Concept/Topic: *Service-Oriented Architecture (SOA)*

Context

Let us recapitulate the state of affairs in architectural synthesis after lessons 3 to 8:[1] candidate components have been identified, specified, and some or all of these components have been realized (as a consequence of a make/build decision) or procured (following a buy decision), Tactic and strategic DDD patterns might but do not have to be used along this way (for instance, during component realization with object-oriented programming). Next, these components need to be stitched together to yield end-to-end applications that implement functional requirements (expressed as use cases and/or user stories) and satisfy NFRs (including quality attributes).

zusammenheften

The context outlined in the previous paragraph is where component (de)composition and remoting concepts come in, for instance those summarized under the umbrella term *Enterprise Application Integration (EAI)*. SOA, REST and microservices are related problem solving concepts in this integration design space: SOA and REST qualify as architectural styles specializing on integration; file transfer, shared database, messaging and remote procedure calls are more general/universal integration styles (see exercise 10 and separate fact sheet).

Definition(s)

M. Fowler on the difference between components and services (Reference: http://martinfowler.com/articles/injection.html):

- "I use component to mean a glob of software that's intended to be used, without change, by an application that is out of the control of the writers of the component. By "without change" I mean that the using application doesn't change the source code of the components, although they may alter the component's behavior by extending it in ways allowed by the component writers.
- A service is similar to a component in that it's used by foreign applications. The main difference is that I expect a component to be used locally (think jar file, assembly, DLL, or a source import). A service will be used remotely through some remote interface, either synchronous or asynchronous (e.g. Web service, messaging system, RPC, or socket.)"

Service Layer pattern

How to hide complex domain models from presentation layer? Define an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation.

Other responsibilities of Service Layer:

- Role-Based Access Control (RBAC) (taking data into account)
- Transaction control, business-level undo (compensation)
- Activity logging (e.g. metering, monitoring)
- Exception handling

Reference: R. Stafford, in Patterns of Enterprise Application Architecture (PoEAA), http://martinfowler.com/eaaCatalog/serviceLayer.html

[1] these are the component modeling steps from lessons 3 and 4

Remote Facade pattern

"Provides a coarse-grained facade on fine-grained objects to improve efficiency over a network."

Reference: M. Fowler, PoEAA, http://martinfowler.com/eaaCatalog/remoteFacade.html

Data Transfer Object (DTO) pattern

"A DTO is an object that carries data between processes in order to reduce the number of method calls."

Reference: M. Fowler, PoEAA (another version of this pattern is described here), https://martinfowler.com/eaaCatalog/dataTransferObject.html.

Integration Styles

The Enterprise Integration Patterns (EIP) book introduces four styles (Hohpe and Woolf (2003)):

- File Transfer[2]
- Shared Database[3]
- Remote Procedure Invocation[4]
- Messgaging[5]

See the EIP website[6] for more information, as well as exercise 10. The Web can be seen as a fifth style.

Service-Oriented Architecture (SOA)

There is no single definition, SOA means different things to different stakeholders. Three definitions taking different views (from the 4+1 viewpoint model introduced in lesson 1) are:

- From a scenario viewpoint when analyzing business stakeholder concerns, a SOA is a set of services that a business wants to expose to their customers and partners, or other portions of the organization.
- For software architects taking a logical, process or deployment/physical viewpoint, SOA is an architectural style which requires a *service provider*, a *service requestor (consumer)* and a *service contract*: "A service is a component with a remote interface." (M. Fowler). These basic definitions resemble the client/server style; however, the SOA style also:
  - promotes *architectural principles* such as modularity, layering, and loose coupling to achieve design goals such as separation of concerns, reuse, and flexibility.
  - suggests a set of *architectural patterns* such as enterprise service bus (service invocations have to be routed, transformed, and their protocols adapted), service composition (services have to be stitched together to implement user needs), and service registry (as services have to be discovered.
- For developers, operators (system administrators) and coding architects, SOA requires and defines one or more programming and deployment models realized by standards, tools and technologies such as Web services (WSDL/SOAP based or RESTful HTTP).

More elaborate versions of these definitions can be found in Zimmermann (2009).[7]

[2] http://www.enterpriseintegrationpatterns.com/patterns/messaging/FileTransferIntegration.html
[3] http://www.enterpriseintegrationpatterns.com/patterns/messaging/SharedDataBaseIntegration.html
[4] http://www.enterpriseintegrationpatterns.com/patterns/messaging/EncapsulatedSynchronousIntegration.html
[5] http://www.enterpriseintegrationpatterns.com/patterns/messaging/Messaging.html
[6] http://www.enterpriseintegrationpatterns.com/patterns/messaging/
[7] Note that each SOA author uses slightly different definitions (which has been criticized, but also happens to newer terms such as microservices). A OASIS reference model norms some of the terminology: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm#overview.

The AppArch lessons and excercises focuses on the *loose coupling* principle; the others ones architectural principles are covered elsewhere (e.g., SE1/SE2 and VSS lectures). The ESB and Service Contract patterns are covered in depth in AppArch. Service Composition and Service Registry (or discovery) are out of scope (at least in the 2017/2018 edition of the lecture). However, the latter two patterns are also featured in this fact sheet (as a bonus).

Loose Coupling Principle: Background and Definitions/Dimensions

Foundational perspectives on architectural principles that are relevant for the SOA style include:

- General software engineering/architecture literature starting from the very beginnings of the field, e.g., Parnas (1972): modularization, high cohesion/low coupling
- A WWW 2009 presentation[8] and paper[9] by C. Pautasso and E. Wilde lists 12 facets used for a remoting technology comparison: discovery, state, granularity
- An ESOCC 2016 keynote[10] by F. Leymann highlights four types of autonomy: reference (i.e., location), platform, time, format
  - *Reference autonomy* is also called location transparency): Producers and consumers don't know each other.
  - *Platform autonomy*: Producers and consumers may be in different environments and written in different languages.
  - *Time autonomy*: Producers and consumers access channel at their own pace; communication is asynchronous, data exchanged is persistent.
  - *Format autonomy*: Producers and consumers may use different formats of data exchanged; this requires transformation "on the wire" (a.k.a. mediation).

Practitioner heuristics (a.k.a. coupling criteria) appear in books, articles, and blog posts. A few notable examples are:

- The "SOA in Practice" book by N. Josuttis (Josuttis (2007)) describes 11 types of (loose) coupling, and emphasises versioning and compatibility.
- IBM Redbook SG24-6346-00[11] on SOA and ESB (Martin Keen (2004)) covers: coupled vs. decoupled continuum: semantic interface, (business) data model, QoS (e.g. transactional context, reliability), security.

The topic is also regularly touched upon in community sited and online knowledge portals such as DZone, IBM developerWorks articles, InfoQ, and MSDN (under varying names and on varying levels of detail).

The integration styles that SOAs can leverage can be compared along these dimensions:

Integration Style	Time Autonomy	Reference Autonomy	Platform Autonomy	Format Autonomy
File Transfer[12]	*+*	*+*	*(+)*	-
Shared Database[13]	*+*	*+ or (+)*	*(+)*	-
Remote Procedure Invocation or Call (RPC)[14]	-	*(+)*	*(+)*	-

[8] http://dret.net/netdret/docs/loosely-coupled-www2009/
[9] https://gitlab.dev.ifs.hsr.ch/ZIO/tree/master/www2009.eprints.org/92/1/p911.pdf
[10] http://esocc2016.eu/keynotes/
[11] http://www.redbooks.ibm.com/abstracts/sg246346.html?Open

Integration Style	Time Autonomy	Reference Autonomy	Platform Autonomy	Format Autonomy
Messaging[15]	+	+	+	*(+) with EIPs and protocol headers*
The Web (REST)	+	+	+	*(+) with CMTs and HTTP content negotiation (but actual format still need to be agreed upon)*

SOA Patterns

SOA combines and extends PoEAA, EIP and DDD patterns (and others) in an enterprise computing context, including Service Layer, Remote Facade, Data Transfer Object (DTO) from PoEAA; Enterprise Integration Patterns (EIP) such as Process Manager, Command Message, Document Message and Content-Based Router. SOAs can also apply patterns from other languages and domains, for instance remoting patterns as taught in the VSS lecture.

The figure "Core SOA patterns" and the following patten texts are excerpted from "An Architectural Decision Modeling Framework for Service-Oriented Architecture Design" (Zimmermann (2009)). As shown in the figure, the essence of the style is the decoupling of service consumer and service provider via the service contract, ESB messaging, and the service registry.

More style-specific patterns can be found in "Service Design Patterns" by R. Daigneau (see book website[16] and "SOA Patterns" by A. Rotem-Gal-Oz (Rotem-Gal-Oz (2012)).

