KU LEUVEN

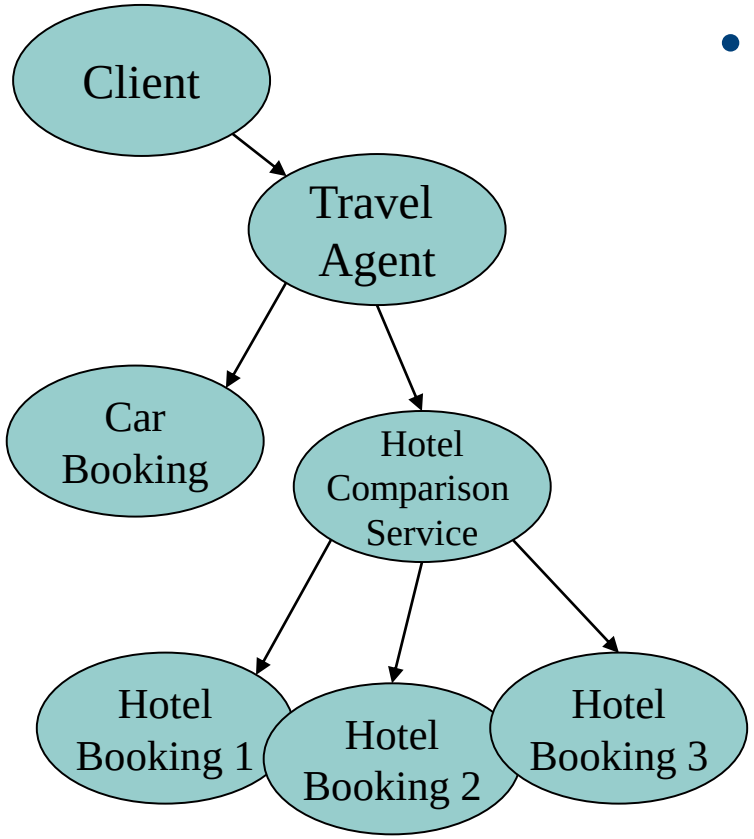
Distributed Systems: service-oriented architectures

Web-services, SOAP & REST.

Recap:
Distributed objects and remote method invocation

- Object-based software
  - Original idea: model software according to real life entities
  - Common definition: objects have state and behavior (methods)
  - Interface: exposed behavior
  - Reference semantics / value semantics
- Distributing the paradigm
  - Distributed objects
  - Remote method invocations
  - Remote interfaces
  - Remote references / serialization of objects

Interoperability ?



- Example: travel agency
  - Travel agent runs Java, the flight booking systems .NET and the car booking Python.
  - Services possibly distributed across the world.

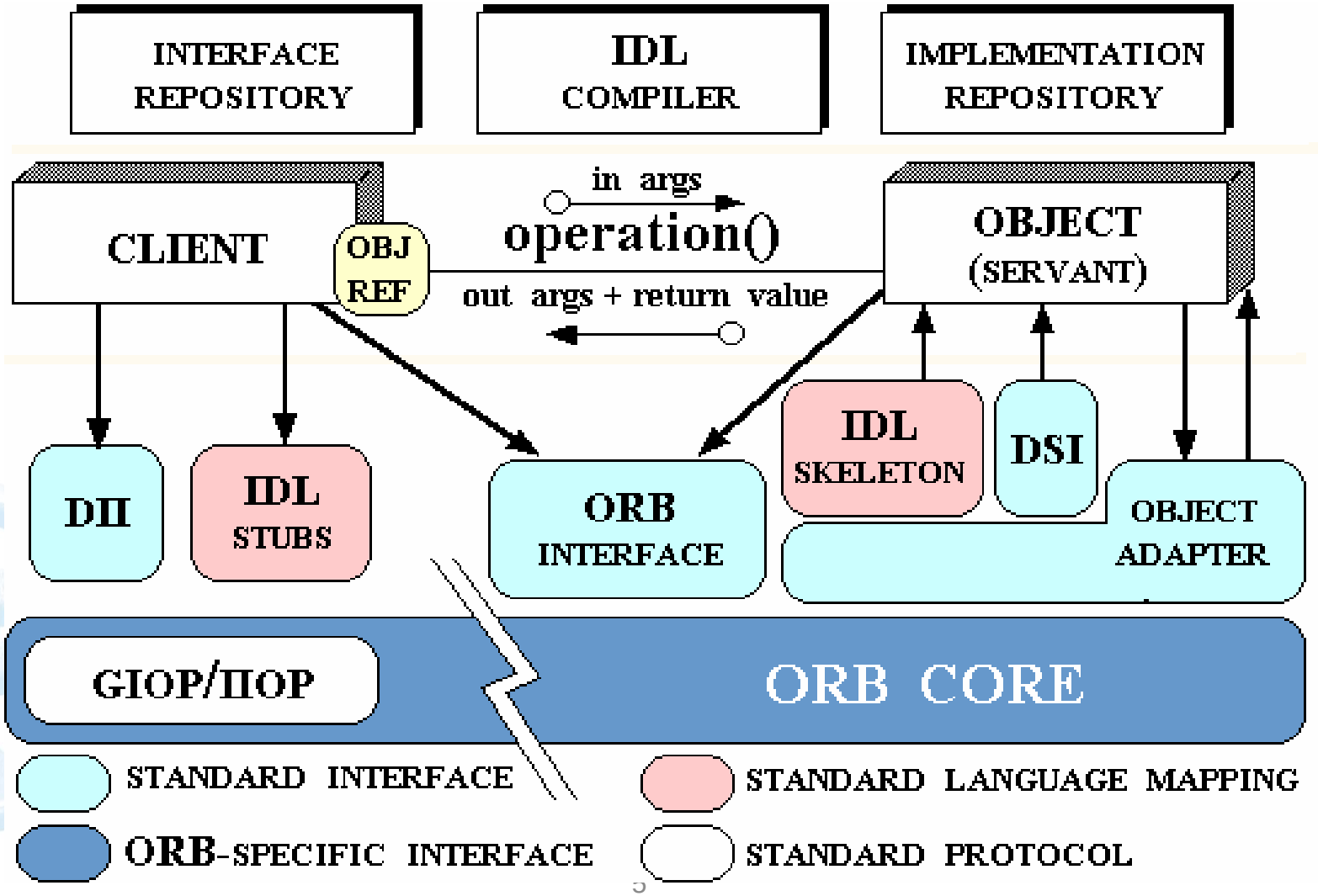
Corba: common object request broker architecture
CORBA IDL for a bank service

