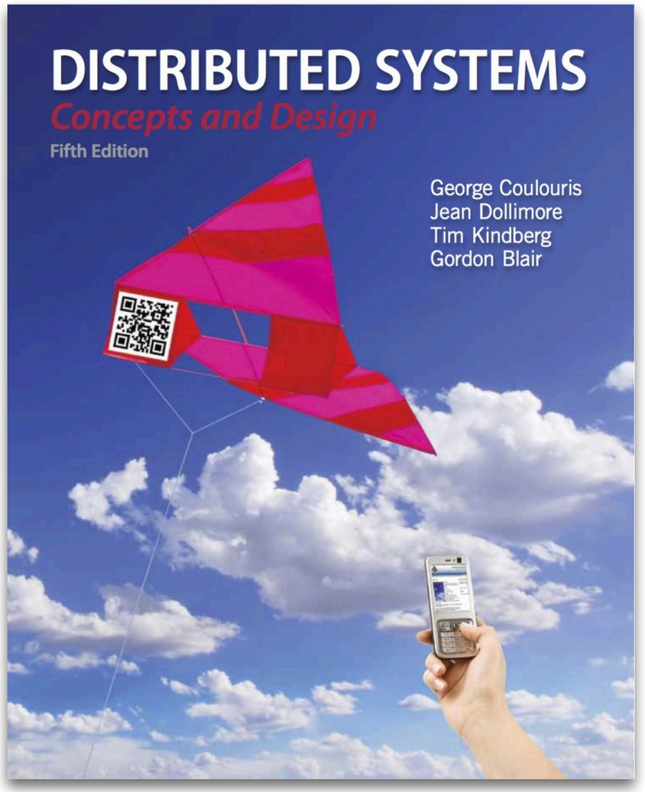
Distributed Systems 2023-2024: Web Services

Wouter Joosen & Tom Van Cutsem
DistriNet KU Leuven
October 2023

KU LEUVEN DistriNet

Learning resources

- For background reading, see Textbook:
  - Chapter 1, section 1.6 (case study: WWW)
  - Chapter 5: Remote Invocation
    - section 5.2 (HTTP protocol basics)
  - Chapter 9: Web Services
    - 9.1 Introduction
    - 9.2 Web Services (intro only, up to and excluding 9.2.1 and later)
    - 9.7.1 Service-oriented architecture



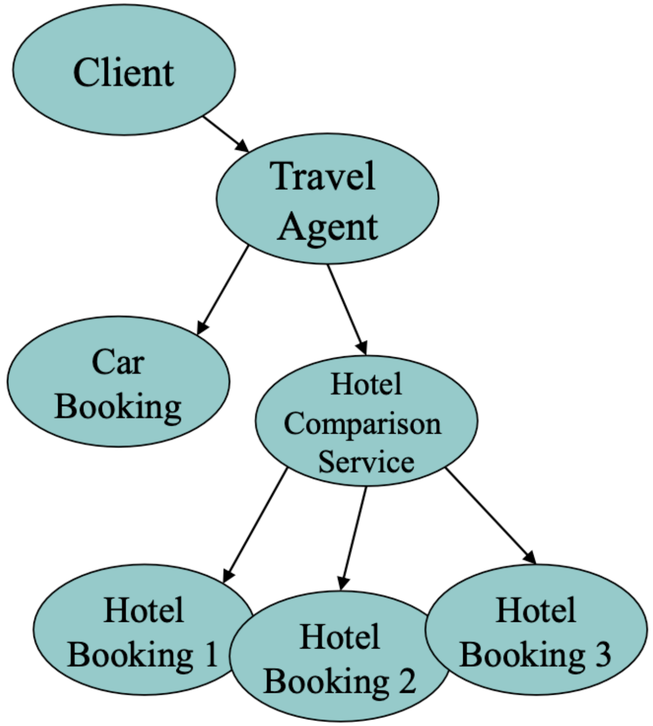
From distributed objects to Web services

Recap: Distributed objects and remote method invocation

- Object-oriented software
  - Original idea: model software according to real-world entities
  - Common definition: objects encapsulate state and expose behavior (methods)
  - Interface: set of allowed operations (method signatures)
  - Pass-by-reference (pointer semantics) vs pass-by-value (copy semantics)
- Distributed objects: extend the object-oriented programming paradigm across the network (e.g. Java RMI, CORBA)
  - Remote objects (represented by local stubs or “proxy” objects)
  - Remote method invocations
  - Remote interfaces (method signatures + remote exceptions)
  - Pass-by-reference (pass a remote reference) vs pass-by-copy (“serialize” and then “deserialise” the object to get a copy)

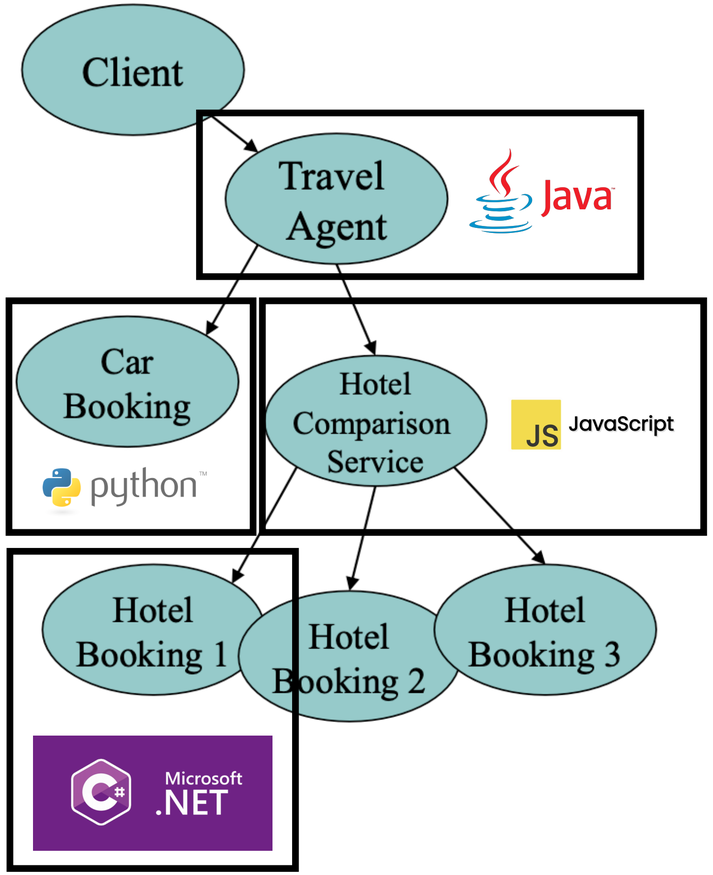
The trouble with distributed objects: a lack of interoperability

