INFOH509 XML & Web Technologies

Lecture 10

RESTFUL WEB SERVICES

What is a Web Information System?

- **Information system (IS)** refers to the networks of hardware and software that people and organizations use to collect, filter, process, create, and distribute data.

--Wikipedia

- **Web information system**, or web-based information system, is an information system that uses Internet web technologies to deliver information and **services**, to users or other information systems/applications. It is a software system whose main purpose is to publish and maintain data by using hypertext-based principles.

--Wikipedia

What is the Web?

- **1989:** At the CERN physics laboratory, Tim Berners-Lee designs a simple global hypermedia system, now known as the **World Wide Web**

Three constitutents (with their **current meaning**):

- URIs as a means for **identifying &** locating **resources**
- HTTP as a protocol for transmitting **information** over networks
- Several data formats for describing information (**representations of resources**):
  - HTML for display in a web browser
  - XML as a data exchange format
  - JSON as a data exchange format, alternative to XML
  - RDF as a machine-interpretable data model

Part I

INTRODUCTION TO SERVICES

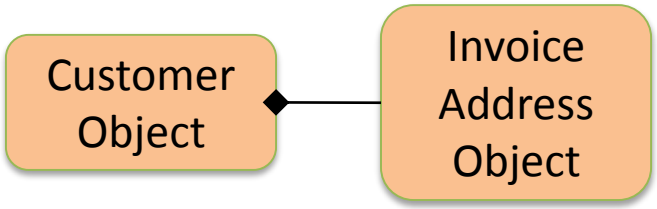
What is a service?

- The term **service** is like the term **multimedia**: lots of people have given different definitions.

- Essentially, a **service** is a software function or component:
  - It may carry out a business task,
  - provides access to files,
  - Perform generic functions like authentication and logging,
  - …

- Services reflect a new 'service-oriented' approach to programming, based on the idea of composing applications by discovering and invoking network-available services rather than building new applications or by invoking available applications to accomplish some task.

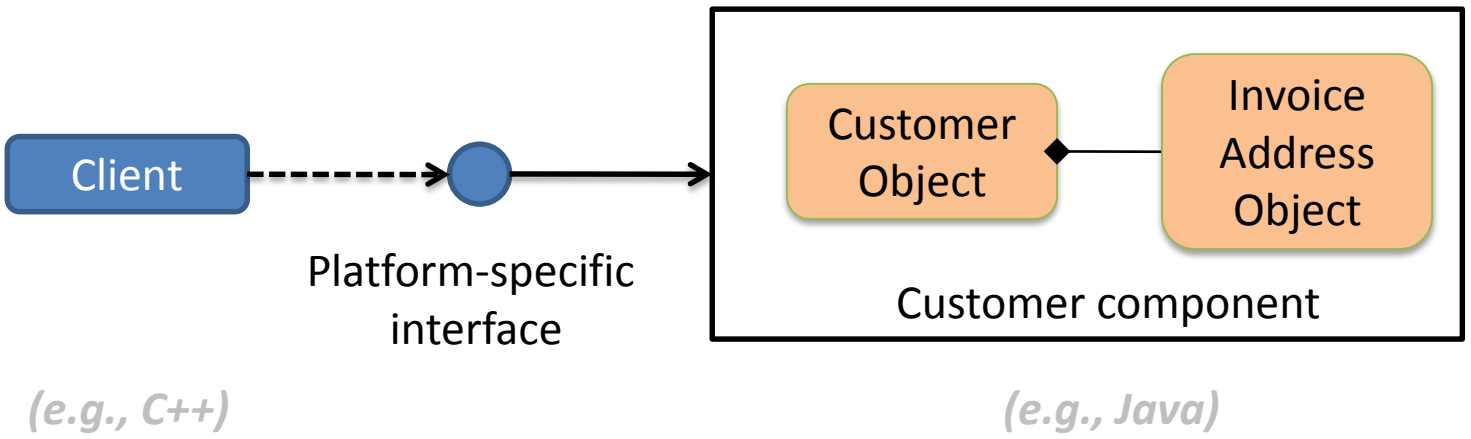
Some history: from local to distributed objects



(e.g., Java)

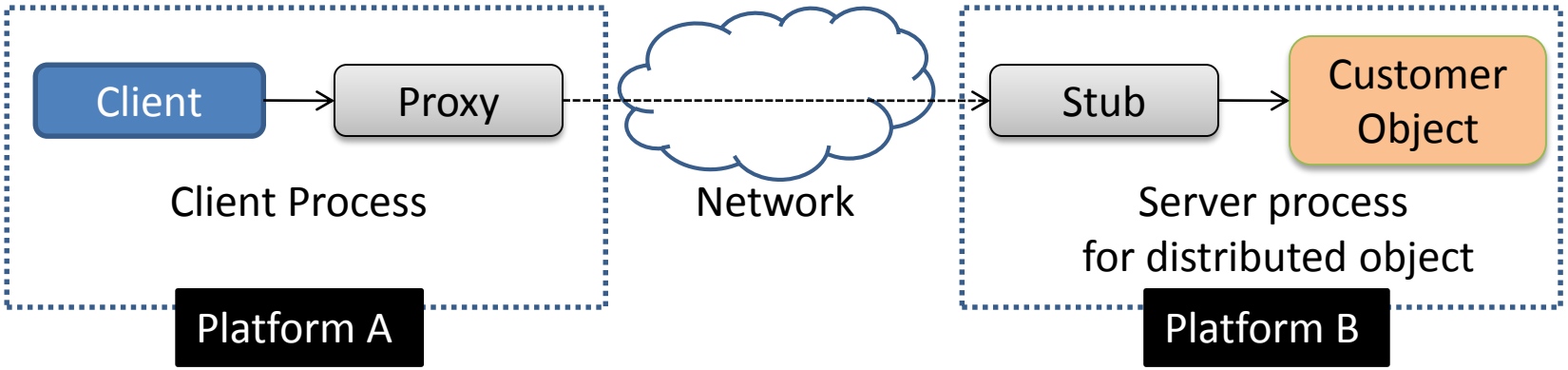
- **Object**-oriented programming allows encapsulation of both behavior and data.
- Clients use objects by first instantiating an object, and then calling their properties & methods
- Re-use is usually restricted to the same programming language and platform

Some history: from local to distributed objects



- **Components** were devised to facilitate software reuse across disparate programming languages.
- Components group related objects into (binary) units that can be plugged into applications (cf. electronic component assembly in circuit boards).
- Component reuse typically restricted to same computing platform due to incompatible binary interfaces.

