the same as the one sent
2. Make sure that the message was sent once and only once to the intended recipient
3. Guarantee that the received messages are in the same order as they were sent

Guarantee #1

"Both the sender and recipient of a message must know whether or not a message was actually sent and received, and that the message received is the same as the one sent"

How can we ensure this?

Solution

- Scope all messages that are sent from a source to a destination
- Provide each message a unique identifier in the context of that scope
- Work with acknowledgments that indicate a successful transfer

Guarantee #2

"Make sure that the message was sent once and only once to the intended recipient"

Any ideas how to ensure this?

Solution

- We need a middleware system that takes the responsibility for „once and only once“ delivery
- The middleware system persists the message to be sent and delivers the message in a way that ensures exactly once delivery

Guarantee #3

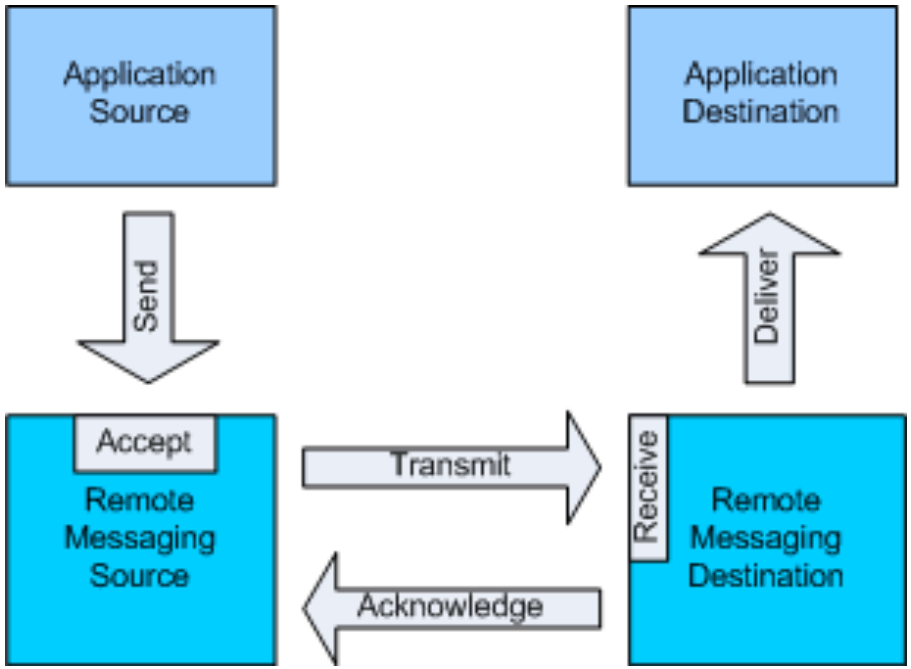
„Guarantee that the received messages are in the same order as they were sent“

How do we do that?

Solution

- Ordering meta-data is required: Turn again to the concept of a scope and unique message identifiers within that scope
- Make sure the RM middleware system sitting between the application source and application destination delivers messages in order

Basic reliable mesaging model



Delivery assurances

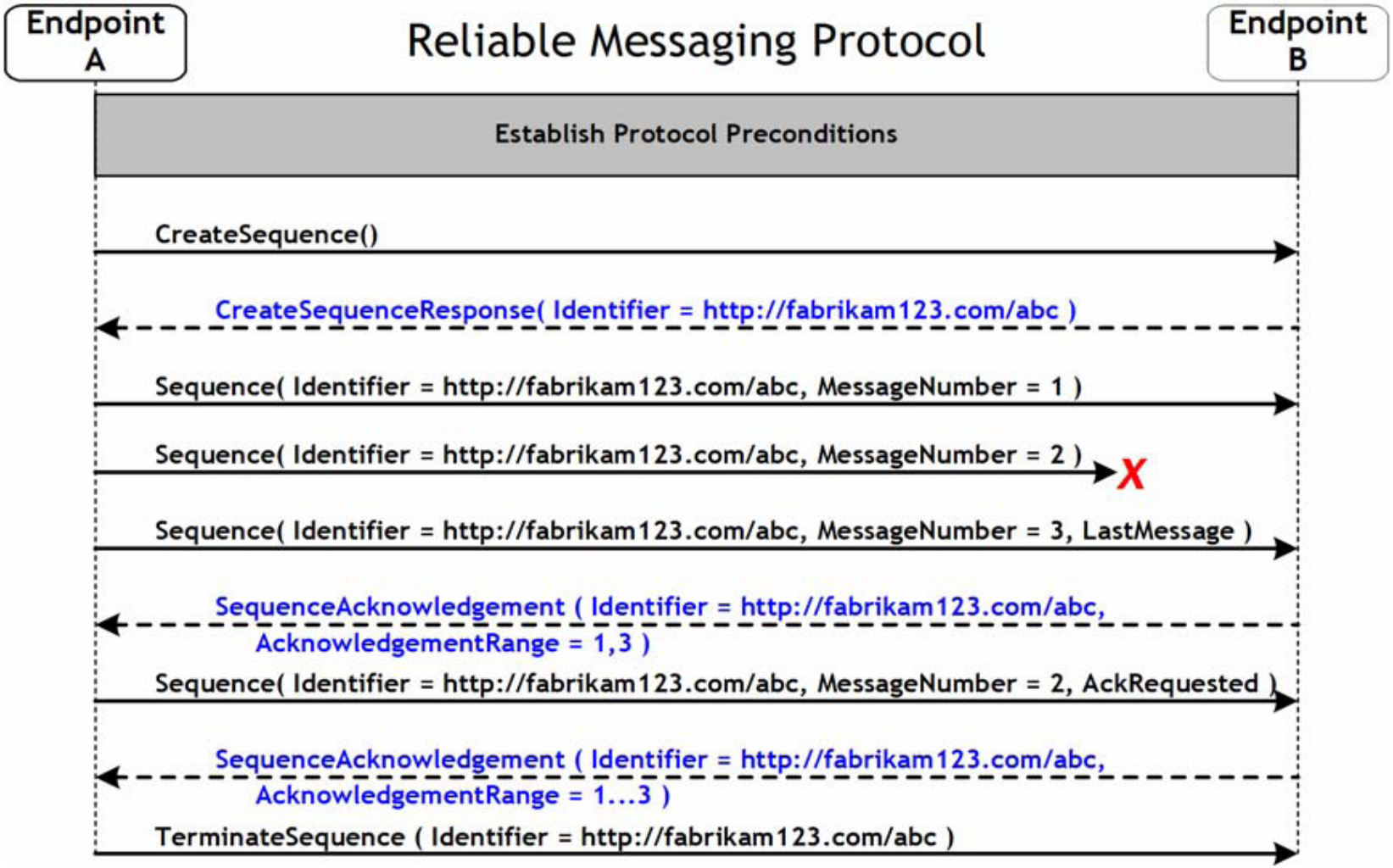
AtLeastOnce: Each message will be delivered to the AD at least once. If a message cannot be delivered, an error must be raised by the RMS and/or the RMD. Messages may be delivered to the AD more than once (i.e. the AD may get duplicate messages).

AtMostOnce: Each message will be delivered to the AD at most once. Messages may not be delivered to the AD, but the AD will never get duplicate messages.

ExactlyOnce: Each message will be delivered to the AD exactly once. If a message cannot be delivered, an error must by raised by the RMS and/or the RMD. The AD will never get duplicate messages.

InOrder: Messages will be delivered from the RMD to the AD in the order that they are sent from the AS to the RMS. This assurance can be combined with any of the above assurances.

Sample message exchange between RMS and RMD



WS-RM Key Concepts

Sequences: The messages sent from the source to the destination are scoped using a sequence

- Sequences are distinguished using a unique identifier (a URI) for each sequence
- It is important to note that the term sequence does not imply any processing order

Message Numbers: Every message has a unique identifier in the context of the sequence

- This identifier is a monotonically increasing integer number, starting with 1 and increasing by exactly 1 for each message
- In a typical implementation the numbering would be performed "under the covers" by the messaging middleware, and the applications sending the messages would not be aware of it
- This numbering scheme makes it simple to detect missing or duplicate messages, and simplifies acknowledgement generation and processing

Source: Microsoft

WS-RM Key Concepts

Technische Universität Berlin

Acknowledgements: An acknowledgement is an indication that a message was successfully transferred to the destination

- Messages are acknowledged using acknowledgement ranges. For example, acknowledging 1 through 4 and 6 through 13 will indicate that messages number 1 through 13 were received, with the exception of message number 5.
- An acknowledgement does not necessarily indicate that the message was processed. Indication of successful processing, or the reporting of processing errors, requires a separate, higher-level protocol. This protocol will often be application-specific.

Persistence: Message persistence (durability) considerations do not affect the wire protocol and are not addressed by WS-RM; WS-RM ensures transfer, not processing.

- Persistence requirements have to do with the storing of the message on the destination until it is processed, and are thus the responsibility of the RM middleware implementation.

Source: Microsoft

WS-RM: Who does what?