- Distributed object systems assume that all components in the system follow the object paradigm, speak the same RMI protocol or use the same language-specific byte-serialized object format
- Example: a travel agency booking service
  - Must interact with many subsystems (e.g. car & hotel bookings)
  - These subsystems are likely **distributed across different organizations** that use different software platforms (operating systems, middleware, programming languages & runtimes)



The trouble with distributed objects: a lack of interoperability

- Distributed object systems assume that all components in the system follow the object paradigm, speak the same RMI protocol or use the same language-specific byte-serialized object format
- Example: a travel agency booking service
  - Must interact with many subsystems (e.g. car & hotel bookings)
  - These subsystems are likely **distributed across different organizations** that use different software platforms (operating systems, middleware, programming languages & runtimes)
  - E.g. the travel agent runs Java on JVM, the hotel comparison service runs JavaScript on node.js, a hotel booking system runs C# on .NET and the car booking service runs Python.



From Objects to Web Services

- In the early 2000s the Web emerged as the **universally adopted information system** with consequently universal adoption of its standards: URLs for naming resources, HTTP for transport, HTML and later XML and JSON for data interchange

- This led to a move away from “distributed objects” towards “web services”:

  - **Naming**: from looking up names of objects in a registry (e.g. Java RMI Registry) to resolving URLs to resources. **Reuse** the **DNS** domain name infrastructure instead of maintaining custom registries.

  - **Transport protocol**: from custom-designed request-reply protocols over TCP/IP (e.g. Java RMI’s “Java Remote Method Protocol”) to standard HTTP over TCP/IP. **Firewalls** typically blocked the ports used by these non-standard protocols, but rarely blocked HTTP port 80.

  - **Data interchange**: from custom-designed *binary* serialization formats (e.g. Java Object Serialization) to standardized *text-based* serialization formats (e.g. XML). While text-based formats generate larger messages and are more expensive to parse, they **simplified development** (testing and debugging)

The Web: URLs + HTTP
+ HTML (for humans)
+ XML (for machines)

Distributed objects	Web services
Custom object registries	DNS and URLs
Custom remote invocation protocols	SOAP over HTTP (or just HTTP in the case of REST)
Custom binary serialization formats	Textual encoding: XML (and later JSON)

Service-oriented architecture (SOA)

- The adoption of standard Web protocols **decoupled** services from their platform implementations (operating systems, middleware, programming languages, …), thus greatly improving service **interoperability**

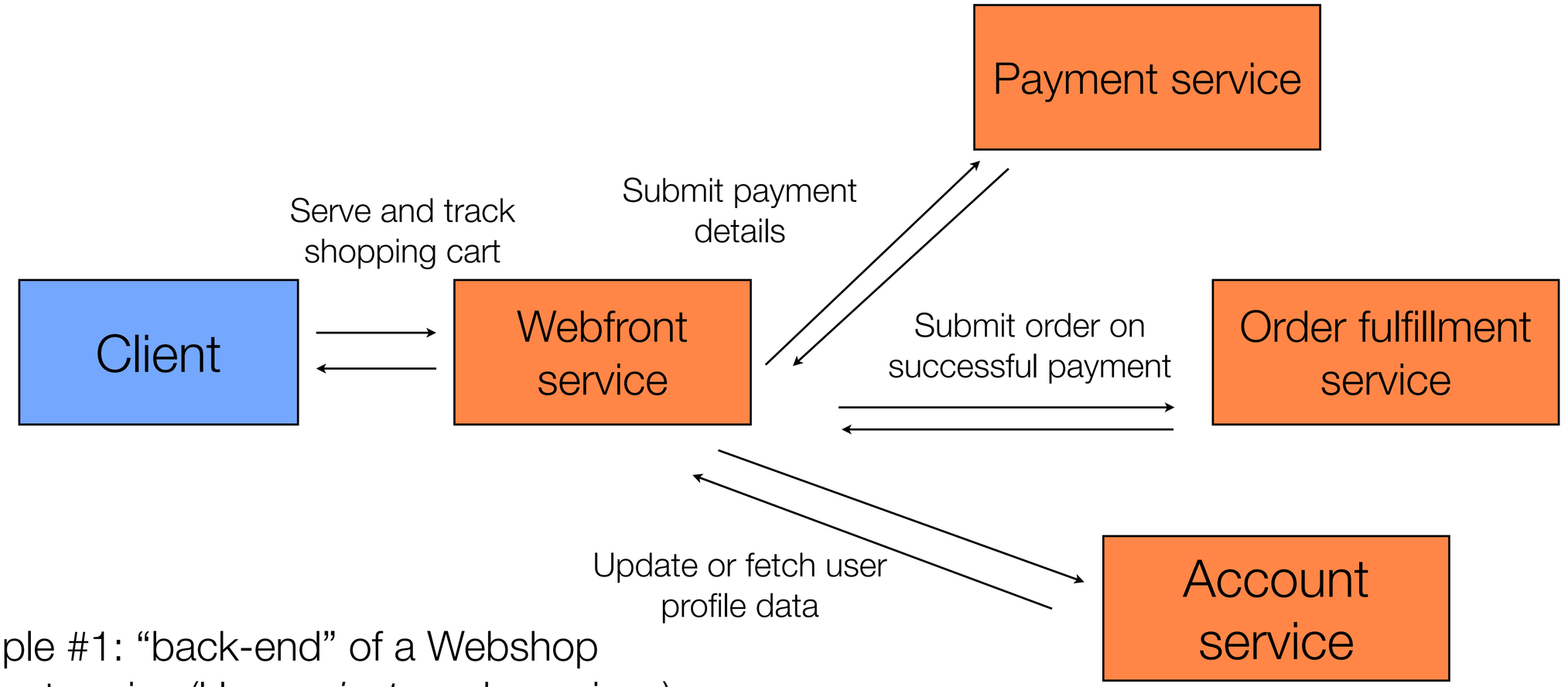
- Software **architectures** moved away from “monolithic” architectures with all parts of a service written for the same platform (often even compiled in a single executable) to “service-oriented” architectures: applications decomposed into **independently deployed and managed** services, with well-documented APIs and data interchange formats.