Service Consumer-Provider Contract (also known as Service Contract). "A service contract defines a service invocation interface. The contract has a functional and a behavioral part. The functional contract is machine-readable and specifies one or more operations which comprise request and, optionally, response messages. A service provider realizes the operations defined in the contract; a service consumer invokes them. We jointly refer to a service provider-contract pair as service. A service consumer sends a request message to invoke an operation defined in the functional contract; optionally, the service provider returns a result as a response message. The behavioral part of the service contract defines the non-functional characteristics of the message exchange and the operation invocation semantics.

The motivating principles for this pattern are modularity and platform transparency. [...] The service contract separates interface and implementation; it is the only knowledge shared by service consumer and service provider. Security policies are examples of non-functional aspects expressed in the contract (e.g., should a request message be encrypted?). Operation invocation semantics include pre- and postconditions. The service lifecycle (e.g., provisioning and decommissioning of providers) is not exposed in the behavioral part of the contract, i.e., no

[12] http://www.enterpriseintegrationpatterns.com/patterns/messaging/FileTransferIntegration.html
[13] http://www.enterpriseintegrationpatterns.com/patterns/messaging/SharedDataBaseIntegration.html
[14] http://www.enterpriseintegrationpatterns.com/patterns/messaging/EncapsulatedSynchronousIntegration.html
[15] http://www.enterpriseintegrationpatterns.com/patterns/messaging/Messaging.html
[16] http://servicedesignpatterns.com/

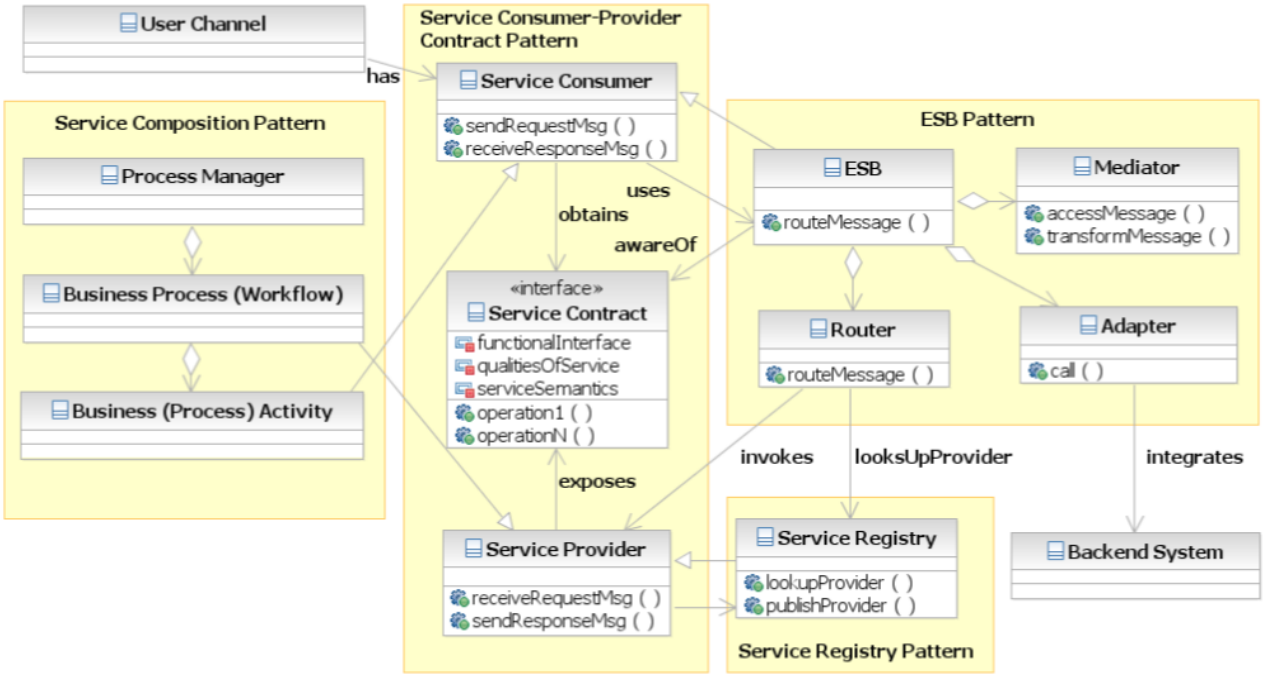


Figure 1: *Core SOA patterns and their interworking*

distributed call stacks or heaps exist, which would require a remote memory management. This gives services an "always-on" appearance from a consumer's perspective. As a consequence, the request and response messages can only be exchanged as documents which do not include any memory references.

Once such contract has been agreed upon, the implementation details of service providers are hidden from the consumers; they can change without effect on service consumers. Consumer and provider do not have to be implemented in the same programming language; they can run on multiple hardware and operating system platforms. Assuming that service invocations do not have any unspecified side effects such as uncoordinated manipulations of enterprise resources, service consumers can share and reuse service providers freely.

The principles and similar patterns are known from object-oriented programming and distributed (client/server) computing. Many technology platforms can be used to implement them. If Web services technologies are used, the Web Service Description Language (WSDL) defines functional service contracts. WS-Policy can be used to specify non-functional characteristics. Web Service Semantics (WSDL-S) annotates WSDL contracts with semantic descriptions; several other notations have been proposed. At runtime, service consumers and providers exchange SOAP messages to transfer request and response documents which are formatted according to the WSDL contract. All these technologies are based on XML languages."

Enterprise Service Bus (ESB). "The ESB pattern is the SOA-specific refinement of the general broker pattern: An ESB provides a remote communication infrastructure that allows service consumers and service providers to exchange request and response messages using one or more message exchange patterns, communication protocols, and message exchange formats.

In general, brokers provide many-to-many connectivity between technically diverse and physically distributed communication parties; they decouple the parties from each other. The primary responsibility of an ESB is to route request and response messages. Introducing a central ESB and a service registry creates a hub-and-spoke architecture known from EAI middleware; a direct communication variant of the broker pattern also exists, in which the communication partners know about each other and address each other directly. ESBs support message exchange patterns such as synchronous request-reply invocations, asynchronous one way messaging,

and publish-subscribe.

Unlike traditional message brokers, an ESB is aware of the type and structure of the messages exchanged: The ‚S' in the pattern name refers to a machine-readable service contract. The ‚E' in the pattern name indicates that the ESB must provide architectural qualities that make it possible to overcome the integration challenges. For instance, high volumes of messages have to be processed, possibly exchanging large amounts of data over local or wide area networks when transferring enterprise resource information (data). Channels serving human users must respond instantly; sub-second response time is often required […].

The World Wide Web (WWW) as a distributed communication infrastructure partially implements the ESB pattern, providing universal connectivity over a single protocol, HTTP. Advanced ESBs also provide protocol transparency. Multiple communication protocols are supported, including HTTP, but also asynchronous message queuing, e.g., via Java Message Service (JMS) providers, and, with the help of adapters, proprietary protocols used by legacy systems. Other advanced ESB capabilities are content- and workload-based routing and *mediations*. Mediations transform request and response messages if service consumer and provider use different formats. This provides format transparency to service consumers and providers. ESBs also provide access to the message payload (i.e., request and response message data) for security and systems management purposes, e.g., authentication, authorization, monitoring, and billing. The service contract is interpreted to process the payload."

Service Composition. "If two or more service providers are assembled into another service provider we speak of service composition. The assembled service provider invokes the composed service providers via their service invocation interfaces. Service composition may form a dedicated architectural layer in an SOA.

If an enterprise application employs SOA principles such as modularity, many different service providers with rather diverse responsibilities may exist, e.g., technical logging services, atomic business functions such as customer lookups and address validations, and entire business processes such customer enquiries and claim processing in the insurance industry example. The characteristics of these service providers differ. To avoid a tight coupling between service consumers and providers with different responsibilities and quality attributes as well as undesired dependencies between the services, the permitted invocations must be defined. For instance, a process service may be permitted to invoke a business function service, but not to invoke a technical service directly. Similarly, a technical logging service should be unaware of its consumers and not call a process service itself. Otherwise, a change to the interface of the process service, which is required to respond to a change in the business requirements, requires the technical utility service to be changed as well (if the change is not backward compatible). This violates the modularity principle and degrades the maintainability and portability of the application.

As a solution, the SOA should be organized into three or more logical layers. Selecting the layers pattern is an architectural decision driven by the desire for structure and flexibility: Architectural elements in a particular logical layer fulfill a certain architectural responsibility jointly and cohesively. They only interact with each other and with architectural elements in adjacent lower layers; interfaces isolate the layers from each other. As a result, layer implementations can seamlessly switch from one technology to another, without causing a need to change the architectural elements contained within one of the adjacent layers.

Traditionally three logical layers are used […]: The presentation layer contains all rich or thin client logic displaying user interfaces to human users. In an SOA, many service consumers reside in the presentation layer. The domain layer contains business logic such as control flow, but also calculations and modifications of enterprise resources. It is typically activated in response to stimuli from the presentation layer or from other systems (e.g., when realizing business event and timeout management). The data source layer lets enterprise resources and other data persist. It also provides interfaces allowing the domain layer to access the data when executing its logic.

