

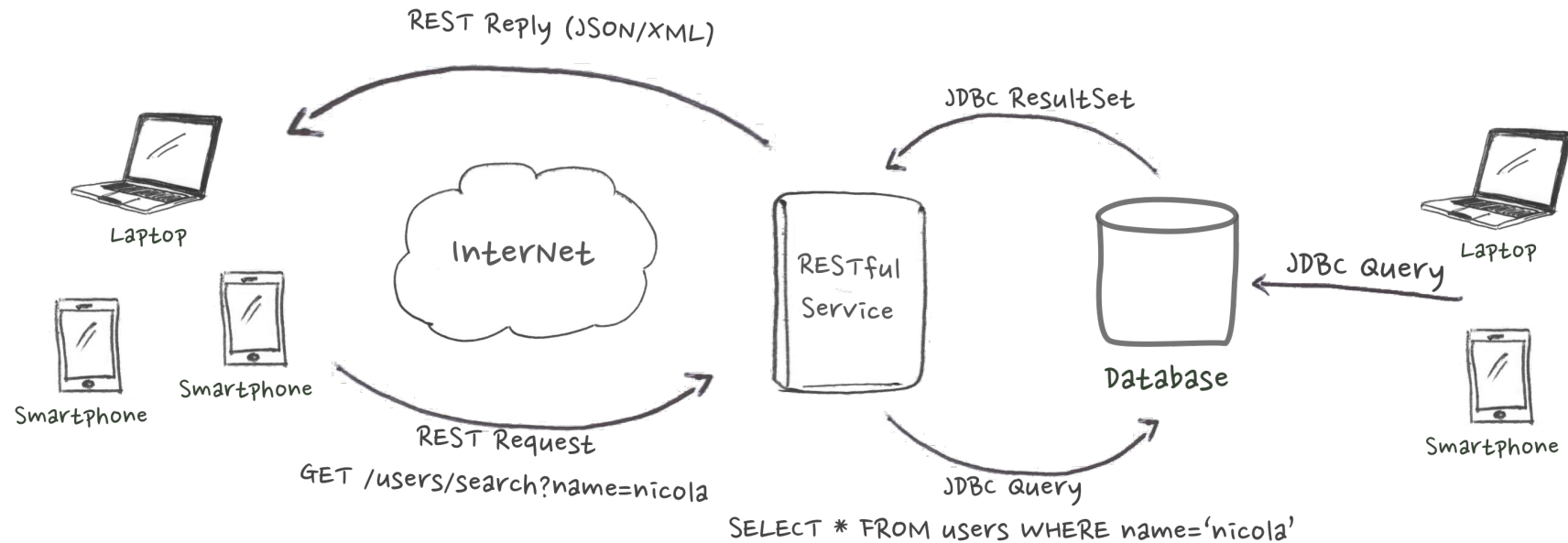
REST (Representational State Transfer)

Università di Modena e Reggio Emilia

Prof. Nicola Bicocchi (nicola.bicocchi@unimore.it)



Why Learn REST?



- REST is used to **build Web services** that are lightweight and scalable
- REST **decouples applications** from vendor-specific details (e.g., JDBC) and prevents exposing DMBS to untrusted networks (e.g. Internet)
- Widely **available libraries** for many languages (e.g., RESTlet for Java)
- <https://github.com/toddmotto/public-apis#books>

Why Learn REST?

- A number of **mobile apps** are built upon **RESTful** services.
 - <https://www.instagram.com/developer/>
 - <https://developer.twitter.com/en/docs>
 - <https://developers.facebook.com/docs/graph-api>
 - <https://www.flickr.com/services/api/>
 - <https://developer.foursquare.com/>