IDL Example

```
module Bank {
  struct StructType{
        long initbalance;
        string acctname;
      };
  interface Account {
        float balance();
        string getCalendar();
    };
  interface AccountManager {
        Account open(in StructType  st);
    };
};
```

Distributed Object Programming course 2003

Corba: common object model and architecture



KU LEUVEN

IDL

Interface definition language for remote objects

Java-rmi IDL, Corba IDL, DCOM IDL

```
public class StackFullException
   extends Exception
{
}

public class StackEmptyException
   extends Exception
{
}

public interface StackInterface
   extends java.rmi.Remote
{
   boolean isEmpty()
      throws java.rmi.RemoteException;

   boolean isFull()
      throws java.rmi.RemoteException;

   void push(Object obj)
      throws java.rmi.RemoteException,
      StackFullException;

   Object pop()
      throws java.rmi.RemoteException,
      StackEmptyException;
}
```

(a)

```
exception StackFullException
{
};

exception StackEmptyException
{
};

interface StackInterface
{
   boolean isEmpty();

   boolean isFull();

   void push(in short i)
      raises (StackFullException);

   short pop()
      raises (StackEmptyException);
};
```

(b)

```
import "unknwn.idl";

[ uuid(c38cea50-498a-11d2-8043-00104b235515) ]
interface StackInterface :IUnknown
{
   HRESULT isEmpty([out] boolean *empty);

   HRESULT isFull([out] boolean *full);

   HRESULT push([in] int i);

   HRESULT pop([out] int *i);
};

[ uuid(ddd99e70-498a-11d2-8043-00104b235515) ]
library StackLib
{
   importlib("stdole32.tlb");

   [ uuid(e9868930-498a-11d2-8043-00104b235515) ]
   coclass Stack
   {
      interface StackInterface;
   };
};
```

(c)

KU LEUVEN

Marshalling / serialization

Figure 4.9
Indication of Java serialized form

Serialized values				*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name:	java.lang.String place:	*number, type and name of instance variables*
1984	5 Smith	6 London	h1	*values of instance variables*

The true serialized form contains additional type markers; h0 and h1 are handles

Serialization of a struct in CORBA

index in sequence of bytes	← 4 bytes →	*notes on representation*
0–3	`5`	*length of string*
4–7	`"Smit"`	*'Smith'*
8–11	`"h___"`	
12–15	`6`	*length of string*
16–19	`"Lond"`	*'London'*
20-23	`"on__"`	
24–27	`1984`	*unsigned long*

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1984}

Byte-based serialization is not easy to consume for web-based heterogenous clients (e.g. javascript)

RMI

- Serialized object is self-describing
- Not easy to parse without java-based client

CORBA

- Just serialized fields
- No self-describing structure
- Not easy to parse without
  - interface definition
  - corba client proxies

Service-oriented architecture

From objects to services

KU LEUVEN

Service-oriented architectures

- Applications are constructed from reusable online services
- Services offer interoperable machine-to-machine interaction over a network
- Interoperability is key: not dependent on a programming model or language
  - CORBA ?
- Information is exchanged in machine-processable textformat (XML, JSON)
- Transport layer is often HTTP-based
- Services are stateless and rely on remote process communication (RPC)

- Two common approaches:
  - Web services (SOAP)
  - RESTful services

What is a web service?

- **Main goal: Interoperability**

  Independent of used programming language, middleware platform, operating system, …
- **Context: Machine-to-machine interactions**

  Web services are not meant for human users, but are typically used in B2B scenarios.
- **Interface: WSDL**

  A web service is described by its interface, written in WSDL (‘the IDL for web services’)
- **Transport protocol: SOAP over HTTP**

  Web service messages are (usually) transported using the SOAP standard over an HTTP connection (but others are possible)
- **Data format: XML**

  Information is serialized to XML, to ensure interoperability.

XML: general

XML is built from human-readable tags and data.

```
<person id="123456789">
  <name>Smith</name>
  <place>London</place>
  <year>1934</year>
  <!-- a comment -->
</person >
```

An *element*: `<tag>data</tag>`

Note: `<tag />` is short for `<tag></tag>`

An *attribute*: `name="value"`

XML: namespaces (Cfr java packages)

```
<pers:person xmlns:pers = "http://www.cdk4.net/">
   …
</pers:person >
```

- Example above: `pers:person` refers to the person in the "http://www.cdk4.net/" XML namespace, which is identified using the prefix `pers`.
- Notes:
  - URL's in namespaces do not have to exist (they are just unique strings)
  - Prefix `pers` can be chosen freely by the creator of the XML document, as long as there are no collisions within the document (so `p1` would be equally suited)

