Welcome to

9. Asynchronous and synchronous APIs, versioning, contracts

KEA System Integration

Henrik Kramselund Jereminsen hkj@zencurity.com @kramse

Slides are available as PDF, kramse@Github

9-async-sync-system-integration.tex in the repo security-courses

This weeks Agenda in system integration

- Follow the plan:
  `https://zencurity.gitbook.io/kea-it-sikkerhed/system-integration/lektionsplan`
- Work on the hand-in assignment I: Describe the system environment for an organisation
- Plan for April 27.
  I will go through the subjects from the book
- We will do a bigger exercise with APIs

Goals for today

Todays goals:

- SOA book chapters about Service API and Contract Design

Photo by Thomas Galler on Unsplash

Time schedule

- 08:30 2x 45 min with 10min break
  MessagePack, gRPC, protobuf, Apache Thrift etc.
  SOA chapter 8: Service API and Contract Design with Web Services
- 10:15 2x 45 min with 10min break
  SOA Chapter 9: Service API and Contract Design with REST Services and Microservices
  SOA Chapter 10: Service API and Contract Versioning with Web Services and REST Services
  Appendix B: REST Constraints Reference
- 12:30 2x 45min with 10min break

- 14:15 45 min
  Chatting, doing exercises, questions about Linux

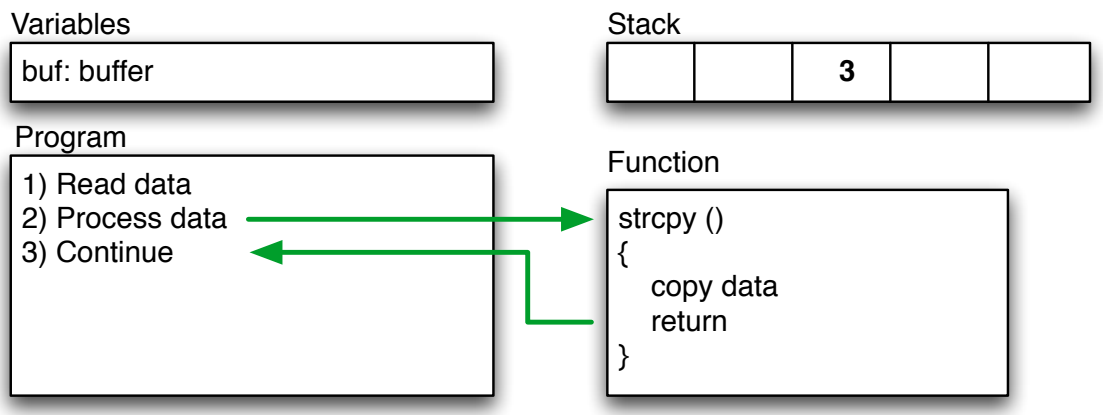
Plan for today

- MessagePack, gRPC, protobuf, Apache Thrift etc.
- Service API and Contract Design with Web Services, REST Services and Microservices
- Service API and Contract Versioning with Web Services and REST Services

Exercises

-

MessagePack, gRPC, protobuf, Apache Thrift etc.



- We know JSON and XML, but there are alternatives
- We will do a quick survey of some of the other popular methods for sending data back and forth
- But first, how do systems send data across architectures, platforms, operating systems

Serialization

In computing, **serialization (or serialisation)** is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer) or transmitted (for example, **across a network connection link) and reconstructed later (possibly in a different computer environment)**.[1] When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object. For many complex objects, such as those that make extensive use of references, this process is not straightforward. Serialization of object-oriented objects does not include any of their associated methods with which they were previously linked.

- Java can do this with `Serializable` - for example writing objects to file and reading them again

Source:

`https://en.wikipedia.org/wiki/Serialization`

Marshalling used for RPC

In computer science, **marshalling or marshaling** is the process of transforming the memory representation of an object to a data format suitable for storage or transmission, and it is typically used when data must be moved between different parts of a computer program or from one program to another. **Marshalling is similar to serialization** and is used to communicate to remote objects with an object, in this case a serialized object. It simplifies complex communication, using composite objects in order to communicate instead of primitives. The inverse of marshalling is called unmarshalling (or demarshalling, similar to deserialization).

- marshaling is about getting parameters from here to there
- while serialization is about copying structured data to or from a primitive form such as a byte stream
- serialization is one means to perform marshaling, usually implementing pass-by-value semantics.