Major Concepts

- Messages
- Resources (URIs)
- Representations
- Stateless

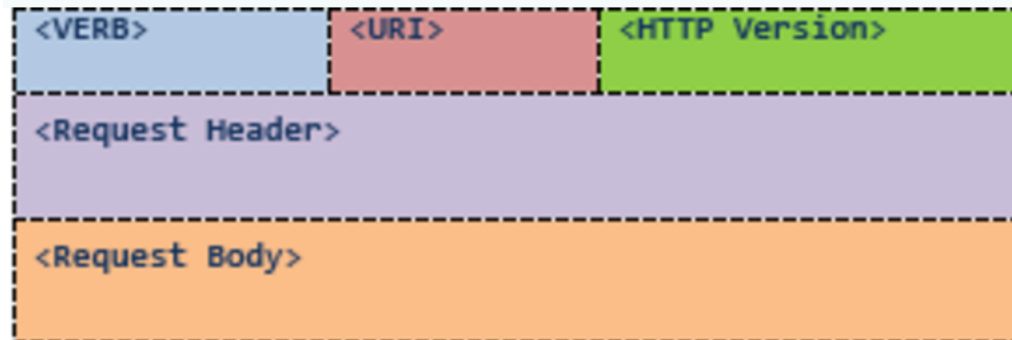


Messages

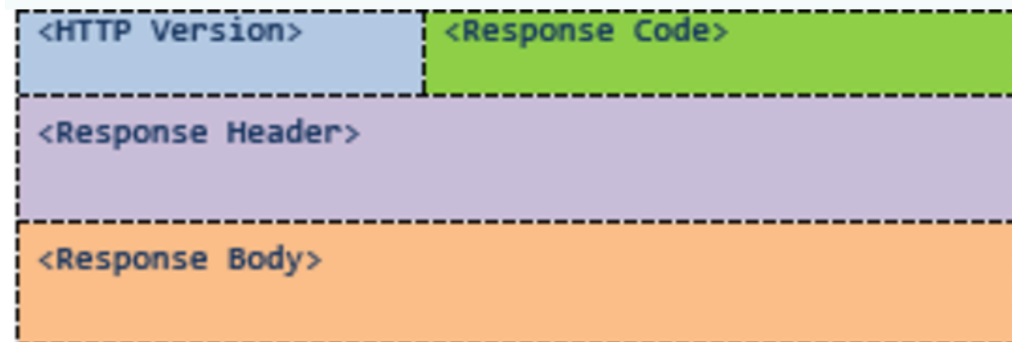
- The client and service talk to each other via messages. Clients send a request to the server, and the server replies with a response. Apart from the actual data, these messages also contain some metadata about the message.
- It is important to have some background about the HTTP 1.1 request and response formats for designing RESTful Web services.



HTTP Messages



HTTP/1.1 Request



HTTP/1.1 Response

Resources

- **Every system uses resources.** Resources can be pictures, videos, users data ecc... The purpose of a service is to provide an access to resources to its clients. Service architects and developers want services to be **easy to implement, maintainable, extensible, and scalable.**



Resources

Place details

https://api.foursquare.com/v2/venues/VENUE_ID

Photos details

https://api.foursquare.com/v2/photos/PHOTO_ID

Search for a user

<https://api.foursquare.com/v2/users/search>

Recent checkins by friends

<https://api.foursquare.com/v2/checkins/recent>



Representations

- The focus of a RESTful service is on resources and how to provide access to these resources. A resource can be thought of as an object as in OOP. A resource can consist of other resources.
- While designing a system, the first thing to do is identify the resources and determine how they are related to each other.
- This is similar to designing a database: Identify entities and relations.



Representations

- Once we have identified our resources, the next thing we need is to **find a way to represent these resources**.
- You can use **any format** for representing the resources, as **REST does not put a restriction** on the format of a representation.
- Nevertheless, the most used representations are XML and JSON



Representations

Listing One: JSON representation of a resource.

```
1 {  
2   "ID": "1",  
3   "Name": "M Vaqqas",  
4   "Email": "m.vaqqas@gmail.com",  
5   "Country": "India"  
6 }
```

?

Listing Two: XML representation of a resource.

```
1 <Person>  
2  
3   <ID>1</ID>  
4  
5   <Name>M Vaqqas</Name>  
6  
7   <Email>m.vaqqas@gmail.com</Email>  
8  
9   <Country>India</Country>  
10 </Person>
```

?



HTTP Verbs (Operations)

- HTTP Verbs (see HTTP Request) define **operations on specific resources.**
- GET /users/145 (*retrieve user 145*)
- DELETE /users/145 (*delete user 145*)
- POST /users/ (*add a new user*)
- PUT /users/17 (*update user 17*)



HTTP Verbs (Operations)

Method	Operation performed on server	Quality
GET	Read a resource.	Safe
PUT	Insert a new resource or update if the resource already exists.	Idempotent
POST	Insert a new resource. Also can be used to update an existing resource.	N/A
DELETE	Delete a resource .	Idempotent



Safe and Idempotent

- A **Safe HTTP method** does not make any changes to the resource on the server.
- An **Idempotent HTTP method** has same effect no matter how many times it is performed.
- Classifying methods as Safe and Idempotent makes it easy to predict the results in the unreliable environment of the Web where the client may fire the same request again.