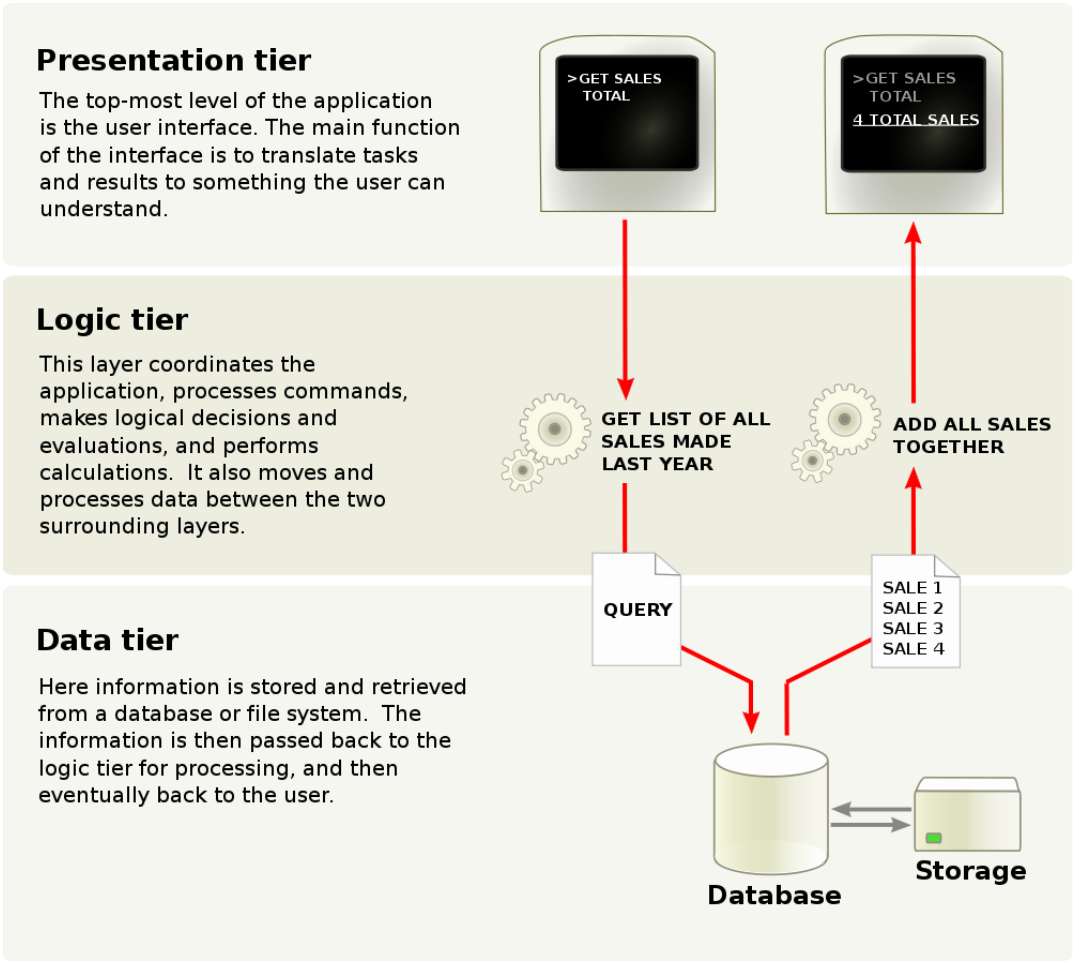
Some history: from local to distributed objects



- **Distributed objects.** To share and reuse objects, objects were deployed to remote servers. Clients connect to such objects through a remoting technology (CORBA, DCOM, Java or .NET Remote Method Invocation).
- Clients and distributed objects live in separate machines, and can therefore live in separate programming languages and platforms.
- Reuse restricted to the same remoting technology
- Scalability problems due to client state

Some history: from local to distributed objects

Here be services! →



- Distributed objects were quickly grouped into a logic tier that stores all of the application logic – these are already "services"

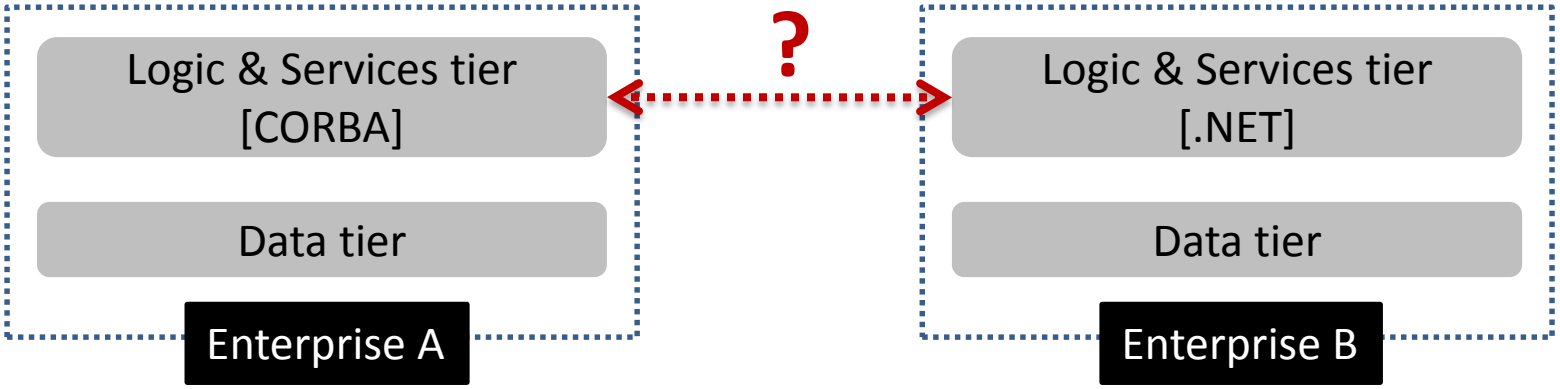
What is a service? (cont.)

- Services hence provide logical functions that are shared across different applications. They enable software that runs on disparate computing platforms to collaborate.

- A platform may be any combination of hardware, operating systems (e.g., Windows, Linux, Android, iOS), software framework (java, .Net, Rails), and programming language.

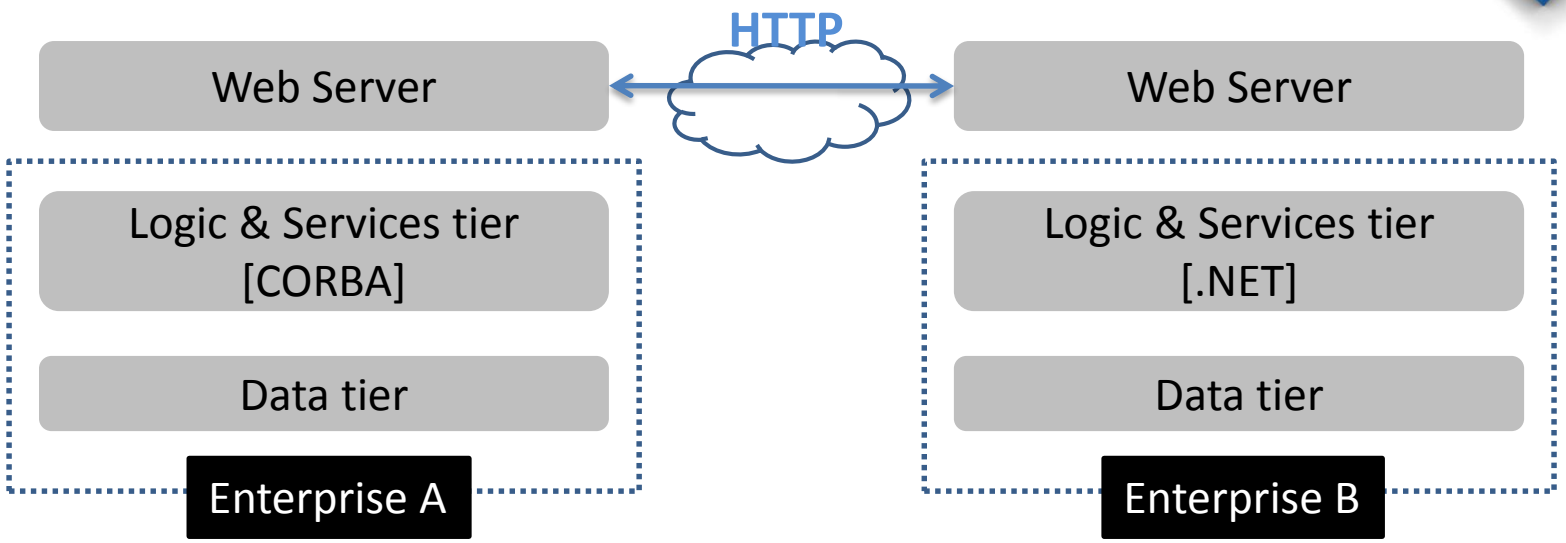
- The service-oriented paradigm to programming utilizes services as the constructs to support the rapid development of easily composable distributed applications (again cf. electronic component assembly in circuit boards).

What is a Web Service?



- Service reuse is restricted to the same remoting technology when built on traditional distributed object architectures.
- This is especially problematic in enterprise integration and communication scenarios, where services must be callable from outside enterprise boundaries.