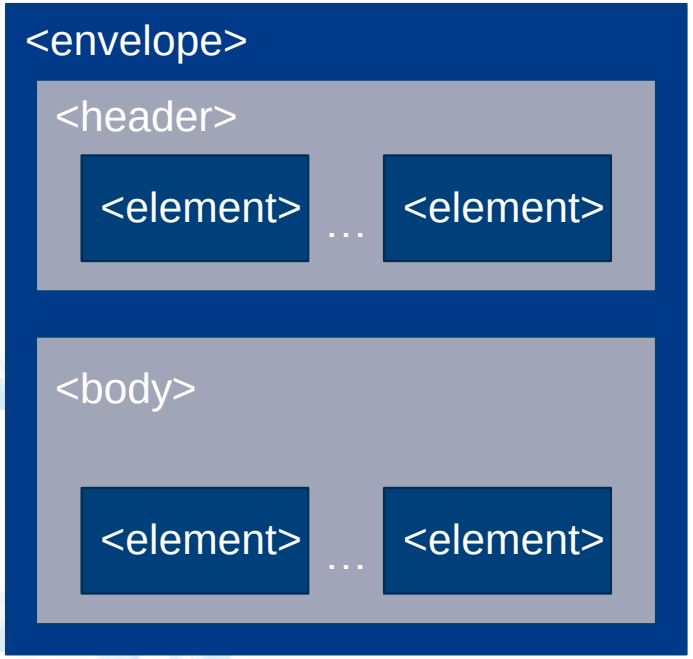
XML: schema

```
<person id="123456789">
  <name>Smith</name>
  <place>London</place>
  <year>1934</year>
</person>
```

```
<!- Each XML document conforming to
    this schema … -->
<xsd:schema xmlns:xsd=URL of XML schema definitions >
  <!- contains one 'person' element,
      of the type personType (described next) -->
  <xsd:element name="person" type="personType" />
  <!- the personType type defines the following: -->
  <xsd:complexType name="personType">
    <!-- a sequence of three elements… -->
    <xsd:sequence>
      <!- first a name, which is a string -->
      <xsd:element name="name" type="xsd:string" />
      <!- then a place, which is also a string -->
      <xsd:element name="place" type="xsd:string" />
      <!- finally the year, which is a positive integer -->
      <xsd:element name="year" type="xsd:positiveInteger" />
    </xsd:sequence>
    <!- a person also has an attribute (id) -->
    <xsd:attribute name="id" type="xsd:positiveInteger" />
  </xsd:complexType>
</xsd:schema>
```

KU LEUVEN

SOAP: Message structure (1): General



- Contained in a SOAP envelope
- Defined in SOAP schema and namespace
- Headers
  - Establish context
  - Can be inspected, removed or added by intermediaries
- Body
  - can carry documents in XML format, together with schema, or
  - can contain either a request or a reply (see next slides)

Message structure (2): Request

- Request is enclosed in envelope
  - Contains a reference (“s”) to the SOAP namespace in which envelope is defined
- Body contains element with
  - name of the operation to call (*add*)
  - URI of namespace for service description (referenced by prefix “m”)
- Inner elements contain arguments of operation (here, two integers)

```
<s:envelope xmlns:s=… >

<s:header>…</s:header>

<s:body>

  <m:add xmlns:m=…>
     <m:arg1>5</m:arg1>
     <m:arg2>2</m:arg2>
  </m:add>

</s:body>

</s:envelope>
```

Message structure (3): Reply

- Reply is also enclosed in envelope
- Body contains reply content:
  addResponse, containing one integer, named *result*
- The names *addResponse* and *result* are again in a namespace defined by the developer

```
<s:envelope xmlns:s=… >

<s:body>

 <m:addResponse xmlns:m=…>
   <m:result>7</m:result>
 </m:addResponse>

</s:body>

</s:envelope>
```

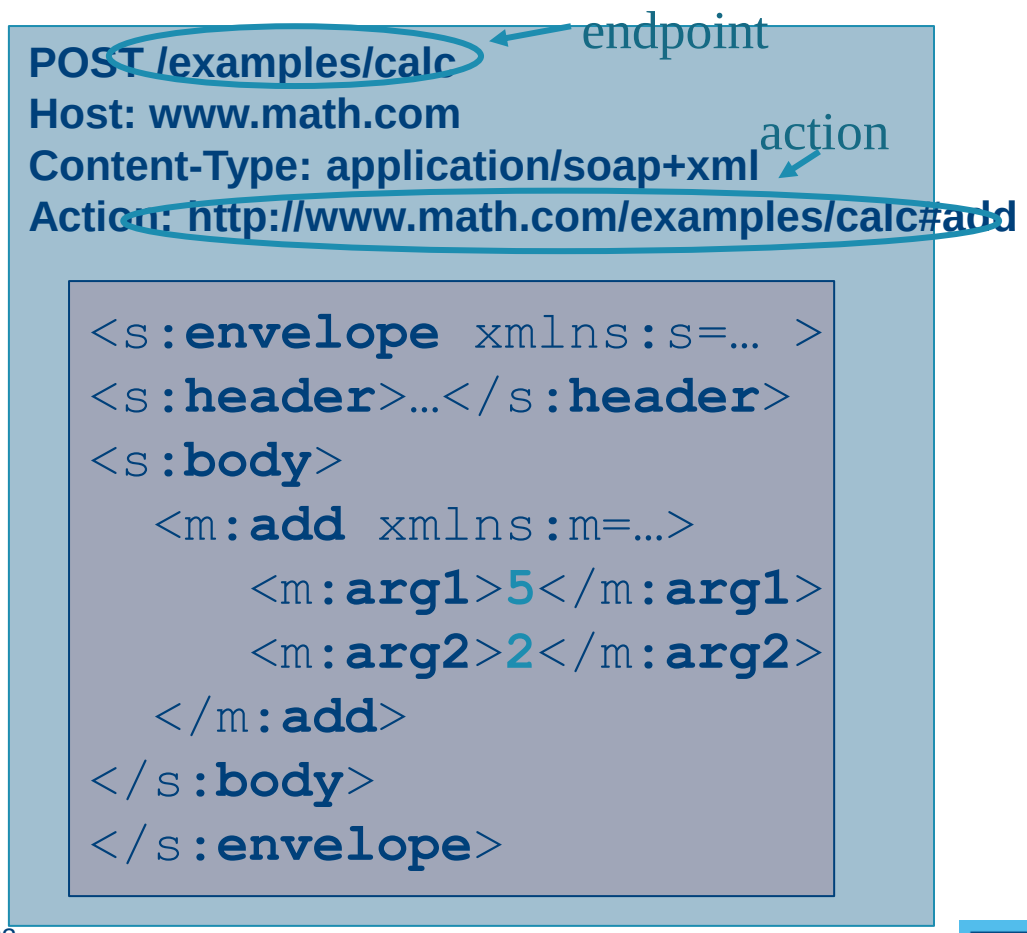
Message structure (4): Fault

- Instead of a response, a fault can be returned in the SOAP body.
- Contains a fault *code*
- Contains a *reason*
- Can also contain a *node*, *role* and/or *detail* element

```
<s:envelope xmlns:s=… >

<s:fault>

 <s:code>…</s:code>
 <s:reason>…</s:reason>

</s:fault>

</s:envelope>
```