The service composition pattern refines the above logical layering scheme: The domain layer is divided into two sub-layers, a service composition layer and an atomic service layer. Service providers either reside in the atomic service layer or, as composed services, in the service composition layer. The implementations of atomic services in a programming language may also reside in the atomic service layer or be placed in an additional component layer. […].

Business activities that are assigned to end users are placed in the service composition layer. The service composition layer keeps track of the conversational state. Business activities invoke services in the atomic service layer. The atomic service layer contains calculations and manipulations of enterprise resources, which are not permitted to invoke services in the service composition layer. A business process manager (e.g., a workflow management system) is the central middleware component in the service composition layer. It is aware of the business activities that have to be performed and the appropriate order, which defines an executable business process control flow (a.k.a. workflow). Each process manager can host more than one process; it is responsible for creating and terminating process instances, and relating incoming requests to such process instances (correlation). Such process instances may run for a long time: The process manager can ensure that the logical order of the process execution is adhered to and that the integrity of enterprise resources is preserved throughout the process lifetime and across user channels (coordination). This includes handling logical and technical processing errors (e.g., invalid request data, network connectivity problems). The process manager can also ensure that process instances complete in a timely manner if that is a business requirement.

Having divided the business logic this way, flow independence can be achieved; just like presentation and domain layer are unaware of the way a relational database stores the enterprise resource data and optimizes access to it (providing data independence), the basic computations in the atomic service layer are unaware of the way they are composed into business processes.

If workflow patterns and technologies are used to realize the service composition layer, the formal foundations for its execution semantics can be Petri nets, Pi-calculus, or graph theory. One technology option is the Web Services Business Process Execution Language (WS-BPEL or, in short, BPEL), which evolved from several proprietary languages and has been standardized."

Service Registry. "A service registry provides information about services that can be invoked via the ESB. It makes service contracts and service provider access information available to the ESB and to service consumers. Organizational information such as service ownership, service level agreements, and billing information can optionally be stored in the service registry as well.

To ensure flexibility during deployment and service invocation, service consumers should not use fixed service provider addresses; ideally, they should even be unaware of the actual service provider and let the ESB decide where to route a service request to (e.g., for load balancing purposes). To provide such location transparency is the objective of the service registry pattern. A service registry provides a design time interface to architects and developers which allows these users to publish and lookup service contracts and providers. At runtime, a service registry may also act as a service provider, so that service consumers in other applications have access to the information about service contracts and service providers that is stored in the registry.

Selecting service providers at runtime is an advanced usage scenario for a service registry; such dynamic lookup requires semantic annotations that can be used to automate the provider lookup based on the Quality of Service (QoS) expected by a consumer. Service consumers and providers are no longer aware of each other (service virtualization). Many open research and industry adoption challenges exist, such as trust, negotiation, and monitoring of dynamically negotiable service level agreements.

This pattern is the SOA pendant of naming and directory services known from the Common Request Broker Architecture (CORBA), Java Enterprise Edition (JEE), Distributed Computing Environment (DCE), and other remoting middleware. The Universal Description, Discovery, and Integration (UDDI) specifications realize the

service registry pattern in a Web services context; vendor-specific UDDI extensions and alternative realizations exist."

Example(s)

Let us assume that an insurance company uses three SAP R/3, MS Visual Basic, and COBOL applications to manage customer information, check for fraud, and calculate payments. The user interfaces (UIs) are the only access points. A multi-step, multi-user business process for claim handling, executing in IBM WebSphere, is supposed to reuse the functions in the existing applications. How to integrate the new business process with the three legacy applications in a flexible, secure, and reliable way?

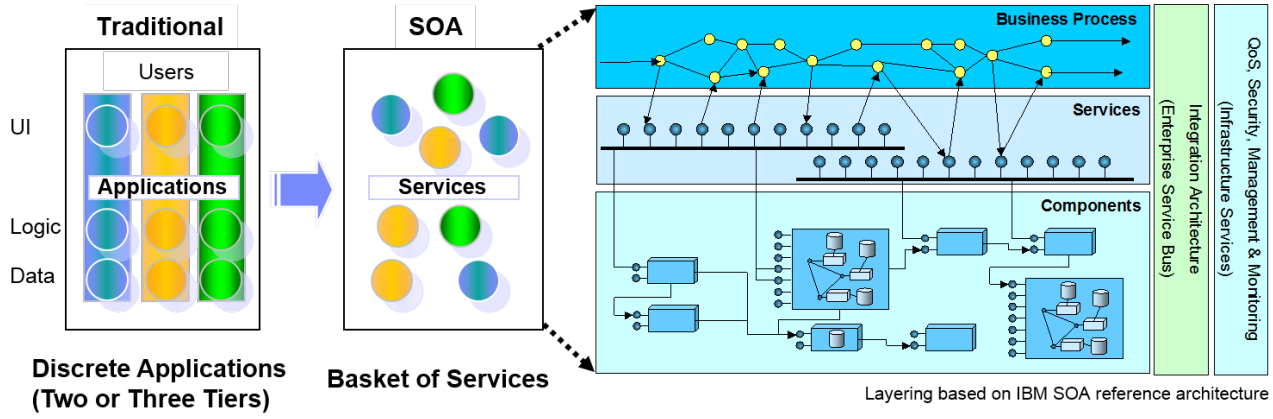


Figure 2: *SOA Layers in an Insurance Scenario*

An exemplary table format for a service contract/service model is shown in the figure "Service Model with Business Content" below.

Application of the Concept in Products and Projects

Usage of Concept/Topic

Many SOA reference projects exist. While some SOA project did not succeed indeed, an anti-SOA attitude that sometimes can be observed is not backed by evidence. Two successful SOA projects are described in Zimmermann et al. (2005) and Brandner et al. (2004)

Concept/Topic: *REpresentational State Transfer (REST)*

REST is an architectural style for Web-based application integration. REST is defined via constraints that are added to a theoretical empty style:

1. Client-server
2. Stateless
3. Cacheable
4. Uniform interface
5. Layered system
6. Code on demand (optional), see Section 5 of the REST thesis[17](Fielding (2000))

[17] http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

***Service Contract:* [Name]**	
Business domain (scenario viewpoint, functional area): • […]	*User stories and quality attributes (design forces):* • […]
Service quick reference (synopsis of what this service provides to consumers): • […]	
Invocation syntax (functional contract): IDL specification, security policy; request/response data samples; endpoint addresses (test deployment, production instances); sample service consumer program (source code); error handling information (error codes, exceptions) • […]	
Invocation semantics (behavioral contract): informal description of preconditions, postconditions, invariants, parameter meanings; FSM; service composition example; integration test cases • […]	
Service Level Agreement (SLA) with Service Level Objectives (SLO); Quality-of-Service policies • […]	
Accounting information (service pricing); external dependencies/resource needs • […]	
Lifecycle information: current and previous version(s); limitations, future roadmap; service owner with contact information, link to support and bug tracking system • […]	

Figure 3: *Service Model with Business Content*

Key concepts in REST are *linked resources* identified by *URIs*, resource *representations* (external views), and the unified method set (HTTP verbs). Constraint 2 (statelessness) and 4 (uniform interface) require "identification of resources; manipulation of resources through representations; self-descriptive messages" according to Section 5.2 of R. Fielding's thesis (Fielding (2000)). In other words, REST as such is *not* an API technology or protocol; talking about a "REST API* therefore is either an oxymoron or an incomplete statement.

RESTful HTTP is the dominating implementation of the REST style. The acronym REST is more popular than the actual constraints, which go as far as banning cookies entirely; all state information has to travel in links (which should be typed and disclose media type information in addition to link type and target; more on this further down in this fact sheet).

Maturity Levels

RESTful HTTP is one prominent incarnation of the REST style (when done right); REST is defined via constraints. Since the concepts in the original work on REST come across as somewhat abstract, book authors from industry came up with additional explanations. One of these is the *Richardson Maturity Model* explained in an article by M. Fowler[18]:

- *Level 0:* No particular rules other than usage of HTTP; e.g., Plain Old XML (POX) as message exchange format
- *Level 1:* Resources
- *Level 2:* HTTP verbs
- *Level 3:* Hypermedia controls

[18] http://martinfowler.com/articles/richardsonMaturityModel.html

One can only claim to use REST if level 3 is reached. Not all Web APIs can be considered truly RESTful; only hypermedia-based APIs that meet all constraints are (for instance, they have to use hypertext as the engine of application state).

Maturity Levels 1 and 2: REST and HTTP

See Figure "REST and HTTP" courtesy of C. Pautasso (USI Lugano).