- Historically, this shift coincided with the shift from Objects to Web services, but **SOA** is a more **general** and independent **concept**. A service-oriented architecture can in principle also be built using language-agnostic RPC protocols (e.g. a popular approach today is to build services using gRPC)

- The adoption of Linux **containers** around 2013-2015 made it easier to deploy and manage many independent services. As a result, applications were further decomposed into even more services, with each service focusing on an even smaller part of the application (a “**microservices**” architecture)

Service-oriented architecture (SOA)

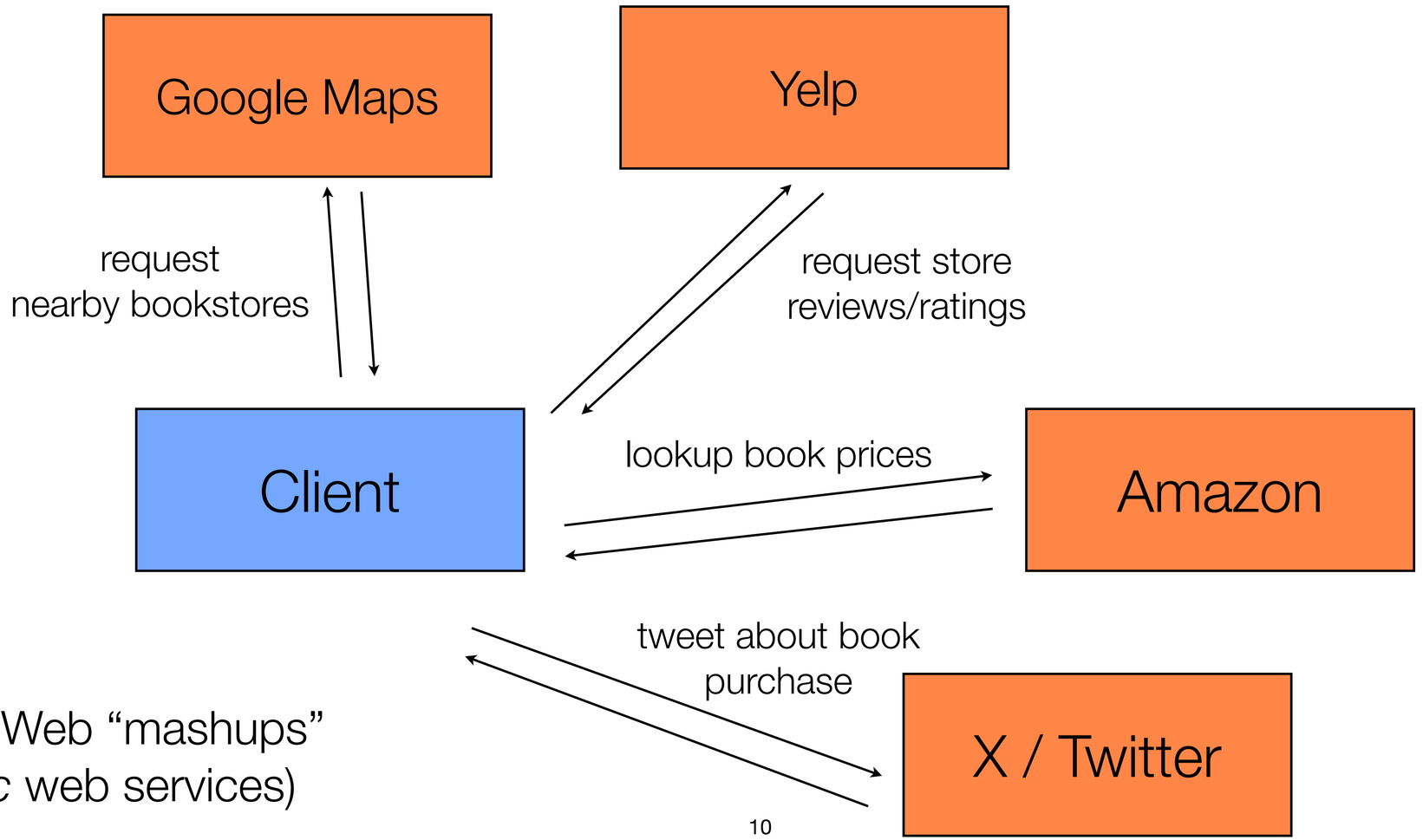
- Decompose applications into ensembles of loosely coupled (often even independently deployed) services



