REST

What is REST ?

REST stands for REpresentational State Transfer.

REST is web standards based architecture and uses HTTP Protocol for data communication.

It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods.

REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and presents the resources.

Here each resource is identified by URIs

REST uses various representations to represent a resource like text, JSON and XML.
Most popular light weight data exchange format used in web services = JSON

HTTP Methods

GET - Provides a read only access to a resource.

POST - Used to create a new resource.

DELETE - Used to remove a resource.

PUT - Used to update a existing resource or create a new resource.

PATCH -- Used to perform partial updates to the resource.

REST API Meaning

REST stands for REpresentational State Transfer.
API stands for Application Program Interface.

REST is a software architectural style that defines the set of rules to be used for creating web services(that are resource oriented)

Web services which follow the REST architectural style are known as RESTful web services.

It allows requesting systems)(REST Client) to access and manipulate web resources by using a uniform and predefined set of rules , using standard Hypertext Transfer Protocol (HTTP).

What does a Restful web system consists of

1. client who requests for the resources.
2. server who has the resources.

It is MANDATORY to create REST API as per industry standards which results in ease of development and increase client support.

Architectural Constraints of RESTful API

Uniform Interface
Stateless
Cacheable
Client-Server
Layered System
Code on Demand
The only optional constraint of REST architecture is code on demand. If a service violates any other constraint, it cannot strictly be referred to as RESTful.

Details :

What is Restful Web Service?

REST is used to build Web services that are lightweight, maintainable, and scalable in nature. A service which is built on the REST architecture is called a RESTful service. The underlying protocol for REST is HTTP, which is the basic web protocol. REST stands for REpresentational State Transfer
----

The key elements of a RESTful implementation are as follows:

1. Resources The first key element is the resource itself. Let assume that a web application on a server has records of several employees. Let's assume the URL of the web application is http://www.server.com. Now in order to access an employee record resource via REST, one can issue the command http://www.server.com/employee/1 - This command tells the web server to please provide the details of the employee whose employee number is 1.
----
2. Request Verbs - These describe what you want to do with the resource. A browser issues a GET verb to instruct the endpoint it wants to get data. However, there are many other verbs available including things like POST, PUT, and DELETE. So in the case of the example http://www.server.com/employee/1 , the web browser is actually issuing a GET Verb because it wants to get the details of the employee record.
----
3. Request Headers These are additional instructions sent with the request. These might define the type of response required or the authorization details.
----
4. Request Body - Data is sent with the request. Data is normally sent in the request when a POST request is made to the REST web service. In a POST call, the client actually tells the web service that it wants to add a resource to the

server. Hence, the request body would have the details of the resource which is required to be added to the server.

----

5. Response Body This is the main body of the response. So in our example, if we were to query the web server via the request http://www.server.com/employee/1 , the web server might return an XML document with all the details of the employee in the Response Body.

----

6. Response Status codes These codes are the general codes which are returned along with the response from the web server. An example is the code 200 which is normally returned if there is no error when returning a response to the client.

The six architectural constraints of REST APIs

1. Client-server architecture
An API’s job is to connect two pieces of software without limiting their own functionalities. This objective is one of the core restrictions of REST: the client (that makes requests) and the server (that gives responses) stay separate and independent.

When done properly, the client and server can update and evolve in different directions without having an impact on the quality of their data exchange. This is especially important in various cases where there are plenty of different clients a server has to cater to. Think about weather APIs – they have to send data to tons of different clients (all types of mobile devices are good examples) from a single database.

----

2. Statelessness
For an API to be stateless, it has to handle calls independently of each other. Each API call has to contain the data and commands necessary to complete the desired action.

An example of a non-stateless API would be if, during a session, only the first call has to contain the API key, which is then stored server-side. The following API calls depend on that first one since it provides the client’s credentials.

In the same case, a stateless API will ensure that each call contains the API key and the server expects to see proof of access each time.

Stateless APIs have the advantage that one bad or failed call doesn’t derail the ones that follow.

----

3. Uniform Interface
While the client and the server change in different ways, it’s important that the API can still facilitate communication. To that end, REST APIs impose a uniform interface that can easily accommodate all connected software.

In most cases, that interface is based on the HTTP protocols. Besides the fact that it sets rules as to how the clients and the server may interact, it also has the advantage of being widely known and used on the Internet. Data is stored and exchanged through JSON files because of their versatility.
----

4. Layered system
To keep the API easy to understand and scale, RESTful architecture dictates that the design is structured into layers that operate together.

With a clear hierarchy for these layers, executing a command means that each layer does its function and then sends the data to the next one. Connected layers communicate with each other, but not with every component of the program. This way, the overall security of the API is also improved.

If the scope of the API changes, layers can be added, modified, or taken out without compromising other components of the interface.
----

5. Cacheability
It’s not uncommon for a stateless API’s requests to have large overhead. In some cases, that’s unavoidable, but for repeated requests that need the same data, caching said information can make a huge difference.

The concept is simple: the client has the option to locally store certain pieces of data for a predetermined period of time. When they make a request for that data, instead of the server sending it again, they use the stored version.

The result is simple: instead of the client sending several difficult or costly requests in a short span of time, they only have to do it once.
----

6. Code on Demand
Unlike the other constraints we talked about up to this point, the last one is optional. The reason for making “code on demand” optional is simple: it can be a large security risk.

The concept is to allow code or applets(now obsolete!) to be sent through the API and used for the application. As you can imagine, unknown code from a shady source could do some damage, so this constraint is best left for internal APIs where you have less to fear from hackers and people with bad intentions. Another drawback is that the code has to be in the appropriate programming language for the application, which isn’t always the case.