What is a Web Service?



- **Web services** integrate disparate systems and expose reusable business functions over the web (HTTP).
- They leverage HTTP either:
  - as a simple transport over which data is carried (e.g., SOAP/WSDL services), or
  - a complete pre-defined application protocol (RESTful services).

Where are Services used?

- Within an enterprise (Enterprise Application Integration)
  - Accelerate and reduce the cost of integration
  - Save on infrastructure deployment and management costs
  - Reduce skill requirements
  - Improve reuse

- Between enterprises (E-business integration, B2B)
  - Providing service to a company's customers
  - e.g., an Insurance company wishes to link its systems to the systems of a new institutional customer

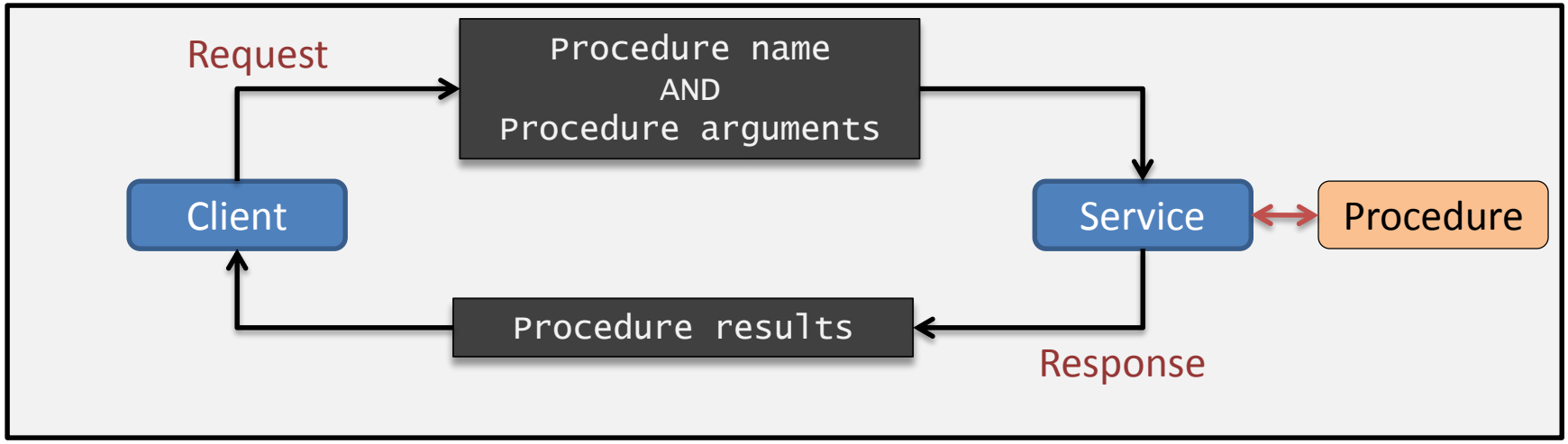
  - Accessing services from a company's partners and suppliers
  - e.g., dynamically link to new partners and suppliers to offer their services to complement the value the company provides

  - Standards and common infrastructure reduce the barriers
  - Simplicity accelerates deployment
  - Dynamics opens new business opportunities

Web Service API styles

- RPC (Remote Procedure Call)
- Message-based
- Resource-based

RPC Style (1/2)



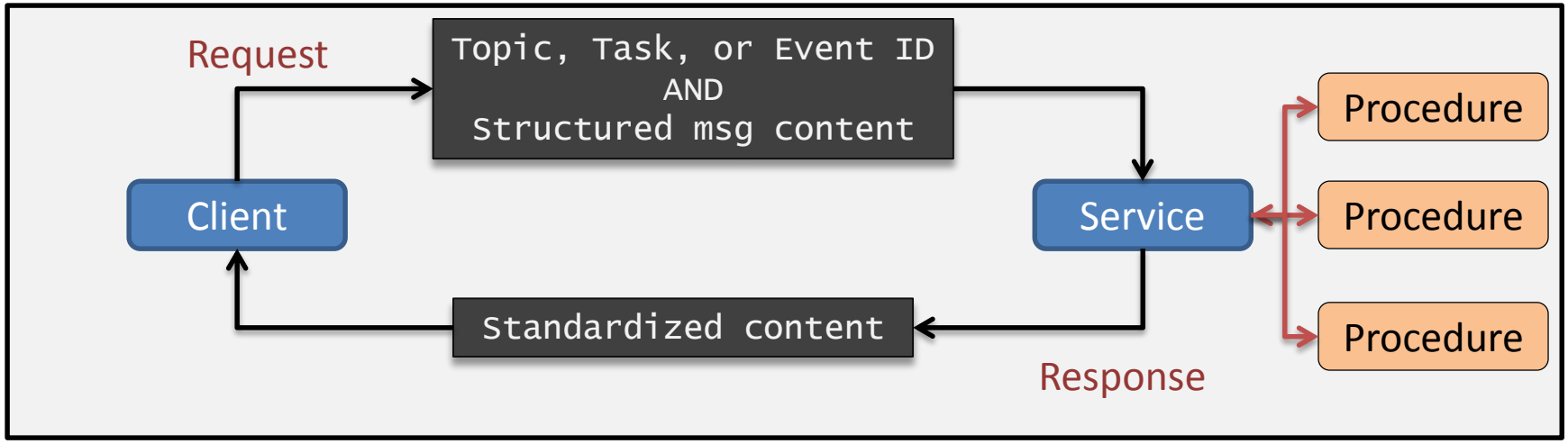
- Client sends message to a remote server and blocks while waiting for response
- Request message identifies the procedure to be executed and its arguments
- Server decodes message, maps message arguments directly to input parameters, executes procedure, and sends (serialized) results back to client

RPC Style (2/2)

- Pros:
  - Very easy to implement (lots of frameworks that automate the process, e.g. JAX-WS framework for Java)

- Cons:
  - Usually inflexible and fragile: tight coupling between client and service, if procedure needs to change (e.g., number of arguments), all clients need to be rewritten.
  - Usually restricted to **synchronous communication** (client blocks while waiting for response)

Message-based style



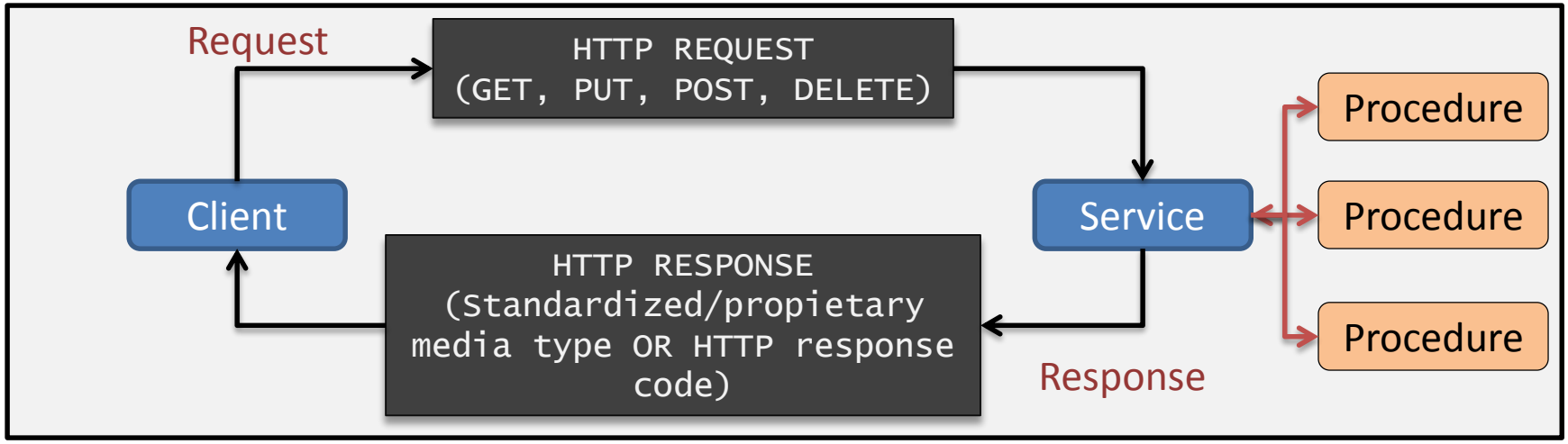
- In a message-based API, messages are not derived from the signatures of remote procedures.
- Instead, messages may carry information on specific topics, tasks to execute, and events.
- The server selects the correct procedure to execute based on the message content

Message-based Style (2/2)

- Pros:
  - Looser coupling between clients and servers
  - Support for asynchronous communication [necessary on web-scale networks]

- Cons:
  - Messages must be standardized somehow. This is easy if communication is within the same organization, but more difficult when many parties are involved.

