What is a web service?

> Definition: A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.
>
> — W3C, Web Services Glossary
> `https://www.w3.org/TR/ws-arch/\#whatis`

Two Major Classes

> We can identify two major classes of Web services:
>
> - REST-compliant Web services, in which the primary purpose of the service is to manipulate XML representations of Web resources using a uniform set of "stateless" operations; and
> - arbitrary Web services, in which the service may expose an arbitrary set of operations.
>
> — W3C, Web Services Architecture (2004)
> `https://www.w3.org/TR/ws-arch/\#relwwwrest`

From SOAP to REST

- First technology for interactive web applications used AJAX – Asynchronous Javascript And Xml, but processing of XML is not convenient in Javascript
- Raise of using JavaScript Object Notation – JSON
  - Simple testing
  - Plenty of helping apps: Postman, Insomnia, curl, web browser
  - Javascript is simpler to start with than Java (e.g. there are more JS programmers and they are cheaper)

Web Service API Distribution

Basic terms

- **Uniform Resource Identifier (URI)** is a string of characters used to identify a resource. (e.g., `http://www.fel.cvut.cz/cz/education/`)

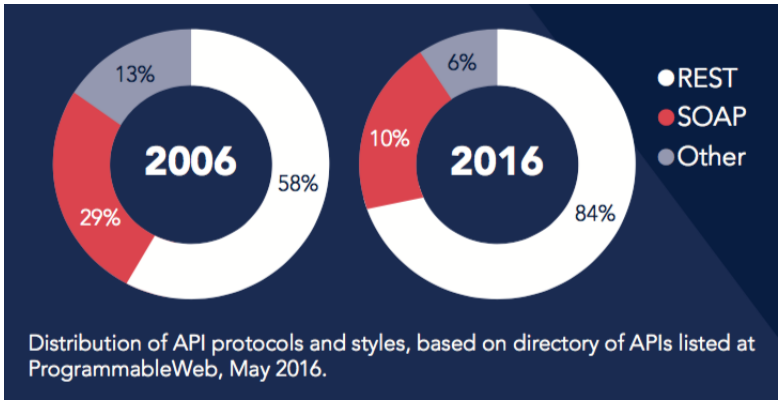
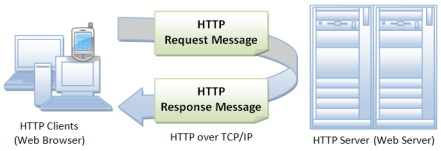


Figure 1: Interest in web service APIs. Source: `https://blog.wishtack.com/rest-apis-best-practices-and-security/`



- **The Hypertext Transfer Protocol (HTTP)** is an application *protocol* for distributed, collaborative, hypermedia information systems. It is the foundation of data communication for the World Wide Web.
  - initiated by Tim Berners-Lee at CERN in 1989
- **Representational State Transfer (REST)** is an *architectural style* for distributed hypermedia systems.
  - defined in 2000 by Roy Fielding in his doctoral dissertation

1 HTTP

HTTP protocol basics

- HTTP is a client-server application-level protocol
- Typically runs over a TCP/IP connection
- Extensible – e.g., video, image support
- Stateless
- Cacheable
- Requires *reliable* transport protocol – no UDP

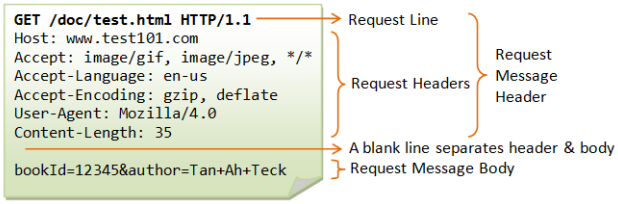


Figure 2: HTTP request example. Source: `https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html`

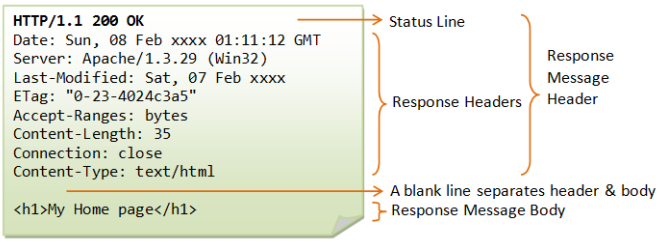


Figure 3: HTTP request example. Source: `https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html`

HTTP Request

- Message header
  - Request line – identifies HTTP method, URI and protocol version
  - Request headers
- Message body

HTTP Response

- Message header
  - Status line – identifies protocol version and response status code
  - Response headers
- Message body