Source:

`https://en.wikipedia.org/wiki/Marshalling_(computer_science)`

TCP/IP and External Data Representation 1987

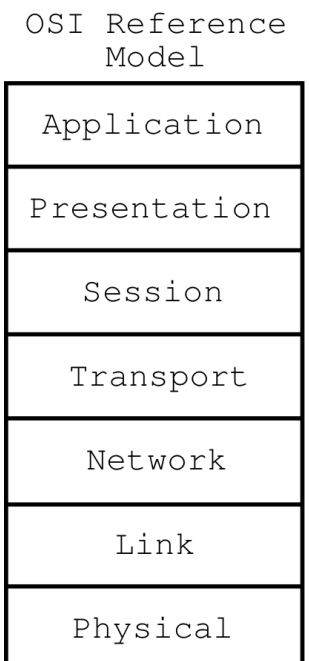
.

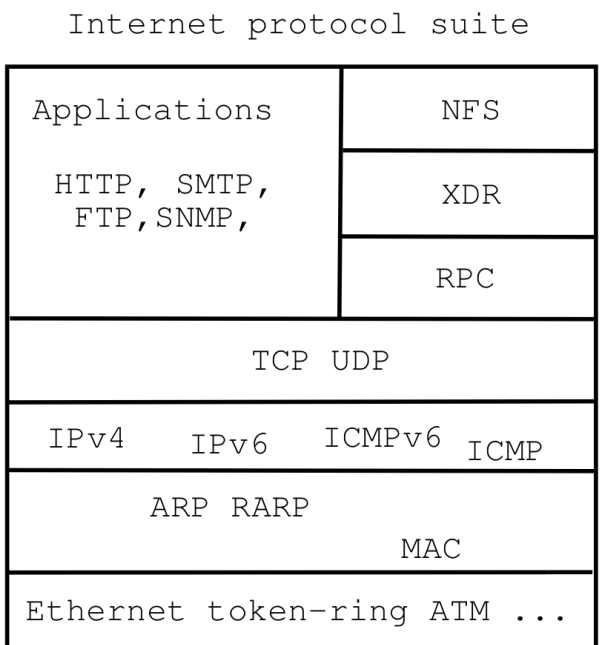
External Data Representation (XDR) is a standard data serialization format, for uses such as computer network protocols. It allows data to be transferred between different kinds of computer systems. Converting from the local representation to XDR is called encoding. Converting from XDR to the local representation is called decoding.

Source:

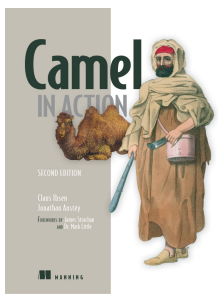
https://en.wikipedia.org/wiki/External_Data_Representation

https://tools.ietf.org/html/rfc1014





Chapter 3: Transforming data with Camel



We covered a whole chapter about transforming data:

- Transforming data by using EIPs and Java
- Transforming XML data
- Transforming by using well-known data formats
- Writing your own data formats for transformations
- Understanding the Camel type-converter mechanism

Camel type-converter system

How the Camel type-converter mechanism works

To understand the type-converter system, you first need to know what a type converter in Camel is. Figure 3.7 illustrates the relationship between `TypeConverterRegistry` and the `TypeConverters` it holds.

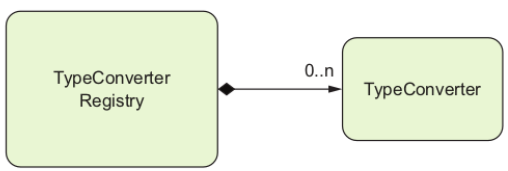


Figure 3.7 The `TypeConverterRegistry` contains many `TypeConverters`

- How the Camel type-converter mechanism works

```
from("file://riders/inbox"){\bf
  .convertBodyTo(String.class)}
  .to("jms:queue:inbox");
```

MessagePack

MessagePack is a computer data interchange format. It is a binary form for representing simple data structures like arrays and associative arrays. MessagePack aims to be as compact and simple as possible. The official implementation is available in a variety of languages such as C, C++, C#, D, Erlang, Go, Haskell, Java, JavaScript, Lua, OCaml, Perl, PHP, Python, Ruby, Scala, Smalltalk, and Swift.[1]

Data structures processed by MessagePack loosely correspond to those used in JSON format.