Resource-based style



- In a resource-based API, all procedures, instances of domain data, and files are given a URI.
- HTTP is used as a complete application protocol to define standard service behavior.
- Information is exchanged based on standardized media types (JSON, XML, …) and HTTP response codes where possible
- Clients manipulate the state of resources through representations (e.g., a database table row may be represented as XHTML, XML, or JSON).

Two competing technology stacks

- **Big Web Services (WS-*)** → **Next lecture**
  - Various (complex) protocols on top of HTTP (SOAP, UDDI, WSDL, WS-Addressing, …)
  - Is mostly used to implement RPC-style services, but can be used to implement any of the three
  - Lots of standards! Primarily meant to create web services that involve more than 2 peers.

- **RESTful Web Services** → **This lecture**
  - Use ONLY HTTP and standard media types
  - Restricted to Resource-style services
  - Conceptually simpler, but mainly restricted to web services that are limited to two endpoints

A word of caution

- Web Service calls entail distributed communication & programming
  - Network latency
  - Failures
  - Complexity of distributed programming

- So using web services only makes sense in situations where out-of-processes and cross-machine calls make sense.

Part II

REST
= REPRESENTATIONAL STATE TRANSFER

REST: some history

> *At the beginning of our efforts within the Internet Engineering Taskforce to define the existing (HTTP/1.0) and design the extensions for the new standards of HTTP/1.1 and Uniform Resource Identifiers (URI), we recognized the need for a model of how the World Wide Web (WWW, or simply Web) should work. This idealized model of the interactions within an overall Web application—what we refer to as the Representational State Transfer (REST) architectural style—became the foundation for the modern Web architecture, providing the guiding principles by which flaws in the existing architecture could be identified and extensions validated prior to deployment.*

- The term REpresentational State Transfer was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation.
- Fielding is one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification versions 1.0 and 1.1.

REST: some history

"

A software architecture is an abstraction of the runtime elements of a software system during some phase of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture. An architecture determines how system elements are identified and allocated, how the elements interact to form a system, the amount and granularity of communication needed for interaction, and the interface protocols used for communication.

REST is a coordinated set of architectural constraints that attempts to minimize latency and network communication, while at the same time maximizing the independence and scalability of component implementations. This is achieved by placing constraints on connector semantics, where other styles have focused on component semantics. REST enables the caching and reuse of interactions, dynamic substitutability of components, and processing of actions by intermediaries, in order to meet the needs of an Internet-scale distributed hypermedia system.

From "*Principled design of the modern Web architecture*."

REST: some history

> *The name "Representational State Transfer" is intended to evoke an image of how a well-designed Web application behaves: a network of Web pages forms a virtual state machine, allowing a user to progress through the application by selecting a link or submitting a short data-entry form, with each action resulting in a transition to the next state of the application by transferring a representation of that state to the user.*

- The modern (human) Web is one instance of a REST-style architecture
- RESTful web services transfer these ideas from the human web to web services, founded by idea that there should be no essential difference between the human web (designed for human consumption) and the "programmable web" designed for consumption by software

ROA: Some History

- REST is actually a meta-architecture: it is a collection of architectures – the current Web is just one instance.
- In 2007, Richardson & Ruby introduced the term Resource Oriented Architecture (ROA) to refer to a set of design guidelines and best practices (adhering to the REST constraints) that should be used to design RESTful Web Services

- We mention the ROA guidelines in what follows.
- Beware, however, that many current web services do not implement or follow all of these guidelines. [Your milage may vary].

Key REST concepts

- Rest consists of 4 key concepts:
  - Resources
  - Resource names (URIs)
  - Resource representations
  - Links between resources
- And 4 key properties:
  - Addressability
  - Statelessness
  - Connectedness
  - The Uniform Interface

Resources

> *The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g., "today's weather in Los Angeles"), a collection of other resources, a nonvirtual object (e.g., a person), a concept and so on.*

- Resource examples
  - A historical building
  - The newspaper "le soir"
  - The newspaper "le soir" at a particular date
  - The collection of all Belgian newspapers
  - The Belgian prime minister
  - The preferred newspaper of the prime minister

RESTFUL WEB SERVICE DESIGN GUIDELINE 1:

Every object manipulated by the web service (or web application) should be identified and exposed as a resource.

URIs: names for resources

- A URI is the name of a resource
- Examples:
  - http://www.lesoir.be
  - http:// www.lesoir.be/edition/20-01-2012
  - http:// www.example.org/newspapers/belgium
  - http:// www.example.org/newspapers?country=belgium

RESTFUL WEB SERVICE DESIGN GUIDELINE 2:

- Every identified resource must be assigned at least one URI. This ensures it is **addressable**
- A URI should never represent more than one resource.
- Resources can have multiple URIs, but should have as few URIs as possible.

Representations

- A representation is a description of (some part of) the resource.
- A resource can have multiple representations (one in HTML, one in XML, one in a Google protocol buffer, …).
- Example:
  - http:// www.example.org/newspapers/belgium could support both HTML and JSON

RESTFUL WEB SERVICE DESIGN GUIDELINE 3:

Specify, for every resource:

- The representations that the service serves to the client
- The representations that the service accepts from the client

