REST API and Data Formats

Tullio Facchinetti
<tullio.facchinetti@unipv.it>

24 maggio 2023

http://robot.unipv.it/toolleeo

XML and JSON

The two currently most widely adopted data formats:

1. **XML** - Extensible Markup Language
2. **JSON** - JavaScript Object Notation

Origin:

- XML - Markup language proposed by the World Wide Web Consortium in 1998.
- JSON - Data interchange originally specified by Douglas Crockford originally in the early 2000s to be used with Javascript.

Examples

XML

```
<update>2023-03-03</update>
<students>
  <student>
    <lastName>Doe</lastName>
    <age>18</age>
  </student>
  <student>
    <lastName>Smith</lastName>
    <age>22</age>
  </student>
  <student>
    <lastName>Jones</lastName>
    <age>20</age>
  </student>
</students>
```

JSON

```
{
  "update": "2023-03-03",
  "students":
  [
      {
        "lastName": "Doe",
        "age": 18
      },
      {
        "lastName": "Smith",
        "age": 22
      },
      {
        "lastName": "Jones",
        "age": 20
      }
  ]
}
```

Line breaks, indentation and spacing are for human readability.

Characteristics of XML

- Tree data structure.
- Supports attributes to elements.
- Validation through an additional XML schema (XSD) that defines the necessary metadata for interpreting.
- Supports comments.
- Supports namespaces.
- Supports complex data types (images, audio, etc.).
- Several file formats are based on XML (e.g., SVG, Open XML - docx, xlsx, pptx, OpenDocument - ods, odt, odp).
- Verbose.

Characteristics of JSON

- File format based on array and maps.
- Data structures directly mapped on programming language types (e.g., Javascript, Python).
- Support for primitive types such as strings, numbers, arrays, boolean and null.
- Fast and easy to parse.
- (Relatively) Compact.

Comparison

	XML	JSON
Human readable	☺	☺
Speed	😐	☺
Size	☹	☺
Comments	☺	☹
UTF support	☺	☺
Array support	😐	☺
Data types	☺	😐
Namespace support	☺	☹

In general, XML is adequate to more articulated and complex data structures, while JSON works better for simpler and faster data exchange (e.g., through API).

Data structure in JSON

JSON is based on two fundamental data structure:

- **List**: like arrays but with variable size and heterogeneous types
- **Map** (or hash map, or dictionary): key-value association

Nesting

- Lists can contain maps as elements
- The value of a map can be a list

Sorting of elements

- List: based on the position of appearance in the list
- Map: not sorted

Access to elements

- List: by index (e.g. `mylist[0]`)
- Map: by key (e.g. `mymapp["Facchinetti"]`)

Example

```
{
  "update": "2023-03-03",
  "students":
  [
      {
        "lastName": "Doe",
        "age": 18
      },
      {
        "lastName": "Smith",
        "age": 22
      },
      {
        "lastName": "Jones",
        "age": 20
      }
  ]
}
```

- Let's assume that the structure is addressed by the variable `data` in a Python program.
- `data` is a map containing two keys: `update` and `students`.
- `data["update"]` is a *string* representing a date.
- `data["students"]` is a list containing 3 maps.
- `data["students"][0]` is the first map in the list.
- `data["students"][0]["age"]` is the value 18.

RESTful services

- **REST**: acronym for **RE**presentational **S**tate **T**ransfer.
- Architectural style for distributed hypermedia systems.
- Firtly introduced by Roy Fielding in his dissertation (2000).

> A Web API (or Web Service) conforming to the REST architectural style is a REST API

REST principles: Uniform interface (1/6)

- **Identification of resources**: The interface must uniquely identify each resource involved in the interaction between the client and the server.
- **Manipulation of resources through representations**: The resources should have uniform representations in the server response; clients use these representations to modify the resources state in the server.
- **Self-descriptive messages**: Each resource representation should carry enough information to describe how to process the message.
- **Hypermedia as the engine of application state**: The client should have only the initial URI of the application; the client application should dynamically drive all other resources and interactions with the use of hyperlinks.

REST principles: Client-Server (2/6)

- Separation of concerns between the user interface concerns (client) from the data storage concerns (server).
- Client and server components can evolve independently.
- Improvement of the portability of the user interface across multiple platforms
- Improvement of the scalability by simplifying the server components.

While the client and the server evolve, we have to make sure that the interface/contract between the client and the server - i.e., the API - does not change (break)

REST principles: Stateless (3/6)

- **Statelessness** requires that each request from the client to the server must contain all of the necessary information to understand and complete the request.
- The server cannot take advantage of any previously stored context information on the server.
- For this reason, the client application must entirely keep the session state.

REST principles: Cacheable (4/6)

- A response should implicitly or explicitly label itself as cacheable or non-cacheable.
- If the response is cacheable, the client application gets the right to reuse the same (cached) response data for equivalent requests and a specified period.

REST principles: Layered system (5/6)

- An architecture to be composed of hierarchical layers by constraining component behavior.
- In a layered system, each component cannot see beyond the immediate layer they are interacting with.