- MessagePack is more compact than JSON, but imposes limitations on array and integer sizes
- Allows binary data and non UTF-8 encoded strings
- MessagePack is designed for efficient transmission over the wire

Source:

`https://en.wikipedia.org/wiki/MessagePack`

gRPC

gRPC (gRPC Remote Procedure Calls[1]) is an open source remote procedure call (RPC) system initially developed at Google in 2015[2]. It uses HTTP/2 for transport, Protocol Buffers as the interface description language, and provides features such as authentication, bidirectional streaming and flow control, blocking or nonblocking bindings, and cancellation and timeouts. It generates cross-platform client and server bindings for many languages. Most common usage scenarios include connecting services in microservices style architecture and connect mobile devices, browser clients to backend services.[3]

- We will not begin running the code but tutorials are available at `https://grpc.io/`

Source:

`https://en.wikipedia.org/wiki/GRPC`

Protobuf used in gRPC

> Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages.

- Supports generated code in multiple languages: *Java, Python, Objective-C, and C++. With our new proto3 language version, you can also work with Dart, Go, Ruby, and C#, with more languages to come.*

Source:

`https://developers.google.com/protocol-buffers`

Protobuf: example definition

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

Source:

https://developers.google.com/protocol-buffers/docs/overview

Protobuf: example code C++

```
Person person;
person.set_name("John Doe");
person.set_id(1234);
person.set_email("jdoe@example.com");
fstream output("myfile", ios::out | ios::binary);
person.SerializeToOstream(&output);
```

Then, later on, you could read your message back in:

```
fstream input("myfile", ios::in | ios::binary);
Person person;
person.ParseFromIstream(&input);
cout << "Name: " << person.name() << endl;
cout << "E-mail: " << person.email() << endl;
```

Source:

https://developers.google.com/protocol-buffers/docs/overview

Why protobuf?

Protocol buffers have many advantages over XML for serializing structured data. Protocol buffers:

- are simpler
- are 3 to 10 times smaller
- are 20 to 100 times faster
- are less ambiguous
- generate data access classes that are easier to use programmatically

When this message is encoded to the protocol buffer binary format (the text format above is just a convenient human-readable representation for debugging and editing), it would probably be 28 bytes long and take around 100-200 nanoseconds to parse. The XML version is at least 69 bytes if you remove whitespace, and would take around 5,000-10,000 nanoseconds to parse.

Source:

`https://developers.google.com/protocol-buffers/docs/overview`

Apache Thrift

> The Apache Thrift software framework, for scalable cross-language services development, combines a software stack with a code generation engine to build services that work efficiently and seamlessly between C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml and Delphi and other languages. ... Apache Thrift allows you to define data types and service interfaces in a simple definition file. Taking that file as input, the compiler generates code to be used to easily build RPC clients and servers that communicate seamlessly across programming languages.

- Using the input file defining the service, Thrift format
- The compiling this generates code to be used for building clients and servers that can communicate
- Next slides show parts of the tutorial `https://thrift.apache.org/tutorial/`

Source:

`https://thrift.apache.org/`

Apache Thrift example input

```
/**
 * Ahh, now onto the cool part, defining a service. Services just need a name
 * and can optionally inherit from another service using the extends keyword.
 */
service Calculator extends shared.SharedService {
  /**
   * A method definition looks like C code. It has a return type, arguments,
   * and optionally a list of exceptions that it may throw. Note that argument
   * lists and exception lists are specified using the exact same syntax as
   * field lists in struct or exception definitions.
   */

   void ping(),

   i32 add(1:i32 num1, 2:i32 num2),
   i32 calculate(1:i32 logid, 2:Work w) throws (1:InvalidOperation ouch),

   /**
    * This method has a oneway modifier. That means the client only makes
    * a request and does not listen for any response at all. Oneway methods
    * must be void.
```

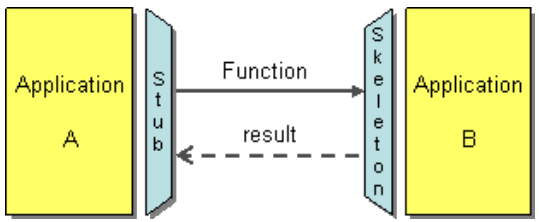
Apache Thrift example

```
try {
  TServerTransport serverTransport = new TServerSocket(9090);
  TServer server = new TSimpleServer(new Args(serverTransport).processor(processor));
  System.out.println("Starting the simple server...");
  server.serve();
} catch (Exception e) {
  e.printStackTrace();
}
public class CalculatorHandler implements Calculator.Iface {
  private HashMap<Integer,SharedStruct> log;

  public CalculatorHandler() {
    log = new HashMap<Integer, SharedStruct>();  }

  public void ping() {
    System.out.println("ping()");  }

  public int add(int n1, int n2) {
    System.out.println("add(" + n1 + "," + n2 + ")");
    return n1 + n2;  }
  ...
```