HTTP Headers

Typical, often used HTTP headers

	Request	Response
Content	• Content-Type • Content-Length • Content-Encoding • Accept	• Content-Type • Content-Length • Content-Encoding
Caching	• If-Modified-Since • If-Match	• Last-Modified • ETag
Miscellaneous	• Cookie • Host • Authorization • User-Agent	• Set-Cookie • Location

HTTP Methods

GET

- Used to retrieve resource at request URI
- Safe and idempotent
- Cacheable
- Can have side effects, but not expected
- Can be conditional or partial (If-Modified-Since, Range)

POST

- Requests server to create new resource from the specified body
- Can be used also to update resources
- Should respond with 201 status and location of newly created resource on success
- Neither safe nor idempotent
- No caching

HTTP Methods

PUT

- Requests server to store the specified entity under the request URI
- Server may possibly create a resource if it does not exist
- Usually used to update resources

- Idempotent, unsafe

DELETE

- Used to ask server to delete resource at the request URI
- Idempotent, unsafe
- Deletion does not have to be immediate

HTTP Response Status Codes

- **1xx** – rarely used
- **2xx** – success
  - 200 OK – requests succeeded, usually contains data
  - 201 Created – returns a *Location* header for new resource
  - 202 Accepted – server received request and started processing
  - 204 No Content – request succeeded, nothing to return
- **3xx** – redirection
  - 304 Not Modified – resource not modified, cached version can be used

HTTP Response Status Codes

- **4xx** – client error
  - 400 Bad Request – malformed syntax
  - 401 Unauthorized – authentication required
  - 403 Forbidden – server has understood, but refuses request
  - 404 Not Found – resource not found
  - 405 Method Not Allowed – specified method is not supported
  - 409 Conflict – resource conflicts with client data
  - 415 Unsupported Media Type – server does not support media type
- **5xx** – server error
  - 500 Internal Server Error – server encountered error and failed to process request

2 RESTful web services

Understanding REST

- REST is an architectural style, not standard
- It was designed for distributed systems to address *architectural properties* such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability
- REST architectural style is defined by 6 *principles/architectural constraints* (e.g., client-server, stateless)
- System/API that conforms to the constraints of REST can be called *RESTful*

REST principles

1. Client-server
2. Uniform interface
   - Resource-based
   - Manipulation of resource through representation
   - Self-descriptive messages
   - Hypermedia as the engine of application state
3. Stateless interactions
4. Cacheable
5. Layered system
6. Code on demand (optional)

Building RESTful API

- Can be built on top of existing web technologies
- Reusing semantics of HTTP 1.1 methods
  - Safe and idempotent methods
  - Typically called HTTP verbs in context of services
  - Resource oriented, correspond to CRUD operations
  - Satisfies **uniform interface** constraint
- HTTP Headers to describe requests & responses
- Content negotiation

HTTP Verb	CRUD	Collection (e.g. /categories)	Specific Item (e.g. /categories/{id})
POST	Create	201 Created[1]	405 Method Not Allowed /409 Conflict[3]
GET	Read	200 OK, list of categories	200 OK, single category/404 Not Found[4]
PUT	Update/Replace	405 Method Not Allowed[2]	200 OK/**204 No Content**/404 Not Found[4]
PATCH	Update/Modify	405 Method Not Allowed[2]	200 OK/**204 No Content**/404 Not Found[4]
DELETE	Delete	405 Method Not Allowed[2]	200 OK/**204 No Content**/404 Not Found[4]

Table 1: Recommended return values of HTTP methods in combination with the resource URIs.

Recommended Interaction of HTTP Methods w.r.t. URIs

- [1] – returns *Location* header with link to /categories/{id} containing new ID
- [2] – unless you want to update/replace/modify/delete whole collection
- [3] – if resource already exists
- [4] – if ID is not found or invalid

Naming conventions

- resources should have name as nouns, not as verbs or actions
- plural if possible to apply
- URI should follow a predictable (i.e., consistent usage) and hierarchical structure (based on structure-relationships of data)

Correct usages
POST /customers/12345/orders/121/items **GET** /customers/12345/orders/121/items/3
GET|PUT|DELETE /customers/12345/configuration

Anti-patterns
GET /services?op=update_customer&id=12345&format=json **PUT** /customers/12345/update

HTTP Verbs – GET

```
GET /eshop/rest/categories HTTP/1.1
Host: localhost:8080
Accept: application/json
Cache-Control: no-cache
```

```
HTTP/1.1 200
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Content-Type: application/json;charset=UTF-8

[{
  "id": 2,
  "name": "CPU"
}, {
  "id": 7,
```

```
  "name": "Graphic card"
}, {
  "id": 11,
  "name": "RAM"
}]
```

HTTP Verbs – POST

```
POST /eshop/rest/categories HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Cookie: EAR_JSESSIONID=18162708908C126C0BA5A3D3081CCAC9
Cache-Control: no-cache

{
  "name": "Motherboard"
}
```

```
HTTP/1.1 201
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Location: http://localhost:8080/eshop/rest/categories/151
```