Use standard representations whenever possible

On content negotiation and URIs

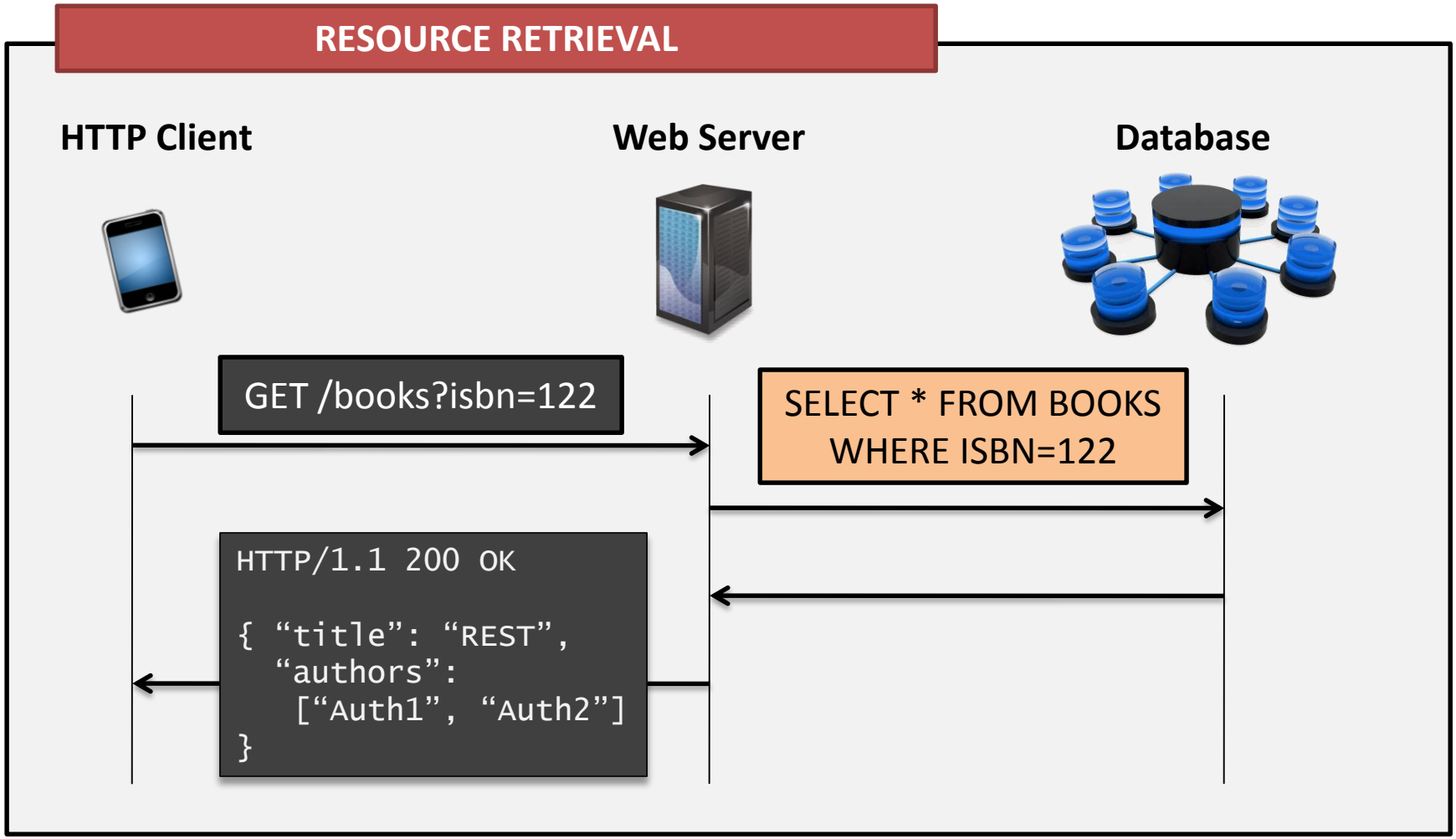
- Content negotiation can be used to get dinstinct representations of the same resource.
- By giving these distinct representations their own URIs, however, we also make them addressable
- Example:
  - http:// www.example.org/newspapers/belgium
  - http:// www.example.org/newspapers/belgium.json
  - http:// www.example.org/newspapers/belgium.xml
  - http://www.belgium.be/consitution
  - http://www.belgium.be/constitution.nl
  - http://www.belgium.be/constitution.fr
- Addressability is useful, e.g.,
  - http://myservice.com/rank?site=http%3A%2F%2Fwww.example.org%2Fnewspapers%2Fbelgium.json

The Uniform Interface

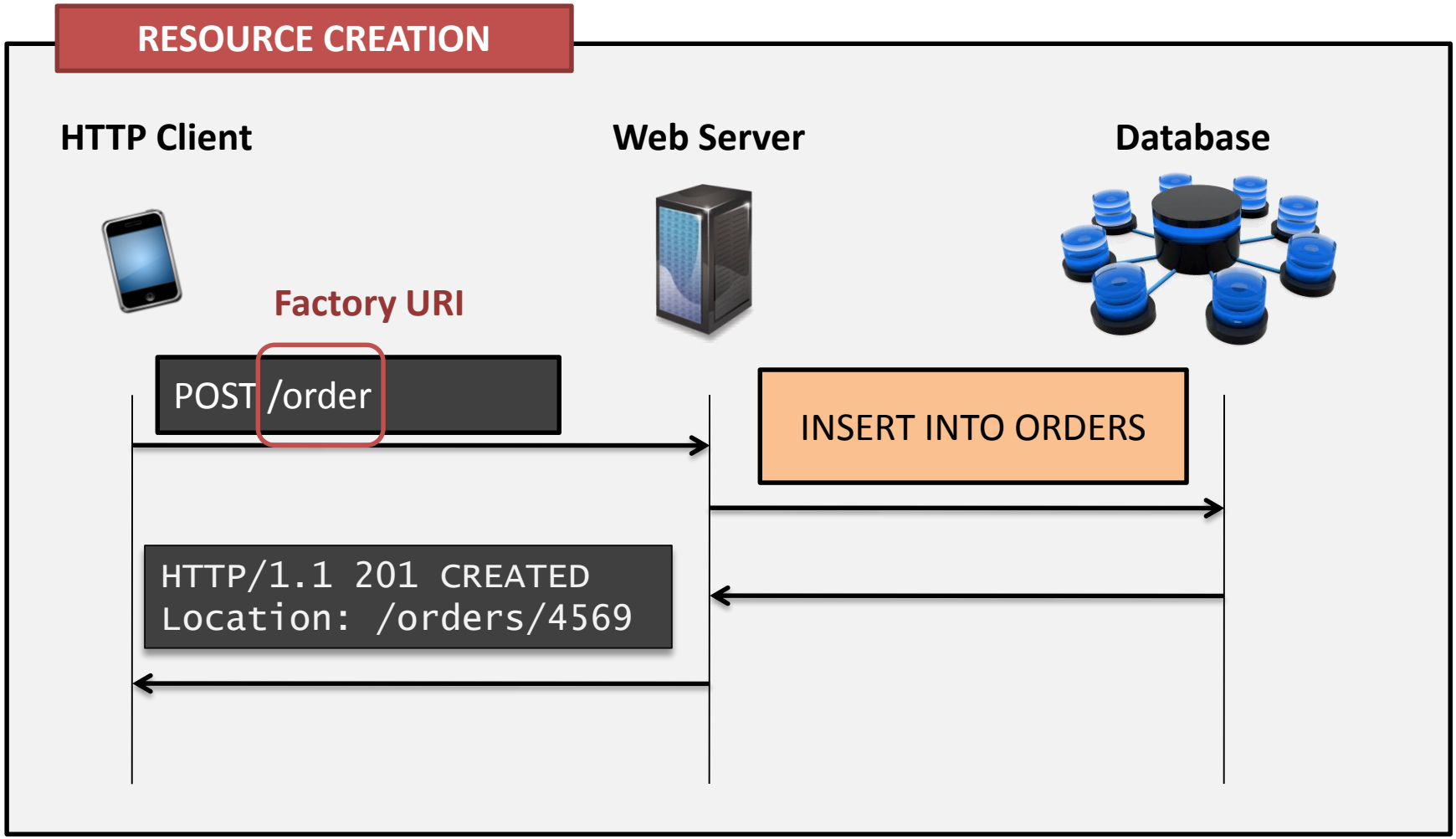
- All access to resources happens through HTTP uniform interface (GET, POST, PUT, DELETE, HEAD, OPTIONS).