Message transfer: HTTP

- SOAP message is then transferred on top of a transport protocol e.g. HTTP
- Here: the HTTP POST header includes
  - the address of the service (the “endpoint”)
  - the action to invoke at the endpoint (allows dispatching without parsing the SOAP message)

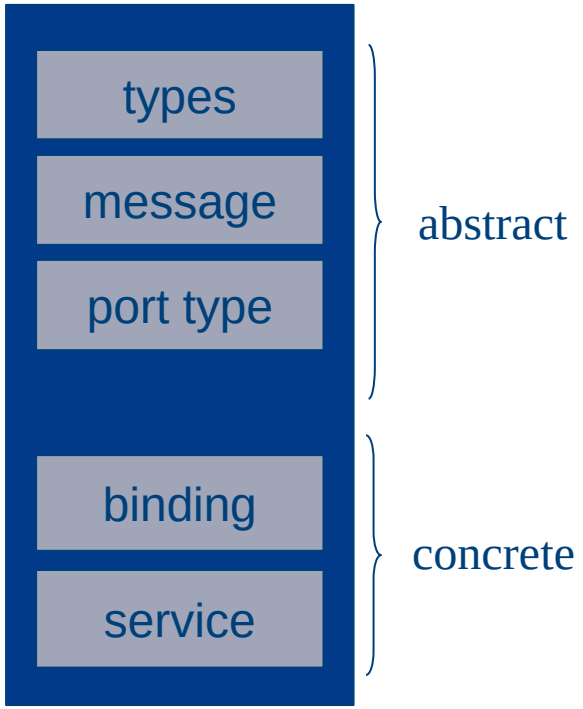


WSDL = IDL for SOAP
Webservice description language

```
<wsdl:definitions targetNamespace="http://math.example.com" name="MathFunctionsDef">
<wsdl:message name="addIntResponse">
        <wsdl:part name="addIntReturn" type="xsd:int" />
</wsdl:message>
<wsdl:message name="addIntRequest">
        <wsdl:part name="a" type="xsd:int" />
        <wsdl:part name="b" type="xsd:int" />
</wsdl:message>
<wsdl:portType name="AddFunction">
        <wsdl:operation name="addInt" parameterOrder="a b">
                <wsdl:input message="impl:addIntRequest" name="addIntRequest" />
                <wsdl:output message="impl:addIntResponse" name="addIntResponse" />
        </wsdl:operation>
</wsdl:portType>
<service name="MathFunctions"/>
</wsdl:definitions>
```

WSDL 1.1 Structure

- **Types** define the XML types of the used elements
  - Contains type definitions as XML schemas (or reference to schema)
- **Message** defines the transmitted data
  - What *messages* can be received/sent
  - A message consists of different *parts*, defined by a type
- **Port type** defines an interface
  - The set of *operations* offered by one or more endpoints
  - Each operation consists of an input and/or output message
- **Binding** tells how a port type is bound to the transport protocols
  - Specifies the transport *protocol* to use
  - Contains information specific to the transport protocol, necessary to bind the operations
- **Service** defines an offered service
  - Collection of related ports (*endpoints*)
  - Each port is a combination of a binding (*how*) and a network address (*where*)



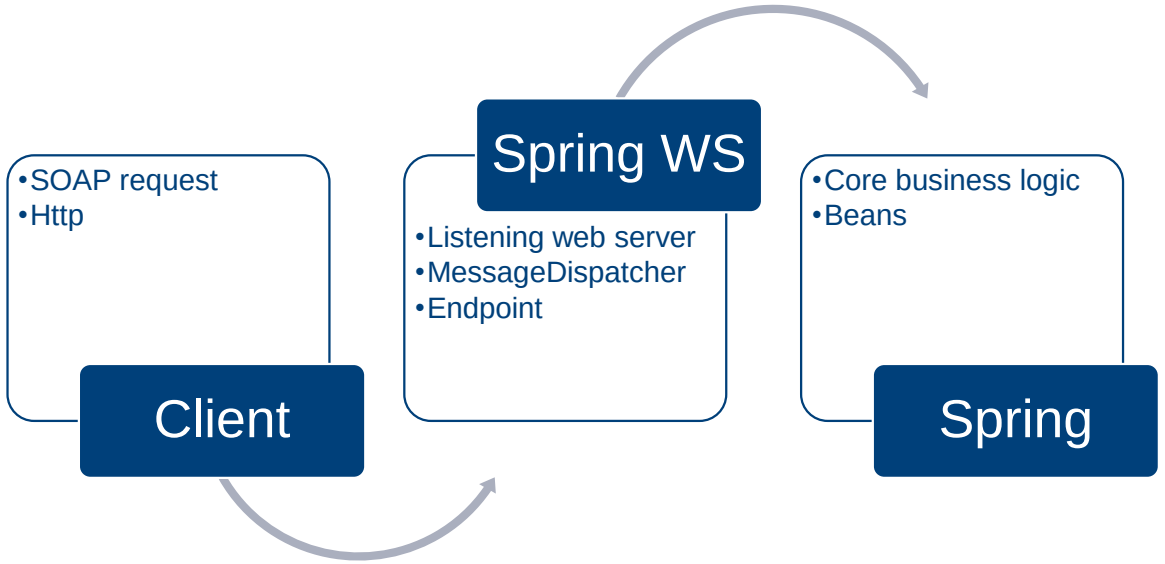
WSDL 1.1 specification: http://www.w3.org/TR/wsdl