HTTP Verbs – PUT

```
PUT /eshop/rest/products/8 HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Cookie: EAR_JSESSIONID=18162708908C126C0BA5A3D3081CCAC9

{
  "id":8,
  "name":"MSI GeForce GTX 1050 Ti 4GT OC",
  "amount":50,
  "price":4490.0,
  "categories":[{
    "id":7,
    "name":"Graphic card"
  }],
  "removed":false
}
```

```
HTTP/1.1 204
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
```

HTTP Verbs – DELETE

```
DELETE /eshop/rest/products/8 HTTP/1.1
Host: localhost:8080
Cookie: EAR_JSESSIONID=18162708908C126C0BA5A3D3081CCAC9
Cache-Control: no-cache
```

```
HTTP/1.1 204
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
```

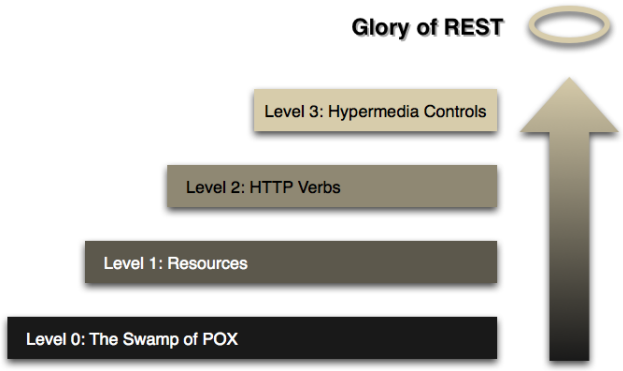


Figure 4: A model (developed by Leonard Richardson) that breaks down the principal elements of a REST approach into three steps about *resources*, *http verbs*, and *hypermedia controls*. Source: `http://martinfowler.com/articles/richardsonMaturityModel.html`

Demo

Let's examine SpaceX REST API.

`https://documenter.getpostman.com/view/2025350/RWaEzAiG\#intro`

The Richardson Maturity Model

- provides a way to evaluate compliance of API to REST constraints

2.1 HATEOAS

HATEOAS

- *Hypermedia as the Engine of Application State*
- Final level of the Richardson Maturity Model
- Client needs zero or little prior knowledge of an API
- Client just needs to understand hypermedia
- Server provides links to further endpoints
- Often difficult to implement
  - Not many usable libraries

HATEOAS Example

**EAR e-shop does not support HATEOAS.*

```
{
  "id": 2,
  "name": "CPU",
  "links": [{
    "rel": "self",
    "href": "http://localhost:8080/eshop/rest/categories/2"
  }, {
    "rel": "edit",
    "href": "http://localhost:8080/eshop/rest/categories/2"
  }, {
    "rel": "products",
    "href": "http://localhost:8080/eshop/rest/categories/2/products"
  }]
}
```

We are using the *Atom* link format.

REST Documentation – Source

```
@Path("demos")
public class JavaEE8Resource {

   @GET
   public Response ping(){
      return Response
            .ok("ping")
            .build();
   }

   @GET
   @Path("{id}")
   public SampleObject objects(@PathParam("id") Integer id) {
      return new SampleObject(id, "NAZDAR!");
   }
}
```

REST Documentation – Output

- Documentation of REST is done in two (similar) formats: Swagger or OpenApi
- `https://download.eclipse.org/microprofile/microprofile-open-api-3.1` → spec, Annotations

```
openapi: 3.0.0
info:
  title: Deployed Resources
  version: 1.0.0
servers:
- url: http://pidibook:8080/DemoRest1
  description: Default Server.
paths:
  /resources/demo:
    get:
      operationId: ping
```

```
    responses:
      default:...
  /resources/demo/objects:
    get:
      operationId: objects
      responses:
        default:
          content:
            '*/*':
              schema:
                $ref: '\#/components/schemas/SampleObject'
components:
  schemas:
    SampleObject:
      type: object
      properties:
        demoInt:
          type: integer
        demoString:
          type: string
```

3 Linked Data

Linked Data

- Method of publishing structured data allowing to interlink them with other data
- Builds upon the original ideas of the Web
  - Interconnected resources, but this time, machine-readable
- Knowledge-based systems, context-aware applications, precise domain description, knowledge inference
- Still possible to build REST APIs, but resources have global identifiers now
- Attributes and relationships also globally identifiable and may have well-defined meaning