- **HTTP protocol primitives**
  - Sent by client
  - Understood by server
  - Client e.g. Web browser (or application client in integration scenario)

CRUD	REST	
CREATE	POST/PUT	Initialize the state of a new resource at the given URI
READ	GET	Retrieve the current state of the resource
UPDATE	PUT	Modify the state of a resource
DELETE	DELETE	Clear a resource, after the URI is no longer valid

- **URI-reference = [absoluteURI | relativeURI] ["#" fragment]**

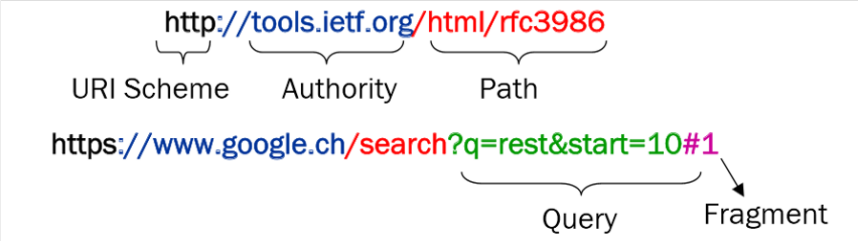


Figure 4: *REST and HTTP*

Maturity Level 3: Hypertext as the Engine of Application State (HATEOAS)

Motivation and examples for Level 3 (HATEOAS) can be found here[19], here[20] and in the RESTBucks example[21] often presented by the authors of "REST in Practice" (Webber, Parastatidis, and Robinson (2010)). Chapter 5 of "REST in Practice" is a recommended read for those who are serious about REST and therefore want to reach level 3.

Let us decompose the HATEOAS acronym into its components:

- Application State (AS) represents the state of processing (e.g., activities in business workflow) including preconditions, post conditions (e.g., for use case execution) and the content/values of backend information systems and databases (a.k.a. enterprise resources)[22]
- Media Types (MTs) are seen to be the carriers of AS (the "E" in HATEOAS), serving as typed data transfer objects for HTTP request and response payload (schema language depends on MIME type, e.g. XML Schema or RelaxNG for XML, JSON Schema for JSON. See this blog post[23] for more options.

[19] http://restcookbook.com/Basics/hateoas/
[20] https://www.subbu.org/blog/2008/10/generalized-linking
[21] http://www.infoq.com/articles/webber-rest-workflow
[22] also see separate fact sheet
[23] http://www.foxycart.com/blog/the-hypermedia-debate

- Typed links (hypertext) as identifiers of AS (the "H" in HATEOAS): In a blog post[24], R. Fielding defines hypertext almost synonymously to hypermedia: "simultaneous presentation of information and controls such that the information becomes the affordance through which the user (or automaton) obtains choices and selects actions."

Under the HATEOAS paradigm (or dogma?), the client discovers possible interactions with resources in responses; only the *home URI* of a given resource set is fixed and shared upfront; this leads to a partial specification of the overall workflow and the need for *dynamic contracts*. Internet media types (RFC 2045) play a key role; media types are also called MIME type or content type. This information is part of HTTP response (header), e.g., `application/xml`. This makes REST (and HTTP) extremely flexible and extensible (in case of changes). However is not obvious how to test in such a flexible and dynamic setting. Other open questions are:

- How to meet Service Level Agreements (SLAs)?
- How to satisfy audit requirements?
- How to react on the client side when unexpected link types or unexpected processing orders are disclosed in the dynamic contract (that changes over time)?

HATEOAS example. An example from Paypal[25] introduces "three components for each link in a HATEOAS links array:

- `href`: URL of the related HATEOAS link you can use for subsequent calls.
- `rel`: Link relation that describes how this link relates to the previous call.
- `method`: The HTTP method required for the related call."

Standard media types or custom developed ones can be used. Related standardization efforts include RFC 5988, RFC 6573, RFC 6906 and microformats.

Even RESTafarians admit that REST client design and implementation is hard in the absence of frameworks (Webber, Parastatidis, and Robinson (2010)). Also see this StackOvervflow post[26].

Media Types and Link Types

According to RFC 2016[27], hundreds of media types are already defined. Some important, frequently used ones are:

```
application/xml
application/atom+xml
text/plain
image/png
```

Selecting an existing media type or creating a new, domain-specific *Custom Media Type (CMT)* is an important design decision when crafting RESTful HTTTP APIs (which is different but can be compared to XML schema design for WSDL/SOAP services). Some of the available implementation technologies are:

- JSON-LD and HAL, as motivated in the REST Cookbook[28].
- Atom Syndication Format (ASF) is one example of a media type; it is defined in RelaxNG (due to concerns about the usability of XML Schema)

[24] http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven
[25] https://developer.paypal.com/docs/integration/direct/paypal-rest-payment-hateoas-links/
[26] https://stackoverflow.com/questions/1164154/is-that-rest-api-really-rpc-roy-fielding-seems-to-think-so
[27] http://www.ietf.org/rfc/rfc2046.txt
[28] http://restcookbook.com/Mediatypes/json/

- M. Amundsen has a web page on media types[29] such as `Collection+JSON` (which has not been updated in a while)
- Collection JSON[30]
- RFC 5988[31]

Service Contracts in REST: Swagger and Alternatives

Swagger is a specification and complete framework implementation for describing, producing, consuming, and visualizing RESTful web services (according to its website).

The Swagger language specification defines two concrete syntaxes, a JSON-based one and a YAML-based one. Its abstract syntax is inspired by HTTP. Swagger uses an "extended subset of the JSON Schema Specification", which couples it tightly to that media type to some extent (but also references XML); several types of parameters are supported (path, query, header, and cookie), see the parameter object section of the specification[32].

Starting with version 3.0, Swagger has been renamed to OpenAPI Specification (OAS) because more firms joined. There is an online editor, and code code generators are available for Java and other language:

- Swagger specification[33]
- SwaggerHub[34]
- Swagger Onine Editor[35]

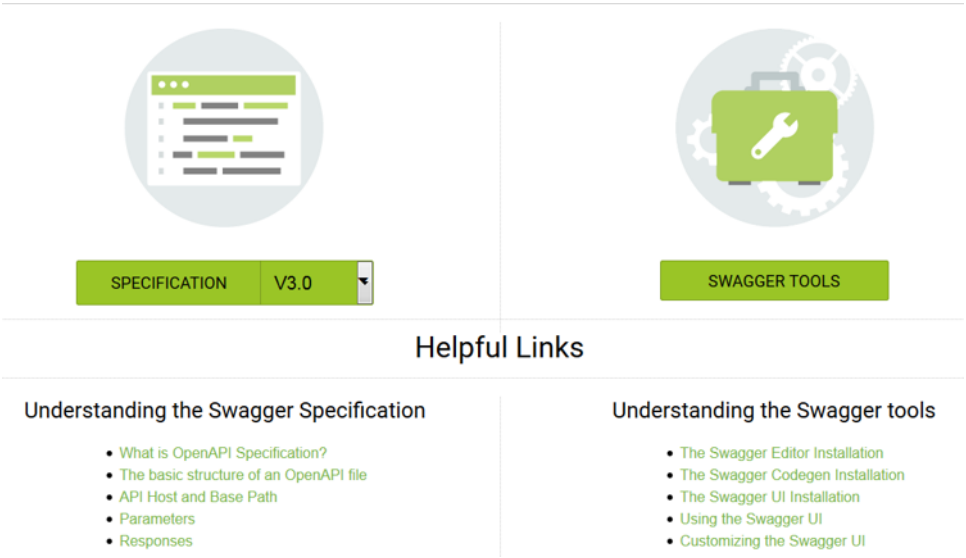


Figure 5: *Swagger deliverables (source: OAS website as of December 2016)*

Critique. The Swagger language centers on HTTP rather than REST (which are conceptually close not not identical, architectural style vs. protocol implementing the style and other things). OAS/Swagger supports REST maturity levels 1 and 2 properly (`path` concept and verbs such as GET and POST, see `PathItemObject` in the specification[36]). Level 3 is not fully supported (which would be hard to do, see discussion in this article

[29] http://amundsen.com/media-types/
[30] https://github.com/collection-json
[31] http://tools.ietf.org/pdf/rfc5988.pdf
[32] https://swagger.io/specification/#parameterObject
[33] https://swagger.io/specification/
[34] https://swaggerhub.com/
[35] https://editor.swagger.io//#/
[36] https://swagger.io/specification/#pathItemObject

on REST contract design[37] by S. Allamaraju: "also note that a completely machine-readable description of a contract for a [level 2 3 RESTful HTTP] remote interface is a fallacy"). HATEOAS and state are not mentioned at all. There is a notion of static links that can be defined in a link object[38] as explained here[39] but dynamic links are declared to be out of scope.

