You can jump to last two pages! (Skip history and long explanations)

Distributed systems and messaging exists from 1970:s (e.g. EDI). In 1970-1980:s in Finland too, e.g. between banks or main logistics streams, big money businesses: OVT)

What Are Web Services?

Source: Java EE 6 Tutorial, Oracle corporation, 2013
https://docs.oracle.com/javaee/6/tutorial/doc/javaeetutorial6.pdf
pages 363-365:

Web services are client and server applications that communicate over the World Wide Web's (WWW) Hyper Text Transfer Protocol (HTTP). As described by the World Wide Web Consortium (W3C), web services provide a standard means of interoperating between software applications running on a variety of platforms and frameworks. Web services are characterized by their great interoperability and extensibility, as well as their machine-processable descriptions, thanks to the use of XML. Web services can be combined in a loosely coupled way to achieve complex operations. Programs providing simple services can interact with each other to deliver sophisticated added-value services.

Types of Web Services

On the conceptual level, a service is a software component provided through a network-accessible endpoint. The service consumer and provider use messages to exchange invocation request and response information in the form of self-containing documents that make very few assumptions about the technological capabilities of the receiver.

On a technical level, web services can be implemented in various ways. The two types of web services discussed in this section can be distinguished as "big" web services and "RESTful" web services.

"Big" Web Services

Big web services use XML messages that follow the Simple Object Access Protocol (SOAP) standard (1998), an XML language defining a message architecture and message formats. Such systems often contain a machine-readable description of the operations offered by the service, written in the Web Services Description Language (WSDL), an XML language for defining interfaces syntactically. The SOAP message format and the WSDL interface definition language have gained widespread adoption. Many development tools, such as NetBeans IDE, can reduce the complexity of developing web service applications.

A SOAP-based design must include the following elements.

- A formal contract must be established to describe the interface that the web service offers. WSDL can be used to describe the details of the contract, which may include messages, operations, bindings, and the location of the web service. You may also process SOAP messages in a JAX-WS service without publishing a WSDL.

- The architecture must address complex nonfunctional requirements. Many web service specifications address such requirements and establish a common vocabulary for them. Examples include transactions, security, addressing, trust, coordination, and so on.

- The architecture needs to handle asynchronous processing and invocation. In such cases, the infrastructure provided by standards, such as Web Services Reliable Messaging (WSRM), and APIs, such as JAX-WS, with their client-side asynchronous invocation support, can be leveraged out of the box.

Source: Wikipedia *"Enterprise Service Bus"* *A bit extra and old topic*

https://en.wikipedia.org/wiki/Enterprise_service_bus
https://en.wikipedia.org/wiki/Enterprise_service_bus#ESB_as_software

Enterprise Service Bus (ESB) Products

(Built for SOA, but also could be used for RESTful Web Services too)

A more complete overview can also be found in the comparison of business integration software.

- Commercial
  - IBM WebSphere Message Broker Integration Bus
  - IBM WebSphere ESB
  - InterSystems Ensemble
  - Information Builders iWay Service Manager
  - Microsoft BizTalk Server
  - Mule ESB
  - Oracle Enterprise Service Bus
  - Progress Software Sonic ESB (acquired by Trilogy)
  - SAP Process Integration
  - Talend enterprise ESB
  - TIBCO Software ActiveMatrix BusinessWorks
  - webMethods enterprise service bus (acquired by Software AG)
  - Windows Azure Service Bus
  - Sonic ESB from Aurea

- Open-source software
  - Apache Camel
  - Apache ServiceMix
  - Apache Synapse
  - Fuse ESB from Red Hat
  - JBoss ESB
  - NetKernel
  - Open ESB
  - Petals ESB
  - Spring Integration
  - Talend Open Studio for ESB
  - UltraESB
  - WSO2 ESB

Microservices (2005-2007-2011-2012)

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

Source: same Java EE 6 tutorial

RESTful Web Services

In Java EE 6, JAX-RS provides the functionality for Representational State Transfer (RESTful) web services. REST is well suited for basic, ad hoc integration scenarios. RESTful web services, often better integrated with HTTP than SOAP-based services are, do not require XML messages or WSDL service–API definitions. Project Jersey is the production-ready reference implementation for the JAX-RS specification. Jersey implements support for the annotations defined in the JAX-RS specification, making it easy for developers to build RESTful web services with Java and the Java Virtual Machine (JVM).

Because RESTful web services use existing well-known W3C and Internet Engineering Task Force (IETF) standards (HTTP, XML, URI, MIME) and have a lightweight infrastructure that allows services to be built with minimal tooling, developing RESTful web services is inexpensive and thus has a very low barrier for adoption. You can use a development tool such as NetBeans IDE to further reduce the complexity of developing RESTful web services.

A RESTful design may be appropriate when the following conditions are met:

- The web services are completely stateless. A good test is to consider whether the interaction can survive a restart of the server.

- A caching infrastructure can be leveraged for performance. If the data that the web service returns is not dynamically generated and can be cached, the caching infrastructure that web servers and other intermediaries inherently provide can be leveraged to improve performance. However, the developer must take care because such caches are limited to the HTTP GET method for most servers.