Apache Thrift example

```
def main():
    # Make socket
    transport = TSocket.TSocket('localhost', 9090)

    # Buffering is critical. Raw sockets are very slow
    transport = TTransport.TBufferedTransport(transport)

    # Wrap in a protocol
    protocol = TBinaryProtocol.TBinaryProtocol(transport)

    # Create a client to use the protocol encoder
    client = Calculator.Client(protocol)

    # Connect!
    transport.open()

    client.ping()
    print('ping()')

    sum_ = client.add(1, 1)
```

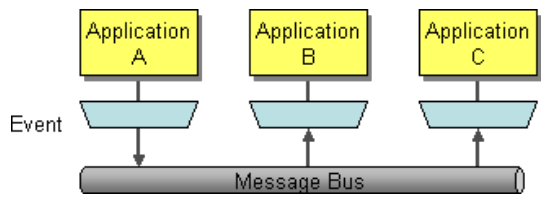
Remote Procedure Invocation



Remote Procedure Invocation — Have each application expose some of its procedures so that they can be invoked remotely, and have applications invoke those to run behavior and exchange data. Common systems and technologies used:

- Java remote method invocation (RMI), Unix RPC
- XMLHttpRequest (XHR) JavaScript in the browser makes connections and requests: `https://en.wikipedia.org/wiki/XMLHttpRequest`
- Common Object Request Broker Architecture (CORBA) used in the 1990s but not very relevant anymore
- See more at `https://en.wikipedia.org/wiki/Remote_procedure_call`

Messaging

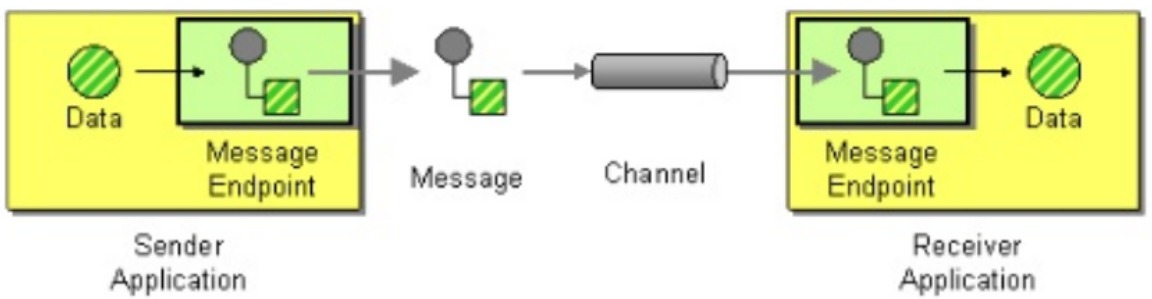


Messaging — Have each application connect to a common messaging system, and exchange data and invoke behavior using messages.

Common systems and technologies used:

- Java Message Service (JMS) API is a Java message-oriented middleware Application Programming Interface (API)
- Apache ActiveMQ, RabbitMQ, Oracle WebLogic
- See more at `https://en.wikipedia.org/wiki/Message_passing`

Endpoints



- Endpoints are built-in or added to applications
- Interface between internal representation and code, to the external message format

Lets try running the Thrift Python and Java

- From the tutorial `https://thrift.apache.org/tutorial/`
- Run the Python client with the Python server
- Run the Java client with the Java server
- Then try mixing, Python client to Java server etc.
- Note: the code is already generated, so you can get away by just running the existing programs, not doing the generation

Reading Summary

SOA chapter 8: Service API and Contract Design with Web Services

Chapter 9: Service API and Contract Design with REST Services and Microservices

Chapter 10: Service API and Contract Versioning with Web Services and REST Services and Appendix B: REST Constraints Reference

Service-Oriented Architecture: Analysis and Design for Services and Microservices, Thomas Erl, 2017 ISBN: 978-0-13-385858-7

SOA chapter 8: Service API and Contract Design with Web Services

a contract-first approach with Web services:

- Web service contracts can be designed to accurately represent the context and function of their corresponding service candidates.
- Conventions can be applied to Web service operation names to produce standardized endpoint definitions.
- The granularity of operations can be modeled in abstract to provide consistent and predictable API designs that also establish a message size and volume ratio suitable for the target communications infrastructure.
- Service consumers are required to conform to the expression of the service contract, not vice versa.
- The design of business-centric Web service contracts can be assisted by business analysts who may be able to help establish an accurate expression of business logic.