Linked Data Example

```
{
  "@context": {
    "name": "http://www.w3.org/2000/01/rdf-schema#label",
    "description": "http://purl.org/dc/terms/description",
    "products": "http://onto.fel.cvut.cz/ontologies/eshop/has-product"
  },
  "@id": "http://onto.fel.cvut.cz/eshop/categories/cpu",
  "products": {
    "@id": "https://ark.intel.com/products/97455/Intel-Core-i3-7100-Processor-3M-Cache
        -3-90-GHz",
    "name": "Intel Core i3-7100"
  },
  "description": "Category of Central Processing Units for computers.",
  "name": "CPU"
}
```

REST in Spring

```
@RestController
public class CarController {
   @Inject private CarService carService;

      @GetMapping("/cars")
      public Cars allCars(@RequestParam(value = "name", defaultValue = "World") String
          name) {
            return carService.listAllCars();
      }
}
```

JAX-RS

```
@Path("v1/cars")
@Produces(MediaType.APPLICATION_JSON)
public class CarsResource {
   @GET
   public Cars allCars() {
      return service.allCars();
   }

   @Path("{id}")
   @GET
   public Car oneCar(@PathParam("id") Integer id) {
      return service.findById(id);
   }

   @Path("{id}")
   @DELETE
   public Response deleteOneCar(@PathParam("id") Integer id) {
      service.remove(id);
      return Response.noContent().build();
   }
```

JAX-RS Client

```
Client client = javax.ws.rs.client.ClientBuilder.newClient();
WebTarget webTarget = client.target(BASE_URI).path("v1/cars");
webTarget.request(javax.ws.rs.core.MediaType.APPLICATION_JSON)
   .get(Cars.class)
```

MicroProfile – REST Support

```
@RegisterRestClient(baseUri = "https://api.spacexdata.com/")
@Path("v3")
public interface SpaceXRestClient {
   @GET
   @Path("rockets/")
   @Produces(MediaType.APPLICATION_JSON)
   public List<RestRocket> all();

   @GET
   @Path("rockets/{rocket_id}")
   @Produces(MediaType.APPLICATION_JSON)
   public RestRocket rocket(@PathParam("rocket_id") String rocketId);
```

```
}
use:
   @Inject
   @RestClient
   SpaceXRestClient spaceXRestClient;

{ spaceXRestClient.all(); }
```

REST – Security

- Same as HTML – HTTPS, passwords
- Usage of JWT (JSON Web Token), mainly makes sense for μServices (holds signed roles, other information so some services don't need user database)
- Necessary to use either reverse (https) proxy or CORS headers
- Security is a huge problem
  - No way, how to protect access, easy to play with
  - Double security – on client, on server
  - Every single data must have REST, every dropdown list, every table, every form
  - Very difficult to check EVERYTHING – objects are returned and only parts of them are allowed to change (e.g. mail, username, password, but not id, roles). In some other cases it is allowed (e.g. by superadmin).

REST – Battlefield Experience

- Good support in Spring, JAX-RS, great in MicroProfile
- Good idea to add API version to url, e.g. /rest/v1/cars
- Use DTO frequently, always for list/array
- ID returned in URL – needs to be parsed
- Messages returned in HTTP header are in ASCII, e.g. no Czech messages
- Various errors return messages in various parts of the JS response object
- Using JavaScript Object Notation (JSON) between languages having nothing with JS in μServices
- Generation of client from service description

Demo

JAX-RS, MicroProfile

- `http://localhost:4848/openapi`
- `http://localhost:8080/JAXRSServer/rest/v1/cars`
- `http://localhost:8080/JAXRSServer/rest/v1/cars/0`
- `http://localhost:8080/JAXRSServer/rest/v1/cars/5`
- `http://aubiwork:8080/JAXRSClient/rest/v1/rockets/`
- `http://aubiwork:8080/JAXRSClient/rest/v1/rockets/by-id?id=falcon1`

4 Conclusions

REST

Pros

- API first (agree on API, then code on both sides)
- Easy to build
- Easy to use
- Standard technologies – HTTP, JSON, XML
- Platform-independent (JS-based web pages, mobiles)
- Stateless, cacheable

Cons

- No standard for REST itself – APIs build in various ways
- No full generator for all the possibilities (lack in documentation)
- No "registry" of REST services

The End

Thank You

Resources

- Fielding, R.T., 2000. Architectural styles and the design of network-based software architectures (Doctoral dissertation, University of California, Irvine),
- Fowler, M., 2010. Richardson Maturity Model: steps toward the glory of REST. Online at `http://martinfowler.com/articles/richardsonMaturityModel.html`.
- Lanthaler, M. and Gütl, C., 2012, April. On using JSON-LD to create evolvable RESTful services. In Proceedings of the Third International Workshop on RESTful Design (pp. 25-32). ACM.
- `https://spring.io/understanding/REST`
- `https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview`
- `http://linkeddata.org/`