Supporting technologies and frameworks

Standard vs implementation

- SOAP = Standard
- Many implementations
  - .net
  - Java
    - JAX-WS
    - Spring SOAP
    - …

Spring web services



SOA exercise session
Application domain: Foodservice.io

Domain concepts

- Expose menu online
- Meals
  - Name
  - Description
  - kcal (energy value)
  - (price)
  - mealtype
    - vegan
    - veggie
    - meat
    - fish

Operations:

- Meal getMeal(name)
- Meal GetBiggestMeal()

- Exercise:
  - Meal GetCheapestMeal()
  - Confirmation OrderMeal(name, address)

Meals.xsd

Meal concept

```
<xs:complexType name="meal">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="kcal" type="xs:int"/>
    <xs:element name="description" type="xs:string"/>
    <xs:element name="mealtype" type="tns:mealtype"/>
  </xs:sequence>
</xs:complexType>

<xs:simpleType name="mealtype">
  <xs:restriction base="xs:string">
    <xs:enumeration value="vegan"/>
    <xs:enumeration value="veggie"/>
    <xs:enumeration value="meat"/>
    <xs:enumeration value="fish"/>
  </xs:restriction>
</xs:simpleType>
```

operations

```
<xs:element name="getMealRequest">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="getMealResponse">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="meal" type="tns:meal"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

KU LEUVEN

Meals Bean: pure application logic for spring. No soap. No service

Mealrepository

```
@Component
public class MealRepository {
   private static final Map<String, Meal> meals = new HashMap<>();

   @PostConstruct
   public void initData() {
      Meal a = new Meal();
      a.setName("Steak");
      a.setDescription("Steak with fries");
      a.setMealtype(Mealtype.MEAT);
      a.setKcal(1100);
      meals.put(a.getName(), a);

      Meal b = new Meal();
...
   }
...
}
```

Meal operations

```
public Meal findMeal(String name) {
      Assert.notNull(name, "The meal's code must not be null");
      return meals.get(name);
   }

public Meal findBiggestMeal(){

      if (meals ==null) return null;
      if (meals.size() == 0) return null;

      var values = meals.values();
      return values.stream()
.max(Comparator.comparing(Meal::getKcal))
.orElseThrow(NoSuchElementException::new);

   }
```

MenuEndpoint
mapping soap operations

```
public class MenuEndpoint {
   private static final String NAMESPACE_URI ="http://foodmenu.io/cs/webservice";
   private MealRepository mealrepo;

   @Autowired
   public MenuEndpoint(MealRepository mealrepo) {
      this.mealrepo = mealrepo;
   }