Example #1: “back-end” of a Webshop in a large enterprise (Uses *private* web services)

Service-oriented architecture (SOA)

- Decompose applications into ensembles of loosely coupled (often even independently deployed) services



Example #2: Web “mashups”
(Uses *public* web services)

Back to Web Services

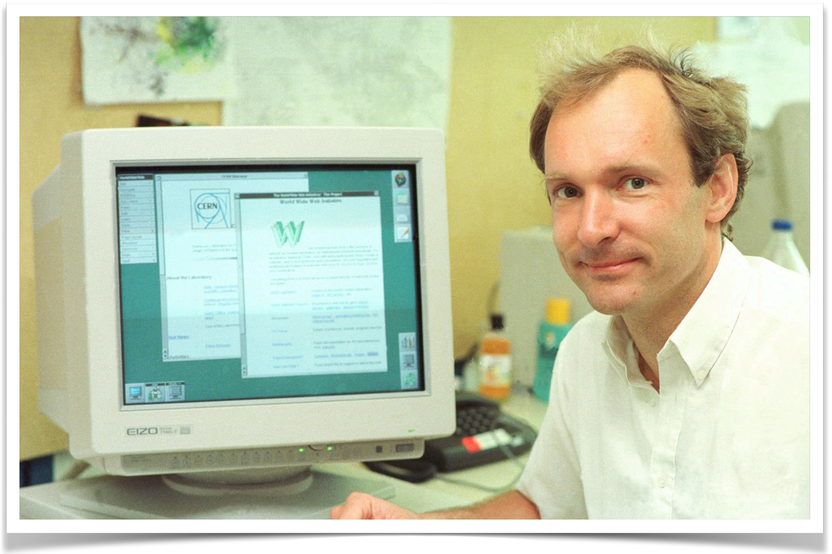
- Two broad approaches developed: first **SOAP**-based and later **REST**-based web services

- Before discussing each in more detail, let us first review the core principles and standard protocols of the Web, on which both approaches are based.

The Web (aka the **W**orld **W**ide **W**eb)

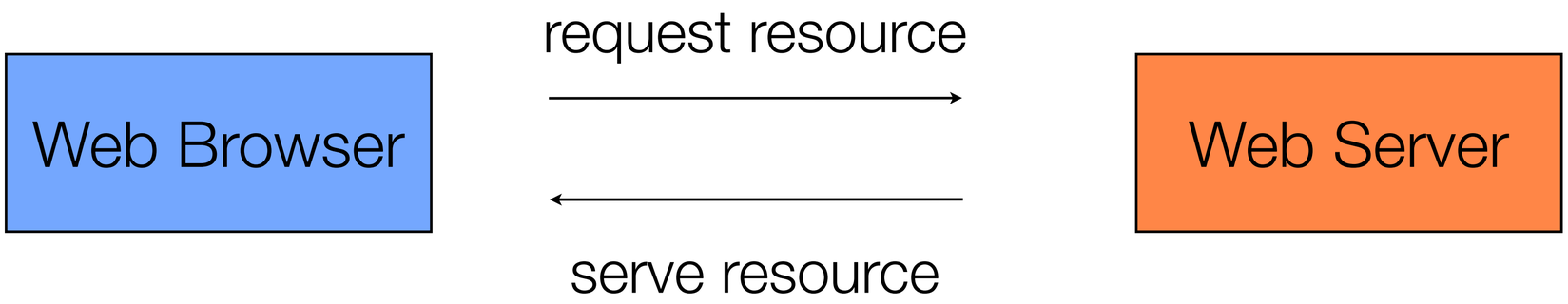
The Web (aka the **W**orld **W**ide **W**eb)

- Tim Berners-Lee, CERN, 1989
- Distributed information system for publishing and accessing resources and services across the Internet
- Key feature: hyperlinked documents
- Easy to share and *link to* resources by name via their URL



The Web: system architecture

- Client-server architecture
- User Agent (Browser)
- Web Server



The Web: key technologies

- Hypertext Markup Language (HTML)
  - Specify content and layout of web pages, rendered by browsers
- Uniform Resource Locators (URLs)
  - Identify documents and other resources on the web
- Hypertext Transfer Protocol (HTTP)
  - Client/Server Request/Reply protocol to fetch resources

URLs

- A protocol + a protocol-specific identifier. Most general form:

```
scheme : scheme-specific-identifier
```

- Most commonly, HTTP or HTTPS URLs:

```
http:// servername [:port] [/pathName] [?query] [#fragment]
```

- Port defaults to 80
- Examples:

