**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*What is Contract\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

In the context of programming and software development, a contract can refer to a few different concepts:

Interface: In Java, an interface can be seen as a contract for a class. It defines a set of methods that the class must implement. This ensures that the class adheres to a certain standard and can be used interchangeably with other classes that implement the same interface.

Design by Contract (DbC): This is a software correctness methodology which uses preconditions, postconditions and invariants to define the responsibilities and benefits of a method or a class. It's a way of designing software such that the interactions between different parts of the system are specified in a precise and verifiable manner.

API Contract: This is a set of rules that describe how an API should behave. It includes details about the endpoints, request/response formats, error handling mechanisms, and more. This contract allows developers to understand how to interact with the API without needing to know the underlying implementation details.

Smart Contracts: In the context of blockchain and cryptocurrencies, a smart contract is a self-executing contract with the terms of the agreement directly written into code. They automatically execute transactions when certain conditions are met.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Data Format\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

A data format refers to the structure or organization of data, which allows it to be processed by a program. Data formats can be human-readable like XML, JSON, CSV, or binary formats like Protobuf, Avro, Parquet, etc.

**XML (eXtensible Markup Language):** It's a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It's often used for data exchange between applications.

**JSON (JavaScript Object Notation):** It's a lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate. It's often used in web applications for data exchange.

**CSV (Comma Separated Values):** It's a simple file format used to store tabular data, such as a spreadsheet or database. Each line of the file is a data record and each record consists of one or more fields, separated by commas.

**Protobuf (Protocol Buffers**): It's a binary format developed by Google to serialize structured data. It's smaller and faster than XML and JSON.

**Avro:** It's a binary data serialization system that relies on schemas for reading and writing data, providing compact, fast, binary data format.

**Parquet:** It's a columnar storage file format optimized for use with big data processing frameworks like Hadoop.

The choice of data format depends on the specific requirements of your application, such as the complexity of the data, the need for human readability, performance considerations, and so on.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*JSON Data Format\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

JSON (JavaScript Object Notation) is a lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999.

JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language**.**

**Here is an example of a JSON object:**

**{**

**"name": "John Doe",**

**"age": 30,**

**"isDeveloper": true,**

**"skills": ["Java", "Maven", "JSON"],**

**"address": {**

**"street": "123 Main St",**

**"city": "Anytown",**

**"state": "Anystate",**

**"zip": "12345"**

**}**

**}**

* In Java, you can use libraries like org.json, Gson, Jackson, etc., to parse and generate JSON data. Here is an example of how you can parse the above JSON using the org.json library:

import org.json.JSONObject;

public class Main {

public static void main(String[] args) {

String jsonString = "{ \"name\": \"John Doe\", \"age\": 30, \"isDeveloper\": true, \"skills\": [\"Java\", \"Maven\", \"JSON\"], \"address\": { \"street\": \"123 Main St\", \"city\": \"Anytown\", \"state\": \"Anystate\", \"zip\": \"12345\" } }";

String name = jsonObject.getString("name");

int age = jsonObject.getInt("age");

boolean isDeveloper = jsonObject.getBoolean("isDeveloper");

System.out.println("Name: " + name);

System.out.println("Age: " + age);

System.out.println("Is Developer: " + isDeveloper);

}

}

* In the above code, we first create a JSONObject from the JSON string. Then we use the getString, getInt, and getBoolean methods to extract the values from the JSON object.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*UseCases For JSON\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Web APIs:** JSON is commonly used to send data from a server to a web page or vice versa. It's a standard format for web APIs due to its lightweight nature and ease of use with JavaScript.

**Configuration Files**: JSON is often used for configuration files in modern programming environments. For example, package.json in Node.js projects or tsconfig.json in TypeScript projects.

**Data Storage:** JSON can be used as a format to store data. Some NoSQL databases like MongoDB use JSON-like documents.

**Data Serialization:** JSON is a common format for serializing data to be sent over a network, or for storing complex data structures for later use.

**Real-time Server-to-Browser Communication:** JSON is used in real-time server-to-browser communication. For example, it's used in WebSockets and Server-Sent Events.