   @PayloadRoot(namespace = NAMESPACE_URI, localPart = "getMealRequest")
   @ResponsePayload
   public GetMealResponse getMeal(@RequestPayload GetMealRequest request) {
      GetMealResponse response = new GetMealResponse();
      response.setMeal(mealrepo.findMeal(request.getName()));
      return response;
   }
...
```

Message dispatcher: WsConfigurerAdapter
Web service config and setup of ports/bindings

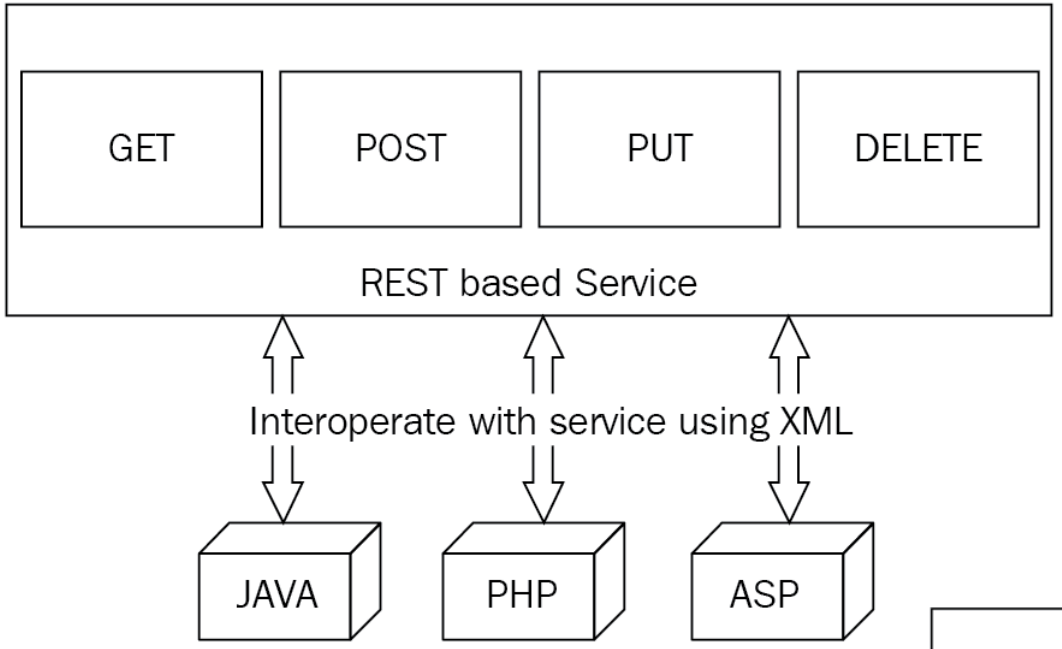
```
@EnableWs
@Configuration
public class WebServiceConfig extends WsConfigurerAdapter {
   @Bean
   public ServletRegistrationBean<MessageDispatcherServlet> messageDispatcherServlet(ApplicationContext applicationContext) {
      MessageDispatcherServlet servlet = new MessageDispatcherServlet();
      servlet.setApplicationContext(applicationContext);
      servlet.setTransformWsdlLocations(true);
      return new ServletRegistrationBean<>(servlet, "/ws/*");
   }
   @Bean(name = "meals")
   public DefaultWsdl11Definition defaultWsdl11Definition(XsdSchema mealsSchema) {
      DefaultWsdl11Definition wsdl11Definition = new DefaultWsdl11Definition();
      wsdl11Definition.setPortTypeName("MealsPort");
      wsdl11Definition.setLocationUri("/ws");
      wsdl11Definition.setTargetNamespace("http://foodmenu.io/cs/webservice");
      wsdl11Definition.setSchema(mealsSchema);
      return wsdl11Definition;
   }
   @Bean
   public XsdSchema mealsSchema() {
      return new SimpleXsdSchema(new ClassPathResource("meals.xsd"));
   }
}
```

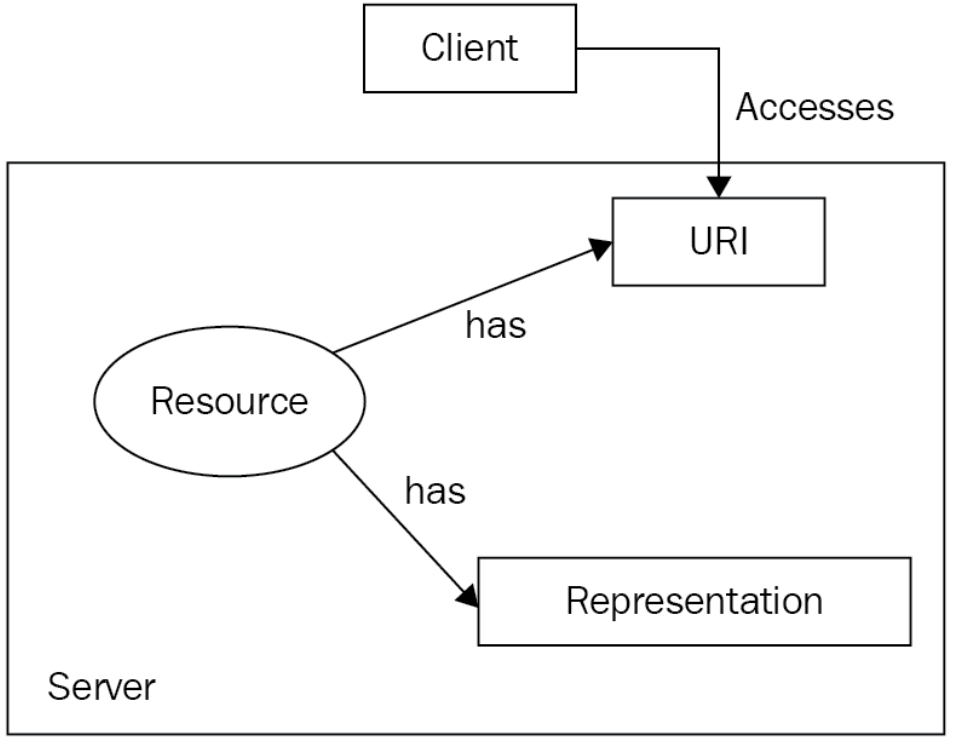
REST

Representational state transfer
Roy Fielding, PhD. 2000.
Author of HTTP protocol.

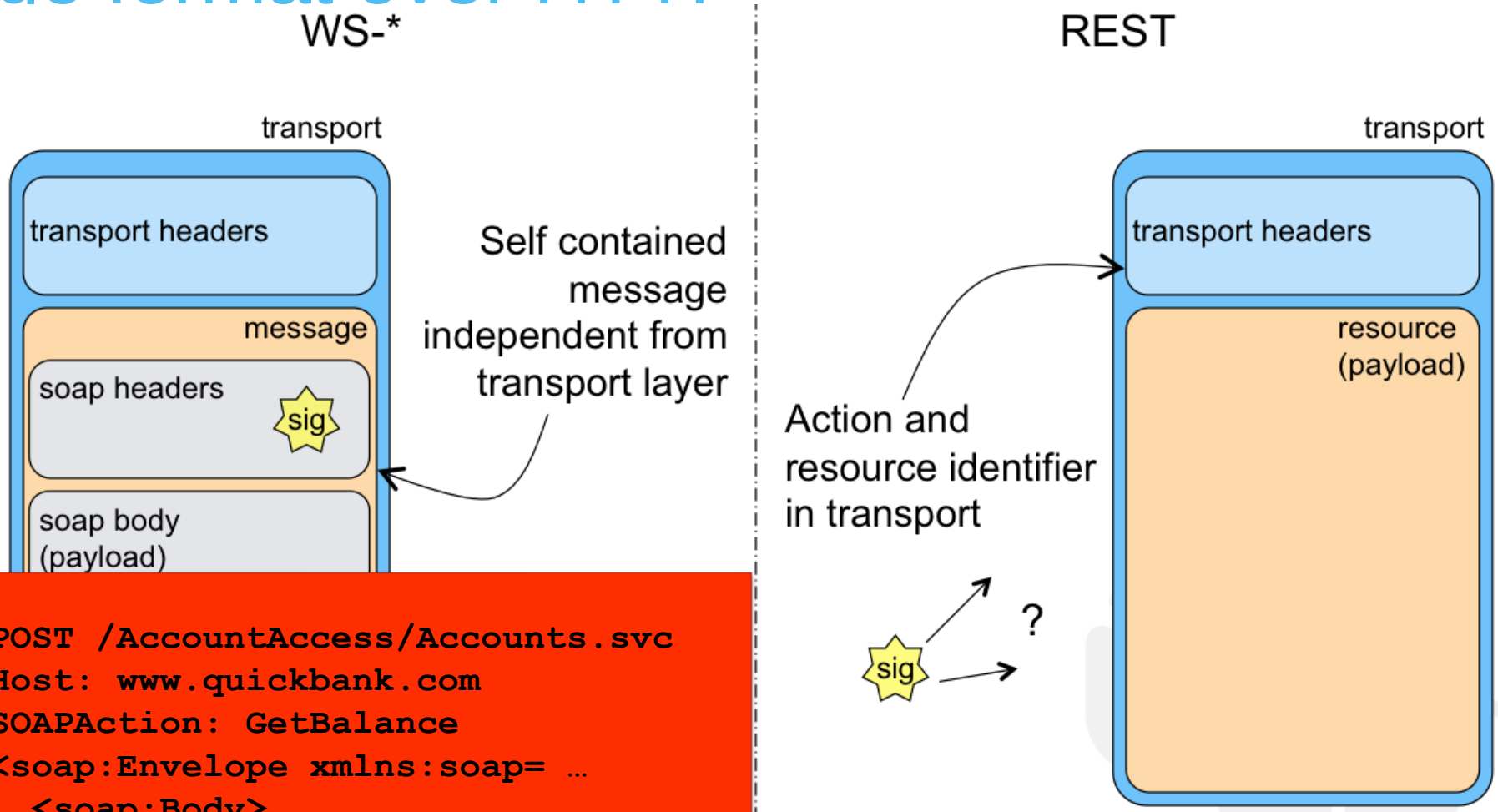
KU LEUVEN

REST: back to basics
HTTP as basic paradigm





Message format over HTTP