Source:

Service-Oriented Architecture: Analysis and Design for Services and Microservices, Thomas Erl, 2017

SOA chapter 8: Service API and Contract Design with Web Services

References multipls patterns:

- Dual Protocols [339] pattern The service inventory architecture is designed to support services based on primary and secondary protocols
- Concurrent Contracts [332] pattern, a single body of service logic can expose two alternative service contracts that allow it to be invoked via two different communication protocols
- Service Façade [360] pattern A separate façade component is incorporated into the service design.
- Decoupled Contract [337]. The service contract is physically decoupled from its implementation

Source:

Service-Oriented Architecture: Analysis and Design for Services and Microservices, Thomas Erl, 2017

lightweight XML schema Example 8.1

```
<xml:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.example.org/tls/employee/schema/accounting/">
  <xml:element name="EmployeeHoursRequestType">
    <xml:complexType>
      <xml:sequence>
        <xml:element name="ID" type="xml:integer"/>
      </xml:sequence>
    </xml:complexType>
  </xml:element>
  <xml:element name="EmployeeHoursResponseType">
    <xml:complexType>
      <xml:sequence>
        <xml:element name="ID" type="xml:integer"/>
        <xml:element name="WeeklyHoursLimit" type="xml:short"/>
      </xml:sequence>
    </xml:complexType>
  </xml:element>
</xml:schema>
```

- Sometimes we will use large complex schemas, other times redefine a small narrow and lightweight scheme

Namespaces and origins

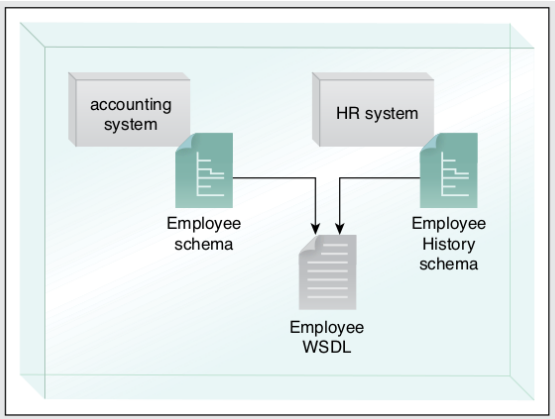


Figure 8.7
Two schemas originating from two different data sources.

- The schema definitions include targetNamespace which can show origins, who owns this schema!

Source:

Service-Oriented Architecture: Analysis and Design for Services and Microservices, Thomas Erl, 2017

Initial service contract

Next, TLS architects follow these steps to define an initial service contract:

1. They confirm that each capability candidate is suitably generic and reusable by ensuring that the granularity of the logic encapsulated is appropriate. They then study the data structures defined earlier and establish a set of operation names.
2. They create the `portType` (or `interface`) area within the WSDL document and populate it with `operation` constructs that correspond to capability candidates.
3. They formalize the list of input and output values required to accommodate the processing of each operation's logic. This is accomplished by defining the appropriate `message` constructs that reference the XML Schema types within the child `part` elements.

Figure 8.8
The Employee service operations.

The TLS architects decide on operation names *GetEmployeeWeeklyHoursLimit* and *UpdateEmployeeHistory* (Figure 8.8).

- Resulting from the modelling is an initial service contract describing operation names
  Later standardized as just `GetWeeklyHoursLimit` and `UpdateHistory`

Source: *Service-Oriented Architecture: Analysis and Design for Services and Microservices*, Thomas Erl, 2017

Apply Naming Standards

- Service candidates with high reuse potential should always be stripped of any naming characteristics that hint at the business processes for which they were originally built. For example, instead of naming an operation GetTimesheetSubmissionID, it can be simply reduced to GetTimesheetID or even just GetID.
- Entity services need to remain representative of the entity models from which their corresponding service candidates were derived. Therefore, the naming conventions used must reflect those established in the organization's original entity models. Typically, this type of service uses the noun-only naming structure. Examples of suitable entity service names are Invoice, Customer, and Employee.
- Service operations for entity services should be verb-based and should not repeat the entity name. For example, an entity service called Invoice should not have an operation named AddInvoice.
- Utility services need to be named according to the processing context under which their operations are grouped. Both the verb+noun or noun only conventions can be used. Simplifi ed examples of suitable utility service names are CustomerDataAccess, SalesReporting, and GetStatistics.
- Utility service operations need to clearly communicate the nature of their individual functionality. Examples of suitable utility service operation names are GetReport, ConvertCurrency, and VerifyData.
- While microservices are not always subjected to the same design standards as agnostic services, it is still recommended that the conventions for service and operation names be applied consistently to whatever extent possible.