**Data Exchange:** JSON is used for data exchange between client and server in web applications. It's a more compact and easy-to-use alternative to XML.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*XML Data Format\*\*\*\*\*Demo\*\*\*\*\*employee.xsd\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"

elementFormDefault="qualified" attributeFormDefault="unqualified">

<xs:simpleType name="ageNum">

<xs:restriction base="xs:positiveInteger">

<xs:maxInclusive value="100" />

<xs:minInclusive value="1" />

</xs:restriction>

</xs:simpleType>

<xs:element name="employees">

<xs:complexType>

<xs:sequence>

<xs:element name="employee">

<xs:complexType>

<xs:sequence>

<xs:element name="employeeName" type="xs:string" />

<xs:element name="age" type="ageNum" />

<xs:element name="salary" type="xs:integer" />

<xs:element name="address">

<xs:complexType>

<xs:sequence>

<xs:element name="city" />

<xs:element name="state" />

<xs:element name="country" />

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:sequence>

<xs:attribute name="eid" type="xs:integer"

use="required" />

</xs:complexType>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:schema>

<!--<xs:attribute name="lang" type="xs:string" fixed="EN"/> <xs:attribute

name="lang" type="xs:string" default="EN"/> -->

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*employee.xml\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

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

<employees xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="employee.xsd">

<employee eid="01" >

<employeeName>BillClinton</employeeName>

<age>23</age>

<salary>40000</salary>

<address>

<city>Pune</city>

<state>MS</state>

<country>India</country>

</address>

</employee>

</employees>

**\*\*\*\*\*\*\*\*\*\*\*Comparing JSON and XML\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Data Types:** JSON has a limited data types compared to XML. JSON types include: number, string, boolean, array, object, and null. XML data are all considered as strings.

**Readability:** Both XML and JSON are human-readable and machine-readable. However, JSON is often considered more readable than XML.

**Metadata:** XML supports attributes and thus can carry more metadata. JSON does not have a feature like XML attributes.

**Namespaces:** XML supports namespaces which can prevent name collisions when combining documents. JSON does not support namespaces.

**Parsing**: JSON parsing and serialization is typically faster than XML, because JSON's structure is simpler.

**APIs:** Many programming languages have APIs that can parse and generate JSON. XML parsers and APIs exist, but they can be more complex to use.

**Usage**: JSON is often used for asynchronous browser/server communication as its use with JavaScript is straightforward. XML is used in many technologies like RSS, SOAP, XHTML, SVG, etc.

**Comments**: JSON does not support comments, while XML does.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*URL -all parts\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

An HTTP URL (Uniform Resource Locator) is typically composed of several parts. Here are the different sections:

**Scheme**: This is the first part of the URL, and it defines the protocol to be used. For HTTP URLs, this is "http" or "https".

**Host**: This is the domain name or IP address of the server where the resource is located.

**Port**: This is optional and specifies the port number to access on the server. If not provided, the default port for HTTP is 80 and for HTTPS is 443.

**Path**: This is the specific resource on the server. It often represents a file structure where '/' separates directories and subdirectories.

**Query**: This is optional and starts with a '?'. It is used to send parameters to the server for processing.

**Fragment**: This is optional and starts with a '#'. It is used to specify a part or a position within the resource.

Here is an example of an HTTP URL with all its parts:

**https://www.example.com:8080/path/to/resource?query=value#fragment**

**In this URL:**

**"https" is the scheme.**

**"www.example.com" is the host.**

**"8080" is the port.**

**"/path/to/resource" is the path.**

**"query=value" is the query.**

**"fragment" is the fragment.**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Headers\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

HTTP headers are a fundamental component of data transfer on the internet. They allow the client and the server to pass additional information with an HTTP request or response. An HTTP header consists of its case-insensitive name followed by a colon ':', then by its value (without line breaks).

**There are four types of HTTP message headers:**

**General Header**: These header fields have general applicability for both request and response messages.

**Client Request Header**: These header fields have applicability only for request messages.

**Server Response Header:** These header fields have applicability only for response messages.

**Entity Header:** These header fields define meta information about the entity-body or, if no BODY is present, about the resource identified by the request.