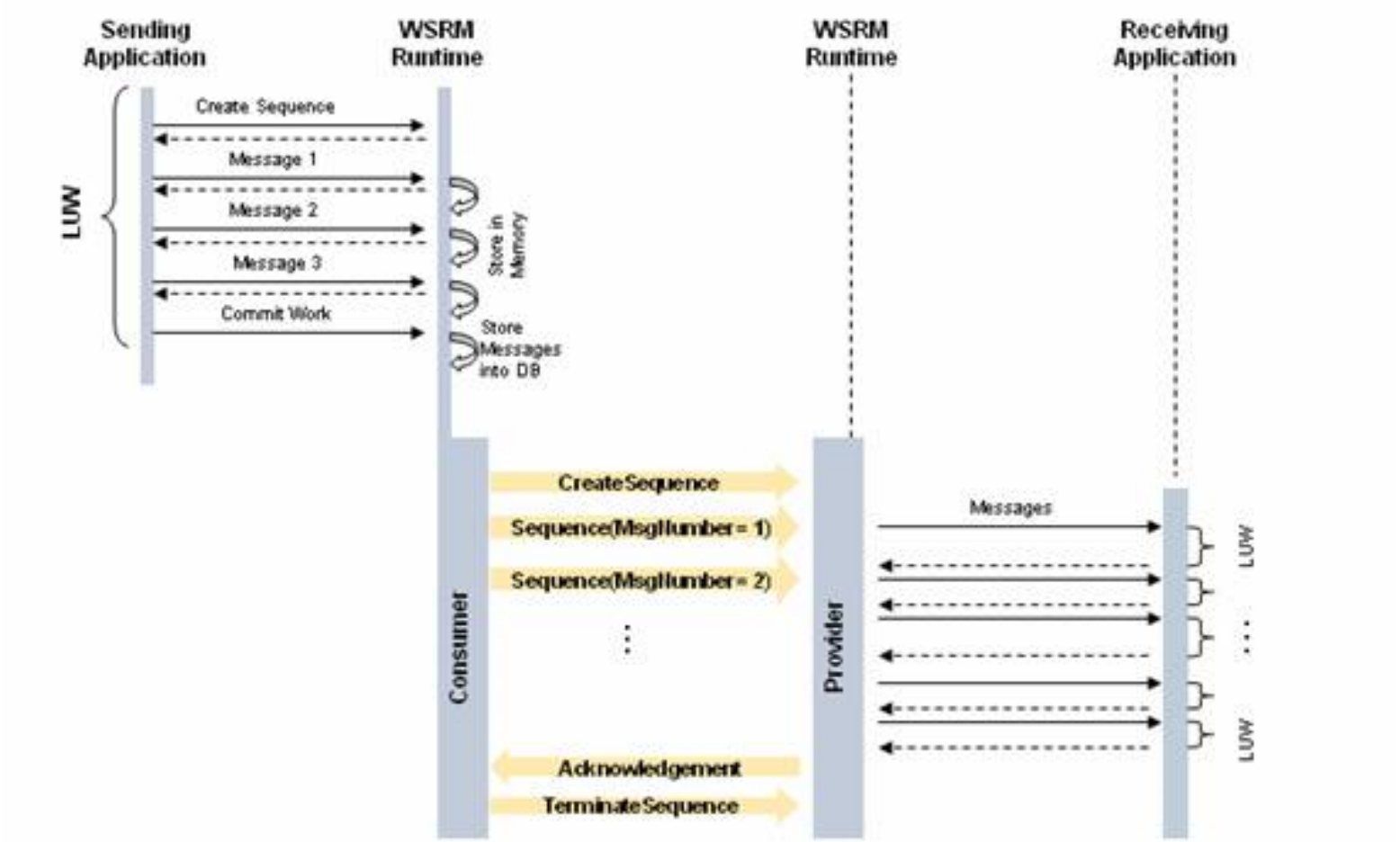


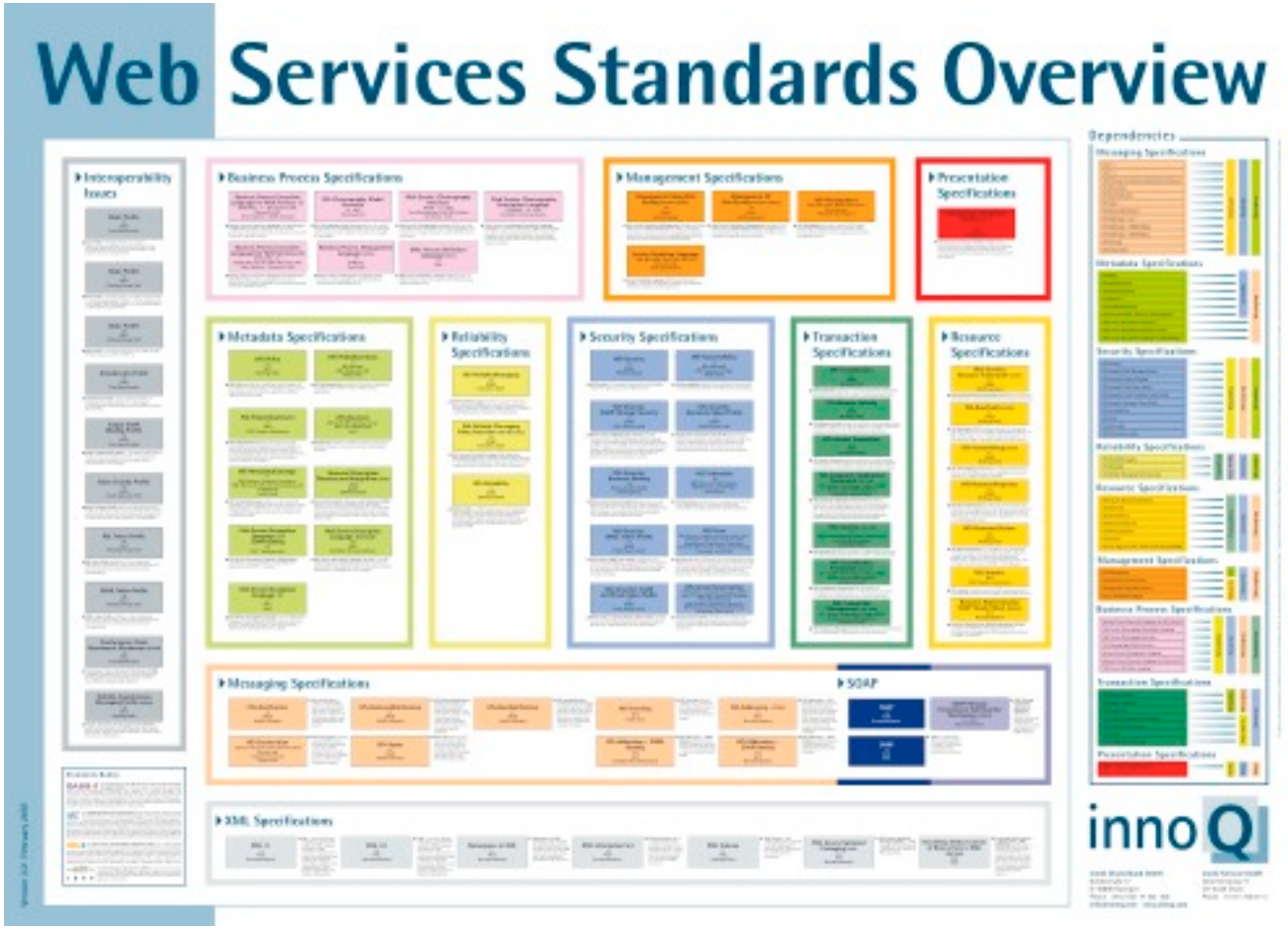
Figure Source: SAP

WS-ReliableMessaging examples

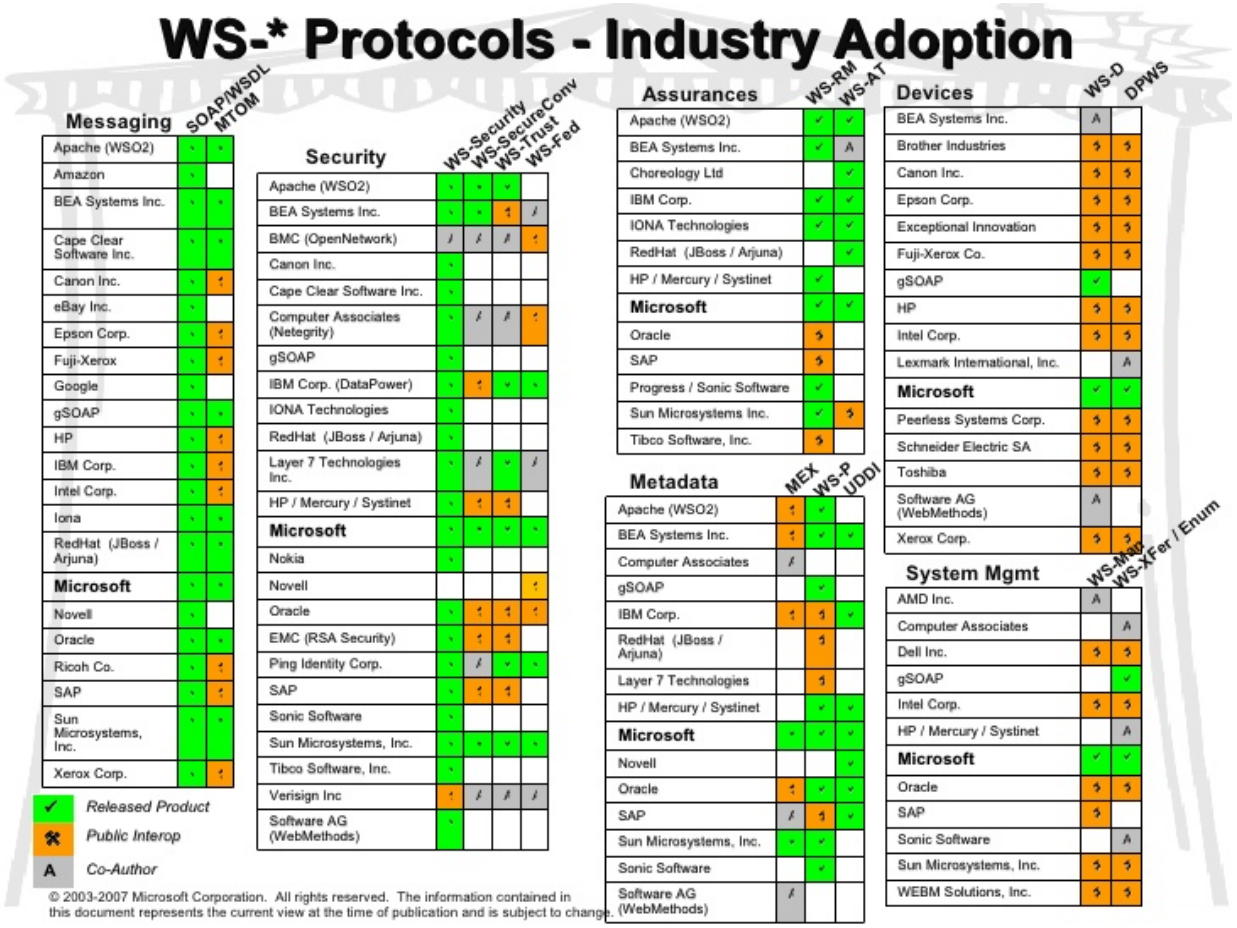
```
<Soap:Envelope
 xmlns:Soap="http://www.w3.org/2003/05/soap-envelope"
 xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm"
 xmlns:wsa="http://www.w3.org/2004/12/addressing">
 <Soap:Header>
    ...
  <wsrm:Sequence>
    <wsrm:Identifier> xs:anyURI </wsrm:Identifier>
      ...
    <wsrm:MessageNumber> 3 </wsrm:MessageNumber>
    <wsrm:LastMessage/>
      ...
  </wsrm:Sequence>

 </Soap:Header>
 <Soap:Body>
   <GetOrder xmlns="http://supply.com/orderservice"
     ...
   </GetOrder>
 </Soap:Body>
</Soap:Envelope>
```

```
<Soap:Envelope
 xmlns:Soap="http://www.w3.org/2003/05/soap-envelope"
 xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm"
 xmlns:wsa="http://www.w3.org/2004/12/addressing">
 <Soap:Header>
    ...
  <wsrm:SequenceAcknowledgement>
    <wsrm:Identifier>  http://supply.com/abc </wsrm:Identifier>
    <wsrm:AcknowledgementRange Upper="2" Lower="1"/>
    <wsrm:AcknowledgementRange Upper="4" Lower="4"/>
    <wsrm:Nack> 3 </wsrm:Nack>
  </wsrm:SequenceAcknowledgement>
  <Soap:Body>
    ...
  </Soap:Body>
</Soap:Envelope>
```

Many WS-* specifications were proposed (ca. 2000-2008)



...with varying degree of industry adoption



One of the key guiding principles: Feature Composability

Technische Universität Berlin

Despite the decreasing popularity of WS-* specifications, there are good principles associated with WS-*. One key guiding principle is that

…all WS-* specifications 'should be' governed by the criteria that

- Each WS-* specification addresses one specific concern and has a value in its own right, independently of any other specification
- All WS-* specifications (by the same authors) are designed to work seamlessly in conjunction with each other

The complexity of a solution (number of WS-* specs used) is a direct consequence of the specific application problem being addressed

The basic Web service specifications (WSDL and SOAP) have been designed to support such feature composition inherently

SOAP's Multi-part Message Structure

```
1  <S:Envelope...>
2    <S:Header>
3      <wsa:ReplyTo>                                                    } WS-Addressing
4        <wsa:Address>http://business456.com/User12</wsa:Address>       }
5      </wsa:ReplyTo>                                                   }
6      <wsa:TO>HTTP://Fabrikam123.com/Traffic</wsa:To>                  }
7      <wsa:Action>http://Fabrikam123.com/Traffic/Status</wsa:Action>   }
8      <wssec:security>
9        <wssec:BinarySecurityToken                                     } WS-Security
10           ValueType="wssec:x509v3"                                   }
11           EncodingType="wssec:Base64Binary"                          }
12           dXJcY3TnYHB....Ujmi8eMTaW                                  }
13       </wssec:BinarySecurityToken                                    }
14     </wssec:Security
15       <wsrm:Sequence>                                                } WS-Reliable Messaging
16           <wsu:Identifier>http://Fabrikam123.com/seq1234</wsu:Identifier>  }
17           <wsrm:MessageNumber>10</wsrm:MessageNumber>                }
18       </wsrm:Sequence>
19 </S:Header>
20 </S:Body
   •   <app:TrafficStatus
   •     xmlns:env="http//highwaymon.org/payloads">
23           <road>520W</road>
24           <speed>3mph</speed>
25   </app:TrafficStatus>
26  </S:Body>
27 </S:Envelope>
```

WS-* in practice, versus REST – 2016

SOAP/WSDL still in use in many enterprise systems

WS-* with industry adoption to varying degrees

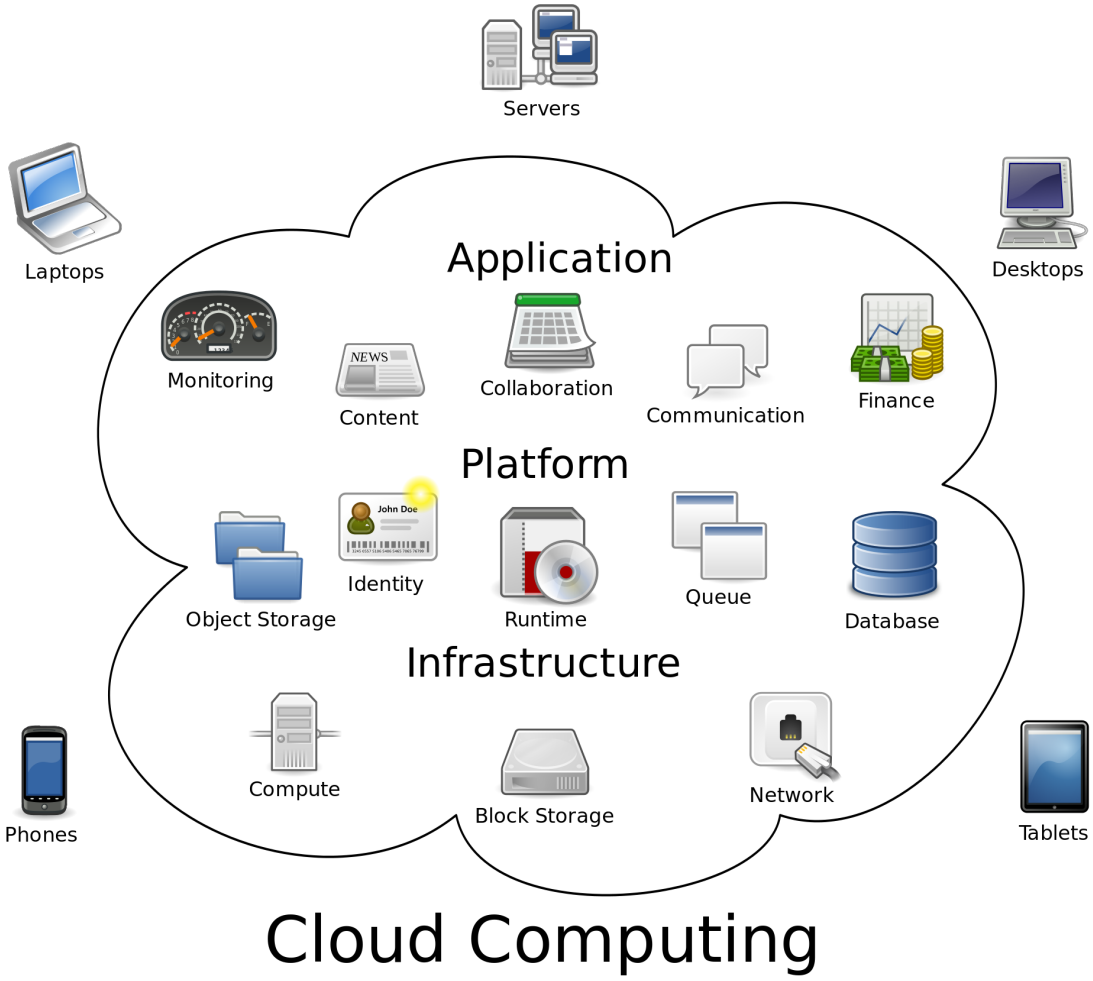
REST by far the more popular model; common engineering practice

Still,
WS-* encodes many proven, basic enterprise computing principles

Agenda

1. Web Engineering Fundamentals
    a. HTTP
    b. REST
    c. SOAP
    d. WSDL
    e. WS-*
2. Cloud Fundamentals
    a. Sample Cloud Services: AWS
    b. Design Principles
3. Microservices
    a. DevOps, CI/CD
    b. Microservices and Server-less Architectures

What is Cloud Computing?



The NIST Definition *http://csrc.nist.gov/groups/SNS/cloud-computing/*

Technische Universität Berlin

"Cloud computing is a model for enabling:

…ubiquitous, convenient, on-demand network access to

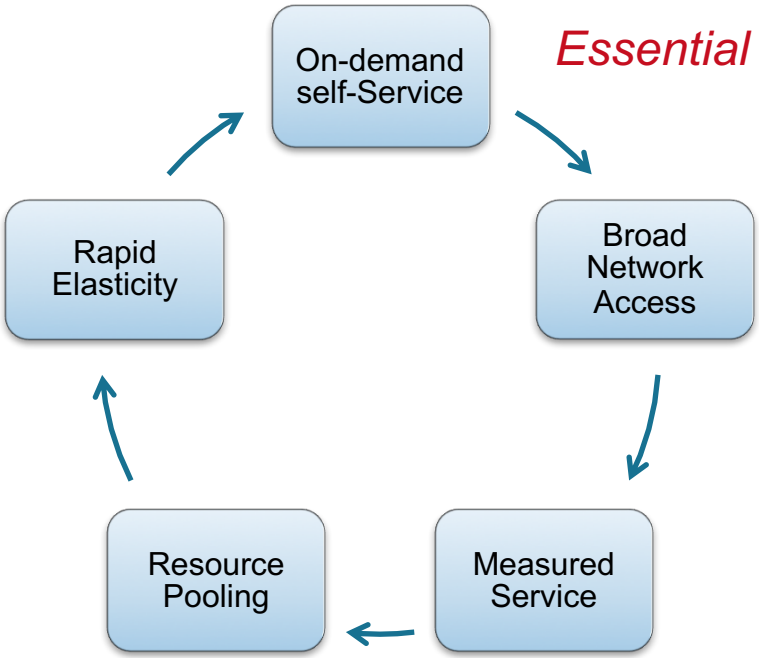
…a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services)

…that can be rapidly provisioned and released with minimal management effort or service provider interaction.