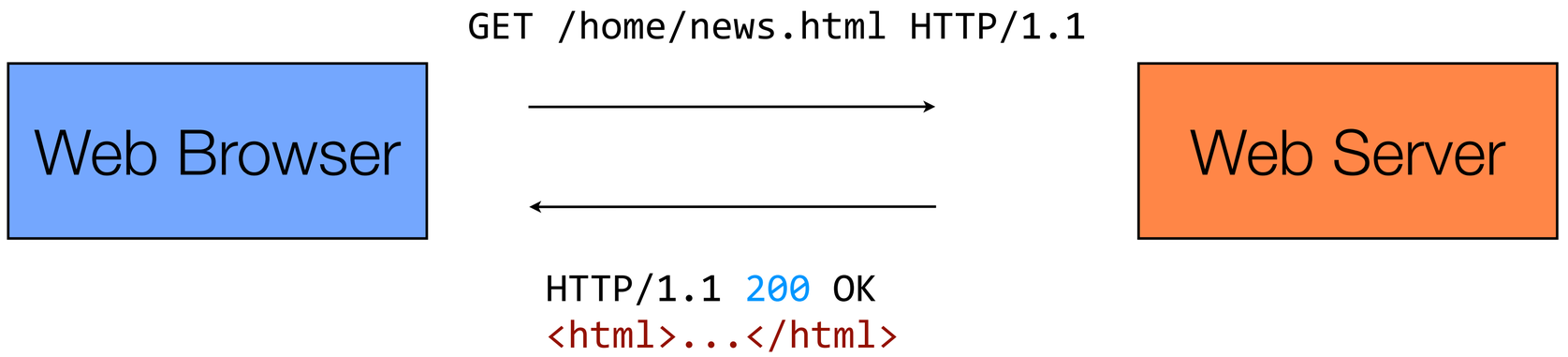
```
http://www.google.com/search?q=distributed+systems
https://en.wikipedia.org/wiki/Distributed_computing#Introduction
```

Hypertext Transfer Protocol (HTTP)

- Application-layer (not network-layer) **request-response** protocol
- HTTP defines a fixed set of operations (called **methods**) that can be performed on a **resource**
- Most common methods:
  - `GET`, to retrieve data from a resource
  - `POST`, to provide data to a resource
- Less frequently used: `HEAD`, `PUT`, `DELETE`, `...`

HTTP GET

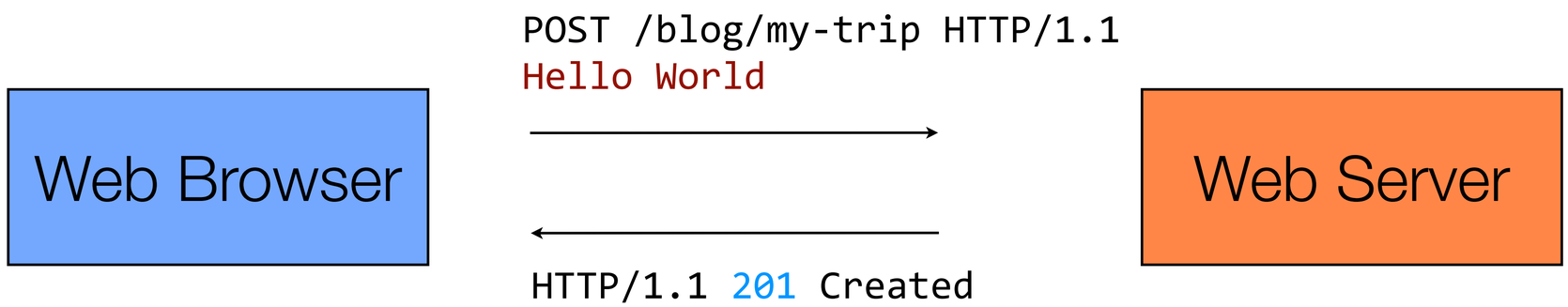
- GET request specifies path to resource
- Server responds with status code and optional response body



- GET requests should be **idempotent**: it should be safe to send the request multiple times without affecting the state of the resource on the server

HTTP POST

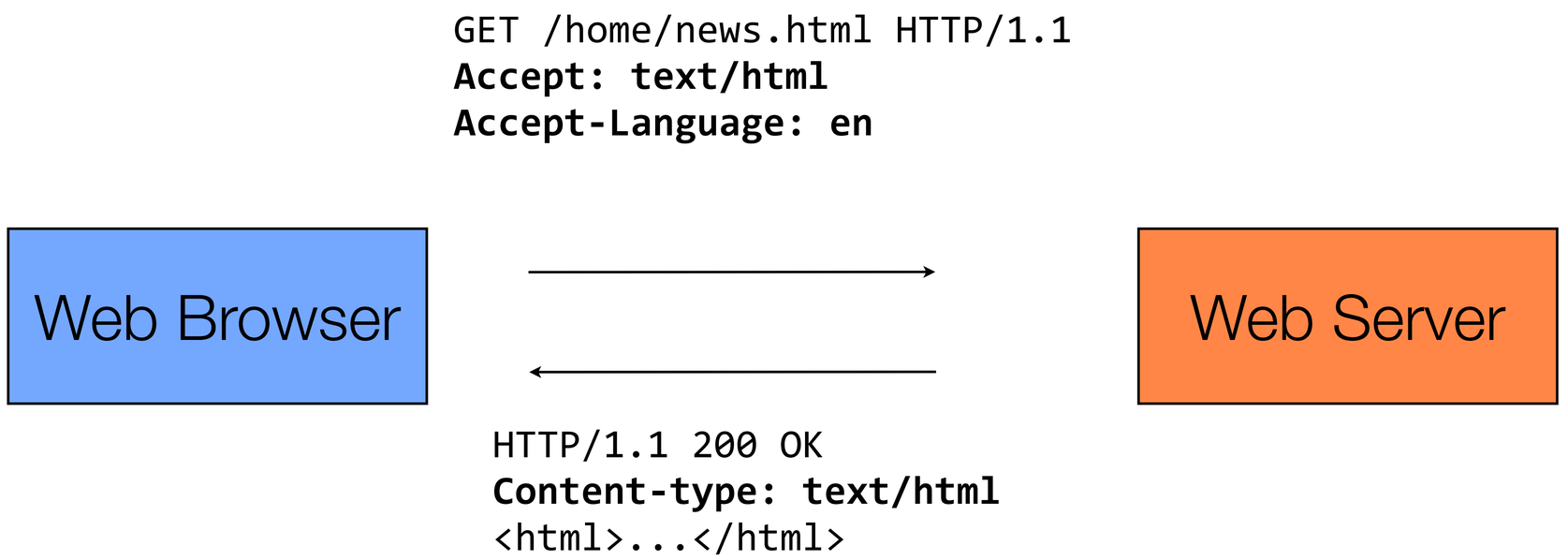
- POST request specifies path to resource, and optional message body
- Server responds with status code