CRUD	REST	
CREATE	POST	Create a (sub)resource
RETRIEVE	GET	Retrieve a representation of a resource
UPDATE	PUT	Modify a resource/create a new resource
DELETE	DELETE	delete a resource
	OPTIONS	Discover what HTTP methods are supported by the resource
	HEAD	requests headers only (similar to GET but omits representation)

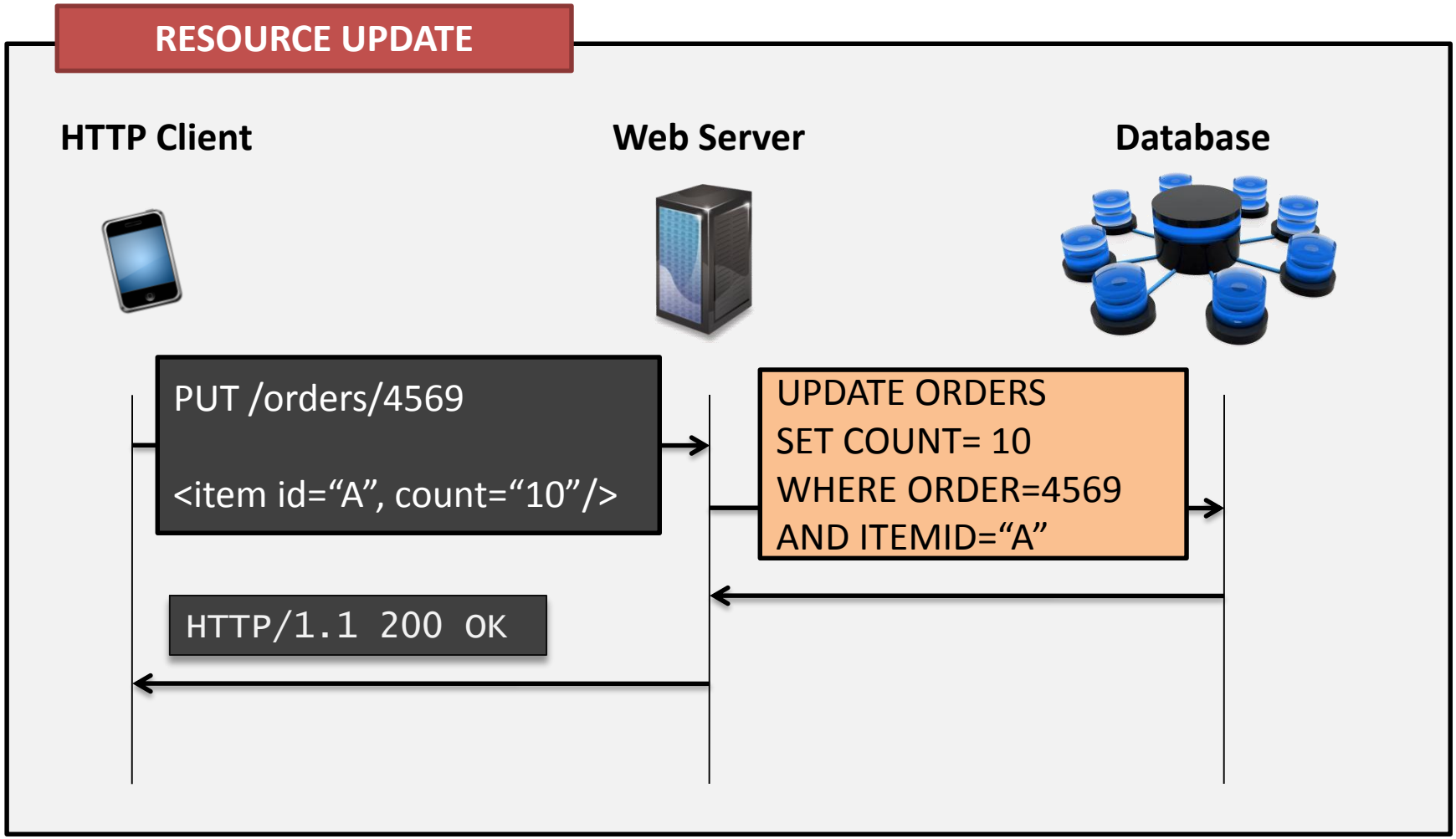
The Uniform Interface (cont.)



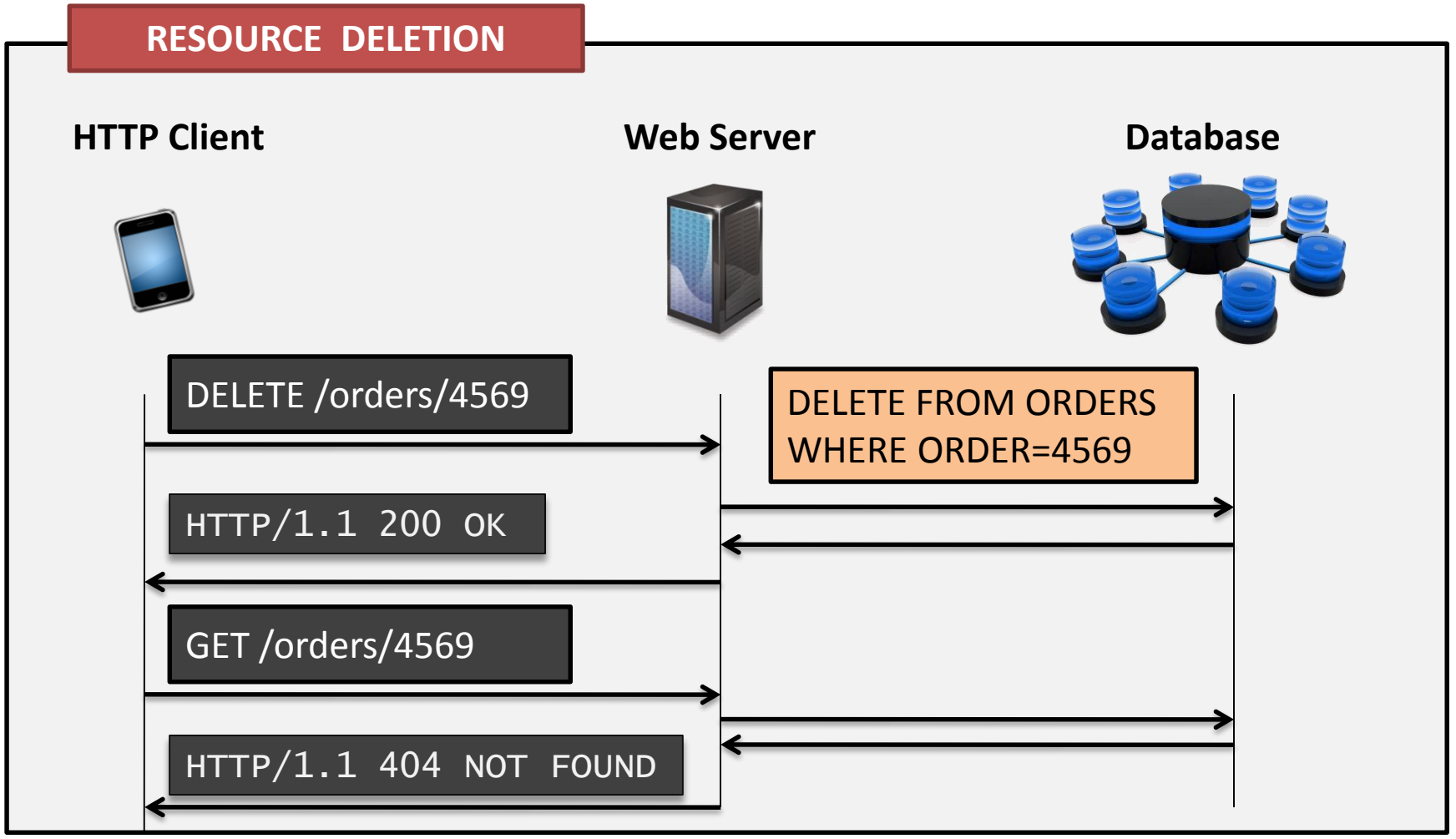
The Uniform Interface (cont)



The Uniform Interface (cont)



The Uniform Interface (cont)



The Uniform Interface (cont)

- All access to resources happens through HTTP uniform interface (GET, POST, PUT, DELETE, HEAD, OPTIONS).
- All information necessary to understand the request must be contained in the request message.