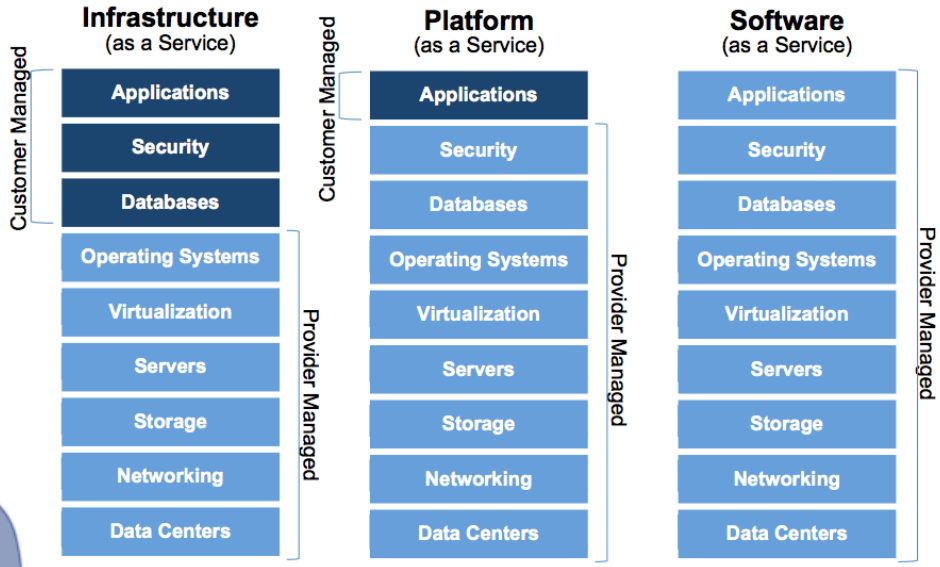
This cloud model is composed of

- five essential characteristics
- three service models
- four deployment models."

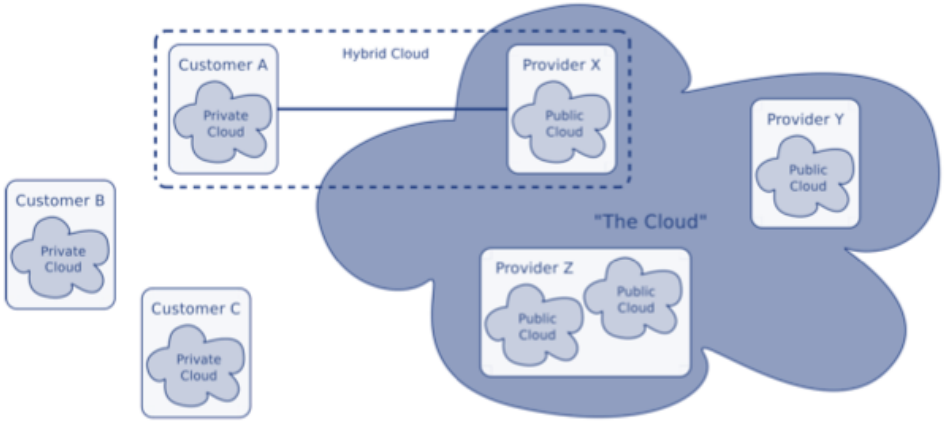
Essential Characteristics



Service Models



Deployment Models



Example: Amazon Web Services (AWS)

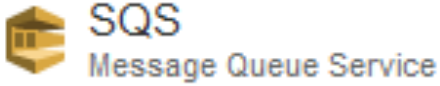
aws.amazon.com

AWS consists of many cloud services that can be used in combination, depending on specific business or organizational needs

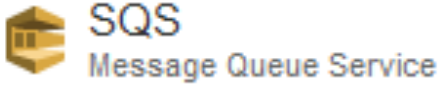
AWS services fall into the following categories, among other:

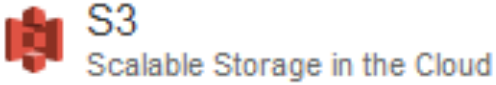
- compute
- networking
- storage and content delivery
- databases
- analytics
- application services
- deployment and management

AWS offerings in Nov ‘04



AWS offerings in Aug ‘06





AWS offerings in Dec ‘10

AWS offerings in April ‘15

Technische Universität Berlin

Amazon Web Services

Compute

- **EC2** — Virtual Servers in the Cloud
- **Lambda** — Run Code in Response to Events
- **EC2 Container Service** — Run and Manage Docker Containers

Storage & Content Delivery

- **S3** — Scalable Storage in the Cloud
- **Storage Gateway** — Integrates On-Premises IT Environments with Cloud Storage
- **Glacier** — Archive Storage in the Cloud
- **CloudFront** — Global Content Delivery Network

Database

- **RDS** — MySQL, Postgres, Oracle, SQL Server, and Amazon Aurora
- **DynamoDB** — Predictable and Scalable NoSQL Data Store
- **ElastiCache** — In-Memory Cache
- **Redshift** — Managed Petabyte-Scale Data Warehouse Service

Networking

- **VPC** — Isolated Cloud Resources
- **Direct Connect** — Dedicated Network Connection to AWS
- **Route 53** — Scalable DNS and Domain Name Registration

Administration & Security

- **Directory Service** — Managed Directories in the Cloud
- **Identity & Access Management** — Access Control and Key Management
- **Trusted Advisor** — AWS Cloud Optimization Expert
- **CloudTrail** — User Activity and Change Tracking
- **Config** — Resource Configurations and Inventory
- **CloudWatch** — Resource and Application Monitoring

Deployment & Management

- **Elastic Beanstalk** — AWS Application Container
- **OpsWorks** — DevOps Application Management Service
- **CloudFormation** — Templated AWS Resource Creation
- **CodeDeploy** — Automated Deployments

Analytics

- **EMR** — Managed Hadoop Framework
- **Kinesis** — Real-time Processing of Streaming Big Data
- **Data Pipeline** — Orchestration for Data-Driven Workflows
- **Machine Learning** — Build Smart Applications Quickly and Easily

Application Services

- **SQS** — Message Queue Service
- **SWF** — Workflow Service for Coordinating Application Components
- **AppStream** — Low Latency Application Streaming
- **Elastic Transcoder** — Easy-to-use Scalable Media Transcoding
- **SES** — Email Sending Service
- **CloudSearch** — Managed Search Service

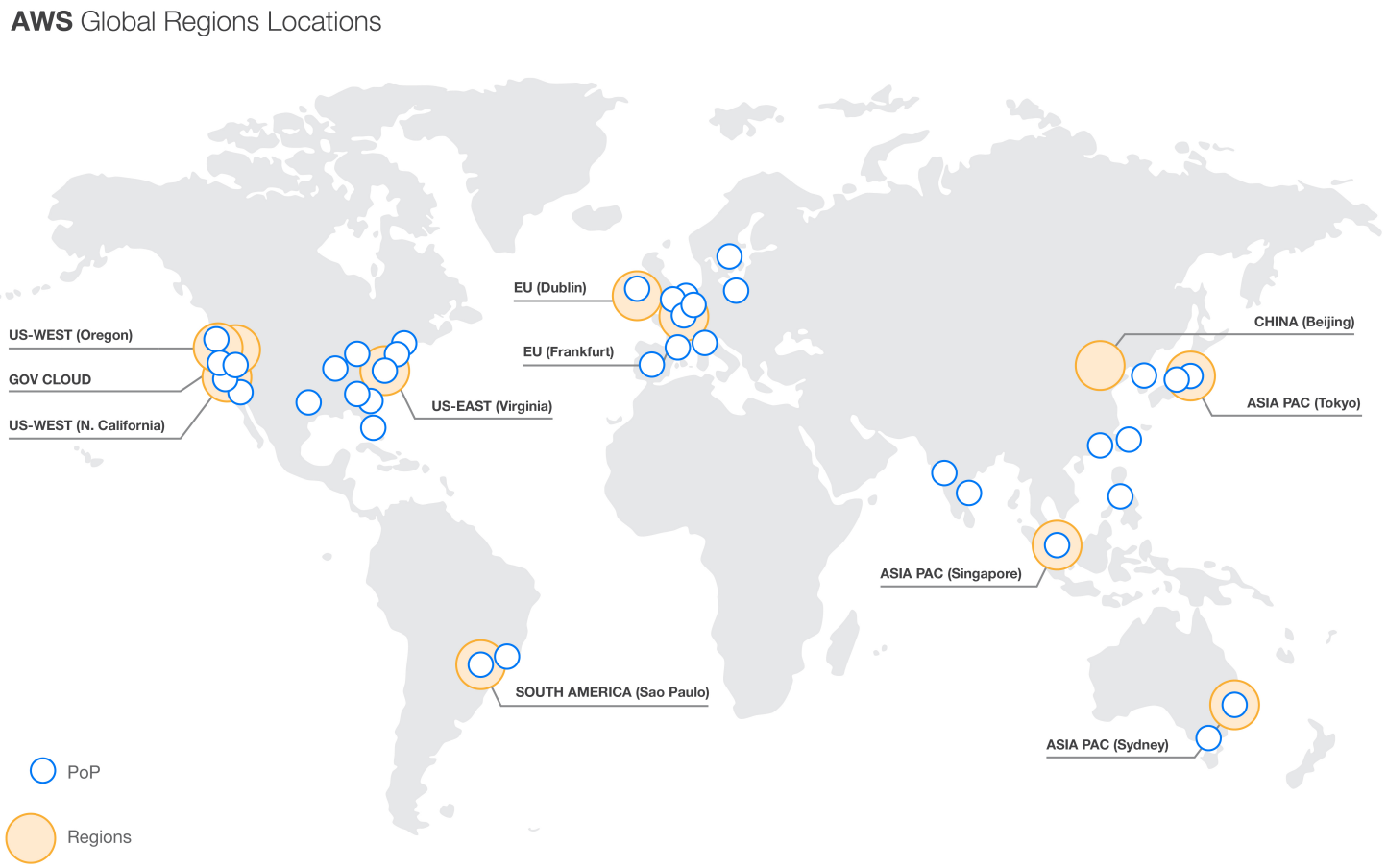
Mobile Services

- **Cognito** — User Identity and App Data Synchronization
- **Mobile Analytics** — Understand App Usage Data at Scale
- **SNS** — Push Notification Service

Enterprise Applications

- **WorkSpaces** — Desktops in the Cloud
- **WorkDocs** — Secure Enterprise Storage and Sharing Service
- **WorkMail** PREVIEW — Secure Email and Calendaring Service

AWS Global Infrastructure – Regions, Edge Locations, Availability Zones



AWS Compute Services

Technische Universität Berlin

Amazon Web Services

Compute
- **EC2** — Virtual Servers in the Cloud
- **Lambda** — Run Code in Response to Events
- **EC2 Container Service** — Run and Manage Docker Containers

Storage & Content Delivery
- **S3** — Scalable Storage in the Cloud
- **Storage Gateway** — Integrates On-Premises IT Environments with Cloud Storage
- **Glacier** — Archive Storage in the Cloud
- **CloudFront** — Global Content Delivery Network

Database
- **RDS** — MySQL, Postgres, Oracle, SQL Server, and Amazon Aurora
- **DynamoDB** — Predictable and Scalable NoSQL Data Store
- **ElastiCache** — In-Memory Cache
- **Redshift** — Managed Petabyte-Scale Data Warehouse Service

Networking
- **VPC** — Isolated Cloud Resources
- **Direct Connect** — Dedicated Network Connection to AWS
- **Route 53** — Scalable DNS and Domain Name Registration

Administration & Security
- **Directory Service** — Managed Directories in the Cloud
- **Identity & Access Management** — Access Control and Key Management
- **Trusted Advisor** — AWS Cloud Optimization Expert
- **CloudTrail** — User Activity and Change Tracking
- **Config** — Resource Configurations and Inventory
- **CloudWatch** — Resource and Application Monitoring

Deployment & Management
- **Elastic Beanstalk** — AWS Application Container
- **OpsWorks** — DevOps Application Management Service
- **CloudFormation** — Templated AWS Resource Creation
- **CodeDeploy** — Automated Deployments

Analytics
- **EMR** — Managed Hadoop Framework
- **Kinesis** — Real-time Processing of Streaming Big Data
- **Data Pipeline** — Orchestration for Data-Driven Workflows
- **Machine Learning** — Build Smart Applications Quickly and Easily

Application Services
- **SQS** — Message Queue Service
- **SWF** — Workflow Service for Coordinating Application Components
- **AppStream** — Low Latency Application Streaming
- **Elastic Transcoder** — Easy-to-use Scalable Media Transcoding
- **SES** — Email Sending Service
- **CloudSearch** — Managed Search Service

Mobile Services
- **Cognito** — User Identity and App Data Synchronization
- **Mobile Analytics** — Understand App Usage Data at Scale
- **SNS** — Push Notification Service

Enterprise Applications
- **WorkSpaces** — Desktops in the Cloud
- **WorkDocs** — Secure Enterprise Storage and Sharing Service
- **WorkMail** PREVIEW — Secure Email and Calendaring Service

AWS EC2: Resizable compute capacity

Technische Universität Berlin

Obtain and boot server instances, called EC2 instances,
using Web service APIs (or using the AWS Management Console)

There are multiple instance types, OS, and software packages
Memory, CPU, instance storage and boot partition size can be configured

Amazon has SLA commitments of 99.95% availability for each region

Virtual Private Clouds (VPC) can be set-up to keep instances private or to expose them to the Internet
Further, single-customer Dedicated Instances can be provisioned
For regular instances, security groups and network ACLs are used to control inbound and outbound network access

EC2 Instance Purchasing Options

Technische Universität Berlin

On-demand instances – compute capacity by the hour without long-term commitments

Reserved Instances (light, medium, heavy utilization) – one-time payment per instance upfront for significant discounts on the hourly charge

Spot Instances – bid on unused EC2 capacity and run those instances as long as the bid exceeds current spot prices (changing periodically based on supply and demand)