**Here are some examples of HTTP headers:**

**Content-Type:** The Media type of the body of the request (used with POST and PUT requests).

**Accept**: The Media type(s) that is/are acceptable for the response.

**Authorization**: Authentication credentials for HTTP authentication.

**Cache-Control:** Used to specify directives that must be obeyed by all caching mechanisms along the request-response chain.

**User-Agent:** The user agent string of the user agent.

**Host**: The domain name of the server (for virtual hosting), and the TCP port number on which the server is listening.

Each header field consists of a name followed by a colon (":") and the field value. Field names are case-insensitive. The field value may be preceded by any amount of LWS, though a single SP is preferred.

---------------------------------Java Example----------------------------------

import java.io.OutputStream;

import java.net.HttpURLConnection;

import java.net.URL;

public class HttpClient {

public static void main(String[] args) throws Exception {

URL url = new URL("http://example.com/api/resource");

HttpURLConnection connection = (HttpURLConnection) url.openConnection();

// Set the request method

connection.setRequestMethod("POST");

// Set the request headers

connection.setRequestProperty("Content-Type", "application/json");

connection.setRequestProperty("Accept", "application/json");

connection.setRequestProperty("Cache-Control", "no-cache");

connection.setRequestProperty("User-Agent", "Mozilla/5.0");

connection.setRequestProperty("Host", "example.com");

// Enable input and output streams

connection.setDoOutput(true);

// Write the request body

String jsonInputString = "{\"name\": \"John\", \"age\": 30}";

try(OutputStream os = connection.getOutputStream()) {

byte[] input = jsonInputString.getBytes("utf-8");

os.write(input, 0, input.length);

}

// Get the response code

int responseCode = connection.getResponseCode();

System.out.println("Response Code : " + responseCode);

connection.disconnect();

}

}

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Query parameters\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Query parameters are a defined set of parameters attached to the end of a URL. They are extensions of the URL that are used to help define specific content or actions based on the data being passed. To append query parameters to the URL, you add a ? followed by a key=value pair. Multiple query parameters can be added by using & to separate them.

Here's an example of how you can use query parameters in a GET request using HttpURLConnection in Java:

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.net.HttpURLConnection;

import java.net.URL;

public class Main {

public static void main(String[] args) throws Exception {

String urlStr = "http://example.com/api/resource";

String queryParam = "?key1=value1&key2=value2";

URL url = new URL(urlStr + queryParam);

HttpURLConnection connection = (HttpURLConnection) url.openConnection();

// Set the request method

connection.setRequestMethod("GET");

// Get the response code

int responseCode = connection.getResponseCode();

System.out.println("Response Code : " + responseCode);

// Read the response

BufferedReader in = new BufferedReader(new InputStreamReader(connection.getInputStream()));

String inputLine;

StringBuilder content = new StringBuilder();

while ((inputLine = in.readLine()) != null) {

content.append(inputLine);

}

// Close connections

in.close();

connection.disconnect();

System.out.println("Response Content : " + content.toString());

}

}

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Path parameters\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

* Path parameters are variables within a URL path segment. They are used to point to a specific resource within a collection, such as an ID or a name.
* In this example, we're sending a GET request to http://example.com/api/resource/123. The path parameter 123 is used to identify a specific resource in the collection at **http://example.com/api/resource**. Here's an example of how you can use path parameters in a GET request using HttpURLConnection in Java:

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.net.HttpURLConnection;

import java.net.URL;

public class Main {

public static void main(String[] args) throws Exception {

String urlStr = "http://example.com/api/resource";

String pathParam = "/123"; // This is the path parameter

URL url = new URL(urlStr + pathParam);

HttpURLConnection connection = (HttpURLConnection) url.openConnection();

// Set the request method

connection.setRequestMethod("GET");

// Get the response code

int responseCode = connection.getResponseCode();

System.out.println("Response Code : " + responseCode);

// Read the response

BufferedReader in = new BufferedReader(new InputStreamReader(connection.getInputStream()));

String inputLine;

StringBuilder content = new StringBuilder();

while ((inputLine = in.readLine()) != null) {

content.append(inputLine);