  Whatever naming standards are chosen, the key is that they must be consistently applied throughout all services within a given service inventory.

Source: *Service-Oriented Architecture: Analysis and Design for Services and Microservices*, Thomas Erl, 2017

Apply a Suitable Level of Contract API Granularity

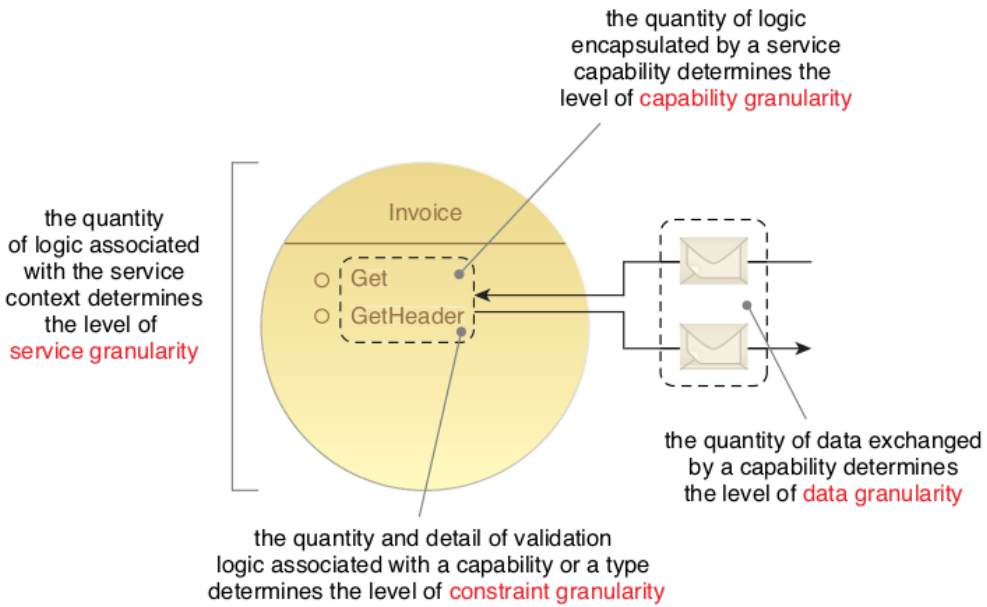


Figure 8.11

The four granularity levels that represent various characteristics of a service and its contract. Note that these granularity types are, for the most part, independent of each other.

Source: *Service-Oriented Architecture: Analysis and Design for Services and Microservices*, Thomas Erl, 2017

Be Inherently Extensible

Design Web Service Operations to Be Inherently Extensible

Regardless of how well services are designed when first deployed, they can never be fully prepared for what the future holds. Some types of business process changes result in the need for the scope of entities to be broadened. As a result, corresponding business services may need to be extended. While the application of Service Reusability (295) and Service Composability (302) are thought through when partitioning logic as part of the service modeling process, extensibility is more of a physical design quality that needs to be considered during design.

- Creating contracts and designing services is not easy

Source: *Service-Oriented Architecture: Analysis and Design for Services and Microservices*, Thomas Erl, 2017

Chapter 9: Service API and Contract Design with REST Services and Microservices

> REST service contracts are typically designed around the primary functions of HTTP methods, which make the documentation and expression of REST service contracts distinctly different from operation-based Web service contracts. Regardless of the differences in notation, the same overarching contract-first approach to designing REST service contracts is paramount when building services for a standardized service inventory.

- REST entity service contracts are typically dominated by service capabilities that include inherently idempotent and reliable GET, PUT, or DELETE methods
- This chapter provides service contract design guidance for service candidates modeled as a result of the service-oriented analysis stage covered in Chapter 7.

Source:

Service-Oriented Architecture: Analysis and Design for Services and Microservices, Thomas Erl, 2017

REST Service

Figure 9.1
An entity service based on the Invoice business entity that defines a functional scope that limits the service capabilities to performing invoice-related processing. This agnostic Invoice service will be reused and composed by other services within the same service inventory in support of different automated business processes that need to process invoice-related data. This particular invoice service contract displays two service capabilities based on primitive methods and two service capabilities based on complex methods.

- Very typical REST URL/method `GET /invoice/{invoice-id}`

Source:

Service-Oriented Architecture: Analysis and Design for Services and Microservices, Thomas Erl, 2017

The following is a series of common guidelines and considerations for designing REST service contracts.