- POST is **not idempotent**: unsafe to send the request multiple times
  - User Agent (Browser) must ask user what to do with resubmissions

HTTP: headers

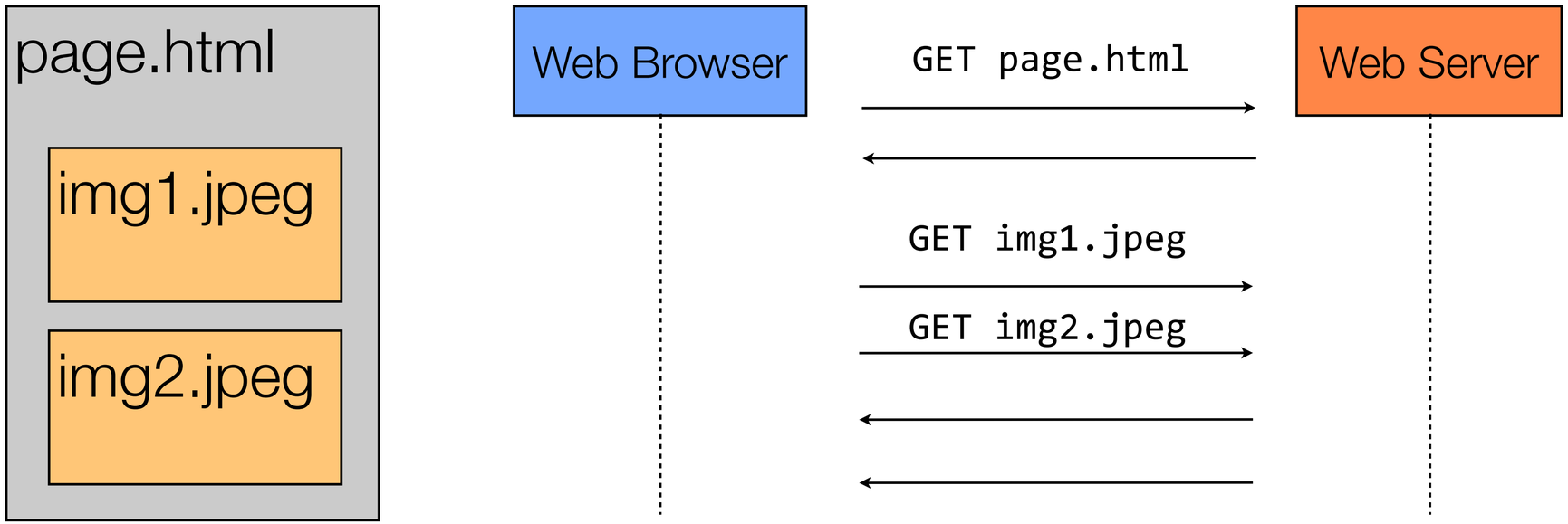
- Both request and response messages may carry **headers** with meta-data



- For example: browser can use a header to specify what content types it can handle, so the web server can customise the response. This is called **“content negotiation”**

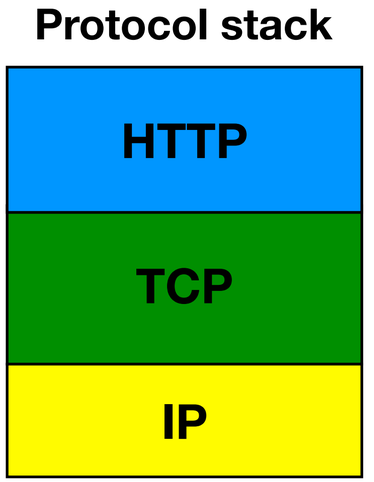
HTTP: one resource per request

- Clients specify one resource per HTTP request.
- If a web page contains 5 images, browser will issue five separate requests, one per image



HTTP and TCP/IP

- HTTP requests are most commonly layered on top of TCP/IP
- Browser opens a TCP/IP socket, connects to server
- HTTP 1.0: connection closed after a single request/response
- HTTP 1.1 introduced *persistent connections*
  - Reuse same TCP/IP connection for multiple requests
  - Less setup/tear-down overhead



HTTP is a **stateless** protocol

- Web server does not maintain client state in between different HTTP requests
  - Every HTTP request is entirely independent
  - Outcome does not depend on previous HTTP requests
- HTTP 1.1 persistent connections are just an implementation-level optimization.
  - The multiple HTTP requests that share the same TCP/IP connection are still logically independent.