Link types and (custom) media types would deserve more attention, as well the OPTIONS method/verb in HTTP. An extension would be needed to model (semi-dynamic) hyperlinking. Links must have three properties: semantic type/nature of the link (class-level type/flow information), data/media type (resource representation, view), and address (instance information). Moreover, the language could feature resources, representations, resource relationships, and state transitions even more prominently in its concrete and abstract syntax.

The language comes across as rather detailed and even verbose; tool support (editors, generators) is needed to work with it efficiently in practice. Note that the use of an "operationId" such as `getPetsById` comes across as somewhat odd (remember the RPC critique for SOAP?). Also note that "enhanced subset of" is a somewhat strange term; reviewer of the specification, reviewers should request clarification and challenge the need to first reduce and then extend again.

Resource Access Markup Language (RAML). An alternative to Swagger is (or was) RAML[40] from MuleSoft that used a YAML syntax to specify similar concepts. An Eclipse plugin supporting JAX-RS code generation and consumption exist(ed). However, RAML also joined the OpenAPI Initiative, so it is unclear whether it will continue to exist as a separate language or whether some of its concepts will find their way into future versions of OAS: https://swagger.io/mulesoft-joins-the-openapi-initiative/

Other contract language options. Other alternatives briefly touched upon in the lecture are WSDL 2.0[41] and WADL[42].

Examples and Technologies

A Spring MVC example can be found at http://websystique.com/springmvc/spring-mvc-4-restful-web-services-crud-example-resttemplate/. A Spring example defining custom media types is: http://www.baeldung.com/spring-rest-custom-media-type.

A (somewhat dated) JAX-RS example is at http://www.mkyong.com/tutorials/jax-rs-tutorials/.

PayPal features HATEOAS here: https://developer.paypal.com/docs/integration/direct/paypal-rest-payment-hateoas-links/.

The AtomPub protocol is RESTful, see http://tools.ietf.org/html/rfc5023 and http://bitworking.org/projects/atom/draft-ietf-atompub-protocol-17.html.

Standards, tools and libraries (also see exercise):

- Jersey for JAX-RS (JEE API for RESTful HTTP)
- Jackson for JSON from/to Java (and XML)
- OAS/Swagger, RAML, Restdocs for service contract
- API gateway products
- Postman for testing

[37] https://www.infoq.com/articles/subbu-allamaraju-rest
[38] https://swagger.io/specification/#linkObject
[39] https://swagger.io/docs/specification/links/
[40] http://raml.org/index.html
[41] http://www.w3.org/TR/wsdl20/
[42] http://wadl.java.net/

The Eclipse Micro Profile[43] combines JAX-RS, JSON-P, CDI (it is promoted by subset of the JEE application server community).

Best Practices fir RESTful HTTP

S. Allamaraju's "RESTful Web services Cookbook"[44] is recommended for everybody who wants to implement RESTful HTTP APIs that meet the style constraints (Allamaraju (2010)). It presets many *recipes* rather than patterns, but uses a similar writing style. "REST in Practice" (Webber, Parastatidis, and Robinson (2010)) has some biased sections, but communicates the vision of REST in practical terms (including code snippets and architecture overview diagrams, e.g., in Chapter 5 on hypermedia).

Other useful resources (note: polyseme!) include an online REST API Tutorial[45] that gives advice such as:

- Unless the costs of offering both JSON and XML are staggering, offer them both.
- Provide resource discoverability through links.
- Support versioning via content negotiation.

More REST API design advice can be found here[46] and here[47].

Some "anti patterns" (that bring us back to the defining constraints of REST) called out at InfoQ[48] are:

- Tunneling everything through GET
- Tunneling everything through POST
- Ignoring caching
- Ignoring response codes
- Misusing cookies
- Forgetting hypermedia
- Ignoring MIME types
- Breaking self-descriptiveness

One could add "inappropriate REST maturity level for the sake of style compliance" (is HATEOAS always possible and needed, so its usage justified?), "under-specifying service contract and request/response payloads" and "technology dogmatism" to this list.

Comparisons

"Should I use REST or SOAP/WSDL" is an invalid design question because SOAP is an XML message format (with an RPC perception) and REST is an architectural style. Such "apple vs. orange" comparisons are often found online. More appropriate questions offering options on the same level of abstraction are:

- Integration style? Optiona: File Transfer, Shared Database, RPC, Queue-Based Messaging.
- Local Application Programming Interface (API)? JAX-RS (supporting RESTful HTTP) vs. JAX-WS (supporting SOAP/HTTP).
- Payload design and format: XML vs. JSON? Representation structure? Application and transport protocol: HTTP over TCP/IP or AMQP?

- Service contract language: Swagger or WSDL or other?

[43] https://microprofile.io/
[44] https://www.subbu.org/blog/2010/03/restful-web-services-cookbook-released
[45] http://www.restapitutorial.com/
[46] http://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/
[47] https://restful-api-design.readthedocs.org/en/latest/
[48] http://www.infoq.com/articles/rest-anti-patterns

- Quality of Service (QoS) considerations, e.g. how to achieve security, transactionality (consistency), reliability

On a more general note: discussions of SOA vs. REST; WS-* vs. RESTful HTTP often are very controversial, sometimes they even get emotional. An early attempt to objectify the discussion is "Restful web services vs. big web services: making the right architectural decision"[49].

More information on SOA, microservices, and REST is available via https://www.ifs.hsr.ch/index.php?id=15266&L=4 and http://soadecisons.org.

Concept/Topic: *Microservices (as an Implementation Approach to and Sub-Style of SOA)*

Context and Motivation

Microservices evolved as an implementation approach to SOA that leverages recent advances in agile practices, cloud computing and DevOps. Microservices Architecture (MSA) constrains the SOA style to make services independently deployable and scalable (e.g., via decentralization and autonomy of teams and decisions).

Definition and Comparison

The essence of the microservices approach, about which consensus can be reached, is:

- Business alignment and loose coupling (goal: agility/flexibility)
- Independent deployability and scalability (and therefore replaceability)
- Decentral decisions (service autonomy)
- Single reason to change per service

A consolidated definition assembled from multiple sources is (Zimmermann (2017)):

1. Fine-grained interfaces to single-responsibility units that encapsulate data and processing logic are exposed remotely to make services independently scalable, typically via RESTful HTTP resources or asynchronous message queues.
2. Business-driven development practices and pattern languages such as Domain-Driven Design (DDD) are employed to identify and conceptualize services.
3. Cloud-native application design principles are followed, e.g., as summarized in Isolated State, Distribution, Elasticity, Automated Management and Loose Coupling (IDEAL).
4. Multiple storage paradigms are leveraged (SQL and NoSQL) in a polyglot persistence strategy; each service implementation has its own data store.
5. Lightweight containers are used to deploy and scale services.
6. Decentralized continuous delivery is practiced during service development.
7. Lean, but holistic and largely automated approaches to configuration and fault management are employed within an overarching DevOps approach.

From SOA to Microservices

SOA vs. microservices comparisons are often biased (just like the REST vs. Web services one). Three options how to position microservices are:

[49] http://dl.acm.org/citation.cfm?id=1367606

1. New architectural style different from SOA.
2. Same old architecture.
3. Implementation approach and physical refinement of SOA (or sub-style establishing additional constraints and principles).

See IFS website "Microservices Resources and Positions"[50] for more information and evidence for position 3: Services are here to stay, but microservices do not constitute a new style.

Characteristic	Viewpoint/Qualities/Benefit	SOA Pendant
Componentization via services	Logical Viewpoint (VP): separation of concerns improves modifiability	Service provider, consumer, contract (same concept)
Organized around business capabilities	Scenario VP: OOAD domain model and DDD ubiquitous language make code understandable and easy to maintain	Part of SOA definition in books and articles since 200x (e.g. Lublinsky/Rosen)
Products not projects	n/a (not technical but process-related)	(enterprise SOA programs)
Smart endpoints and dumb pipes	Logical Viewpoint (VP): information hiding improves scalability and modifiability	Same best practice design rule exists for SOA/ESB (see e.g. here)
Decentralized governance	n/a (not technical but process-related)	SOA governance (might be more centralized, but does not have to; "it depends")
Infrastructure automation	Development/Deployment VP: speed	No direct pendant (not style-specific, recent advances)
Design for failure	Logical/Development/Deployment VP: robustness	Key concern for distributed systems, SOA or other
Evolutionary design	n/a (not technical but process-related): improves replaceability, upgradeability	Service design methods, Backward compatible contracts

Figure 6: *SOA vs. Microservices by Criteria and Concerns (Zimmermann (2017))*

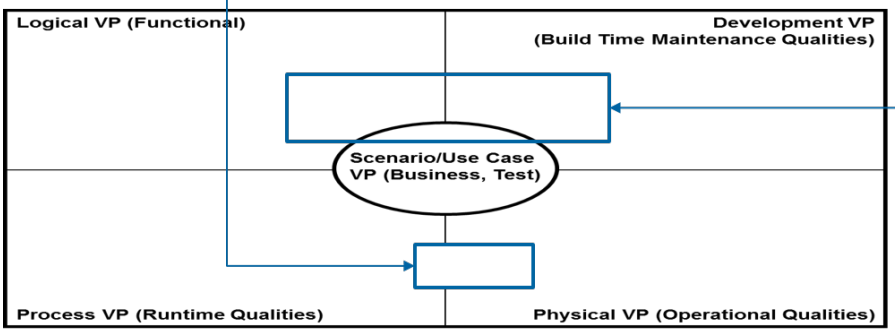
Pattern example: API Gateway (wit ESB Comparison). See lecture slides and pattern description on C. Richardson's website[51] for pattern introduction.

