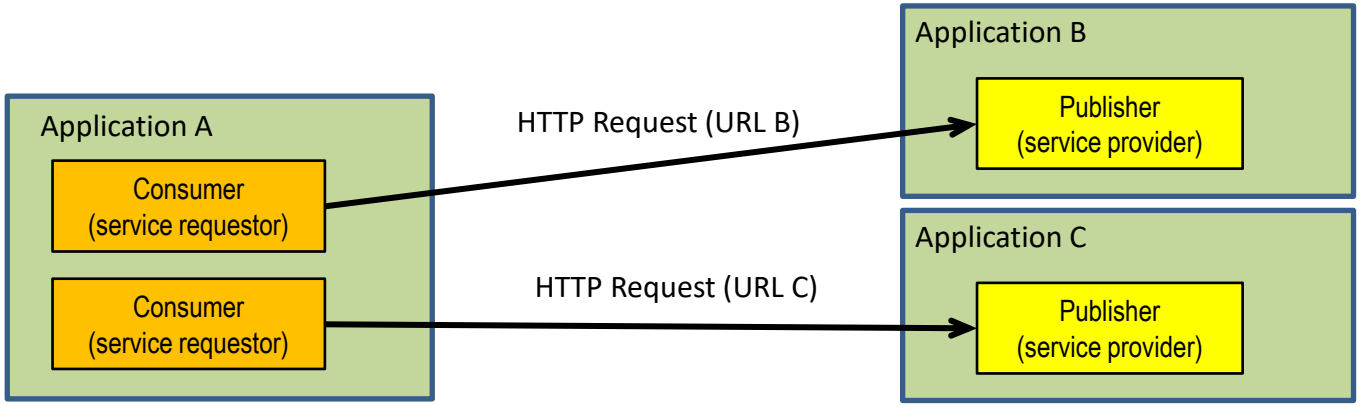
RESTful Web Services

SCOMRED, January 2022

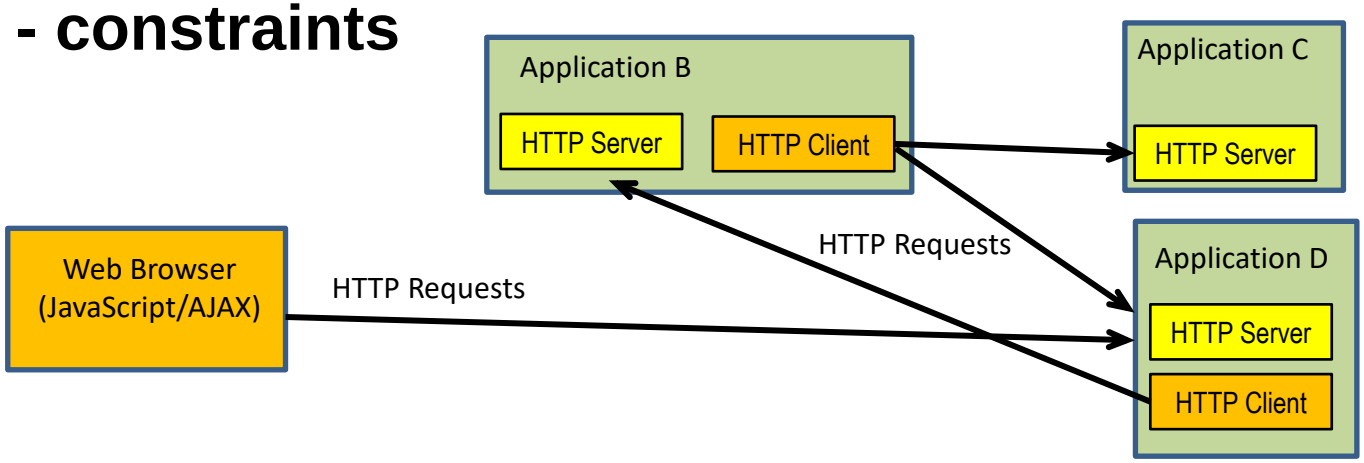
Web Services

The central concept on web services is the use of HTTP for application-to-application communications without direct human intervention.
One application must assume the **HTTP client's role** (**service requestor or consumer**) and the other application the **HTTP server's role** (**service provider or publisher**). So, the web service is made available to the service requestor applications by the service provider application. Of course, the same application can be both a consumer and a provider.



This is a very general concept, and allows programmers to implement it with high freedom, for instance regarding what HTTP methods are going to be used, how resources on the provider side are named, and which content types are going to be used on data transfers between requestors and providers. As far as the HTTP protocol is respected, everything is possible.

Web Services - constraints



There's no obstacle for one application being both a consumer and a provider, and we already know a standard Web Browser can also be a consumer by using **XMLHttpRequest** JavaScript objects. Therefore, quite complex distributed environments can be established through web services.