RESTFUL WEB SERVICE DESIGN GUIDELINE 4:

Specify, for every URI (and hence, resource): the HTTP methods supported (e.g., GET and POST, but not DELETE, PUT)

Allow querying of these operations by supporting OPTIONS

Statelessness

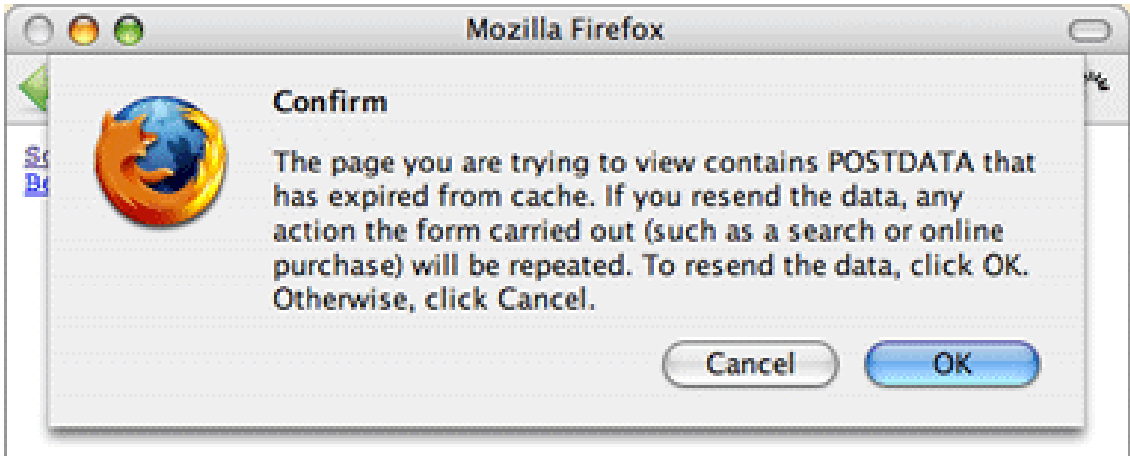
- Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any client state stored on the server.
- Application state is therefore kept entirely on the client.
- Resource state is of course still kept on the server
- Statelessness:
  - Improves reliability because it makes it easier to recover from partial failures
  - Improves scalability because:
    - Servers can quickly free computing resources after each request
    - Different requests can be handled by different servers (load balancing) exactly because the server doesn't have to manage resource usage across requests.

Safety & idempotency

- GET, HEAD, OPTIONS are read-only operations but PUT, POST, DELETE are read-write operations with side effects.
- An operation f is called idempotent if

$$f(f(x)) = f(x)$$

- PUT and DELETE are idempotent.
- Idempotent and read-only operations can safely be re-executed multiple times (e.g., network timeouts) without risking errors
- POST is not idempotent nor read-only, and is not safe to re-execute



Why the Uniform Interface matters

- Consider a GET of

  http://api.del.icio.us/posts/delete

- This misuses GET and does not adhere to the uniform interface
- But software programs don't know this. Programs that follow a link by GETTing it may hence (inadvertly) delete data. [e.g., Google Web Accelerator]

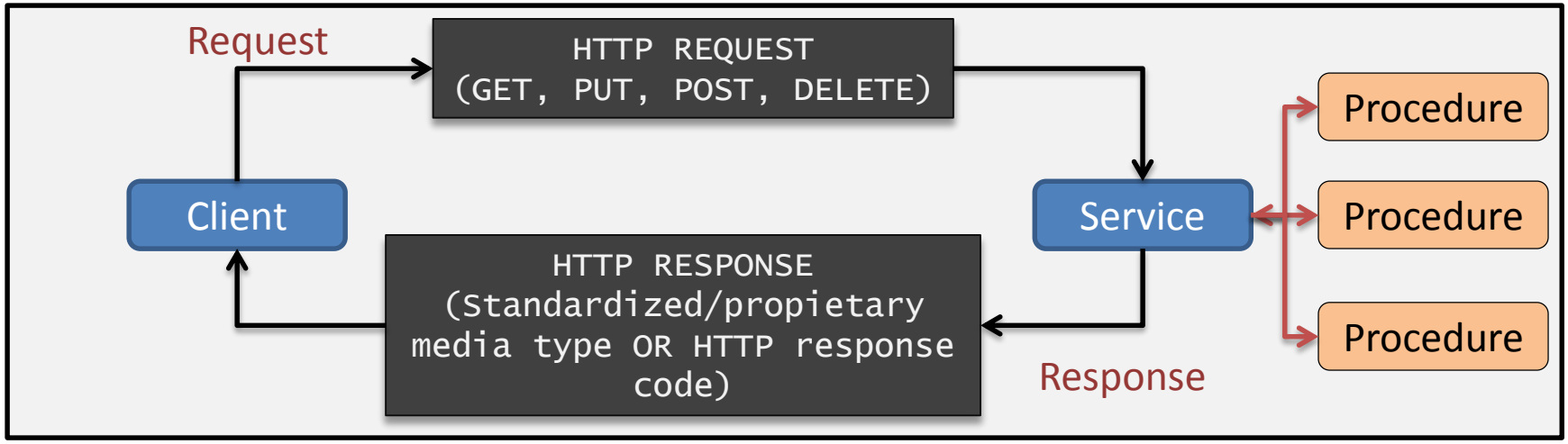
RESTFUL WEB SERVICE DESIGN GUIDELINE 5:

Use the HTTP methods correctly when designing web services.

Connecting resources

- Server response representations should include links to other relevant resources
- This makes a web service self-documenting (the representation can be parsed to see what other resources can be accessed).
- [This will become more clear with the example that follows]

Summary: REST = Resource-based API



- In a resource-based API, all procedures, instances of domain data, and files are given a URI.
- HTTP is used as a complete application protocol to define standard service behavior.
- Information is exchanges based on standardized media types (JSON, XML, ATOM, …) and HTTP response codes where possible
- Clients manipulate the state of resources through representations (e.g., a database table row may be represented as XHTML, XML, or JSON).

Discussion

- In order to allow clients to cache representations that do not change frequently, the server should include the following headers:
  - Last-Modified
  - Etags

- This allows clients to use conditional get (using e.g., If-Modified-Since and If-None-Matches)

REST: AN EXAMPLE

Design procedure

- Richardson and Ruby propose the following design method to obtain a resource-oriented architecture

The ROA procedure

- Figure out the data set
- Split the data set into resources
- For each kind of resource
  - Expose a subset of the uniform interface
  - Design the representation(s) accepted from the client
  - Design the representation(s) served to the client
  - Integrate this resource into existing resources, using hypermedia links
  - Consider the typical course of events: what's supposed to happen?
  - Consider error conditions: what might go wrong?

The example scenario

- Example taken from Richardson and Ruby *RESTful web services.*

- Suppose we want to construct an (imaginary) web service that serves maps of different kinds:
  - Political maps
  - Physical maps
  - Road maps
  - Geological maps

  of different planets and at different scales

Step 1: Figure out the data set

- Maps are made out of *points* (specific longitude & latitude)
- A map concerns a certain *planet* (Earth, Venus)
- Some points on a map are *places* (Brussels, the Himalaya, …)
- Places can be of different *types*

Step 2: split the data into resources

- Web services commonly expose three kinds of resources:
  - Predefined resources for especially important aspects of the application
    - (e.g., top-level directory of other available resources)
  - A resource for every object exposed through the service
  - Resources representing the results of algorithms applied to the data set
    - (e.g., a search resource http://google.com/search?q=books)

Step 2: our resources so far

- For planets and entire maps:
  - The list of planets
  - Mars
  - Earth
  - The satellite map of Mars
  - The radar map of Venus
  - The road map of Earth

Step 2: our resources so far

- For parts of maps
  - 24.9195N 17.821E on Earth
  - 24.9195N 17.821E on the political map of Earth
  - 24.9195N 17.821E on Mars
  - 44N 0W on the geologic map of Earth

- For places:
  - The Cleopatra crater on Venus
  - Campus Diepenbeek of Hasselt University in Diepenbeek, Belgium on Earth
  - The place called Springfield in Massachusetts, in the United States of America, on Earth

Step 2: our resources so far

- Algorithmic resources
  - Places on Earth called Springfield
  - Container ships on Earth
  - Craters on Mars more than 1 km in diameter
  - Places on the moon named before 1900

Step 2: in conclusion, our resources:

1. The list of planets
2. A place on a planet—possibly the entire planet—identified by name
3. A geographic point on a planet, identified by latitude and longitude
4. A list of places on a planet that match some search criteria
5. A map of a planet, centered around a particular point