REST principles: Code on Demand (Optional) (6/6)

- Client functionalities can be extended by downloading and executing code in the form of applets or scripts.
- Servers can provide part of features delivered to the client in the form of code, and the client only needs to execute the code.

Clients are simplified since it reduces the number of features that are required to be pre-implemented

Resources

- A resource can be any information that can be named (from Roy Fielding's dissertation)
- Alternatively: A resource is anything that's important enough to be referenced as a thing in itself.

A resource is an abstraction of information managed by a REST API

Example of resources

Examples of resources:

- Version 1.0.3 of the software release
- The latest version of the software release
- The first weblog entry for October 24, 2006
- A road map of Little Rock, Arkansas
- Some information about jellyfish
- A directory of resources pertaining to jellyfish
- The next prime number after 1024
- The next five prime numbers after 1024
- The sales numbers for Q42004
- A list of the open bugs in the bug database

Source: L. Richardson, S. Ruby, “RESTful Web Services”, O’Reilly Media, 2007.

Resource representation

The state of the resource, at any particular time, is known as the **resource representation**

The representation of a resource consists of:

- The data.
- The metadata describing the data.
- The hypermedia links that can help the clients in transition to the next desired state.

Characteristics of resources: Identifiers (1/5)

Identifiers are used to identify each resource involved in the interactions between the client and the server components.

Resources can be **singletons** or **collections**.

Examples:

- `student` is a singleton resource
- `students` is a collection resource (notice the plural)

Identifiers should refer to a resource that is a thing (noun) instead of referring to an action (verb)

Characteristics of resources: URI (2/5)

Resources are represented and addressd using **Uniform Resource Identifiers** (URIs).

Examples:

- `https://api.mydomain.com/students`
- `https://api.mydomain.com/students/1`

Characteristics of resources: URI (2/5)

Guidelines

Use lowercase letters

- ✗ /MY-FOLDER/MY-DOC
- ✗ /My-Folder/my-doc
- ✓ /my-folder/my-doc

Separate multiple words

- ✗ /studentmanagement/managedstudents
- ✓ /student-management/managed-students

Do not use underscores

- ✗ /student_management/managed_students
- ✓ /student-management/managed-students

Do not use trailing forward slash (/) in URIs

- ✗ /student-management/managed-students/
- ✓ /student-management/managed-students

Characteristics of resources: sub-collections (3/5)

A resource may contain sub-collection resources.

Examples:

- `/students/1/exams`
- `/students/1/exams/3`

Characteristics of resources: Hypermedia (4/5)

- The **media type** is the data format of a representation.
- The media type identifies a specification that defines how a representation is to be processed.

A RESTful API looks like hypertext: every addressable unit of information carries an address, either explicitly (e.g., link and ID attributes) or implicitly (e.g., derived from the media type definition and representation structure).

Characteristics of resources: Self-description (5/5)

- Resource representations shall be self-descriptive.
- The client does not need to know if a resource is an employee or a device.
- The client should act based on the media type associated with the resource.

Every media type defines a default processing model. For example, HTML defines a rendering process for hypertext and the browser behavior around each element.

Object Modeling

Identify the objects that will be presented as resources

Running example with three resources:

- **Students**
- **Courses** (refers to all the courses available to all the students)
- **Exams** (an exam is associated to a student)

where:

- Exam is a sub-resource of a student.
- A student can be associated to many exams.
- All objects/resources have a unique identifier, which is the integer `id` property.

Create Model URIs

```
/students
/students/{studId}

/courses
/courses/{courseId}

/exams
/exams/{examId}

/students/{studId}/exams
/students/{studId}/exams/{examId}
```

Determine Resource Representations (1/8)

Collection of students


```
{
    "count": 2,
    "total": 10234,
    "self-url": "/students",
    "students": [
        {
            "id": "12345",
            "self-url": "/students/12345",
            "first name": "John",
            "family name": "Doe",
            "birthdate": "1999-12-31",
            "graduated": false
        },
        {
            "id": "54321",
            "self-url": "/students/54321",
            "first name": "Jane",
            "family name": "Doe",
            "birthdate": "1999-01-01",
            "graduated": true
        }
    ]
}
```

Determine Resource Representations (2/8)

Single student resource

```
{
    "id": "12345",
    "self-url": "/students/12345",
    "first name": "John",
    "family name": "Doe",
    "birthdate": "1999-12-31",
    "graduated": false
    "exams": [
        {
            "id": "345",
            "self-url": "/exams/345",
            "course": "Robotics",
            "course-url": "/courses/1000",
            "date": "2022-02-18",
            "mark": 33
        },
        {
            "id": "349",
            "self-url": "/exams/349",
            "course": "Systems for Industry 4.0 and environment (IoT)",
            "course-url": "/courses/1001",
            "date": "2022-03-03",
            "mark": 33
        },
        ...
    ]
}
```

Determine Resource Representations (3/8)