}

// Close connections

in.close();

connection.disconnect();

System.out.println("Response Content : " + content.toString());

}

}

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Http status codes - Request and Response\*\*\*\*\*\*\*\*\*\*\*\*\***

* HTTP status codes are three-digit numbers that are returned by servers to indicate the status of a web element. There are five classes of HTTP status codes, each identified by the first digit:
* 1xx (Informational): The request was received, and the process is continuing.
* 2xx (Successful): The request was successfully received, understood, and accepted.
* 3xx (Redirection): Further action must be taken in order to complete the request.
* 4xx (Client Error): The request contains bad syntax or cannot be fulfilled.
* 5xx (Server Error): The server failed to fulfill an apparently valid request.

**Here are some common HTTP status codes:**

* 200 OK: The request has succeeded.
* 201 Created: The request has been fulfilled, and a new resource was created as a result.
* 204 No Content: The server has fulfilled the request but does not need to return an entity-body.
* 301 Moved Permanently: The requested resource has been assigned a new permanent URI.
* 400 Bad Request: The request could not be understood by the server due to malformed syntax.
* 401 Unauthorized: The request requires user authentication.
* 403 Forbidden: The server understood the request, but is refusing to fulfill it.
* 404 Not Found: The server has not found anything matching the Request-URI.
* 500 Internal Server Error: The server encountered an unexpected condition which prevented it from fulfilling the request.
* In Java, you can get the HTTP status code from an HttpURLConnection object using the getResponseCode() method. Here's an example:

import java.net.HttpURLConnection;

import java.net.URL;

public class Main {

public static void main(String[] args) throws Exception {

URL url = new URL("http://example.com");

HttpURLConnection connection = (HttpURLConnection) url.openConnection();

// Set the request method

connection.setRequestMethod("GET");

// Get the response code

int responseCode = connection.getResponseCode();

System.out.println("Response Code : " + responseCode);

connection.disconnect();

}

}

**---------------------SpringBoot XML-------------------------**

<https://www.youtube.com/watch?v=35EQXmHKZYs>

* Content Negotiation In postman
* Spring Boot by defaylt support json.
* How to make spring boot to suooport xml
* Add XML dependency in pom.xml
* And after adding xml dependency you will find that in postman restcontroller is accepting xml data

<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.dataformat/jackson-dataformat-xml -->