The context section of the API Gateway pattern text indicates that the pattern was written up with frontend-to-backend integration in mind, not backend-to-backend. ESBs can support both of these integration scenarios (sometimes "external ESBs" are distinguished from "internal ESBs"); most ESB examples show backend-to-backend integration, though. Turning to the three core ESB capabilities routing, adaptation and transformation: routing is explicitly mentioned in the solution section of the API Gateway pattern; adaptation and transformation are required to resolve the pattern forces and therefore also touched upon in the solution section of the pattern text.

Positioning in an overall architecture (and usage scenario) are somewhat different; according to the API Gateway pattern author, an API Gateway serves requests from clients (which according to one comment towards the end of the online article cannot be other microservices); ESB serves requests from other services (but can also be positioned at the edge of a service ecosystem). Furthermore, the API Gateway description comes across as somewhat more local (decentralized), instances of the pattern can perform load balancing and (somewhat questionable) service composition.

[50] https://www.ifs.hsr.ch/index.php?id=15266&L=4
[51] http://microservices.io/patterns/apigateway.html

Application Component Property (Gartner/TMF)	Mapping to 4+1 Viewpoint Model (Kruchten 1995)	Mapping to ZIO Tenet	Novel or "Same Old Architecture"?
tightly scoped	Scenario/Use Case, Logical	1, 2	Good practice in SOA
strongly encapsulated	Logical, Development	1	SOA definition
loosely coupled	Development, Process (Integr.)	1, 3	SOA definition
independently deployable	Process, Physical	1	(somewhat) novel
independently scalable	Process, Physical	1	(somewhat) ovel



View model adapted from: P. Kruchten, 4+1 views on SWA, IEEE Software.

Figure 7: *SOA vs. Microservices by Viewpoints*

Microservices: Pros and Cons

All AppArch/SOA design challenges still present, and new ones arise. Microservices are not suited for each and every project (at early evolution stages).

More tradeoffs and pros/cons are discussed in many online articles, including:

- M Fowler, "Microservice trade-offs"[52]
- "Failing at microservices"[53]
- "Microservices not a free lunch"[54]

Any such article should be run through a friend-or-foe test before the reported advice is followed: is the definition and positioning of microservices compatible with that of your current project? does the reported experience and guidance come from a similar enough business and technical context (in terms of NFRs and other ASRs?).

(Micro-)Services Size

Judging from the name, the size of a microservice seems to be an important criterion – but how to define/measure it? The optimal size of a microservice cannot be measured in Lines of Code (LoC) or Jeff Bezos's Two-Pizza Rule for optimal team size *alone*; a microservice should rather be rightsized such that it can be:

- Developed and operated by a single team. New or changed business requirements should ideally lead to changes in just a single microservice (including the user interface).
- Fully understood by each developer on the team.
- Replaced by a new implementation if necessary.

[52] http://martinfowler.com/articles/microservice-trade-offs.html
[53] https://rclayton.silvrback.com/failing-at-microservices
[54] http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html

The above criteria call for a lot of small services. On the other hand, it should not be too small either (rightsizing does not necessarily always mean downsizing):

- Communication and deployment overhead
- Transactions spanning multiple microservices are hard to manage
- The same is true for data consistency (consistency boundaries). A microservice should be large enough to contain the data it needs to operate, and loosely coupled with others.

Additional criteria are:

- Fast delivery (of new versions) and flexibility vs. cognitive load and effort (of upfront design, testing and service maintenance)
  - How often are system components (services) updated while others remain unchanged?
  - Do these changes (have to) break service consumers or not?
- Scalability vs. performance
  - Do different parts of the OOAD domain model (or different bounded contexts in DDD context map) have different workloads and bottlenecks?
  - Can the network handle many (small) remote service invocations over HTTP or messaging protocols such as AMQP?
- Many more can be identified (see more general coupling and granularity discussion below).

One has to be careful not to end up with a (distributed) monolith again.

Coupling Criteria and Granularity Patterns. There is no single definite answer to the "what is the right granularity?" question in SOA and microservices design (or any other distributed system design). This question rather leads to several context-specific dimensions and criteria:

- *Business granularity*: semantic density (role in domain model and BPM)
- *Technical granularity*: syntactic weight and QoS entropy

The format dimension of loose coupling is directly related to service *granularity*. Granularity is a service property that deals with representation formats (JSON, XML), which are part of the service contract: If changes in format require API consumer to change too, consumer and provider are coupled (unless a DDD-style Anti-Corruption Layer, SOA Enterprise Service Bus, or RESTful HATEOAS mediate the format differences). The representations (i.e., content of request and response messages, and supporting data such as URIs with parameters) matter: The "representation" donated the "R" to REST. In other communication paradigms, terms such as in and out message parts (with typed elements), interface operation signatures used. Related SOA/REST terms/concepts include message payload, content/media type, schema, etc.

Business granularity (a.k.a. semantic density) has a major impact on agility and flexibility, as well as maintainability

- Position of service operation in Business Architecture, e.g., expressed in a Component Business Model (CBM) or DDD Domain Model (analysis level)
- Amount of business process functionality covered: Entire process? Subprocess? Activity?
- Number and type of analysis-level domain model entities touched

Technical granularity (a.k.a. syntactic weight) determines runtime characteristics such as performance and scalability, interoperability – but also maintainability and flexibility:

- Number of operations in WSDL contract, number of REST resources and HTTP *idioms* (Webber, Parastatidis, and Robinson (2010))
- Structure of payload data in request and response messages
- QoS entropy adds to the maintenance effort of the service component

- Backend system interface dependencies and their properties (e.g. consistency)
- Security, reliability, consistency requirements (coupling criteria)

Granularity Examples. An e-commerce order management service should also handle the order data. In addition, it will also need access to customer data and product information to fulfill its responsibilities. Which data should the order management service own and control? Only transactional data such as order items, bill, delivery? Or master data as well (customer, products)?

Figure 8 "Sample Service Cut 0: Monolith" shows a first obvious decomposition (or lack thereof).