Additional Compute Services

Technische Universität Berlin

Auto Scaling – scales EC2 capacity (number of instances) up or down automatically according to conditions that the consumer defines

Elastic Load Balancing (ELB) – automatically distributes incoming app traffic across multiple EC2 instances, within a single or across multiple availability zones

AWS Lambda – a compute service that runs consumer code in response to events (such as image upload, website click, etc) with automatic management of all necessary compute resources

EC2 Container Service – a high performance container management service (supports Docker containers) to run distributed apps on a managed cluster of EC2 instances

VM Import/Export – a service to import VM images from existing/on-premise environments to EC2 instances and to export them back

AWS Storage Services

Technische Universität Berlin

Amazon Web Services

Compute
- **EC2** — Virtual Servers in the Cloud
- **Lambda** — Run Code in Response to Events
- **EC2 Container Service** — Run and Manage Docker Containers

Storage & Content Delivery
- **S3** — Scalable Storage in the Cloud
- **Storage Gateway** — Integrates On-Premises IT Environments with Cloud Storage
- **Glacier** — Archive Storage in the Cloud
- **CloudFront** — Global Content Delivery Network

Database
- **RDS** — MySQL, Postgres, Oracle, SQL Server, and Amazon Aurora
- **DynamoDB** — Predictable and Scalable NoSQL Data Store
- **ElastiCache** — In-Memory Cache
- **Redshift** — Managed Petabyte-Scale Data Warehouse Service

Networking
- **VPC** — Isolated Cloud Resources
- **Direct Connect** — Dedicated Network Connection to AWS
- **Route 53** — Scalable DNS and Domain Name Registration

Administration & Security
- **Directory Service** — Managed Directories in the Cloud
- **Identity & Access Management** — Access Control and Key Management
- **Trusted Advisor** — AWS Cloud Optimization Expert
- **CloudTrail** — User Activity and Change Tracking
- **Config** — Resource Configurations and Inventory
- **CloudWatch** — Resource and Application Monitoring

Deployment & Management
- **Elastic Beanstalk** — AWS Application Container
- **OpsWorks** — DevOps Application Management Service
- **CloudFormation** — Templated AWS Resource Creation
- **CodeDeploy** — Automated Deployments

Analytics
- **EMR** — Managed Hadoop Framework
- **Kinesis** — Real-time Processing of Streaming Big Data
- **Data Pipeline** — Orchestration for Data-Driven Workflows
- **Machine Learning** — Build Smart Applications Quickly and Easily

Application Services
- **SQS** — Message Queue Service
- **SWF** — Workflow Service for Coordinating Application Components
- **AppStream** — Low Latency Application Streaming
- **Elastic Transcoder** — Easy-to-use Scalable Media Transcoding
- **SES** — Email Sending Service
- **CloudSearch** — Managed Search Service

Mobile Services
- **Cognito** — User Identity and App Data Synchronization
- **Mobile Analytics** — Understand App Usage Data at Scale
- **SNS** — Push Notification Service

Enterprise Applications
- **WorkSpaces** — Desktops in the Cloud
- **WorkDocs** — Secure Enterprise Storage and Sharing Service
- **WorkMail** PREVIEW — Secure Email and Calendaring Service

AWS Storage Services

Technische Universität Berlin

Simple Storage Service (S3) – a highly scalable object storage providing cost-effective storage for a wide variety of use cases, including cloud applications, content distribution, backup and archiving, disaster recovery, big data analytics, and other

With S3, *objects* (up to 5 TB in size) are stored within resources called *buckets*.

Glacier – a low-cost archive storage service (current pricing is $0.01 per gigabyte per month)

Elastic Block Store (EBS) – automatically replicated *persistent* block level storage volumes for use with EC2 instances

AWS Storage Services (cont.)

CloudFront – a content delivery web service to distribute content to end users with low latency and high data transfer speeds; requests are automatically routed to the nearest edge location

Import/Export Snowball – a service to move large amounts of data (peta-byte-scale) into and out of AWS using portable storage devices (Snowball appliances for transport (bypassing the Internet)

AWS Database Services

Technische Universität Berlin

Amazon Web Services

Compute
- **EC2** — Virtual Servers in the Cloud
- **Lambda** — Run Code in Response to Events
- **EC2 Container Service** — Run and Manage Docker Containers

Storage & Content Delivery
- **S3** — Scalable Storage in the Cloud
- **Storage Gateway** — Integrates On-Premises IT Environments with Cloud Storage
- **Glacier** — Archive Storage in the Cloud
- **CloudFront** — Global Content Delivery Network

Database
- **RDS** — MySQL, Postgres, Oracle, SQL Server, and Amazon Aurora
- **DynamoDB** — Predictable and Scalable NoSQL Data Store
- **ElastiCache** — In-Memory Cache
- **Redshift** — Managed Petabyte-Scale Data Warehouse Service

Networking
- **VPC** — Isolated Cloud Resources
- **Direct Connect** — Dedicated Network Connection to AWS
- **Route 53** — Scalable DNS and Domain Name Registration

Administration & Security
- **Directory Service** — Managed Directories in the Cloud
- **Identity & Access Management** — Access Control and Key Management
- **Trusted Advisor** — AWS Cloud Optimization Expert
- **CloudTrail** — User Activity and Change Tracking
- **Config** — Resource Configurations and Inventory
- **CloudWatch** — Resource and Application Monitoring

Deployment & Management
- **Elastic Beanstalk** — AWS Application Container
- **OpsWorks** — DevOps Application Management Service
- **CloudFormation** — Templated AWS Resource Creation
- **CodeDeploy** — Automated Deployments

Analytics
- **EMR** — Managed Hadoop Framework
- **Kinesis** — Real-time Processing of Streaming Big Data
- **Data Pipeline** — Orchestration for Data-Driven Workflows
- **Machine Learning** — Build Smart Applications Quickly and Easily

Application Services
- **SQS** — Message Queue Service
- **SWF** — Workflow Service for Coordinating Application Components
- **AppStream** — Low Latency Application Streaming
- **Elastic Transcoder** — Easy-to-use Scalable Media Transcoding
- **SES** — Email Sending Service
- **CloudSearch** — Managed Search Service

Mobile Services
- **Cognito** — User Identity and App Data Synchronization
- **Mobile Analytics** — Understand App Usage Data at Scale
- **SNS** — Push Notification Service

Enterprise Applications
- **WorkSpaces** — Desktops in the Cloud
- **WorkDocs** — Secure Enterprise Storage and Sharing Service
- **WorkMail** PREVIEW — Secure Email and Calendaring Service

AWS Database Services

Technische Universität Berlin

Relational Database Service (RDS) – a service to set-up, operate and scale a relational DB in the cloud, providing access to capabilities of MySQL, Oracle, and other relational database management systems (overall 5 dbms to-date)

Aurora – a MySQL-compatible relational database engine, “five times better performance than MySQL at a price point one tenth that of a commercial database while delivering similar performance and availability”

SimpleDB – a non-relational data store to store and query data items via web services requests; for example, to index S3 object metadata

DynamoDB – a fast and flexible NoSQL database service supporting both document and key-value data models

AWS Database Services (cont.)

Redshift – a fully-managed petabyte-scale data warehouse solution

ElastiCache – a service to deploy, operate and scale an in-memory cache in the cloud, supporting two open-source caching engines: Memcached and Redis

AWS Application Services

Technische Universität Berlin

Amazon Web Services

Compute
- **EC2** — Virtual Servers in the Cloud
- **Lambda** — Run Code in Response to Events
- **EC2 Container Service** — Run and Manage Docker Containers

Storage & Content Delivery
- **S3** — Scalable Storage in the Cloud
- **Storage Gateway** — Integrates On-Premises IT Environments with Cloud Storage
- **Glacier** — Archive Storage in the Cloud
- **CloudFront** — Global Content Delivery Network

Database
- **RDS** — MySQL, Postgres, Oracle, SQL Server, and Amazon Aurora
- **DynamoDB** — Predictable and Scalable NoSQL Data Store
- **ElastiCache** — In-Memory Cache
- **Redshift** — Managed Petabyte-Scale Data Warehouse Service

Networking
- **VPC** — Isolated Cloud Resources
- **Direct Connect** — Dedicated Network Connection to AWS
- **Route 53** — Scalable DNS and Domain Name Registration

Administration & Security
- **Directory Service** — Managed Directories in the Cloud
- **Identity & Access Management** — Access Control and Key Management
- **Trusted Advisor** — AWS Cloud Optimization Expert
- **CloudTrail** — User Activity and Change Tracking
- **Config** — Resource Configurations and Inventory
- **CloudWatch** — Resource and Application Monitoring

Deployment & Management
- **Elastic Beanstalk** — AWS Application Container
- **OpsWorks** — DevOps Application Management Service
- **CloudFormation** — Templated AWS Resource Creation
- **CodeDeploy** — Automated Deployments

Analytics
- **EMR** — Managed Hadoop Framework
- **Kinesis** — Real-time Processing of Streaming Big Data
- **Data Pipeline** — Orchestration for Data-Driven Workflows
- **Machine Learning** — Build Smart Applications Quickly and Easily

Application Services
- **SQS** — Message Queue Service
- **SWF** — Workflow Service for Coordinating Application Components
- **AppStream** — Low Latency Application Streaming
- **Elastic Transcoder** — Easy-to-use Scalable Media Transcoding
- **SES** — Email Sending Service
- **CloudSearch** — Managed Search Service

Mobile Services
- **Cognito** — User Identity and App Data Synchronization
- **Mobile Analytics** — Understand App Usage Data at Scale
- **SNS** — Push Notification Service

Enterprise Applications
- **WorkSpaces** — Desktops in the Cloud
- **WorkDocs** — Secure Enterprise Storage and Sharing Service
- **WorkMail** PREVIEW — Secure Email and Calendaring Service

AWS Application Services (selection)

Technische Universität Berlin

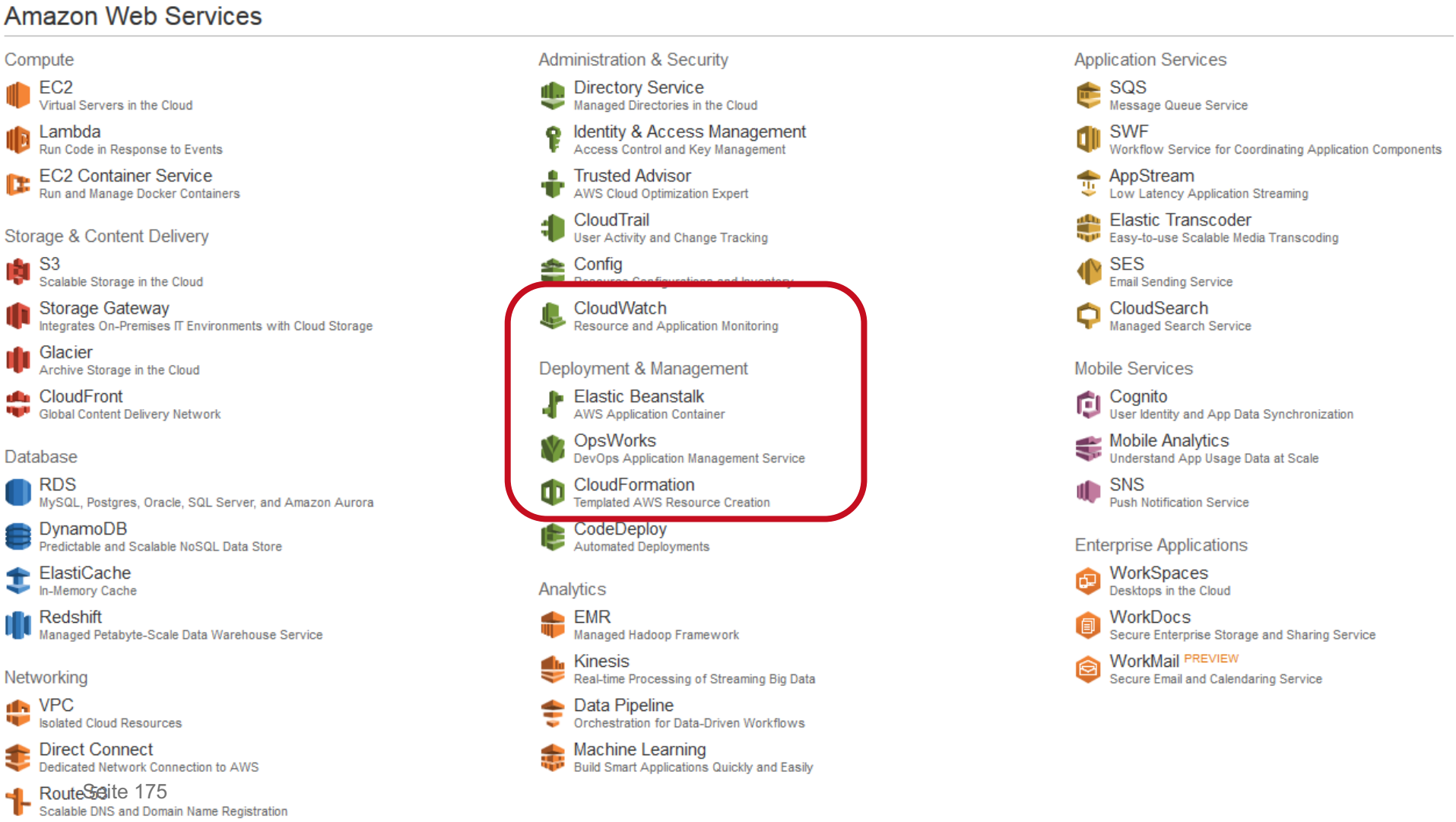
Simple Queue Service (SQS) – a fast, reliable, scalable message queuing service used to transmit any volume of data, at any level of throughput, without losing messages or requiring other services to be always available

SimpleWorkflow Service (SWF) – a managed state-tracker and task coordinator to track the state of processing of jobs that have parallel or sequential steps

AppStream – a low-latency service to stream resource-intensive applications from the cloud

...and some other select AWS offerings

Technische Universität Berlin



Monitoring, Deployment, Management

Technische Universität Berlin

CloudWatch – a monitoring service for AWS resources (such as EC2 instances, DynamoDB tables) to collect and monitor metrics, log files, and to set alarms