The general freedom in implementing web services tends to turn things somewhat chaotic, thus some efforts have been done on instituting some **rules and principles**, often referred to as **constraints**.

One of these sets of constraints for web services architectures is known as **RESTful**, standing for REST compliant.

RESTful web services architectural constraints

REST stands for Representational State Transfer, it is a constrained, resource-based design model to implement web services. Main principles (constraints) are:

- Clients request operations over server-side resources (each identified by an URI), operations over server-side resources are: Create, Read, Update and Delete (CRUD), each corresponds to a specific HTTP request method.
- Resources' contents should be transferred in XML or JSON representations. Nevertheless, HTML and others might be used if appropriate.
- Servers are stateless in the sense they don't store information about each client's dialogue context. Therefore, on every request clients must provide all required context data for the operation.
- If the server has a state, then that state context must be represented by an addressable resource (URI), clients may then refer that state context on requests.

RESTful web services consumer applications can request the following four operations over a resource (URI):

Operation	HTTP methods
Create a resource	POST; PUT
Read/retrieve a resource	GET
Update/Modify a resource	PUT
Delete/remove a resource	DELETE

RESTful – resources and collections

The only **safe method** is GET, meaning it does not change the resource or the server side. Methods PUT, GET, and DELETE are regarded as **idempotent methods;** this means making more than one successive identical request over the same resource has no additional effects beyond the effect of the first request.

A URI may refer to a **single resource** or a **collection** of resources, **singular names** are to be used for single resources, **plural names** for a collection of resources. Depending on being a single resource or a resources collection, HTTP methods will represent different actions over the resource (URI):

HTTP method	Single resource (singular name URI)	Resources collection (plural name URI)
GET	Retrieve the resource.	List the of resources items in the collection. Retrieved data is a list of resources' URIs and optionally other resources' data.
PUT	Replace the resource, if it does not exist, create it.	Replace the whole collection with another collection.
POST	**Not used** because the URI would be regarded as a collection and a new collection item would be created within it.	Create a new resource item within the collection. The new resource URI is automatically assigned.
DELETE	Delete the resource.	Delete the entire collection.

RESTful - URI naming (guidelines and best practices)

- A singular name for a single resource or a collection's item/element.
- A plural name for a collection of resources.
- Verbs for controllers and functions.
- Notice that, excluding the origin (e.g., DNS server's name), the URI is case sensitive.
- Use either camel casing or, preferably, lowercase with words separated with hyphens (spinal case), instead of underscores (snake case).
- Avoid CRUD names (Create/Read/Update/Delete) for a URI.
- URI path elements should represent resources' hierarchical structure.
- A URI path component can be used to represent a variable's value, in REST that's the recommendation, nevertheless, a query string can also be appended to a URI.

RESTful – Contents transfer

Resources' contents must be transferred between providers and consumers (in both directions) in an implementation independent representation.

Text (ASCII characters organized in lines) is a universally supported concept and, within some limits, it's also acceptable for human reading. For those reasons it's widely used to represent data, nevertheless, rules must be established so that data represented in text format can be analysed by applications.

We already are aware about the HTML specification that uses text, and yet HTML is more focused on data presentation on not so much in data representation.

A somewhat similar, but more generic specification is **Extensible Mark-up Language** (XML), RESTful constraints don't impose the use of XML, but they clearly point out to the use of either XML or JSON to represent generic data.

When web services resources are transferred between applications in XML format, the **Content-type: application/xml** HTTP header line should be added.

Extensible Mark-up Language (XML)

XML is a data representation format through text, designed to be both human-readable and also easy to be processed by applications.

As with HTML, XML encapsulates data within tags represented between symbols **<** and **>**, but unlike with HTML where tag names have special meanings, in XML they do not.

In XML tags may be freely established by applications conforming their needs. Also, as mentioned before, HTML specifies a way to present data to end-users, XML specifies only the data representation.

A XML content may optionally start by a special line called **XML prolog**:

<?xml version="1.0" encoding="UTF-8"?>

The XML prolog line is optional, but every XML content must have a **root tag** embracing the whole content. Tag names are case sensitive and every opened tag must be closed by an end tag. As with HTML, if a tag doesn't have any data (content) it may be closed immediately when started by ending it with **/>** instead of **>**.

If a tag's content includes the **<** symbol or the **&** symbol, they must be represented, correspondingly by **<** and **&** to avoid parsing issues.

XML – tag's attributes

XML tags may have attributes, attributes are pairs name="value" declared within the start tag, attribute names are also case sensitive, and the attribute value must always be quoted.