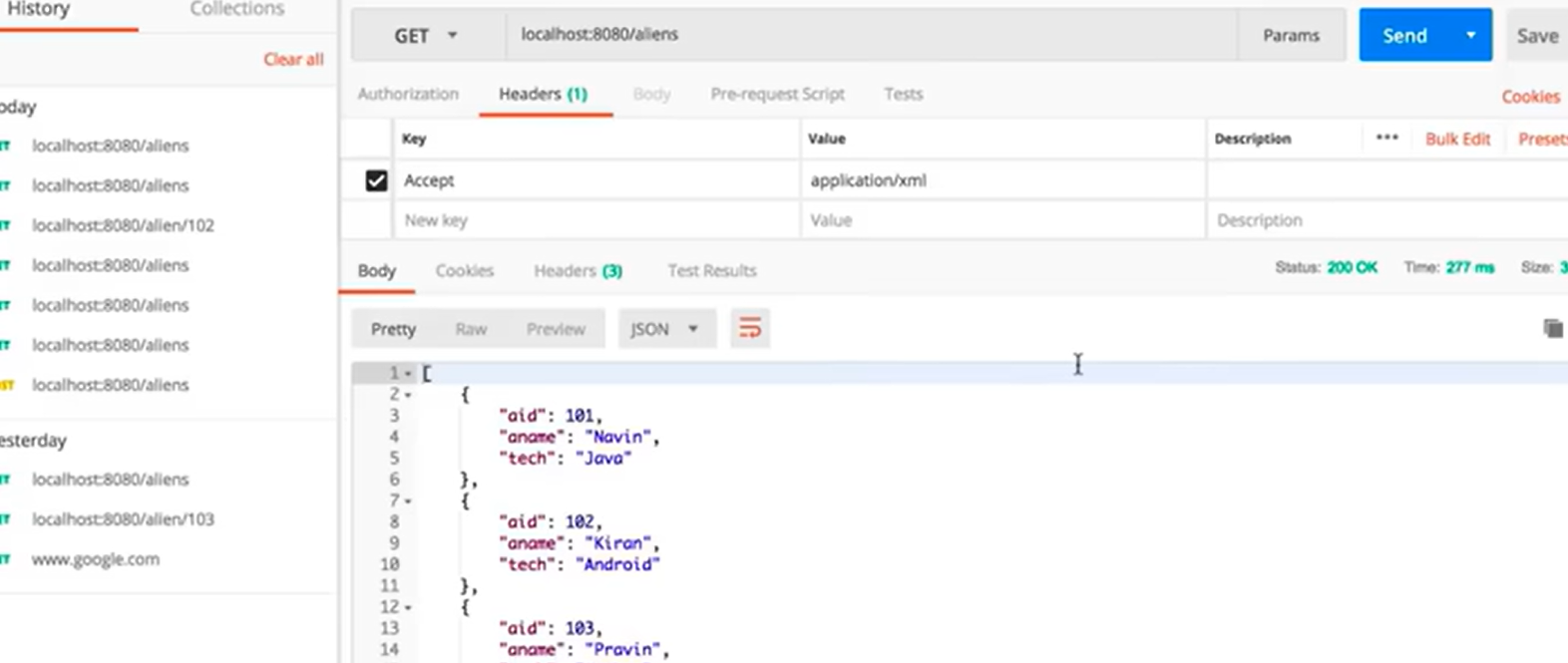
<dependency>

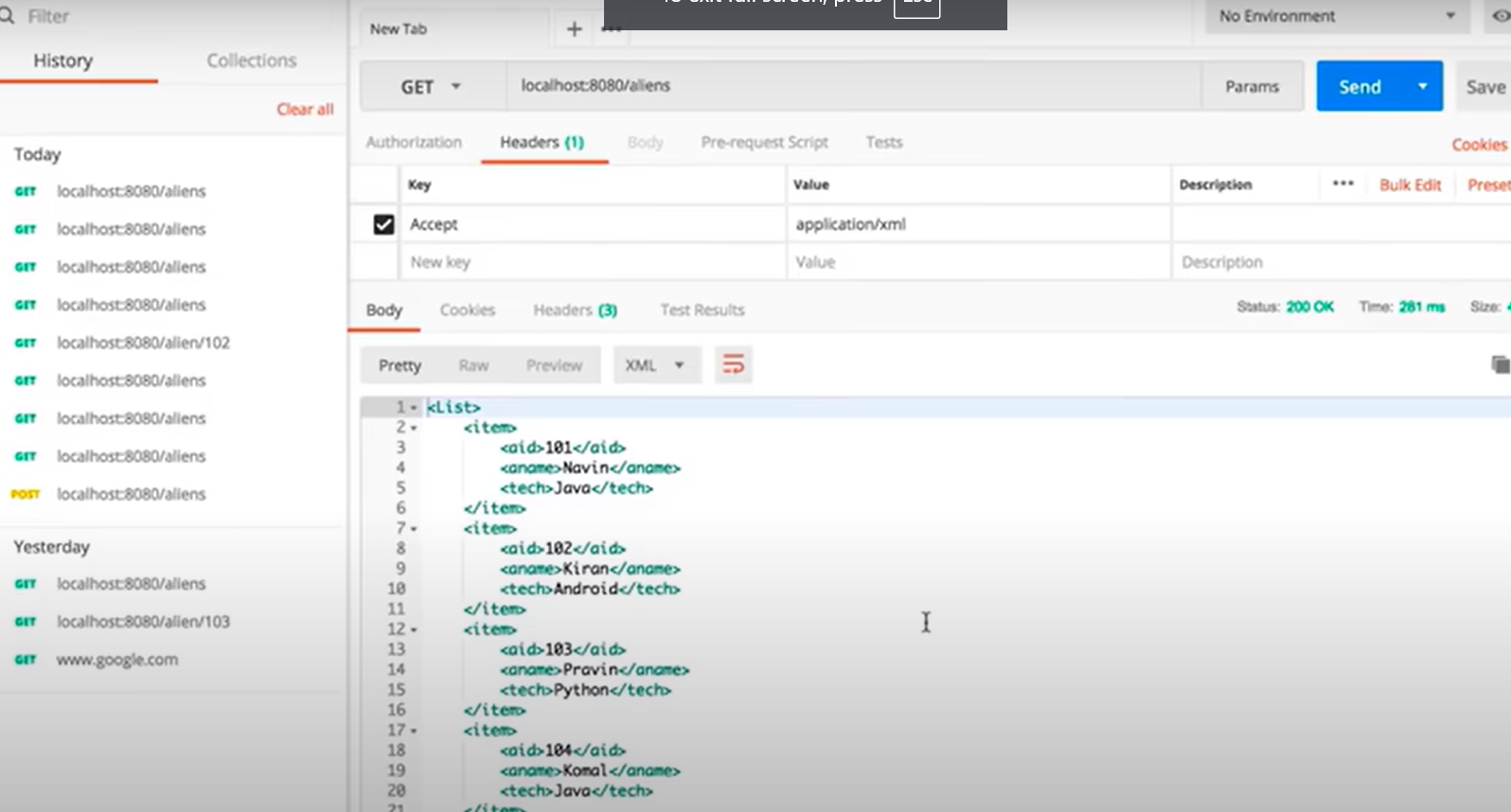
<groupId>com.fasterxml.jackson.dataformat</groupId>