Step 3: name the resources

Some guidelines, born of collective experience:

- Use path variables to encode hierarchy:

  ```
  /parent/child
  ```

- Put punctuation characters in path variables to avoid implying hierarchy where none exists:

  ```
  /parent/child1;child2
  ```

- Use query variables to imply inputs into an algorithm:

  ```
  /search?q=jellyfish&start=20
  ```

Step 3: name the resources

1. The list of planets at the root URI
   http://maps.example.com

2. A place on a planet, possibly the entire planet, identified by name
   http://maps.example.com/Venus
   http://maps.example.com/Earth/France/Paris

3. A geographic point on a planet, identified by latitude and longitude
   http://maps.example.com/Earth/24.9195,17.821

4. A list of places on a planet that match some search criteria
   http://maps.example.com/Earth?show=Springfield
   http://maps.example.com/Mars?show=craters+bigger+than+1km

5. A particular map of a planet, centered around a particular point
   http://maps.example.com/geographic/Venus/24.5,17.2

Step 4: design representations

- An XML representation of the list of planets

```
<?xml version="1.0"?>
<planets>
<planet href="http://maps.example.com/Earth" name="Earth" />
<planet href="http://maps.example.com/Venus" name="Venus" />
...
</planets>
```

- Note: Richardson and Ruby choose XHTML in their examples (using the “class” attribute to denote semantics), but this is inferior to choosing XML.

Step 4: design representations

- An XML representation of a given planet

```
<?xml version="1.0"?>
<planet>
 <name> Earth </name>
 <maps>
   <map href="/road/Earth"> Road </map>
   <map href="/satellite/Earth">Satellite</map>
    …
 </maps>
 <description> Third planet from the So … </description>
</planet>
```

Step 4: design representations

- An XML representation of a point on a map

```
<?xml version="1.0"?>
<point>
  <coordinate> 37.0,-95</coordinate>
  <tile src="/road/Earth/images/37.0,-95.png"/>
  <nav direction="north" href="46.0518,-95.8"/>
  <nav direction="northeast" href="46.0518,-89.7698"/>
  <nav direction="south" href="36.4642,-84.5187"/>
  <nav direction="southeast" href="32.3513,-95.8"/>
  …
  <zoom direction="in" href="…" />
  <zoom direction="out" href="…/>
</point>
```

Step 4: design representations

- An XML representation of results of search “list of places called springfield in the US”

```
<?xml version="1.0"?>
<result>
  <description> … </description>
  <places>
     <place href=“/Earth/USA/IL/Springfield”>Springfield, IL</place>
     <place href=“/Earth/USA/MA/Springfield”>Springfield, MA</place>
     …
  </places>
</result>
```

Expose a subset of the Uniform Interface

- Let's say that the service is read-only for now
  - We only provide GET and HEAD

- What is supposed to happen? What can go wrong
  - GET:
    - 200 OK if resource exists + representation
    - 404 Not Found if resource does not exist …
    - …or 303 See other if we think we have an alternate solution
  - HEAD: same, but no representation

What about read&write resources?

- Let’s say we want to allow users to annotate our maps with Points of Interests (POI’s), which are just places.

Our first R/W resource: user accounts

- User accounts are typically created through a form that has to be filled in by humans.
- However, there are cases where the ability to create user accounts through web services is desirable.
  - E.g., consider “smart” GPS devices with built-in annotation capabilities. Each GPS device can automatically register a user account for itself.

Design of R/W user accounts (1/8)

- Figure out the data set:
  - Account = user name; password
- Split the data set into resources:
  - Each user account becomes a resource
- Name the resources with URI's

```
https://maps.example.com/user/{user-name}
```

Design of R/W user accounts (2/8)

```
https://maps.example.com/user/{user-name}
```

- Expose a subset of the uniform interface
  - *Will clients be creating new resources of this type?*
    YES
  - *When the client creates a new resource of this type, who's in charge of determining the new resource's URI? Is it the client or the server?*
    THE CLIENT

Design of R/W user accounts (3/8)

```
https://maps.example.com/user/{user-name}
```

- Expose a subset of the uniform interface
  - *Will clients be modifying resources of this type?*
    YES (they should be able to change their password)
  - *Will clients be deleting resources of this type?*
    YES (delete an account)
  - *Will clients be fetching representations of resources of this type?*
    DEBATABLE (but let's say yes)

Design of R/W user accounts (4/8)

```
https://maps.example.com/user/{user-name}
```

- Expose a subset of the uniform interface:

REST	INCLUDE?	
POST	NO	*Not supported because the client determines the resource URI*
GET	YES	Retrieve a summary of the account
PUT	YES	Create user/modify his password.
DELETE	YES	Close account
OPTIONS	YES	Discover what HTTP methods are supported by the resource
HEAD	YES	requests headers

- **Note:** all of these methods must provide the minimal security (authentication & authorization)

Design of R/W user accounts (4/8)

```
https://maps.example.com/user/{user-name}
```

- Design representations accepted from client/sent to client

REST	INCLUDE?	REPRESENTATION
POST	NO	*Not supported because the client determines the resource URI*
GET	YES	application/xml
PUT	YES	application/x-www-form-urlencoded
DELETE	YES	*none*
OPTIONS	YES	*none*
HEAD	YES	*none*

Design of R/W user accounts (5/8)

- An XML representation of

```
GET https://maps.example.com/user/svsummer
```

(assuming this user is authenticated as svsummer)

```
<?xml version="1.0"?>
<user>
 <name> Stijn Vansummeren </name>
 <homepage> https://maps.exampl... </homepage>
 <modify xmlns:hthml="http:/…">
  <html:form id="modifyUser" method="put" action="…/users/svsummer">
            <html:input class="password" name="password" /><br />
            <html:input class="submit" /></p>
  </html:form>
</modify>
</user>
```

Design of R/W user accounts (6/8)

- An XML representation of

```
GET https://maps.example.com/user/svsummer
```

(assuming this user is not authenticated as svsummer)

```
<?xml version="1.0"?>
<user>
 <name> Stijn Vansummeren </name>
 <homepage> https://maps.exampl... </homepage>
</user>
```

Design of R/W user accounts (7/8)

- An application/x-www-form-urlencoded representation sent to

```
PUT https://maps.example.com/user/svsummer
```

(assuming this user is authenticated as svsummer or the user does not yet exist):

```
password=ThisIsNotAPassword
```

- *(application/x-www-form-urlencoded just means a list of key=value pairs, separated by &, where special characters in keys and values are escaped)*

Design of R/W user accounts (8/8)

- Link this resource to other resources
  - E.g., put a link or form to create users at the service URI http://maps.example.com

- What can go wrong?
  - PUT request with wrong media type: return 415 (“Unsupported Media Type”).
  - PUT request without media type: 400 (“Bad Request”)
  - PUT request without “password” key: 400 (“Bad Request”)
  - …

What about read&write resources?

- See other examples in handouts available on website

DISCUSSION

Discussion

- REST = synchronous
- Asynchronous can be crafted on top by splitting asynchronous requests into multiple synchronous requests, but this essentially defeats the uniform interface:
  - POST to a URL to submit a new asynchronous request
  - Server replies with 202 “Created” + URI at which the status of the request can be queried (GET) or by which the request can be deleted (DELETE)
  - [Note that asynchronous operations hence cannot be GETten because they create new suboperations]

Discussion

- Lots of web services claim RESTfullness, but actually overloaded POST:
  - They send data to a particular process to call a particular function = RPC call
  - If possible, restrict POST to the use of factory URI's!

References

- R. T. Fielding and R. N. Taylor. *Principled design of the modern Web architecture*. ACM Transactions on Internet Technology (TOIT), 2(2), May 2002, pp. 115-150.
- L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, 2007.
- R. Daignau, Service Design Patterns, Addisson-Wesley, 2011.