Tag's attributes should be used to identify the data element and not data's properties, properties should be specified by sub tags.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
  <user id="100" />
  <user id="101"></user>
  <user id="102"><name>ABC</name></user>
  <user id="103">
        <name>ABC</name>
        <phone>9999909</phone>
  </user>
</users>
```

In this example, <users> is the root tag. It contains four <user> tags, the first two are empty (don't have content).

Hypermedia As The Engine Of Application State (HATEOAS)

HATEOAS is a constraint of the REST application architecture. It means by accessing and retrieving a resource, a REST client also retrieves a **list of links** representing alternative actions from that point on. This is very similar to human web usage: when a web page is reached there's a set of alternative links to follow from that point.

This strategy makes the API discoverable by REST clients, though it's state dependent. Only after accessing a URI follow up links are provided, they represent possible state transitions from the initial state and may depend on the resource itself or other factors, like for instance user authentication. Classic example using XML:

```
GET /accounts/1111 HTTP/1.1
    Host: example.com
    Accept: application/xml
    ...
HTTP/1.1 200 OK
    Content-Type: application/xml
    Content-Length: ...

    <?xml version="1.0"?>
    <account>
       <account_number>1111</account_number>
       <balance currency="usd">100.00</balance>
       <link rel="deposit" href="/accounts/12345/deposit" />
       <link rel="withdraw" href="/accounts/12345/withdraw" />
       <link rel="transfer" href="/accounts/12345/transfer" />
       <link rel="close" href="/accounts/12345/close" />
     </account>
```

```
GET /accounts/1112 HTTP/1.1
    Host: example.com
    Accept: application/xml
    ...
HTTP/1.1 200 OK
    Content-Type: application/xml
    Content-Length: ...

    <?xml version="1.0"?>
    <account>
       <account_number>1112</account_number>
       <balance currency="usd">0.00</balance>
       <link rel="deposit" href="/accounts/12345/deposit" />
       <link rel="close" href="/accounts/12345/close" />
     </account>
```

Because account number 1112 has a zero balance, available actions are only deposit and close.

Example Java API for RESTful Web Services (JAX-RS)

A web services provider is an application that has to include an HTTP server. Take for instance our lab classes, we are using the Apache web server to run external applications through the CGI specification.

Nowadays, many programming languages have the option to include in the developed application a ready to use HTTP server implementation.

One interesting RESTful Web Services API for Java is **JAX-RS**. In essence it provides a set of annotations (**javax.ws.rs** package) that can be used on classes and methods definition to turn them into Web Services **providers**, with a minimal effort. There's also a **JAX-RS Client API** (**javax.ws.rs.client** package) to implement Web Services **consumers**.

The **@Path** annotation can be used both in a class declaration or in a method declaration to establish the relative URI name (path), for instance:

@Path("/users")

Establishes the relative URI for all methods of the class or for a specific method, correspondingly if used in a class declaration or in a method declaration.

JAX-RS - URI names

The **@Path** annotation is relative to the **context path**, by default it's the project's name, but it may be settled on the HTTP Server or project settings. The **@Path** annotation is also relative to the **application path**, established through the **@ApplicationPath** annotation (**javax.ws.rs.core.Application**. subclass).

The final resulting URI is:

/CONTEXT-PATH/APPLICATION-PATH/PATH

The @Path notation may be a template with variable elements of the URI enclosed in braces. When declaring methods, those variable elements' values can be passed as arguments on runtime by using the **@PathParam** annotation.

Example:

```
@Path("/users/{id}")
public String getUser(@PathParam("id") int id) {
…
}
```

JAX-RS – methods

A method may be declared to be a web service by using the following annotations, corresponding to HTTP methods: @GET, @POST, @PUT, @DELETE.

GET requests have no body (content), if a query string (embedded in the URI) is used by the client, the elements of the query string can be obtained by using the **@QueryParam** annotation and pass it as argument to the method.

For requests with a body (POST and PUT), the body's content is passed to the method as argument.

Java methods that are web services should return a result, the return value of the method is the HTTP response to be sent to the client.

When applicable, received and produced content types should also be declared, respectively through annotations **@Consumes** and **@Produces**.

Example:

```
@Path("/echo")
@POST
@Consumes(“text/html”)
@Produces(“text/html”)
public String echoString(String content) {
return(content);
}
```

JAX-RS – @Context

The @Context annotation can be used to access information about the current request. Data regarding the request's properties is obtained as Java objects. Examples for accessing the URI and HTTP headers:

```
@GET
public String get(@Context UriInfo ui) {
    MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
    MultivaluedMap<String, String> pathParams = ui.getPathParameters();
}
```

```
@GET
public String get(@Context HttpHeaders hh) {
    MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
    Map<String, Cookie> pathParams = hh.getCookies();
}
```

This small intro to JAX-RS is mainly intended to show by example that there are libraries and frameworks that make developing web services very simple.