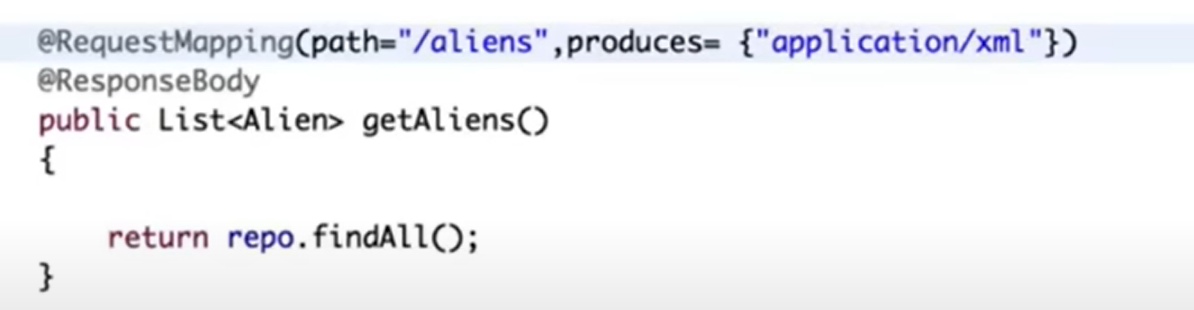
<artifactId>jackson-dataformat-xml</artifactId>

