**SPRING REST**

**Is REST scalable and/or interoperable?**

Yes, REST is Scalable and interoperable.

***REST is Scalable***

The server end of REST is stateless, which means that the server does not store any state about the client session on the server side. This means that there doesn't have to be (much) communication between servers, making it horizontally scalable.

The client session is stored on the client. The server is stateless means that every server can service any client at any time, there is no session affinity or sticky sessions. The relevant session information is stored on the client and passed to the server as needed.

The client's application state should never be stored on the server, but passed around from the client to every place that needs it.

That is where the ST in REST comes from, State Transfer. You transfer the state around instead of having the server store it. This is the only way to scale to millions of concurrent users. If for no other reason than because millions of sessions is millions of sessions.

***REST is interoperable***

It doesn't mandate a specific choice of technology either at client or server end. You can use Java, C++, Python or JavaScript to create RESTful Web Services and Consume them at the client end.

REST allows a greater variety of data formats, whereas SOAP only allows XML. Coupled with JSON (which typically works better with data and offers faster parsing), REST is generally considered easier to work with. Thanks to JSON, REST offers better support for browser clients.

**What is a stereotype annotation? What does that mean?**

When a class is annotated with one of the following Stereotypes Spring will automatically register them in the application context. This makes the class available for dependency injection in other classes and this become vital to building out our applications. These classes can all be found under the org.springframework.stereotype package. These annotations are used for auto-detection of beans using @ComponentScan.

* @Component Indicates that an annotated class is a "component"
* @Controller Indicates that an annotated class is a "Controller"
* @Service Indicates that an annotated class is a "Service" - "an operation offered as an interface that stands alone in the model, with no encapsulated state."
* @Repository Indicates that an annotated class is a "Repository" - "a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects".

@Component is a generic stereotype annotation for any Spring-managed component. This means that if you want to register a class in the application context you can use @Component to do so. The other stereotypes (Controller, Service, Repository) are simply specializations of the @Component class. In fact, if you look at their source code you will see they themselves are annotated with @Component. If this is true, why wouldn't we just annotate everything with @Component? First, it helps us and other developers classify a particular class. We can look at a class with the @Controller annotation and understand right away what its purpose is. Second, it also helps us define well crafted point-cut demarcations when we use AOP.

**Is @Controller a stereotype? Is @RestController a stereotype?**

@Controller is a stereotype annotation which indicates that an annotated class is a "Controller".

@RestController is a convenience annotation that is itself annotated with @Controller and @ResponseBody.

**What is difference between @Configuration and @Component?**

* @Configuration Indicates that a class declares one or more @Bean methods and may be processed by the Spring container to generate bean definitions and service requests for those beans at runtime
* @Component Indicates that an annotated class is a "component". Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.
* @Configuration is meta-annotated with @Component, therefore @Configuration classes are candidates for component scanning.
* @Configuration is also a @Component but a @Component cannot act like a @Configuration.
* @Configuration - It is like beans.xml but java based bean configuration. It means class annotated with this annotation is the place where beans are configured which will be candidate for auto-detection. In this class, methods are annotated with @Bean which return object of the class.

**Do you need Spring MVC in your classpath for developing RESTful Web Service?**

The short answer is: yes — you need Spring MVC in your Java application's classpath to develop RESTful web services using the Spring framework.

It's actually Spring MVC that provides all useful annotations, like @RestController, @ResponseCode , @ResponseBody, @RequestBody, and @PathVariable. Hence, you must use spring-mvc.jar or the appropriate Maven entry in your pom.xml.

The spring-mvc.jar is not part of spring-core, which means that if you want to use Spring MVC framework or RESTful web service in your Java project, you must include spring-mvc.jar in your application's classpath.

**What is an HttpMessageConverter in Spring REST?**

An HttpMessageConverter is a Strategy interface that specifies a converter that can convert from and to HTTP requests and responses. Spring REST uses this interface to convert HTTP response to various formats e.g. JSON or XML.

Each HttpMessageConverter implementation has one or several MIME Types associated with it. Spring uses the "Accept" header to determine the content type client is expecting.

It will then try to find a registered HTTPMessageConverter that is capable of handling that specific content-type and use it to convert the response into that format before sending to the client.

**How to create a custom implementation of HttpMessageConverter to support a new type of request/responses?**

You just need to create an implementation of AbstractHttpMessageConverter and register it using the WebMvcConfigurerAdapter#configureMessageConverters() method with the classes which generate a new type of request/response.