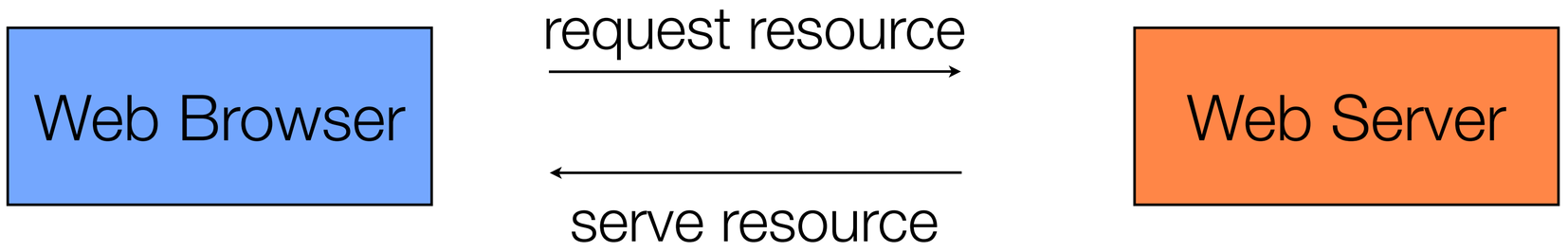
HTTP is a **stateless** protocol

- Yet often an interaction between client and server is stateful.
  - E.g. rendering shopping list in an online store depends on previously added items.
  - Requires notion of a **session** that groups together multiple web requests
- Common work-arounds:
  - Session id **tokens** passed in query string parameters (e.g. `/index.php?session_id=0AxfKaZ54Baq45D`) or in HTTP headers. The session id is passed back and forth in every request/response
  - **Cookies**: small text files stored on the client, automatically appended by the browser as a `Cookie:` HTTP header to every client request

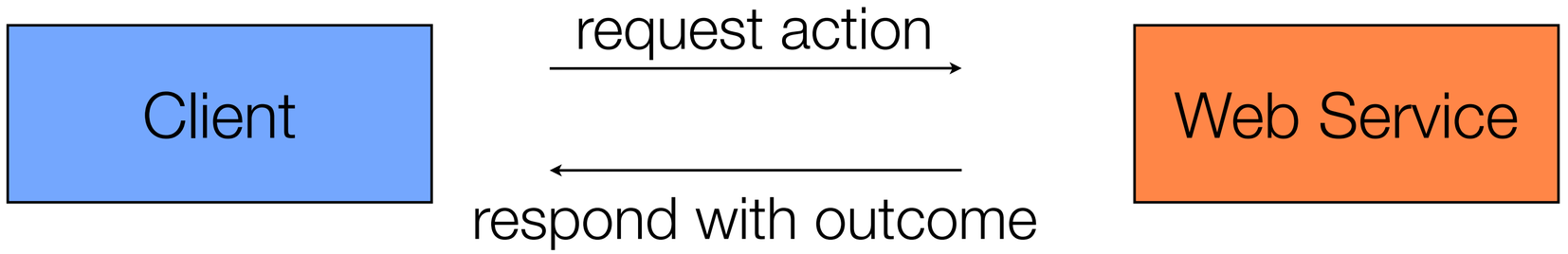
From Web Pages to Web Services

Web Services

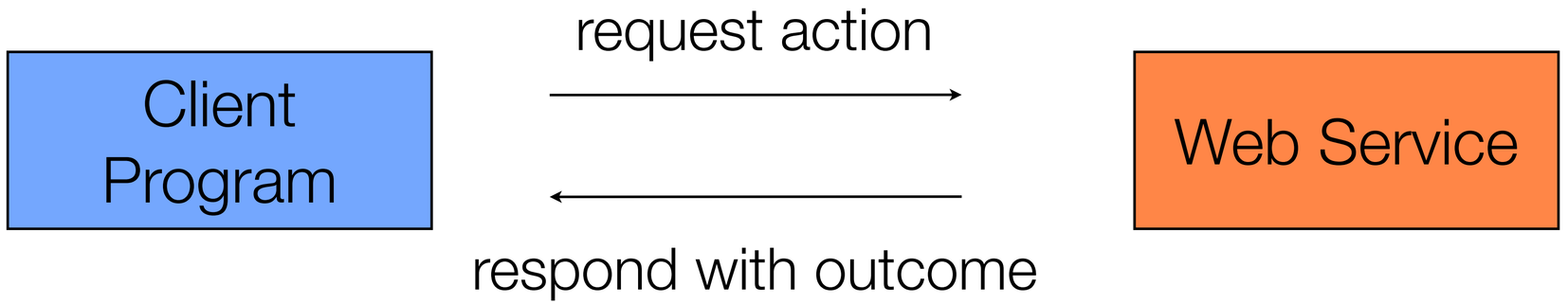
- Web browser is the dominant model for user interaction on the Internet (**human-machine** interaction)



- Many cases where the client is not a human/browser but another program (**machine-to-machine** interaction)

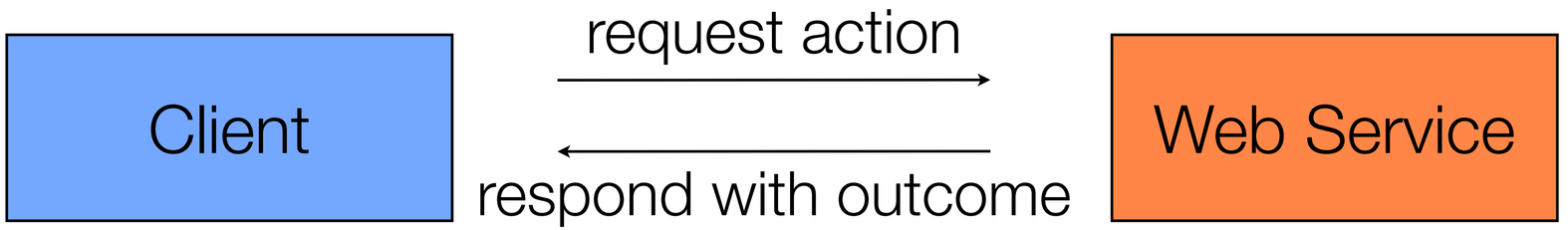


Web Services



- HTML is inadequate for programmatic access to data
- HTML is not extensible. Designed for markup (UI) only.
- Web Services thus use a more general structured data format, often text-based: XML (eXtensible Markup Language) and JSON (Javascript Object Notation)