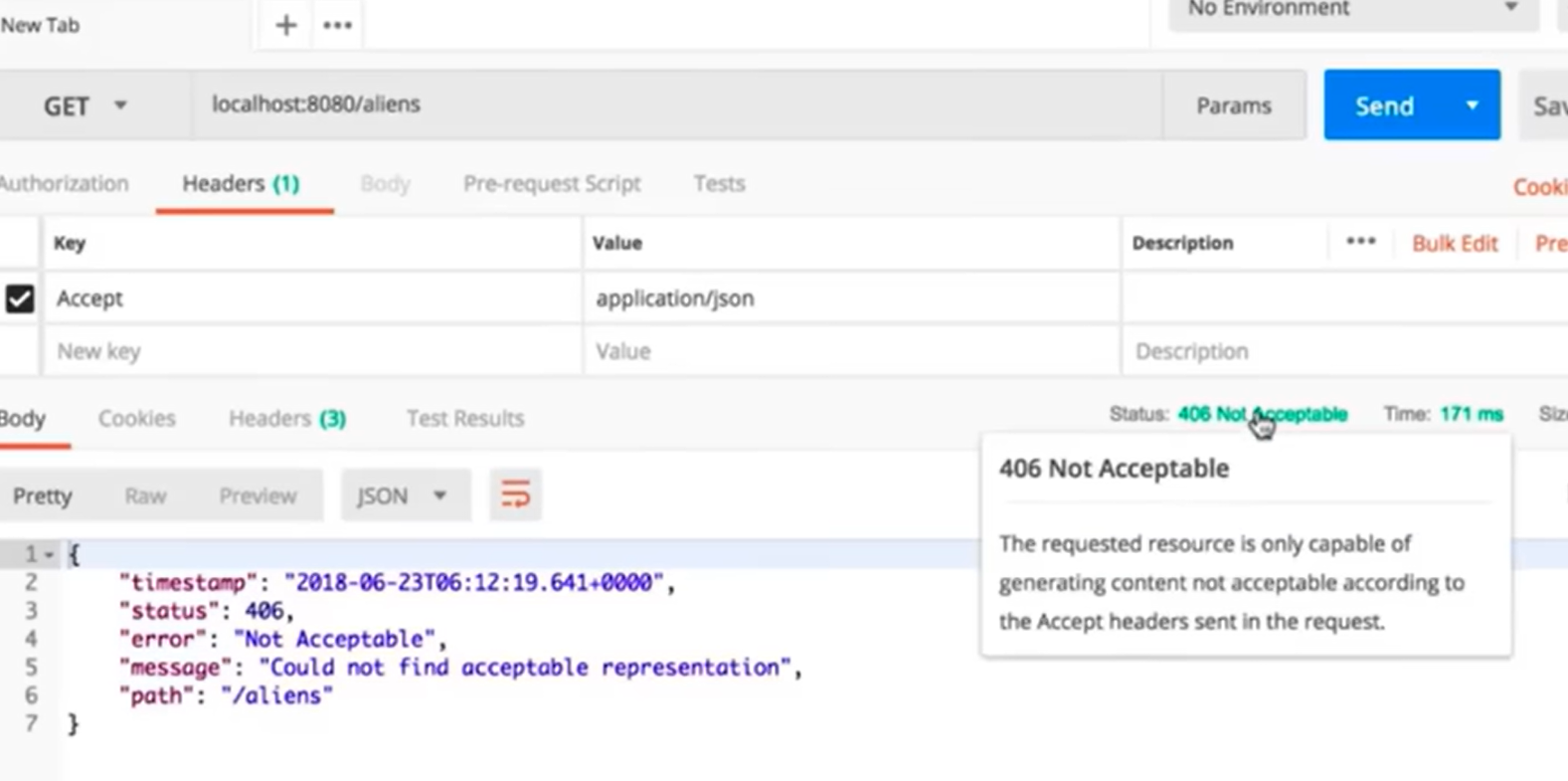
<version>2.17.2</version>

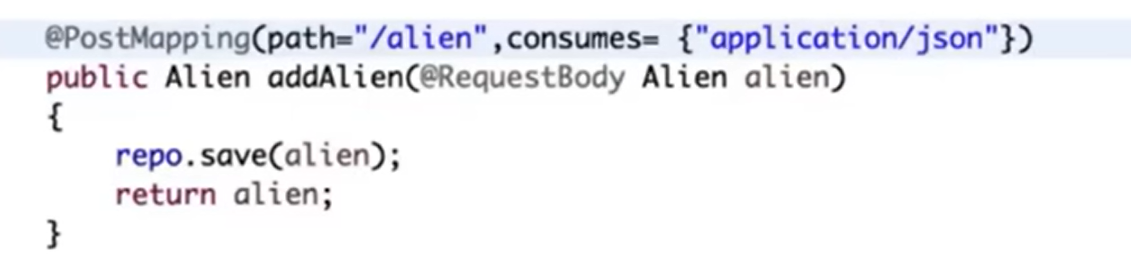
</dependency>

****

****

* In above case spring boot will support both json/xml/
* If we don’t want json to be supported and want to produce only xml then you can add following in the spring rest controller code
* 





\*\*\*\*\*\*\*\*\*\*\*Tools\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Insomnia\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<https://docs.insomnia.rest/insomnia/get-started>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*Postman\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*