The upside is that “code on demand” can help the client implement their own features on the go, with less work being necessary on the API or server. In essence, it permits the whole system to be much more scalable and agile.

RestController vs MVC Controller n Annotations

A key difference between a traditional MVC controller and the RESTful web service controller is the way that the HTTP response body is created.Instead of relying on a view technology(JSP / Thymeleaf) to perform server-side rendering of the data to HTML, typically a RESTful web service controller simply populates and returns a java object. The object data will be written directly to the HTTP response as JSON/XML/Text

To do this, the @ResponseBody annotation on the ret type of the request handling method tells Spring MVC that it does not need to render the java object through a server-side view layer.

Instead it tells that the java object returned is the response body, and should be written out directly.

The java object must be converted to JSON. Thanks to Spring's HTTP message converter support, you don't need to do this conversion manually. Because Jackson Jar is on the classpath, SC can automatically convert the java object to JSON & vice versa (using 2 annotations @ReponseBody & @RequestBody)

API --Starting point
o.s.http.converter.HttpMessageConverter<T>
--Interface that specifies a converter that can convert from and to HTTP requests and responses.
T --type of request/response body.

Implementation classes
1. o.s.http.converter.xml.Jaxb2RootElementHttpMessageConverter
-- Implementation of HttpMessageConverter that can read and write XML using JAXB2.(Java architecture for XML binding)

2. o.s.http.converter.json.
MappingJackson2HttpMessageConverter
--Implementation of HttpMessageConverter that can read and write JSON using Jackson 2.x's ObjectMapper class API

Important Annotations
1. @ResponseBody

Applied at : return value of the request handling method , annotated with @RequestMapping or @GetMapping / @PostMapping / @PutMapping / @DeleteMapping

It is used to marshall(serialize) the return value into the HTTP response body. Spring comes with converters that convert the Java object into a format understandable for a client(text/xml/json)

eg :

```
@Controller
@RequestMapping("/employees")
public class EmpController
{
   @GetMapping(....)
   public @ResponseBody Emp fetchEmpDetails(int empId)
   {
      //get emp dtls from DB through layers
      return e;
   }
}
```
----

2. @RestController
Class level annotation

Good news is @RestController = @Controller(at the class level) + @ResponseBody added on ret types of ALL request handling methods

eg :
```
@RestController
@RequestMapping("/employee")
public class EmpController
{
   @GetMapping(....)
   public Emp fetchEmpDetails(int empId)
   {
      //get emp dtls from DB through layers
      return e;
   }.....
}
```
----

3. @PathVariable --- handles URI templates.(URI variables or path variables)
Where to apply : on the method argument
Purpose : to access a path variable

eg : URL -- http://host:port/products/1234

Method of ProductController
```
@RestController
@RequestMapping("/products") {
@GetMapping("/{pid}")
public Product getDetails(@PathVariable(name="pid") int pid1234)
{...}
}
```
In the above URL , the path variable {pid} is mapped to an int . Therefore all of the URIs such as /products/1 or /products/10 will map to the same method in the controller.

----

4. The @RequestBody annotation, unmarshalls the HTTP request body into a Java object injected in the method.

Applied on the method argument of the reuqest handling methods , containing request body
eg : Typically in POST , PUT , PATCH

@RequestBody must be still added on a method argument of request handling method , for un marshaling(de serialization)
----
5. @CrossOrigin
Class/Method level annotation

What is CORS ?
Cross-Origin Resource Sharing (CORS)

CORS is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading of resources.

A cross-origin HTTP request is a request to a specific resource, which is located at a different origin, namely a domain, protocol and port, than the one of the client performing the request.

For security reasons, browsers can request several cross-origin resources, including images, CSS, JavaScript files etc. By default, a browser' security model will deny any cross-origin HTTP request performed by client-side scripts.

While this behavior is desired, to prevent different types of Ajax-based attacks, sometimes we need to instruct the browser to allow cross-origin HTTP requests from JavaScript clients with CORS.

eg : React client running on http://localhost:3000, and a Spring Boot RESTful web service API listening at http://host:port/products

In such a case, the client should be able to consume the REST API, which by default would be forbidden.

To make this work , enable CORS by simply annotating the class / methods of the RESTful web service API responsible for handling client requests with the @CrossOrigin annotation

```
eg : @CrossOrigin(origins = "http://localhost:3000")
@RestController
public class ProductController{....}
```


SOAP vs REST web services

SOAP stands for simple object access protocol

REST stands for REpresentational State Transfer

Protocol vs Architectural style
 SOAP is a standard protocol to create web services
Rest is architectural style to create web services.

Contract
Client and Server are bound with WSDL contract in SOAP
There is no contract between client and Server in REST

Format Support
SOAP supports only XML format
REST web services supports XML, json and plain text etc.

Maintainability
SOAP web services are hard to maintain as if we do any changes in WSDL , we need to create client stub again
REST web services are generally easy to maintain.

Service interfaces vs URI
SOAP uses Service interfaces to expose business logic
Rest uses URI to expose business logic

Security
SOAP has its own security : WS-security
Rest inherits its security from underlying transport layer.

Bandwidth
SOAP requires more bandwidth and resources as it uses XML messages to exchange information
REST requires less bandwith and resources. It can use JSON also.

Learning curve
SOAP web services are hard to learn as you need to understand WSDL , client stub.
REST web services are easy to understand as you need to annotate plain java class with JAX-RS annotations to use various HTTP methods .