```
POST /AccountAccess/Accounts.svc
Host: www.quickbank.com
SOAPAction: GetBalance
<soap:Envelope xmlns:soap= …
  <soap:Body>
    <GetBalance xmlns= …
      <Account>2</Account>
    </GetBalance>
  </soap:Body>
</soap:Envelope>
```

```
GET www.quickbank.com/Accounts/2
```

XML is hard/slow to consume in light-weight clients and javascript-based browsers

Person object

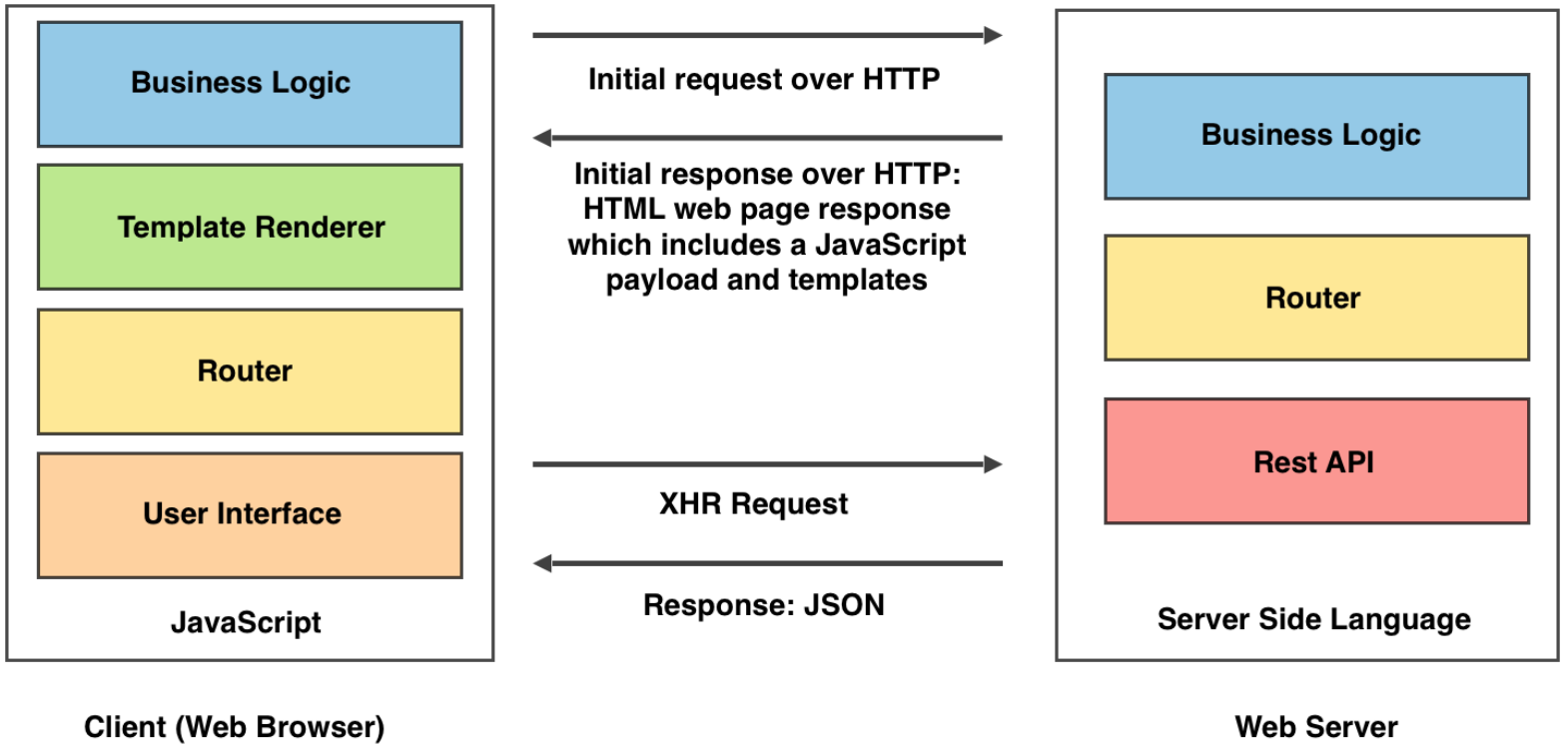
- `var person = { name: "John", age: 31, city: "New York" };`