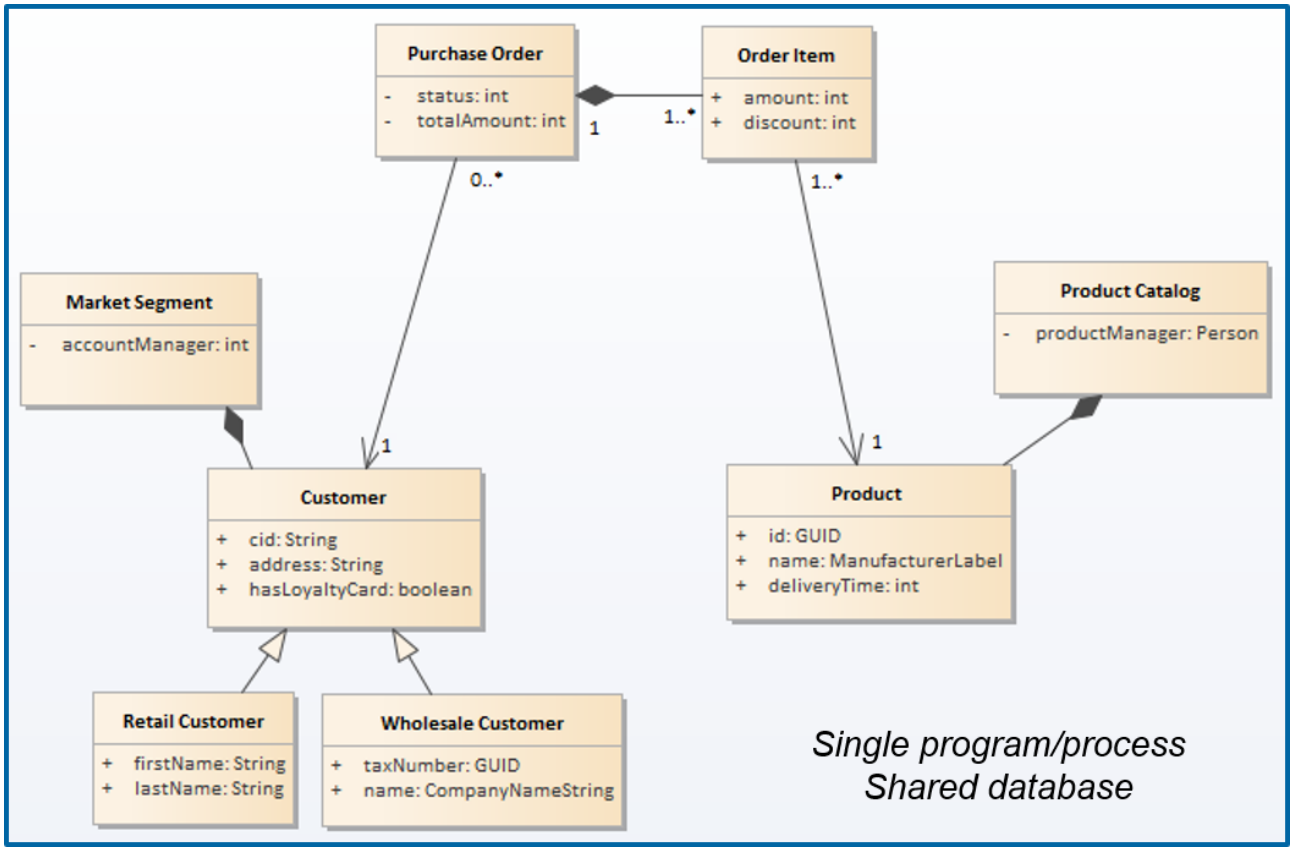


Figure 8: *Sample Service Cut 0: Monolith*

The next figure "Sample Service Cut 1: Bounded Contexts" DDD Bounded Contexts to cut the monolith into two (micro-)services.

An even finer grained cut is shown in the third example in the figure "Sample Service Cut 2: Aggregates".

Service Cutter Method and Tool. *Semantic Proximity* can be observed if service candidates are accessed within same use case (read/write) and/or ervice candidates are associated in OOAD domain model. The coupling impact (note that coupling is a relation not a property) of this criterion is:

- Change management (e.g., interface contract, data definition languages such as SQL)
- Creation and retirement of instances (service instance lifecycle)

The Service Cutter method and tol compils and leverages 16 such criteria in total (see figure "Service Cutter: Couping Criteria Cards (CCC)").

Hypothesis (yet to be validated): Loose coupling between few coarse-grained services is easier to achieve than loose coupling between many fine-grained ones – sharing and delegating responsibilities increases coupling.

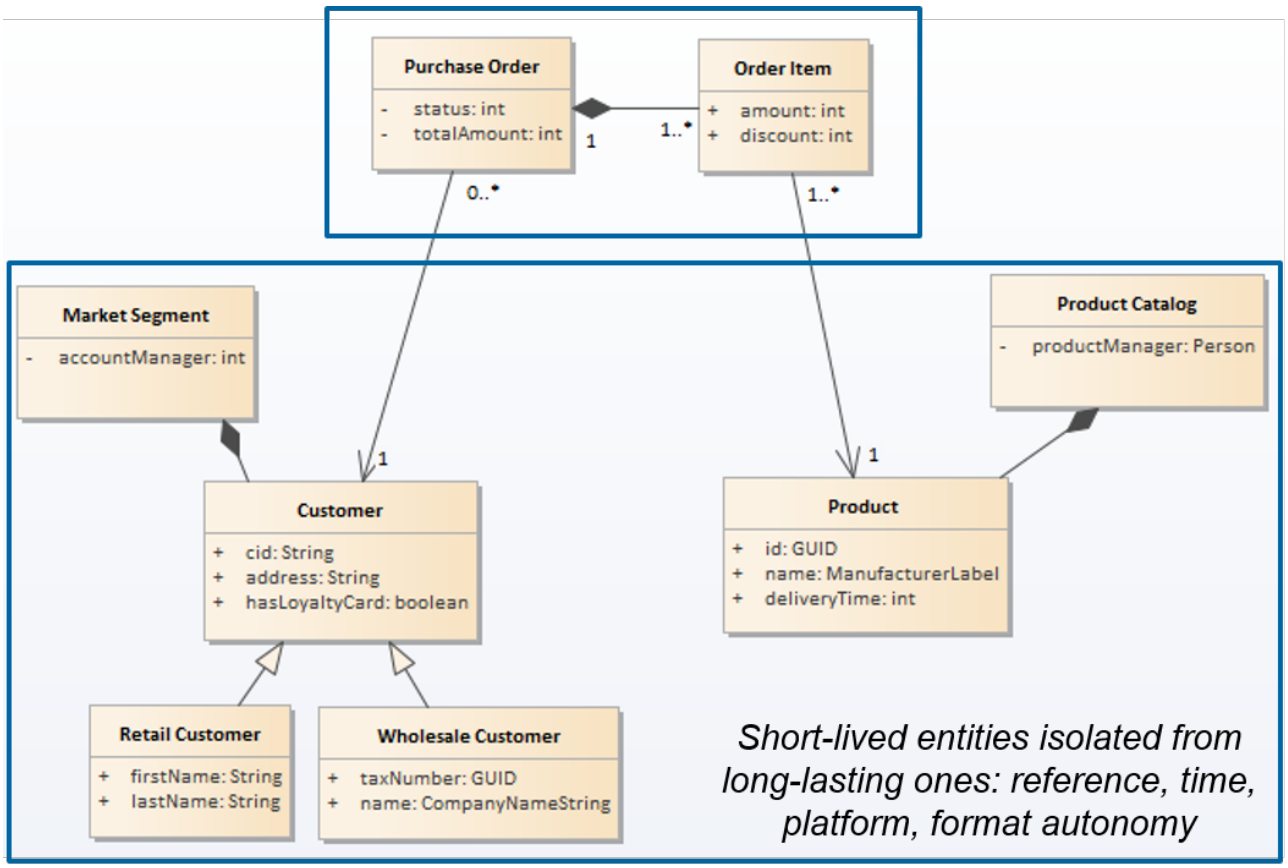


Figure 9: *Sample Service Cut 1: Bounded Contexts*

Web API Design and Evolution (WADE) patterns The figure "Structural Patterns in Web API and Service Design" shows four options for structuring API call request and response messages (e.g., port type operations in WSDL(SOAP and HTTP verbs/methods operating on resources in RESTful HTTP).

More Web API Design and Evolution patterns are currently being mined and documented. See IFS website on WADE[55] and an EuroPLOP 2017 paper[56].

References

Allamaraju, Subbu. 2010. *RESTful Web Services Cookbook*. Sebastopol: O'Reilly Media, Inc.

Brandner, Michael, Michael Craes, Frank Oellermann, and Olaf Zimmermann. 2004. "Web Services-Oriented Architecture in Production in the Finance Industry." *Informatik-Spektrum* 27 (2):136–45. https://doi.org/10.1007/s00287-004-0380-2.

Fielding, Roy Thomas. 2000. "REST: Architectural Styles and the Design of Network-Based Software Architectures." Doctoral dissertation, University of California, Irvine. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

Hohpe, Gregor, and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Josuttis, Nicolai. 2007. *Soa in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc.

Martin Keen, Alan Hopkins, Susan Bishop. 2004. *Patterns: Implementing an Soa Using an Enterprise Service*

[55] https://www.ifs.hsr.ch/index.php?id=15667&L=4
[56] http://eprints.cs.univie.ac.at/5161/1/WADE-EuroPlop2017Paper-FinalSubmissionOct19.pdf