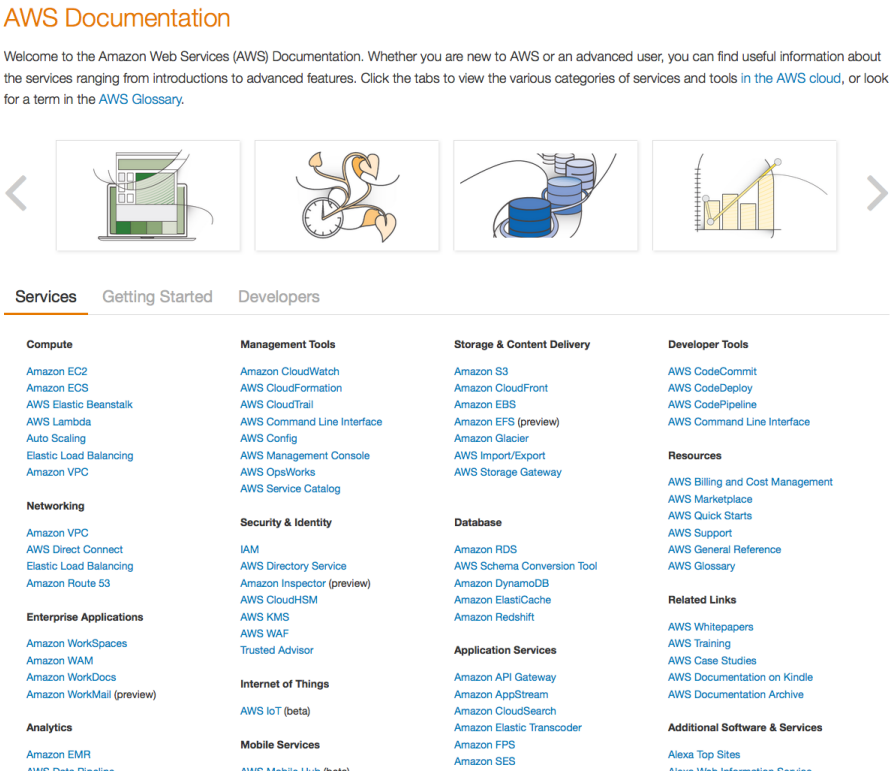
Elastic Beanstalk – a service to deploy web applications developed with Java, .NET, PHP, Python, Ruby and Docker on familiar servers such as Apache, Nginx, Phusion Passenger, and IIS

CloudFormation – create and manage a collection of related AWS resources using templates to describe the resources and their dependencies or runtime parameters

Further Reading / Experimentation

AWS Documentation:
https://aws.amazon.com/documentation/

AWS Educate Program:
https://aws.amazon.com/education/awseducate/



Some thought exercises to start with…

Business-related:

1. What strategy is behind these AWS offerings?
2. What do competitors offer?

Technology-related:

1. How do these services compare to each other?
2. Would you know which service to use for a given application/business/technical problem?

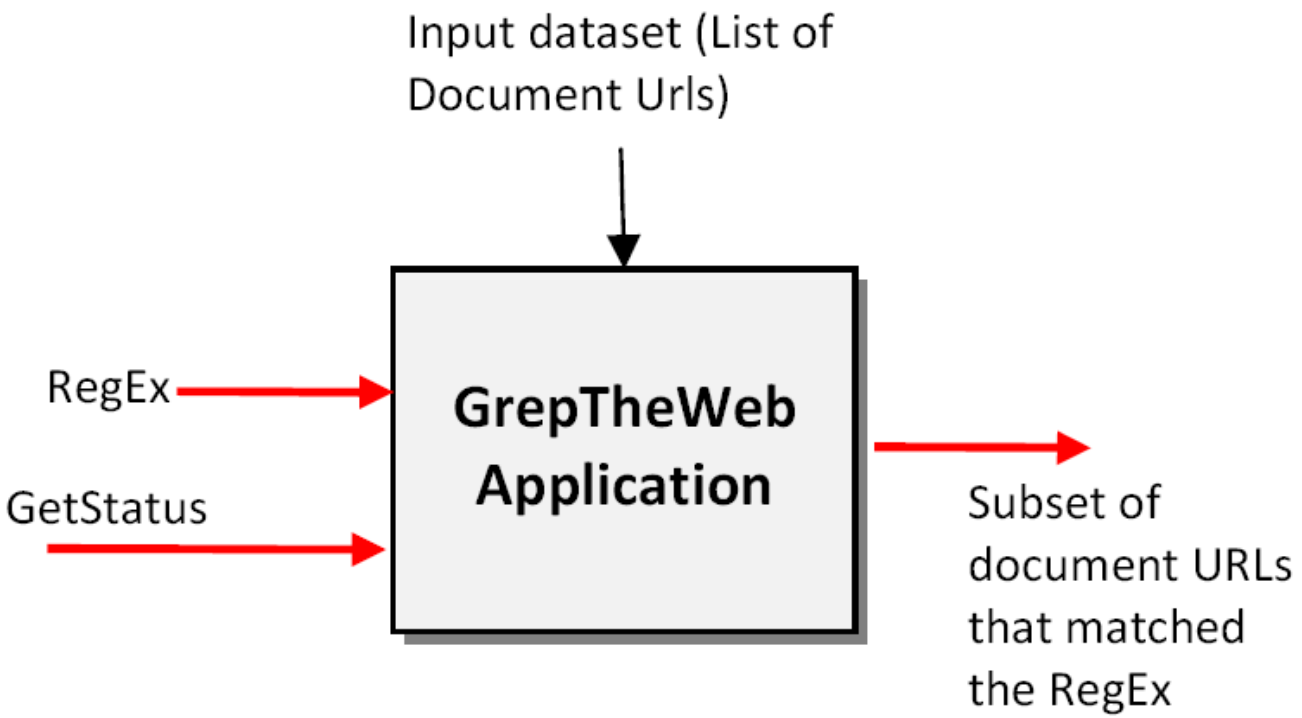
Example: GrepTheWeb (ca. 2008)

AWS offering Alexa Web Search offers customizable search engines for web data crawled by Alexa

Problem: How can we find specific expressions within the (potentially large) dataset returned by Alexa?

J. Varia, Cloud Architectures, Amazon White Paper, 2008.
(Still) available online at
http://jineshvaria.s3.amazonaws.com/public/cloudarchitectures-varia.pdf

GrepThe Web – Zoom Level 1



Source: amazon.com

GrepTheWeb Challenges

Performing a regular expression against millions of documents is not trivial.

Different factors could combine to cause the processing to take lot of time:

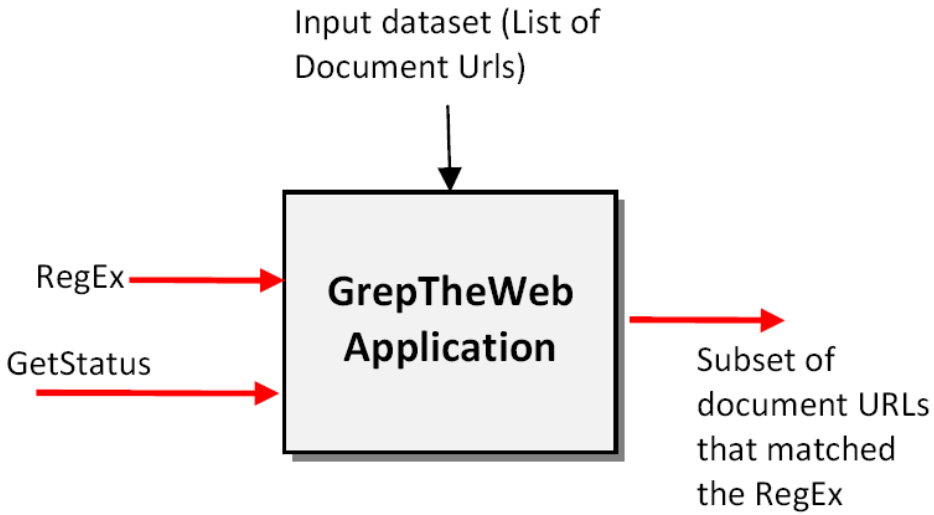
- Regular expressions could be complex
- Dataset could be large, even hundreds of terabytes
- Unknown request patterns, e.g., any number of people can access the application at any given point in time

- ➢ Unknown processing time.
- ➢ But recall: When using cloud resources, running 5 instances for 1h costs the same as running one instance for 5h…

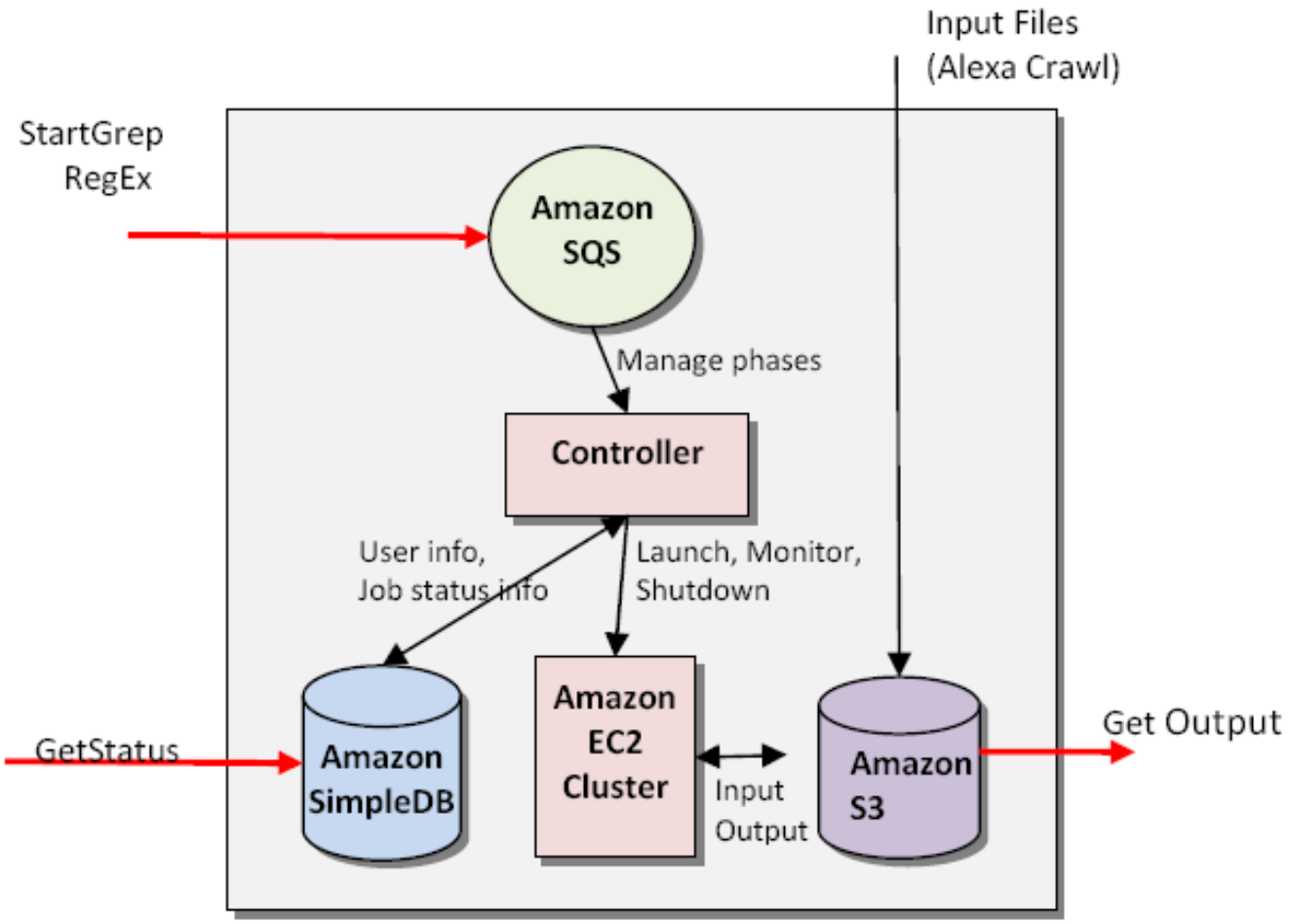
Source: https://aws.amazon.com/articles/1632

Short Quiz

Which AWS offerings come to mind – which functionality and features does the application require?

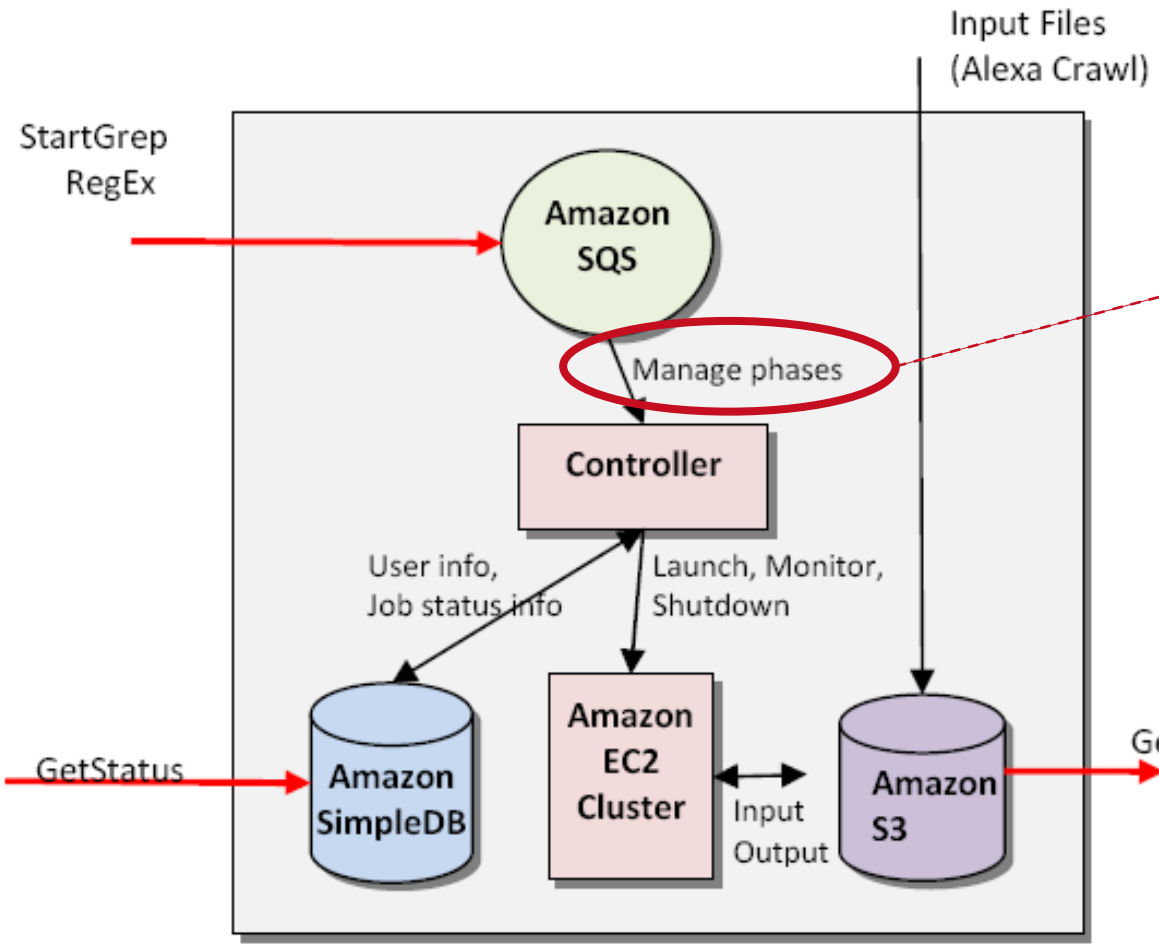


GrepThe Web – Zoom Level 2



Source: amazon.com

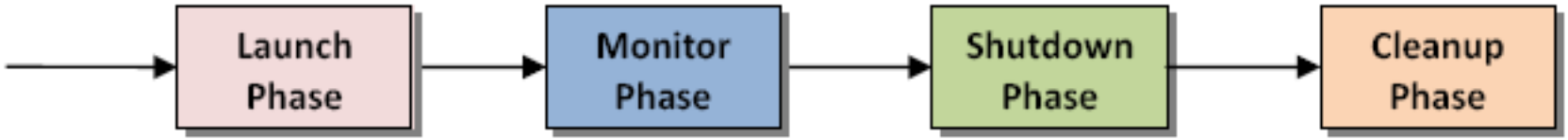
GrepThe Web – Zoom Level 2



Which phases are there?