```
{"people": [
{
        "name": "John Smith",
        "city": "London",
        "country": "United Kingdom",
        "age": 27
},
{
        "name": "George Burns",
        "city": "New York",
        "country": "USA",
        "age": 32
}
]}
```

JSON

- The JSON syntax is a subset of the JavaScript syntax.
- JSON syntax is derived from JavaScript object notation syntax:
- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

Single page applications:
REST + JSON (driven by browser)

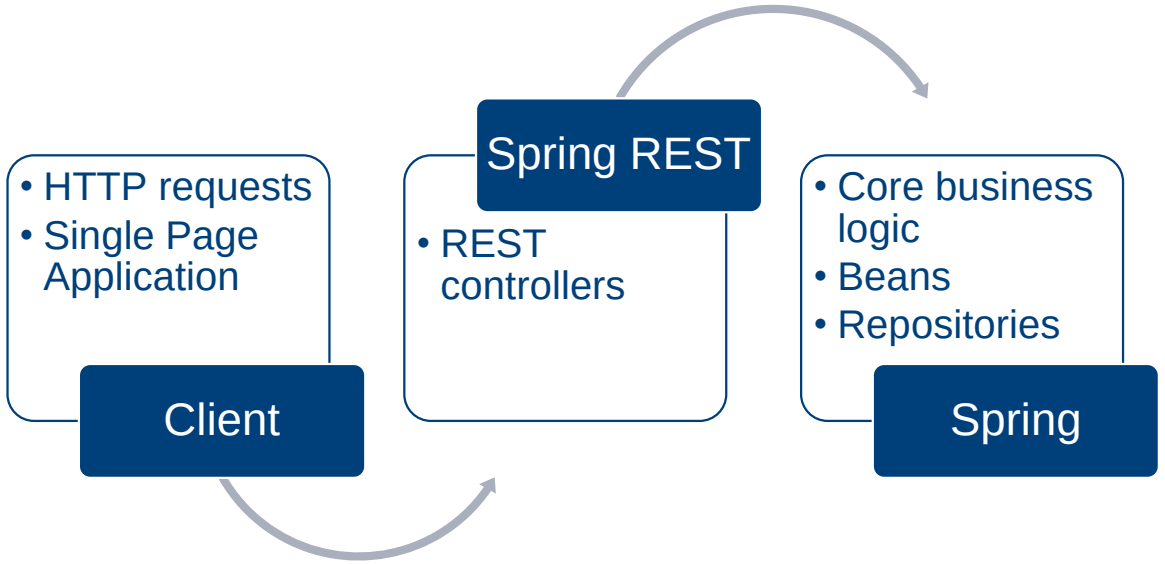


Supporting technologies and frameworks

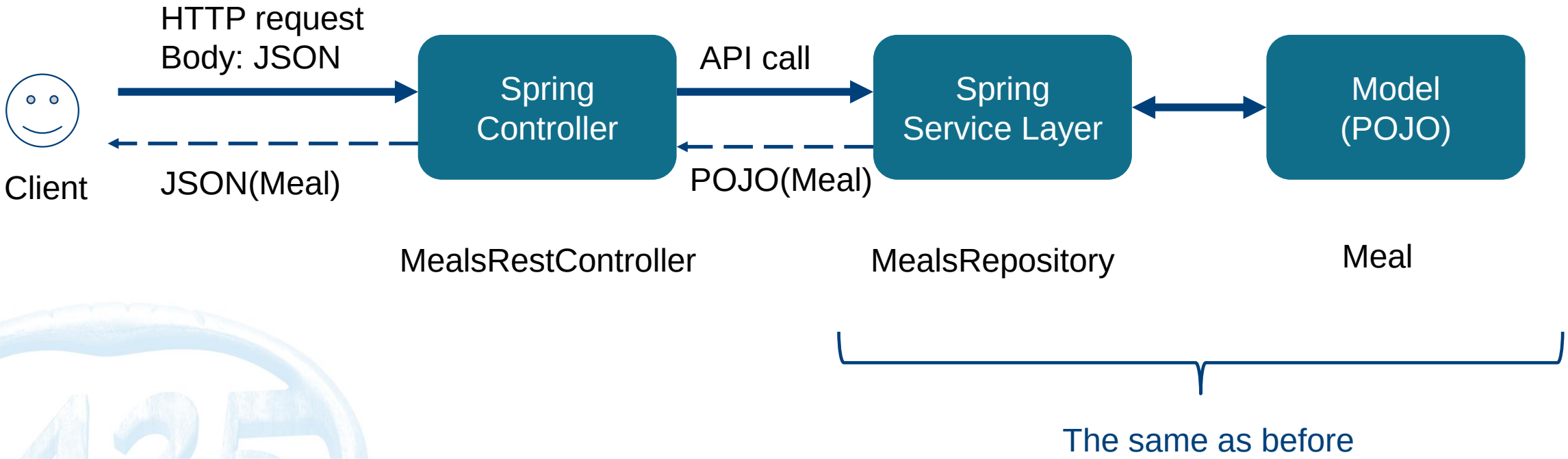
Implementations

- HTTP implementation is available in most frameworks
- Implementations
  - .NET REST
  - Java (spec: JAX-RS):
    - Spring (Tomcat/Jetty)
    - Jersey (Oracle)
    - RestEasy (JBoss)
    - Apache
    - Restlet
    - …

Spring web services



Spring REST Architecture: *food service*



Spring REST Controller

```
@RestController
public class MealsRestController {

    private final MealsRepository mealsRepository;

    … // constructors

    @GetMapping("/meals/{id}")
    Meal getMealById(@PathVariable String id) {
        Optional<Meal> meal = mealsRepository.findMeal(id);

        return meal.orElseThrow(() -> new MealNotFoundException(id));
    }

    @GetMapping("/meals")
    Collection<Meal> getMeals() {
        return mealsRepository.getAllMeal();
    }
}
```

Inter-process remote communication: Interoperability history

90's

- Corba
- COM
- Remote objects

2003

- Service-oriented architecture
- Web Services / SOAP
- XML everywhere
- Lots of standards
- Bloated middleware

2009-2010

- REST
- Initially XML
- Less middleware
- Reduce to HTTP essentials
- Json for javascript clients
- Lightweigth clients/apps

20xx

- gRPC
- Protocol buffers
- Micro-services
- Intra-datacenter