Collection resource of courses


```
}
    "count": 2,
    "total": 1532,
    "self-url": "/courses",
    "courses": [
        {
            "id": "1000",
            "self-url": "/courses/1000",
            "title": "Robotics",
            "a/y": "2022-23",
            "teacher": "Tullio Facchinetti",
            "mandatory": false
        },
        {
            "id": "1001",
            "self-url": "/courses/1001",
            "title": "Systems for Industry 4.0 and environment (IoT)",
            "a/y": "2022-23",
            "course-url": "/courses/1001",
            "teacher": "Tullio Facchinetti",
            "mandatory": true
        }
    ]
}
```

Determine Resource Representations (4/8)

Collection resource of exams


```
{
    "count": 2,
    "total": 18451,
    "self-url": "/exams",
    "exams": [
        {
            "id": "345",
            "self-url": "/exams/345",
            "course": "Robotics",
            "course-url": "/courses/1000",
            "date": "2022-02-18",
        },
        {
            "id": "349",
            "self-url": "/exams/349",
            "course": "Systems for Industry 4.0 and environment (IoT)",
            "course-url": "/courses/1001",
            "date": "2022-03-03",
        },
        ...
    ]
}
```

Determine Resource Representations (5/8)

Single course resource

```
{
    "id": "1001",
    "self-url": "/courses/1000",
    "title": "Systems for Industry 4.0 and environment (IoT)",
    "a/y": "2022-23",
    "teacher": "Tullio Facchinetti",
    "laboratories": true,
    "computers required": true,
    "mandatory": true
}
```

Determine Resource Representations (6/8)

Single exam resource


```
{
    "id": "349",
    "self-url": "/exams/349",
    "course": "Systems for Industry 4.0 and environment (IoT)",
    "course-url": "/courses/1001",
    "date": "2022-03-03",
    "time": "9:30",
}
```

Determine Resource Representations (7/8)

Collection resource of exam under a single student


```
{
    "count": 2,
    "self-url": "/students/12345/exams",
    "exams": [
        {
            "self-url": "/students/12345/exams/345",
            "details": "/exams/345"
        },
        {
            "self-url": "/students/12345/exams/349",
            "details": "/exams/349"
        }
    ]
}
```

Determine Resource Representations (8/8)

Single exam under a single student


```
{
    "id": "349",
    "self-url": "/students/12345/exams/349",
    "course": "Systems for Industry 4.0 and environment (IoT)",
    "exam-url": "/exam/349",
    "date": "2022-03-03",
    "mark": 33
}
```

Methods of RESTful services

Method	Safe	Idempotent	Description
GET	Y	Y	retrieves a representation of a valid resource
POST	N	N	process a representation of a given request
PUT	N	Y	update/create a resource identified by a request URI
DELETE	N	Y	delete a resource identified by the requested URI

- **Safety**: a request does not change the state of the system.
- **Idempotency**: multiple identical requests has the same effect as making a single request.

Define HTTP calls and endpoints (1/6)

Access a list of primary resources

```
HTTP GET /students
HTTP GET /courses
HTTP GET /exams
```

If the collection size is large, paging and filtering can be applied. For example, the following requests will fetch the first 10 records from the collections:

```
HTTP GET /students?startIndex=0&size=10
HTTP GET /courses?startIndex=0&size=10
HTTP GET /exams?startIndex=0&size=10
```

The `total` field in the answer allows to evaluate the number of queries required to retrieve all the information.

Define HTTP calls and endpoints (2/6)

Browse all exams under a student

```
HTTP GET /students/{studId}/exams
```

Browse a specific resource

```
HTTP GET /students/{studId}
HTTP GET /courses/{courseIf}
HTTP GET /exams/{examId}
```

Browse a single exam under a student

```
HTTP GET /students/{studId}/exams/{examId}
```

Define HTTP calls and endpoints (3/6)

Create an element of a primary resource

```
HTTP POST /students
HTTP POST /courses
HTTP POST /exams
```

- The HTTP POST method is not idempotent, thus it is fine for this purpose
- The request does not need to specify any `id`, which will be assigned by the service

Define HTTP calls and endpoints (4/6)

Update a primary resource

```
HTTP PUT /students/{studId}
HTTP PUT /courses/{courseId}
HTTP PUT /exams/{examId}
```

- The HTTP PUT method is idempotent, thus it is fine for this purpose

Define HTTP calls and endpoints (5/6)

Remove a primary resource

```
HTTP DELETE /students/{studId}
HTTP DELETE /courses/{courseId}
HTTP DELETE /exams/{examId}
```

- A response for a successful operation *should be* 202 (Accepted) if the resource has been queued for deletion (*async operation*), or 200 (OK) / 204 (No Content) if the resource has been deleted permanently (*sync operation*).
- In case of async operation, the application shall return a `task id` that can be tracked for success/failure status.
- Usually, a *soft delete* is preferable, i.e., where a resource is set its status as *DELETED* instead of being actually removed.

Define HTTP calls and endpoints (6/6)

Apply a to an exam under a student

`HTTP PUT /students/{studId}/exams/{examId}`

Remove an exam under a student

`HTTP DELETE /students/{studId}/exams/{examId}`