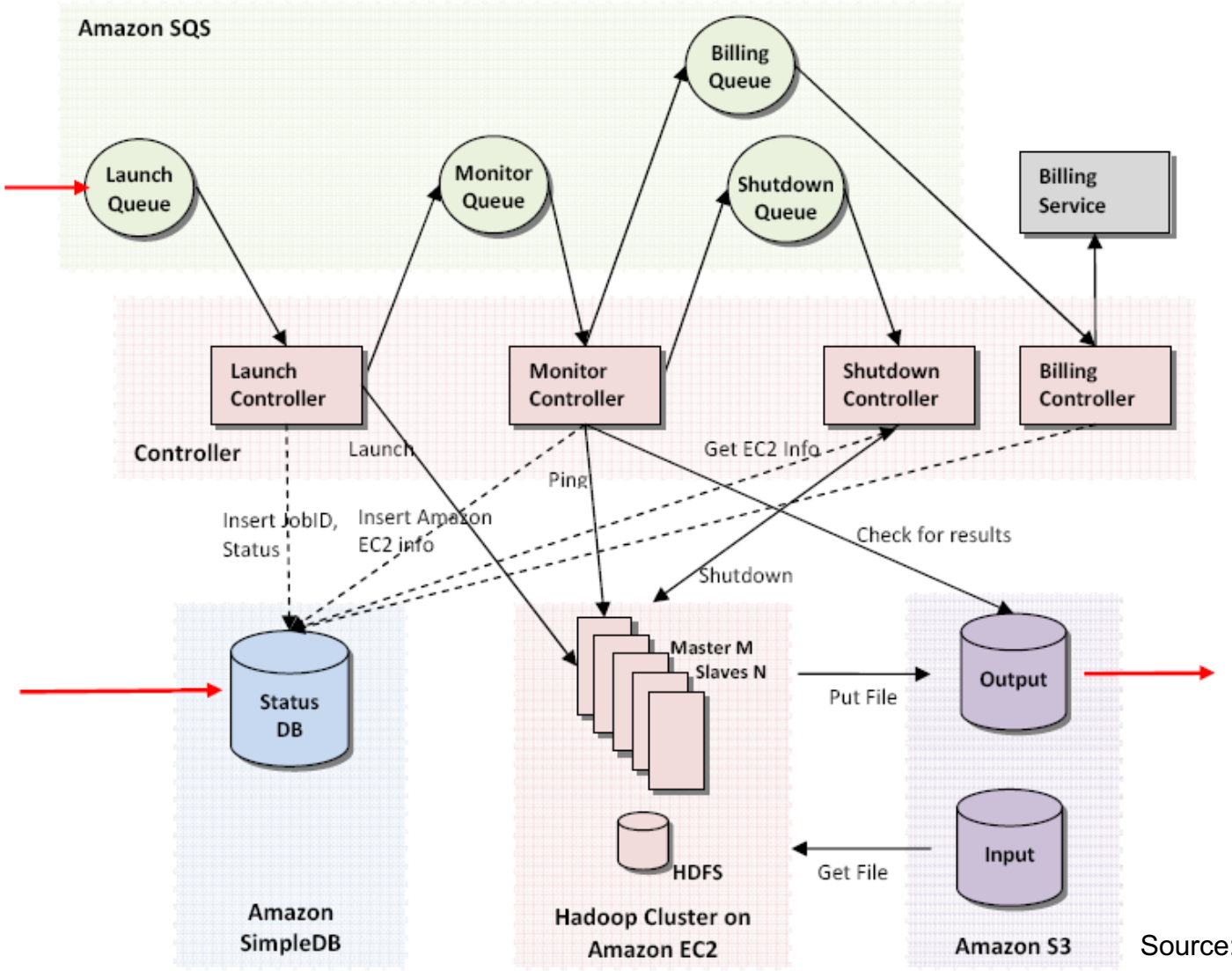
Source: amazon.com

GrepTheWeb Phases



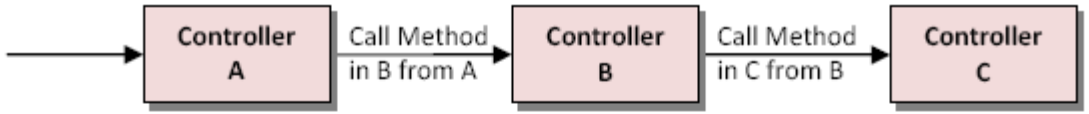
Source: amazon.com

GrepThe Web – Zoom Level 3

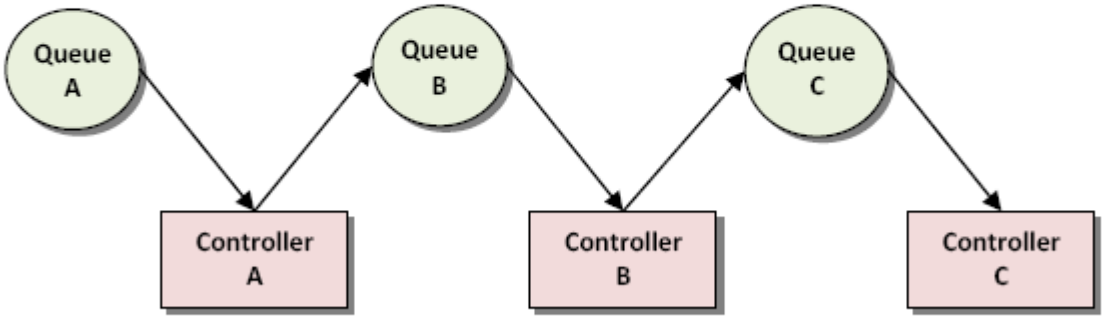


Source: amazon.com

Using Queues for Loose coupling



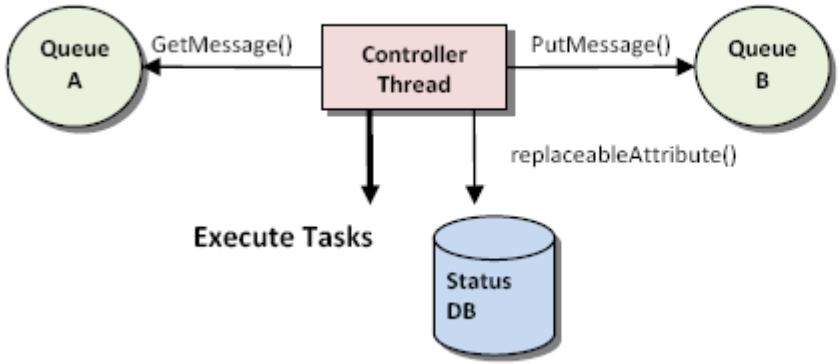
Tight coupling (procedural programming)



Loose coupling (independent phases using queues)

Source: amazon.com

Simple Controller Architecture



1. Controller dequeues message from Queue A
2. Controller executes Tasks (for eg. Launch, monitor etc)
3. Controller Updates Statuses in status DB
4. Controller enqueues new message in Queue B

```
Public Abstract BaseController (SQSMessageQueue fromQueue, SQSMessageQueue
toQueue, SDBDomain domain)
```

Source: amazon.com

Design Principles

Technische Universität Berlin

Decentralization: Use fully decentralized techniques to remove scaling bottlenecks and single points of failure.

Symmetry: Nodes in the system are identical in terms of functionality, and require no or minimal node-specific configuration to function.

Asynchrony: The system makes progress under all circumstances.

Failure tolerant: The system considers the failure of components to be a normal mode of operation, and continues operation with no or minimal interruption.

Simplicity: The system should be made as simple as possible (but no simpler).

Design Principles (cont.)

Technische Universität Berlin

Decompose into small well-understood building blocks: Do not try to provide a single service that does everything for everyone, but instead build small components that can be used as building blocks for other services.

Autonomy: The system is designed such that individual components can make decisions based on local information.

Local responsibility: Each individual component is responsible for achieving its consistency; this is never the burden of its peers.

Controlled concurrency: Operations are designed such that no or limited concurrency control is required.

Controlled parallelism: Abstractions used in the system are of such granularity that parallelism can be used to improve performance and robustness of recovery or the introduction of new nodes.

GrepTheWeb, 2016

Technische Universität Berlin

While the 2008-GrepTheWeb-solution is very valid, would you choose

1. …other AWS services today, given the new(er) offerings available? Why?
2. Would you argue for additional services, e.g., for monitoring purposes? Which ones?
3. In particular, can you imagine a microservice- and/or Lambda-service-architecture? Discuss this option.

Agenda

1. Web Engineering Fundamentals
    a. HTTP
    b. REST
    c. SOAP
    d. WSDL
    e. WS-*
2. Cloud Fundamentals
    a. Sample Cloud Services: AWS
    b. Design Principles
3. Microservices
    a. DevOps, CI/CD
    b. Microservices and Server-less Architectures

Enterprise Systems, 2016

Technische Universität Berlin

"The cloud" is the de-facto platform when building distributed systems

Containerization, *software-defined networking* and other trends advance cloud technology

Continuous delivery, *continuous integration* and *DevOps-based practices* radically change the way how tech-based businesses evolve

Microservices is a "cloud-native" architecture taking advantage of both

DevOps in a nutshell

Technische Universität Berlin

DevOps most generally is about software and business engineering

…emphasizing short, iterative planning and development cycles

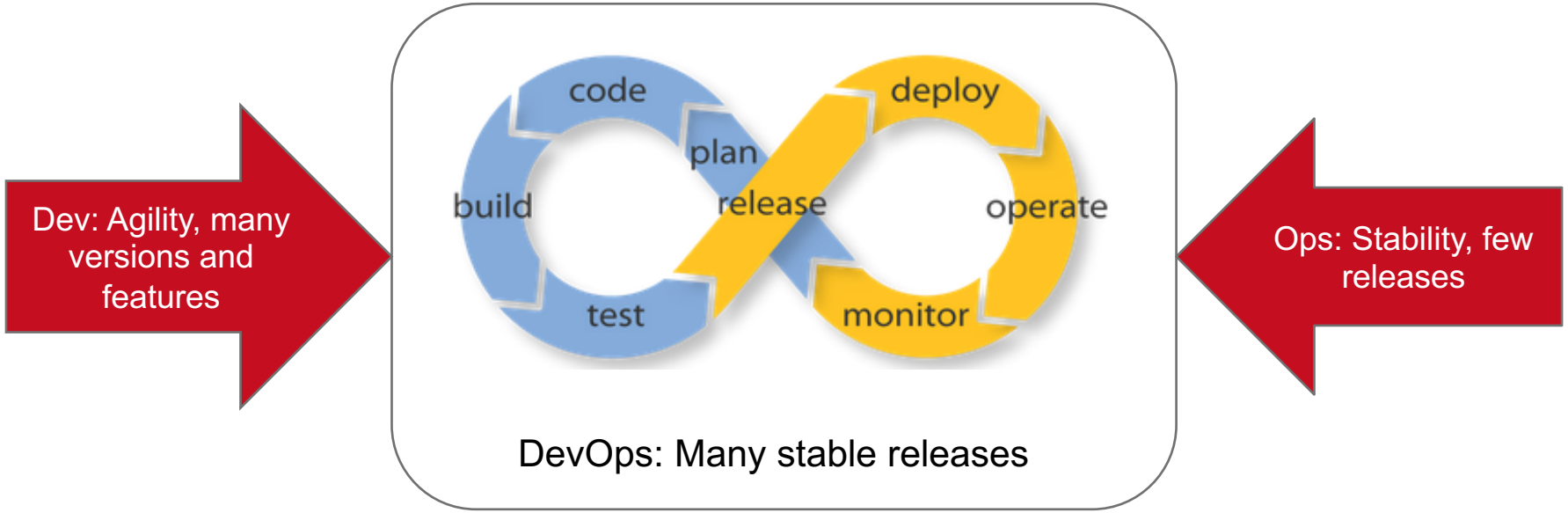
- supporting changing requirements as projects evolve
- enabling fast and frequent delivery of high-quality functionality

…improving communication and collaboration between different teams

- focusing on small, highly decoupled tasks
- Communicating via language-agnostic APIs

overall establishing development practices that leverage frequent code commits, automated verification and builds, and early problem detection

Dev and Ops

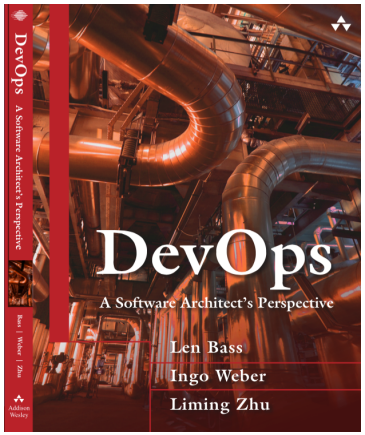

Dev: Agility, many versions and features
code
plan
build
release
deploy
operate
test
monitor
Ops: Stability, few releases
DevOps: Many stable releases

DevOps Definition

"DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality."