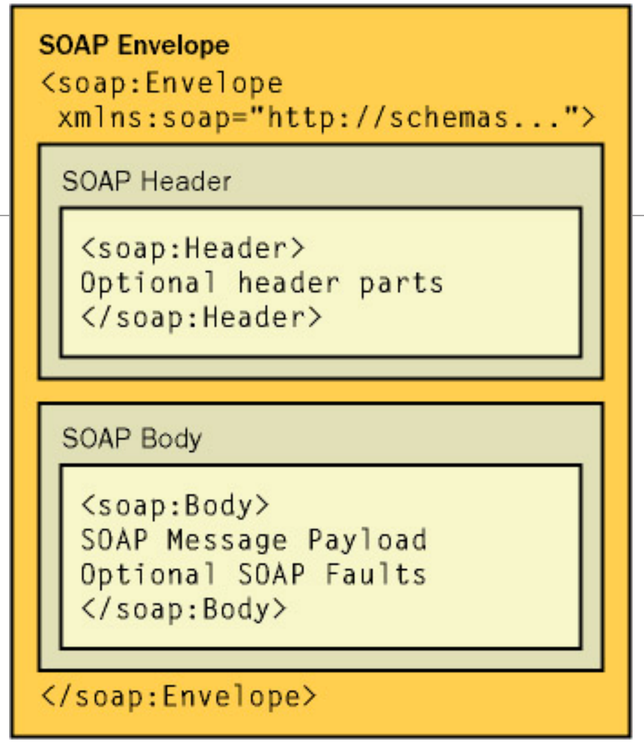
Web Services



- In principle, clients could reuse the HTTP protocol directly to make requests to web services
- But HTTP was designed to fetch resources, not to invoke operations (actions)
- Hence, some web services layer an RPC protocol on top of HTTP, such as SOAP

SOAP-based Web Services

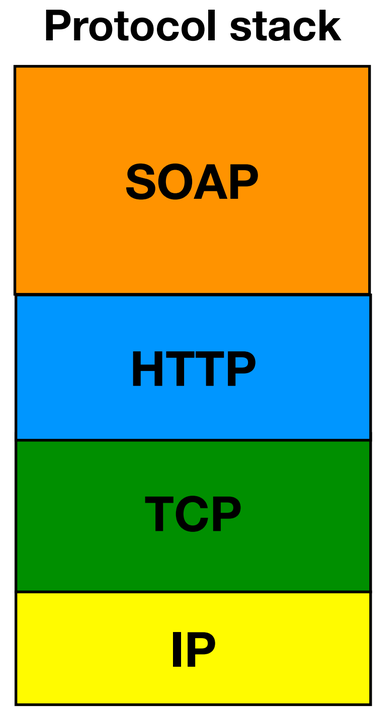
- SOAP = Simple Object Access Protocol
- An XML-based RPC protocol, typically transported over HTTP.
- SOAP messages are wrapped in a SOAP “envelope”
  - Headers (for message meta-data)
  - Body (XML-formatted document, can encode request or reply messages)
- Later complemented with an IDL to describe the interface of the Web Service in “Web Services Description Language” (WSDL) (also an XML-based format)
- XML = eXtensible Markup Language. Uses “namespaces” to try and make document tags globally unique.




```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock">
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

SOAP-based Web Services

- Huge support from IT industry (Microsoft, IBM, …) when originally introduced in 1998-2000
- SOAP-based services layer an extra request-reply RPC protocol on top of HTTP which is already a request-reply protocol. This creates added complexity without significant benefits.
- This added **complexity**, along with the **verbosity** of XML, caused SOAP and XML to be abandoned in favour of simpler REST-based Web Services and JSON.
- For example, Google dropped SOAP support in its Web service APIs in 2006. Amazon Web Services (AWS) deprecated its SOAP API for EC2 in 2015.



Side-note: JavaScript Object Notation (JSON)

- Lightweight data interchange format (“the fat-free alternative to XML”)
- Text-based. Easy to parse.
- Originally based on JavaScript’s syntax for object and array “literals”
- But JSON itself is language-neutral: almost every programming language today has libraries (often built-in) to parse and generate JSON text strings
- JSON values are easily converted to and from a programming language’s built-in data types (e.g. objects <=> dictionaries/maps, arrays <=> lists/vectors)

object
  {}
  { *members* **}**

members
  pair
  pair **,** *members*

pair
  string **:** *value*

array
  []
  [*elements* **]**

elements
  value
  value **,** *elements*

value
  string
  number
  object
  array
  true
  false
  null

JSON BNF grammar

Web services introduction: recap

- **The trouble with distributed object systems**: a **lack of interoperability.** Assume all components speak the same language and/or remote invocation protocol and/or use the same IDL. Examples: Java RMI, CORBA
- The Web emerged as a **universal information system** with near-universal adoption of its standards: URLs for naming resources, HTTP for transport, HTML and later XML and JSON for data interchange
- **Web Basics**: client-server architecture, request-response interaction, HTTP and content-types, stateless request processing, text-based data interchange formats (HTML, XML and JSON)
- **Web Services**:
  - From web servers (content-centric, human-readable) to web services (application-centric, machine-readable)
  - First generation: SOAP over HTTP (for request-reply transport), WSDL (for interface definition), XML (for data interchange) => new set of standards => added complexity without significant benefits. *operation*-oriented APIs.
  - Second generation: RESTful web services => reversal to core Web protocols and *resource*-oriented APIs (see next).
- **Service-oriented architecture (SOA)**: an architectural style where applications are **decomposed** into a collection of services that can be independently deployed, executed and maintained, with well-documented APIs and data interchange formats. SOA does not mandate the use of Web services, but it is common to implement services in a SOA as Web services.

Representational State Transfer (REST)