- Uniform Contract Design Considerations
- Designing and Standardizing Methods
- Designing and Standardizing HTTP Headers
- Designing and Standardizing HTTP Response Codes
- Customizing Response Codes
- Designing Media Types

Source:

Service-Oriented Architecture: Analysis and Design for Services and Microservices, Thomas Erl, 2017

Designing and Standardizing HTTP Response Codes

- 100-199 are informational codes used as low-level signaling mechanisms, such as a confirmation of a request to change protocols
- 200-299 are general success codes used to describe various kinds of success conditions
- 300-399 are redirection codes used to request that the consumer retry a request to a different resource identifier, or via a different intermediary
- 400-499 represent consumer-side error codes that indicate that the consumer has produced a request that is invalid for some reason, example 404 file not found
- 500-599 represent service-side error codes that indicate that the consumer's request may have been valid but that the service has been unable to process it for internal reasElasticsearch exposes REST APIs that are used by the UI components and can be called directly to configure and access Elasticsearch features.ons.

Source:

Service-Oriented Architecture: Analysis and Design for Services and Microservices, Thomas Erl, 2017

Idempotence

Idempotence (UK: / d m po təns/,[1] US: / a dəm-/)[2] is the property of certain operations in mathematics and computer science whereby they can be applied multiple times without changing the result beyond the initial application. The concept of idempotence arises in a number of places in abstract algebra (in particular, in the theory of projectors and closure operators) and functional programming (in which it is connected to the property of referential transparency).

The term was introduced by Benjamin Peirce[3] in the context of elements of algebras that remain invariant when raised to a positive integer power, and literally means "(the quality of having) the same power", from idem + potence (same + power).

- The term idempotent is used multiple times regarding REST and HTTP methods.

Source:

`https://en.wikipedia.org/wiki/Idempotence`

Computer science examples

A function looking up a customer's name and address in a database is typically idempotent, since this will not cause the database to change. Similarly, changing a customer's address to XYZ is typically idempotent, because the final address will be the same no matter how many times XYZ is submitted. ...

In the Hypertext Transfer Protocol (HTTP), idempotence and safety are the major attributes that separate HTTP verbs. Of the major HTTP verbs, GET, PUT, and DELETE should be implemented in an idempotent manner according to the standard, but POST need not be.[15] GET retrieves a resource; PUT stores content at a resource; and DELETE eliminates a resource. ...

In service-oriented architecture (SOA), a multiple-step orchestration process composed entirely of idempotent steps can be replayed without side-effects if any part of that process fails.

Source:

`https://en.wikipedia.org/wiki/Idempotence`

Designing Schemas for Media Types

> Within a service inventory, most custom media types created to represent business data and documents will be defi ned using XML Schema or JSON Schema. This can essentially establish a set of standardized data models that are reused by REST services within the inventory to whatever extent feasible.

- Data used inside methods and operations should be standardized

Source:

Service-Oriented Architecture: Analysis and Design for Services and Microservices, Thomas Erl, 2017