Source:

[BWZ 2015] *DevOps: A Software Architect's Perspective*.
Len Bass, Ingo Weber, Liming Zhu, Addison-Wesley Professional, 2015.
http://amzn.to/1Qlar8K



What does that mean?

Technische Universität Berlin

Quality of the code must be high

- Testing & test-driven development

Quality of the build & delivery mechanism must be high

- Automation & more testing
- A must when deploying to production 25x per day (etsy.com)

Time is split:

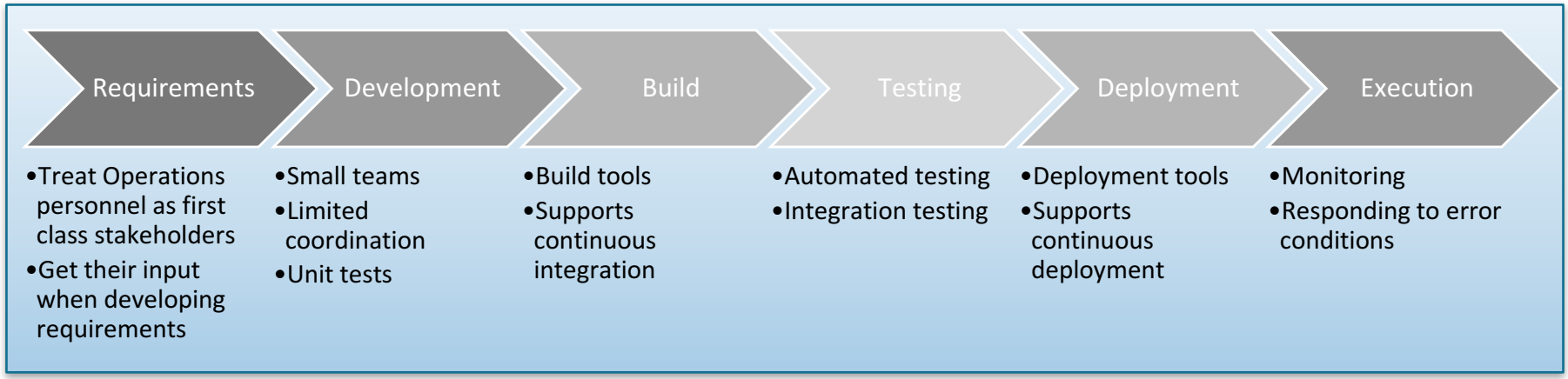
- From commit until deployment to production
- From deployment until acceptance into normal production
- Means testing in production

Goal-oriented definition

- May use agile methods, continuous integration, etc.
- Likely to use automation tools

Source: [BWZ 2015]

DevOps Practices (1/2)



Treat Ops as first-class citizens throughout the lifecycle – e.g., in requirements elicitation

- Many decisions can make operating a system harder or easier
- Logging and monitoring to suit Ops

Make Dev more responsible for relevant incident handling

- Shorten the time between finding and repairing errors

Source: [BWZ 2015]

DevOps Practices (2/2)

Use continuous deployment, automate everything

- Commits trigger automatic build, testing, deployment

Enforce deployment process is used by all

- No ad-hoc deployments
- Ensures changes are traceable

Develop infrastructure code with the same set of practices as application code

- “Infrastructure as Code”: using IaaS APIs, etc., to automate creation of environments
- Misconfiguration can derail your application
- Ops scripts are traditionally more ad-hoc

Source: [BWZ 2015]

DevOps consequences

Technische Universität Berlin

Architecturally significant requirement:

Speed up deployment through minimizing synchronous coordination among development teams.

Synchronous coordination, like a meeting, adds time since it requires

- Ensuring that all parties are available
- Ensuring that all parties have the background to make the coordination productive
- Following up to decisions made during the meeting

Source: [BWZ 2015]

DevOps consequences

Technische Universität Berlin

Keep teams relatively small

- "Two pizza rule": no team should be larger than can be fed with two pizzas
- Advantages: make decisions quickly, less coordination overhead, more coherent units

Business boundaries as service boundaries

- Small, autonomous teams focused on doing one thing well, building small services → Microservices
- Channel most interaction through service interfaces

Source: [BWZ 2015]

Microservices

…are small, autonomous services providing a small, focused amount of functionality

…to address the DevOps objective for fast and high-quality change management:

- Small teams develop small services
- Coordination overhead is minimized by channeling most interaction through service interfaces:
  - Team X provides service A, which is used by teams Y and Z
  - If changes are needed, they are communicated, implemented, and added to the interface

Team size becomes a major driver of the overall architecture!

Adapted from: [BWZ 2015]

Microservices [Fowler]

Technische Universität Berlin

"In short, the 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.

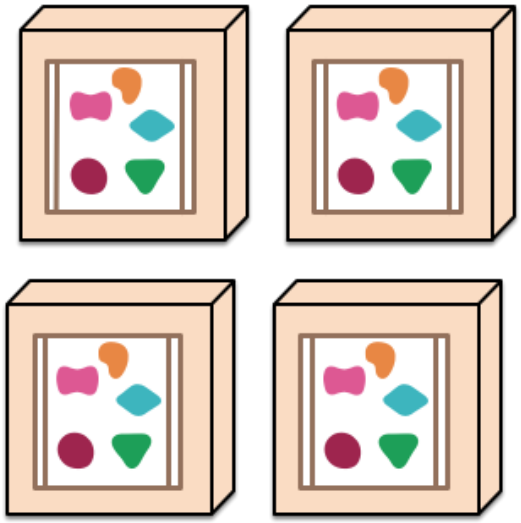
There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."

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

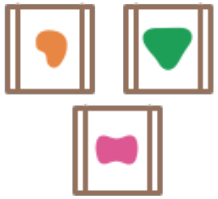
Monoliths versus Microservices

A monolithic application puts all its functionality into a single process...

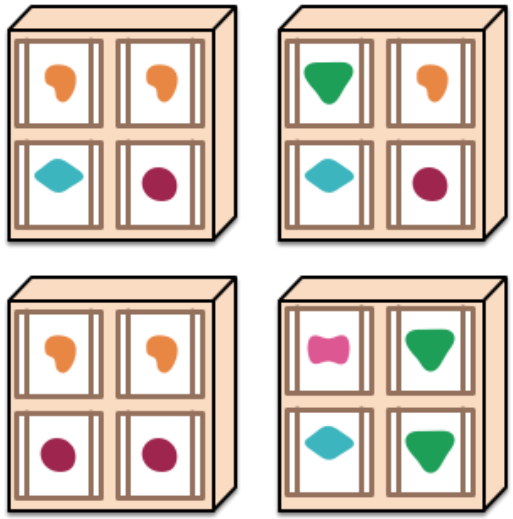
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...

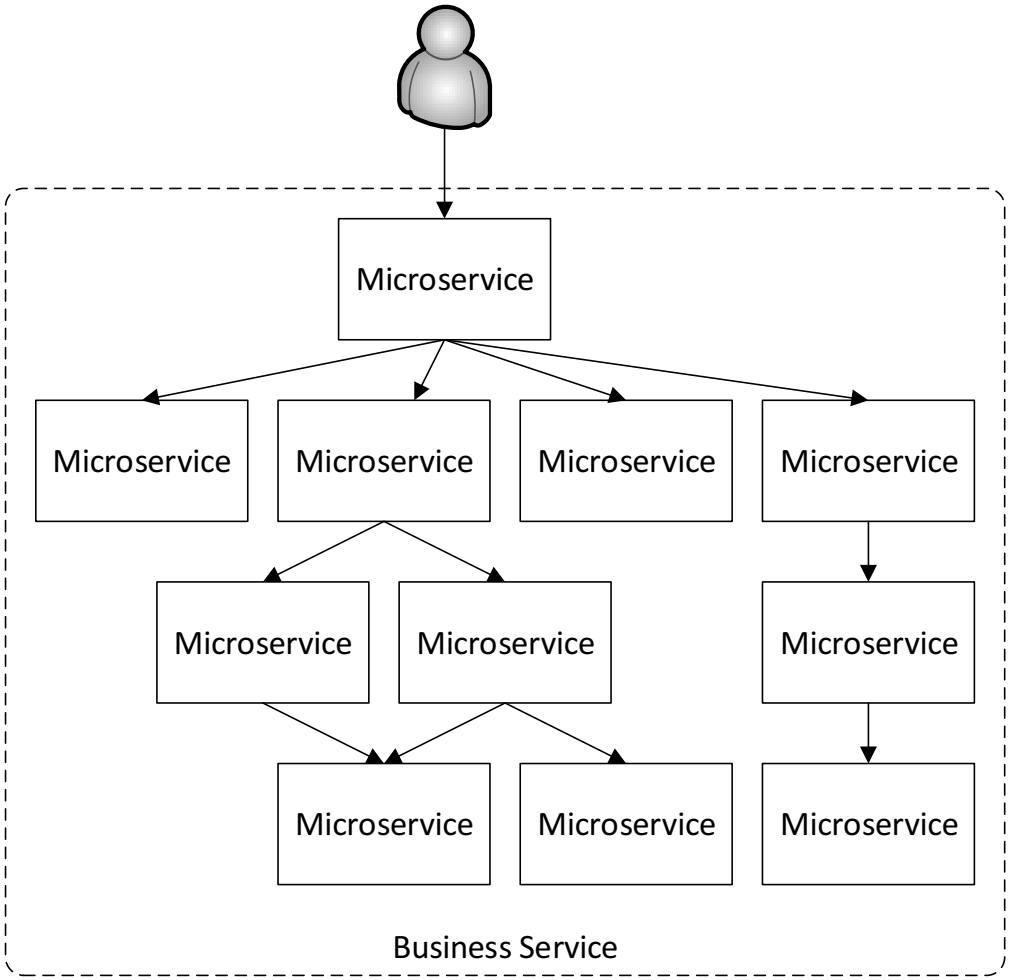


... and scales by distributing these services across servers, replicating as needed.



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

Microservice Architecture



Each user request is satisfied by some sequence of services

Most services are not externally available

Each service communicates with other services through service interfaces

Service depth?

Source: [BWZ 2015]

Amazon design rules

All teams will henceforth expose their data and functionality through service interfaces.
Teams must communicate with each other through these interfaces.

There will be no other form of inter-process communication allowed:

- no direct linking, no direct reads of another team's data store,
- no shared-memory model, no back-doors whatsoever.
- The only communication allowed is via service interface calls over the network.

It doesn't matter what technology they[services] use.

All service interfaces, without exception, must be designed from the ground up to be externalizable.

Source: [BWZ 2015]

Characteristics of Microservices [Fowler]

1. Componentization via Services
2. Organized around Business Capabilities
3. Products not Projects
4. Smart endpoints and dumb pipes
5. Decentralized Governance
6. Decentralized Data Management
7. Infrastructure Automation
8. Design for failure
9. Evolutionary Design

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

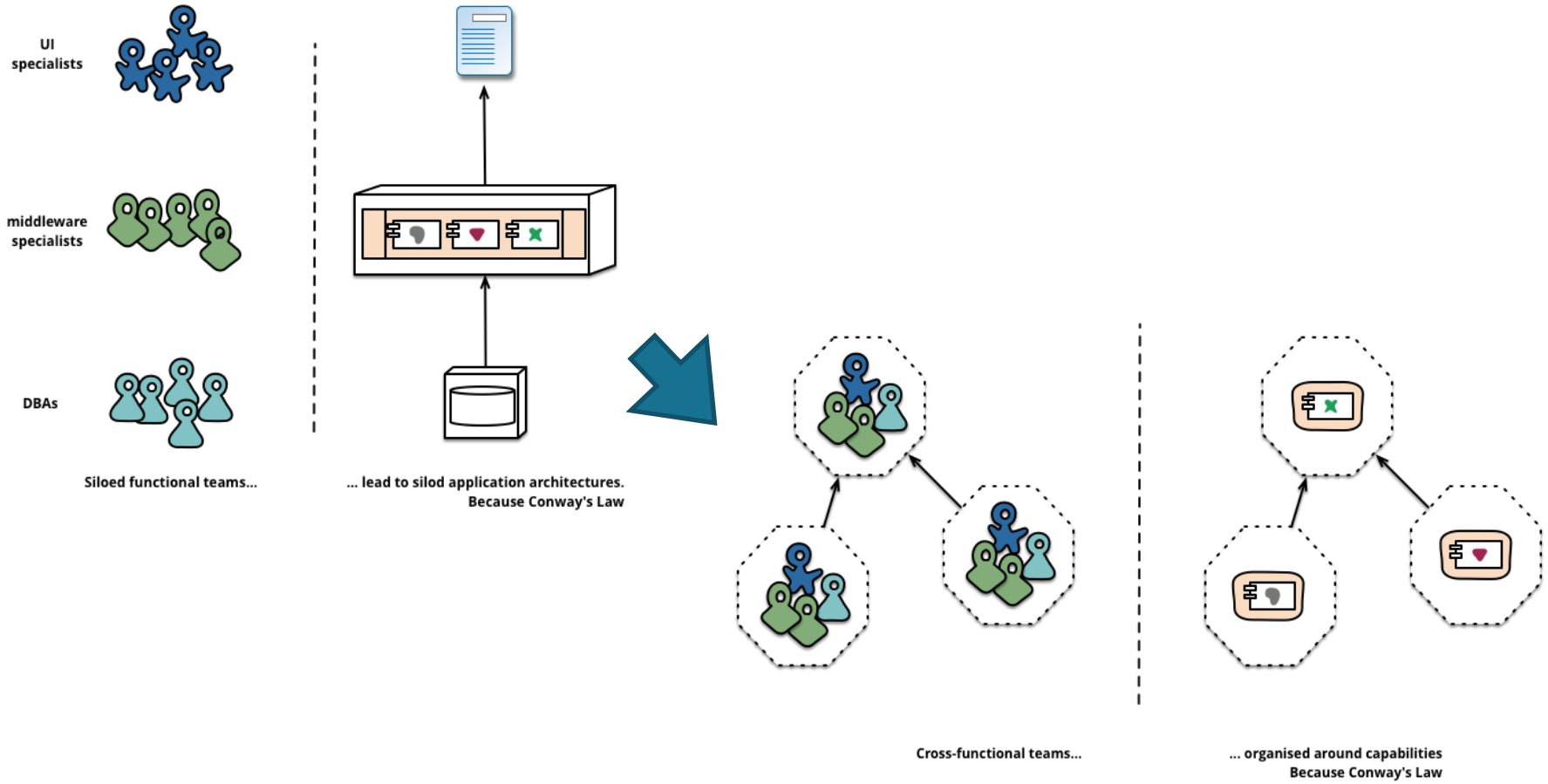
1. Componentization via Services

A component is a unit of software that is independently replaceable and upgradeable.
Microservice architectures will use libraries, but their primary way of componentizing their own software is by breaking down into services.