PUT and POST

Request	Operation
PUT <code>http://MyService/Persons/</code>	Won't work. PUT requires a complete URI
PUT <code>http://MyService/Persons/1</code>	Insert a new person with <code>PersonID=1</code> if it does not already exist, or else update the existing resource
POST <code>http://MyService/Persons/</code>	Insert a new person every time this request is made and generate a new <code>PersonID</code> .
POST <code>http://MyService/Persons/1</code>	Update the existing person where <code>PersonID=1</code>



PUT and POST

- There is no difference between PUT and POST if the resource already exists, both update the existing resource.
- POST <http://MyService/Persons/> will create a resource each time it is fired.



Addressing resources (URIs)

- REST requires each resource to have at least one URI. **A RESTful service uses a directory hierarchy like human readable URIs to address its resources.** The job of a URI is to identify a resource or a collection of resources. The actual operation is determined by an HTTP verb. The URI should not say anything about the operation or action.
- Suppose we have a database of persons and we wish to expose it to the outer world through a service.
`http://MyService/Persons/1`
- **`Protocol://ServiceName/ResourceType/ResourceID`**



Addressing resources (URIs)

- Use plural nouns for naming your resources.
- Avoid using spaces as they create confusion. Use an _ (underscore) or – (hyphen) instead.
- A URI is case insensitive. I use camel case in my URIs for better clarity. You can use all lower-case URIs.
- A cool URI never changes; so give some thought before deciding on the URIs for your service. If you need to change the location of a resource, do not discard the old URI and redirect the client to the new location.
- Avoid verbs for your resource names until your resource is actually an operation or a process. Verbs are more suitable for the names of operations.



Query parameters

- The basic purpose of query parameters is to provide parameters to an operation that needs the data items.
 - `http://MyService/Persons/1?format=json`
 - `http://MyService/Persons/search?name='nicola'`
- Including the encoding in the main URI **is not logically correct**:
 - `http://MyService/Persons/1/json/`



Statelessness

- A RESTful service is stateless and does not maintain the application state for any client.
- A request cannot be dependent on a past request. A REST service treats each request independently.



Statelessness

Stateless design

Request1: GET http://MyService/Persons/1 HTTP/1.1

Request2: GET http://MyService/Persons/2 HTTP/1.1

Stateful design (Dangerous! Which client??)

Request1: GET http://MyService/Persons/1 HTTP/1.1

Request2: GET http://MyService/NextPerson HTTP/1.1



Documentation

- There is no excuse for not documenting your service.
- You should document every resource and URI for client developers. You can use any format for structuring your document, but it should contain enough information about resources, URIs, Available Methods, and any other information required for accessing your service.



Documentation

Service Name: MyService

Address: <http://MyService/>

Resource	Methods	URI	Description
Person	GET, POST, PUT, DELETE	http://MyService/Persons/{PersonID}	Contains information about a person {PersonID} is optional Format: text/xml
Club	GET, POST, PUT	http://MyService/Clubs/{ClubID}	Contains information about a club. A club can be joined by multiple people {ClubID} is optional Format: text/xml
Search	GET	http://MyService/Search?	Search a person or a club Format: text/xml Query Parameters: Name: String, Name of a person or a club Country: String, optional, Name of the country of a person or a club Type: String, optional, Person or Club. If not provided then search will result in both Person and Clubs



Criticism

- No transactions support
 - DBMS (usually behind REST services) support transactions
- No publish/subscribe support.
 - Notification is done by polling.
 - The client can poll the server. GET is extremely optimized on the web.
- High bandwidth
 - HTTP uses a request/response model, so there's a lot of baggage flying around the network to make it all work.



Advantages

- REST is a great way of developing lightweight Web services that are easy to implement, maintain, and discover.
- HTTP provides an excellent interface to implement RESTful services with features like a uniform interface and caching. However, it is up to developers to implement and utilize these features correctly.
- If we get the basics right, a RESTful service can be easily implemented using any of the existing technologies such as Python, .NET, or Java.