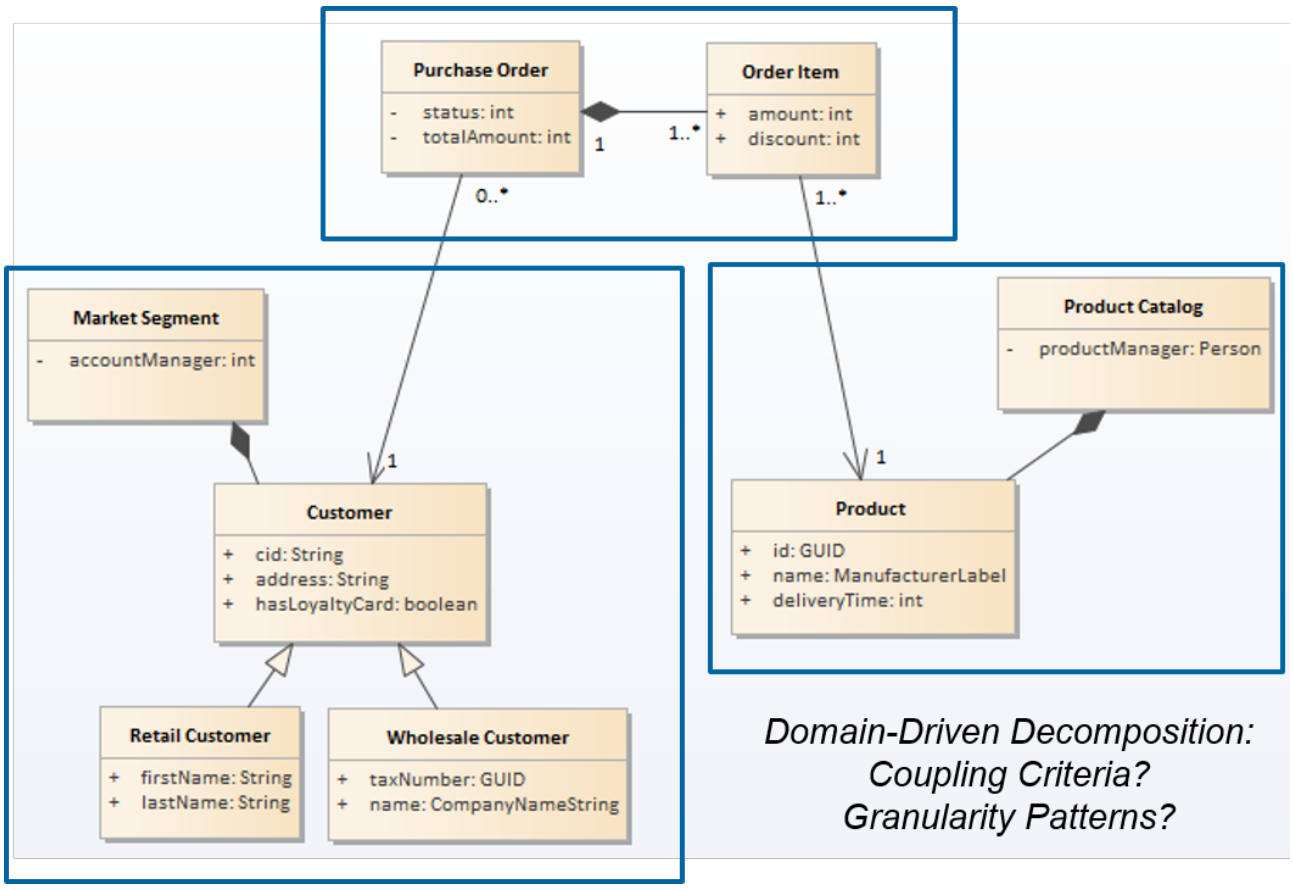


Figure 10: *Sample Service Cut 2: Aggregates*

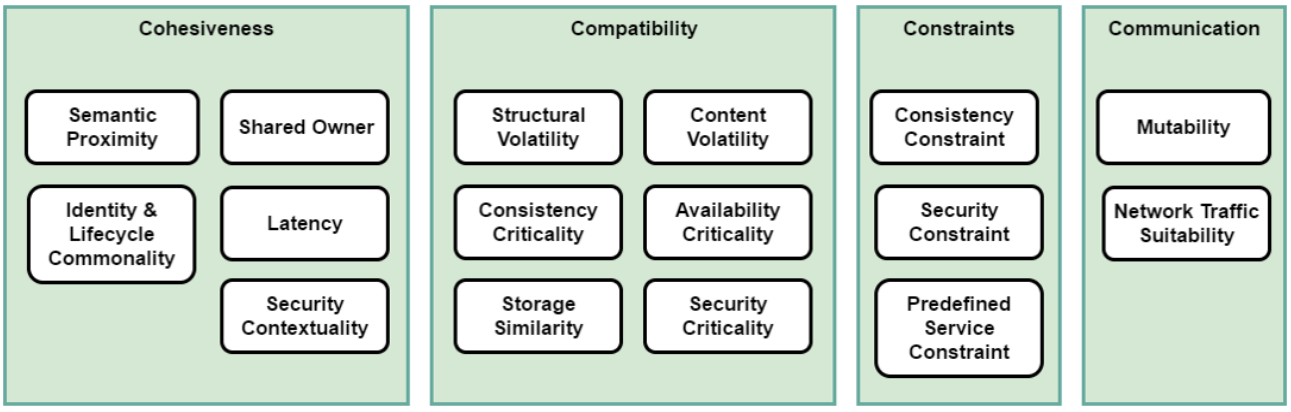


Figure 11: *Service Cutter: Couping Criteria Cards (CCC)*

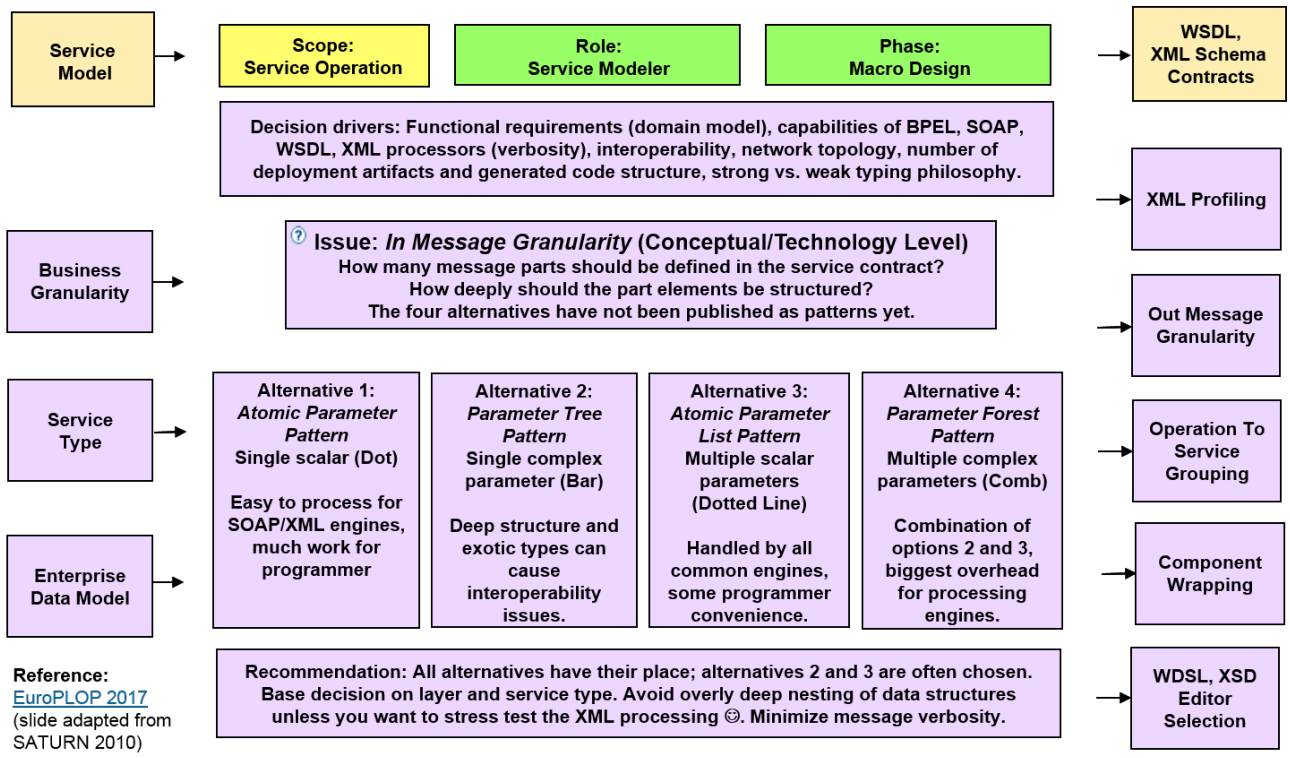


Figure 12: *Structural Patterns in Web API and Service Design*

Bus. IBM Redbooks.

Parnas, D. L. 1972. "On the Criteria to Be Used in Decomposing Systems into Modules." *Commun. ACM* 15 (12). New York, NY, USA: ACM:1053–8. https://doi.org/10.1145/361598.361623.

Rotem-Gal-Oz, Arnon. 2012. *SOA Patterns*. Manning.

Webber, Jim, Savas Parastatidis, and Ian Robinson. 2010. *REST in Practice: Hypermedia and Systems Architecture*. 1st ed. O'Reilly Media, Inc.

Zimmermann, Olaf. 2009. "An Architectural Decision Modeling Framework for Service-Oriented Architecture Design." PhD thesis, University of Stuttgart, Germany. http://elib.uni-stuttgart.de/opus/volltexte/2010/5228/.

———. 2017. "Microservices Tenets." *Comput. Sci.* 32 (3-4). Springer-Verlag New York, Inc.:301–10. https://doi.org/10.1007/s00450-016-0337-0.

Zimmermann, Olaf, Vadim Doubrovski, Jonas Grundler, and Kerard Hogg. 2005. "Service-Oriented Architecture and Business Process Choreography in an Order Management Scenario: Rationale, Concepts, Lessons Learned." ACM, 301–12.