We define libraries as components that are linked into a program and called using in-memory function calls, while services are out-of-process components who communicate with a mechanism such as a web service request, or remote procedure call.

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

2. Organized around Business Capabilities



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

3. Products, not Projects

Technische Universität Berlin

Project model: the aim is to deliver some piece of software which is then considered to be completed. On completion the software is handed over to a maintenance organization and the project team that built it is disbanded.

Product model: A development team takes full responsibility for the software in production (“you build, you run it”).

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

4. Smart endpoints and dumb pipes

Technische Universität Berlin

Applications built from microservices aim to be as decoupled and as cohesive as possible – they own their own domain logic and act more as filters in the classical Unix sense – receiving a request, applying logic as appropriate and producing a response.

These are choreographed using simple RESTish protocols rather than complex protocols such as WS-Choreography or BPEL or orchestration by a central tool.

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

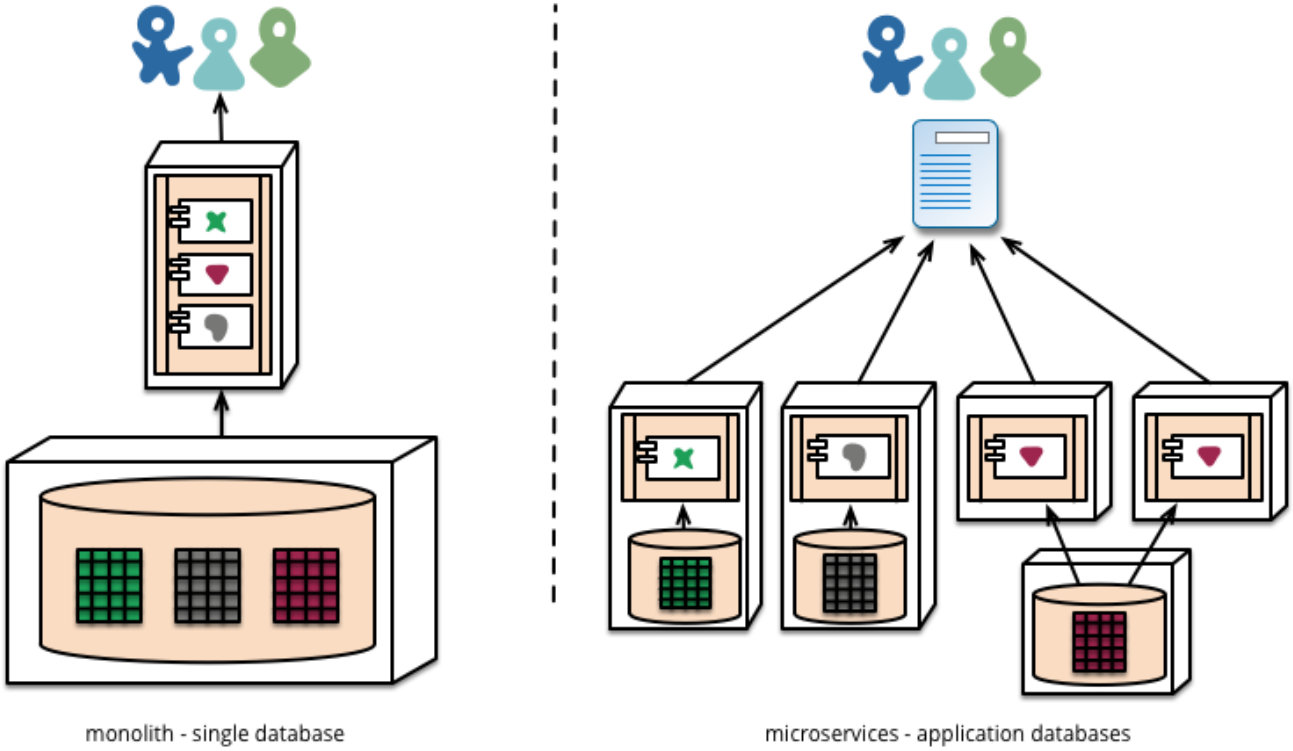
5. Decentralized Governance

One of the consequences of centralized governance is the tendency to standardize on single technology platforms. Experience shows that this approach is constricting - not every problem is a nail and not every solution a hammer.

Rather than use a set of defined standards written down somewhere on paper they prefer the idea of producing useful tools that other developers can use to solve similar problems to the ones they are facing.

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

6. Decentralized Data Management

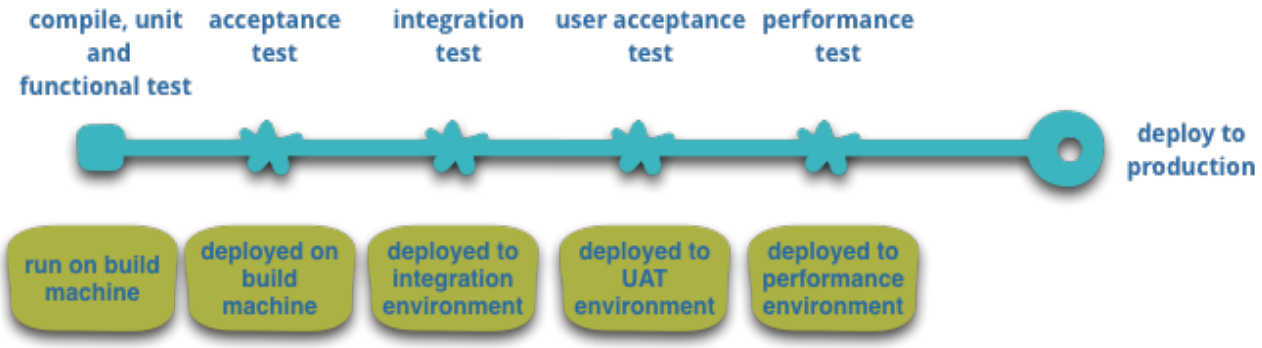


http://martinfowler.com/articles/microservices.html

7. Infrastructure Automation

Technische Universität Berlin

Automated, continuous deployment and testing processes



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

8. Design for failure

A consequence of using services as components, is that applications need to be designed so that they can tolerate the failure of services.

Any service call could fail due to unavailability of the supplier, the client has to respond to this as gracefully as possible. This is a disadvantage compared to a monolithic design as it introduces additional complexity to handle it.

The consequence is that microservice teams constantly reflect on how service failures affect the user experience. Netflix's Simian Army induces failures of services and even datacenters during the working day to test both the application's resilience and monitoring.

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

9. Evolutionary Design

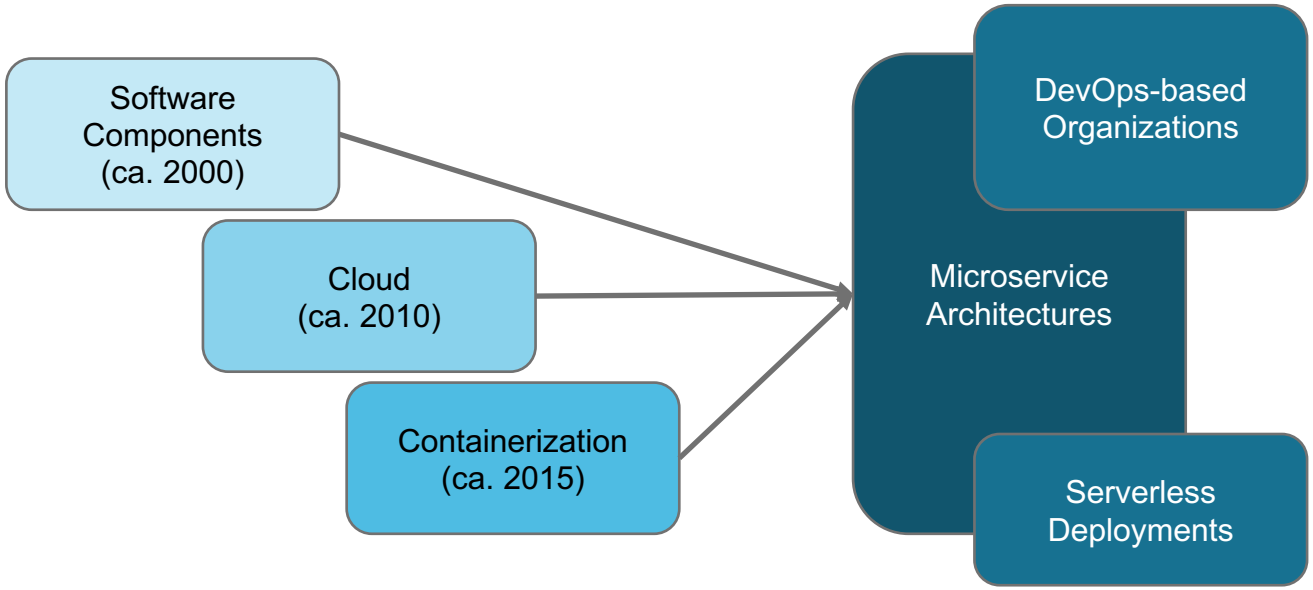
The key property of a component is the notion of independent replacement and upgradeability.

You want to keep things that change at the same time in the same module. Parts of a system that change rarely should be in different services to those that are currently undergoing lots of churn.

Putting components into services adds an opportunity for more granular release planning. With a monolith any changes require a full build and deployment of the entire application. With microservices, however, you only need to redeploy the service(s) you modified. This can simplify and speed up the release process.

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

The evolution towards Microservices

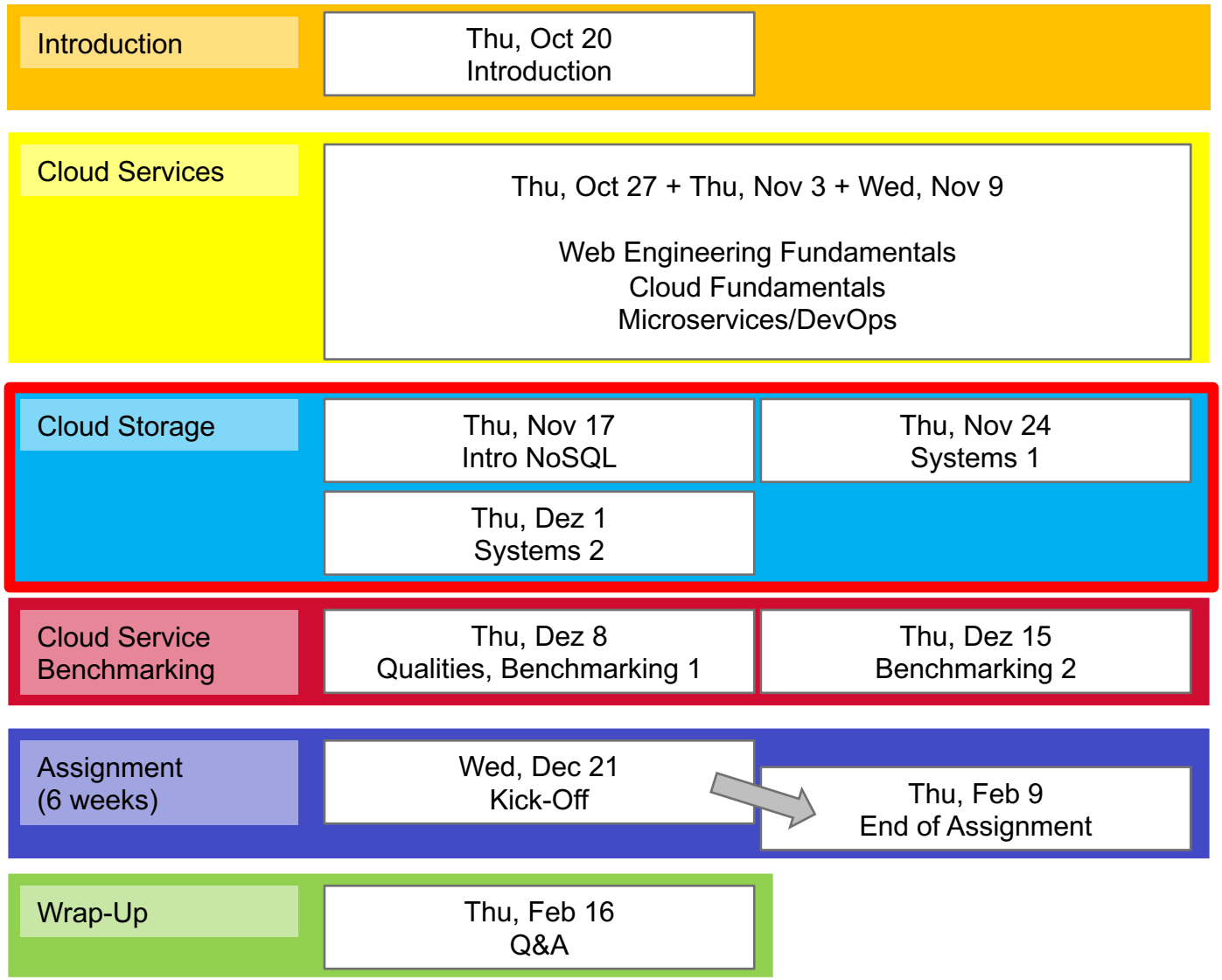


toolchain

Code – Code development and review, continuous integration tools - Git
Build – Version control tools, code merging, build status - Jenkins
Test – Test and results determine performance - Bugzilla
Package – Artifact repository, application pre-deployment staging - Artifactory
Release – Change management, release approvals, release automation - Jenkins
Configure – Infrastructure configuration and management, Infrastructure as Code tools - Ansible
Monitor – Applications performance monitoring, end user experience - Nagios
Log Management – Dealing with log files – Elasticsearch, Logstash & Kibana (ELK)

Containerization - Docker,
continuous Integration - Jenkins,
Infrastructure as Code – Puppet,
virtualization platform - Vagrant

Next


Introduction
Thu, Oct 20
Introduction
Cloud Services
Thu, Oct 27 + Thu, Nov 3 + Wed, Nov 9
Web Engineering Fundamentals
Cloud Fundamentals
Microservices/DevOps
Cloud Storage
Thu, Nov 17
Intro NoSQL
Thu, Nov 24
Systems 1
Thu, Dez 1
Systems 2
Cloud Service
Benchmarking
Thu, Dez 8
Qualities, Benchmarking 1
Thu, Dez 15
Benchmarking 2
Assignment
(6 weeks)
Wed, Dec 21
Kick-Off
Thu, Feb 9
End of Assignment
Wrap-Up
Thu, Feb 16
Q&A