Example 9.2 provides an example of a flexible schema design

```
Media type = application/vnd.com.actioncon.po+xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://example.org/schema/po"
  xmlns="http://example.org/schema/po">
  <xsd:element name="LineItemList" type="LineItemListType"/>
  <xsd:complexType name="LineItemListType">
    <xsd:element name="LineItem" type="LineItemType" minOccurs="0"/>
  </xsd:complexType>
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:anyURI"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="available" type="xsd:boolean" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Source:

Service-Oriented Architecture: Analysis and Design for Services and Microservices, Thomas Erl, 2017

We already mentioned this from the first book

> Data format — Integrated applications must agree on the format of the data they exchange, or must have an intermediate traslator to unify applications that insist on different data formats. A related issue is data format evolution and extensibility—how the format can change over time and how that will affect the applications.

Enterprise Integration Patterns, Gregor Hohpe and Bobby Woolf, 2004

Data or functionality

> Data or functionality — Integrated applications may not want to simply share data, they may wish to share functionality such that each application can invoke the functionality in the others. Invoking functionality remotely can be difficult to achieve, and even though it may seem the same as invoking local functionality, it works quite differently, with significant consequences for how well the integration works.

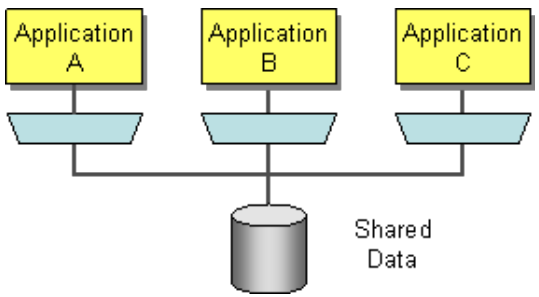
Enterprise Integration Patterns, Gregor Hohpe and Bobby Woolf, 2004

Remote Communication / Asynchronicity

> Remote Communication / Asynchronicity — Computer processing is typically synchronous, such that a procedure waits while its subprocedure executes. It's a given that the subprocedure is available when the procedure wants to invoke it. However, a procedure may not want to wait for the subprocedure to execute; it may want to invoke the subprocedure asynchronously, starting the subprocedure but then letting it execute in the background. This is especially true of integrated applications, where the remote application may not be running or the network may be unavailable—the source application may wish to simply make shared data available or log a request for a subprocedure call, but then go on to other work confident that the remote application will act sometime later.

Enterprise Integration Patterns, Gregor Hohpe and Bobby Woolf, 2004

Shared Database



Shared Database — Have the applications store the data they wish to share in a common database.

Common systems and technologies used:

- database management system (DBMS) using Structured Query Language (SQL), relational database examples:

- PostgresSQL, Oracle DM, Microsoft SQL, MySQL `https://en.wikipedia.org/wiki/SQL`
- NoSQL databases has been a new input with examples like: MongoDB, CouchDB, Redis, RIAK `https://en.wikipedia.org/wiki/NoSQL`

Chapter 10: Service API and Contract Versioning with Web Services and REST Services

After a service contract is deployed, consumer programs will naturally begin forming dependencies on it. When we are subsequently forced to make changes to the contract, we need to figure out:

- Whether the changes will negatively impact existing (and potentially future) service consumers
- How changes that will and will not impact consumers should be implemented and communicated

Included in the chapter are references to:

- Versioning Web Services
- Versioning REST Services
- Fine and Coarse-Grained Constraints

Source:

Service-Oriented Architecture: Analysis and Design for Services and Microservices, Thomas Erl, 2017

Versioning and Compatibility

- Backwards Compatibility
  A backwards-compatible change to a REST-compliant service contract might involve adding some new resources or adding new capabilities to existing resources. In each of these cases the existing service consumers will only invoke the old methods on the old resources, which continue to work as they previously did.
- Forwards Compatibility
  When a service contract is designed in such a manner so that it can support a range of future consumer programs, it is considered to have an extent of forwards compatibility. This means that the contract can essentially accommodate how consumer programs will evolve over time.

Source:

Service-Oriented Architecture: Analysis and Design for Services and Microservices, Thomas Erl, 2017

Incompatible Changes

- Compatible Changes vs Incompatible Changes
  If after a change a contract is no longer compatible with consumers, then it is considered to have received an incompatible change. These are the types of changes that can break an existing contract and therefore impose the most challenges when it comes to versioning.

  Source:

  Service-Oriented Architecture: Analysis and Design for Services and Microservices, Thomas Erl, 2017

10.4 Version Identifiers

> Versions are almost always communicated with version numbers. The most common format is a decimal, followed by a period and then another decimal, as shown here: version="2.0"

From a compatibility perspective, we can associate additional meaning to these numbers. Specifically, the following convention has emerged in the industry:

- A minor version is expected to be backwards-compatible with other minor versions associated with a major version. For example, version 5.2 of a program should be fully backwards-compatible with versions 5.0 and 5.1.
- A major version is generally expected to break backwards compatibility with programs that belong to other major versions. This means that program version 5.0 is not expected to be backwards-compatible with version 4.0.

Source:

Service-Oriented Architecture: Analysis and Design for Services and Microservices, Thomas Erl, 2017

Appendix B: REST Constraints Reference

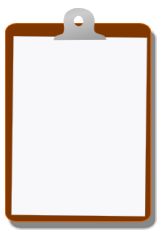
This appendix provides profile tables for the REST constraints referenced throughout this book

- Client-Server *Solution logic is separated into consumer and service logic that share a technical contract.*
- Stateless *Services remain stateless between request/response message exchanges with service consumers.*
- Cache *Service consumers can cache and reuse response message data.*
- Uniform Contract *Service consumers and services share a common, overarching, generic technical contract.*
- Layered System *A solution can be comprised of multiple architectural layers.*
- Code-on-Demand *Service consumers support the execution of deferred service logic.*

Source:

Service-Oriented Architecture: Analysis and Design for Services and Microservices, Thomas Erl, 2017

For Next Time



Think about the subjects from this time, write down questions

Check the plan for chapters to read in the books

Visit web sites and download papers if needed

Retry the exercises to get more confident using the tools