- The service producer and service consumer have a mutual understanding of the context and content being passed along. Because there is no formal way to describe the web services interface, both parties must agree out of band on the schemas that describe the data being exchanged and on ways to process it meaningfully. In the real world, most commercial applications that expose services as RESTful implementations also distribute so-called value-added toolkits that describe the interfaces to developers in popular programming languages.

- Bandwidth is particularly important and needs to be limited. REST is particularly useful for limited-profile devices, such as PDAs and mobile phones, for which the overhead of headers and additional layers of SOAP elements on the XML payload must be restricted.

- Web service delivery or aggregation into existing web sites can be enabled easily with a RESTful style. Developers can use such technologies as JAX-RS and Asynchronous JavaScript with XML (AJAX) and such toolkits as Direct Web Remoting (DWR) to consume the services in their web applications. Rather than starting from scratch, services can be exposed with XML and consumed by HTML pages without significantly refactoring the existing web site architecture. Existing developers will be more productive because they are adding to something they are already familiar with rather than having to start from scratch with new technology

Source: same Java EE 6 tutorial, just from page 381:

What Are RESTful Web Services?

RESTful web services are built to work best on the Web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources and are accessed using *Uniform Resource Identifiers (URIs)*, typically links on the Web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.

The following principles encourage RESTful applications to be simple, lightweight, and fast:

- **Resource identification through URI**: A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery. See “The @Path Annotation and URI Path Templates” on page 385 for more information.
- **Uniform interface**: Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted by using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource. See “Responding to HTTP Methods and Requests” on page 387 for more information.
- **Self-descriptive messages**: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others. Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control. See “Responding to HTTP Methods and Requests” on page 387 and “Using Entity Providers to Map HTTP Response and Request Entity Bodies” on page 389 for more information.
- **Stateful interactions through hyperlinks:** Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction

.

Exam=>

Juhani's **Summary of** what **RESTful Web Services** could mean:

- **stateless** services (=technically stateless) even if the user experience might be constructed to **appear** stateful to the user (1. Find whether a vacation package you want do still exists, get 3 of them listed, 2. Click the one you want to reserve). Stateless though, technically, so just 2. is enough, if you know the correct package id needed for the 2. request:
    - All information needed to serve any request is sent **in the Http Request**
        - RESTful URI https://ourserver.ourdomain.fi/~~ourapp~~**/api/idea/101** (**not a file** URL)
        - Http Method ("Http Verbs") GET/POST/PUT/DELETE
        - Possibly parameters at the end of URI (GET) ?queryString=nokia&order=date
        - Usually Request Body (POST/PUT), usually one root object as JSON text (could be an array or complicated hierarchy of objects)
        - Http Request Header fields used for metadata of the request, especially security
        - Session and Login = Authorization tokens must be in the request!
        - API key must be there as well, otherwise the backend could just DROP the request without serving it
    - Backend doesn't know what was the previous service called!
    - **Do not assume** in your code that that selected vacation package is still available when the user finally makes the decision!!!!)
- Consequence of previous one => **performance**, **scalability** (these are easier to implement because of the statelessness, e.g. no need to keep something about single session in server memory)
- natural, almost **self-explanatory RESTful URIs** that utilize the Http Verbs=Methods: e.g. …/customer/201 (DELETE or GET), …/customer/find/name/Nokia (GET)
  (Those were just examples, don't use as model. E.g. find might actually send a JSON file with the find criteria to the back-end and the back-end would analyze the JSON and use it for the find operation)
- natural, **self-explanatory JSON data** sent back and forth. (See the previous point)
- **shared understanding of API and impacts,** e.g. Web Service API **description documentation OR using service end-point code comments and automatic doc creation**, so that Front and Back developers understand the interface exactly same way, and know e.g. the required inputs and all expected response variations.
- **RESTful** web services could be seen as one example of **https://en.wikipedia.org/wiki/Microservices**

How the **GraphQL** differs from the **traditional REST API** way

- In GraphQL backends the same service endpoint can be used to fetch a variety of data, as part of the POST data can be a query that is written the way we want to have data from the backend. The **language** the query is written in, **is called *GraphQL***.
- (It's not JSON, but still structured with {} and names of objects and other fields/members. And search criteria).
- ((In the older way, REST API can of course also have "somewhat varying queries" but you have to write the parsing of the URI into different queries yourself, e.g. in your JavaScript code. That would mean you need to change the service to support more varying queries in the same REST API endpoint.))
- In GraphQL services, the query language takes care about the dynamic nature of the request. Thus, in GraphQL you can use same endpoint for multiple frontend needs and purposes by altering the GraphQL query parts of the request without changing the backend code!
- The GraphQL backend will use libraries etc. able to automatically translate the query into database understood query or queries.

Serverless functions

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

- The cloud provider will dynamically create the needed backend server environment for the service when it's called.
- Thus, service is the main focus of the backend developer, and the needed resources will come from the cloud service provider, usually by clicking around in a web portal wizard, setting up the dependencies that the service would need, when/if it runs.
- This is of course nice for the cloud provider (MS Azure, AWS, GCP, …) as the customers' backend servers need not be running or have a state. They can manage and optimize the resource use with actual need basis.

Message Queues

In future exams. **No need** to read yet. E.g. Kafka, MQTT, …