If there is a need to customize the message converters in some way, Spring Boot makes it simple. As an example consider if the POST method needs to be little more flexible and should ignore properties which are not present in the Hotel entity - typically this can be done by configuring the Jackson ObjectMapper, all that needs to be done with Spring Boot is to create a new HttpMessageConverter bean and that would end up overriding all the default message converters, this way:

***@Bean***

***public MappingJackson2HttpMessageConverter mappingJackson2HttpMessageConverter() {***

***MappingJackson2HttpMessageConverter jsonConverter = new MappingJackson2HttpMessageConverter();***

***ObjectMapper objectMapper = new ObjectMapper();***

***objectMapper.configure(DeserializationFeature.FAIL\_ON\_UNKNOWN\_PROPERTIES, false);***

***jsonConverter.setObjectMapper(objectMapper);***

***return jsonConverter;***

***}***

This works well for a Spring-Boot application, however for straight Spring MVC applications which do not make use of Spring-Boot, configuring a custom converter is a little more complicated. In the above MVC configuration file, we are using @EnableWebMvc annotation, it automatically registered default Http message converters with application as listed above according to available library in the class path. If there is a need to customize the default message converters in some way. Spring provides a WebMvcConfigurerAdapter class, which allow us to change the default list of Http Converters with our own.

***@Configuration***

***@EnableWebMvc***

***@ComponentScan("com.doj.restapi.web.controller")***

***public class WebConfiguration extends WebMvcConfigurerAdapter{***

***@Override***

***public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {***

***converters.add(createXmlHttpMessageConverter());***

***converters.add(new MappingJackson2HttpMessageConverter());***

***super.configureMessageConverters(converters);***

***}***

***private HttpMessageConverter<Object> createXmlHttpMessageConverter() {***

***MarshallingHttpMessageConverter xmlConverter = new MarshallingHttpMessageConverter();***

***XStreamMarshaller xstreamMarshaller = new XStreamMarshaller();***

***xmlConverter.setMarshaller(xstreamMarshaller);***

***xmlConverter.setUnmarshaller(xstreamMarshaller);***

***return xmlConverter;***

***}***

***}***

**What are the advantages of the RestTemplate?**

Basically, RestTemplate is used to make HTTP REST Calls (REST Client). The RestTemplate is the basic Spring class for simultaneous client-side HTTP access. It simplifies the interaction with HTTP servers and enforces RESTful systems. It is very related to the JdbcTemplate, JmsTemplate and the various other templates.

If we want to make an HTTP Call, we need to create an HttpClient, pass request and form parameters, setup accept headers and perform unmarshalling of response, all by yourself, Spring Rest Templates tries to take the pain away by abstracting all these details from you.

It is thread-safe, once created you can use it as a callbacks to customize its works.

We can configure RestTemplate to time out. ReadTimeOut and ConnectionTimeOut.

***Support to URI Templates -***

Every methods of URI takes as initial arguments. That can be a URI template and variables can be used to enlarge the template to a normal URI. The template variables are passed in two forms

1. String variable arguments array,
2. Map<String, String>.

The string variable-arguments alternatively expand the given template variables.

***E.g.***

***String result = restTemplate.getForObject("http://example.com/car/{car}/bookings/{booking}", String.class, "4355", "21");***

It will perform a GET call on <http://example.com/car/4355/bookings/21>. The map we used expands the template based on variable name and is more useful when using many variables, or when a single variable is used multiple times. For example:

***Map<String, String> map = new HashMap<String, String>();***

***map.put("car", "4355");***

***map.put("booking", "21");***

***String result = restTemplate.getForObject("http://example.com/car/{car}/bookings/{booking}", String.class, map);***

**What are default Message Converters in Spring MVC?**

In Spring MVC Framework, there is a set of default converters automatically registered which supports a whole range of different resource representation formats – json, xml for object. In RestTemplate in Spring REST, objects passed to and returned from the methods of RestTemplate class like getForObject(), postForLocation(), and put() are converted to HTTP requests and from HTTP responses by registered HttpMessageConverters.

* ***StringHttpMessageConverter:*** it converts Strings from the HTTP request and response.
* ***FormHttpMessageConverter:*** it converts form data to/from a MultiValueMap<String, String>.
* ***ByteArrayHttpMessageConverter:*** it converts byte arrays from the HTTP request and response.
* ***MappingJackson2HttpMessageConverter:*** it converts JSON from the HTTP request and response.
* ***Jaxb2RootElementHttpMessageConverter:*** it converts Java objects to/from XML.
* ***SourceHttpMessageConverter:*** it converts javax.xml.transform.Source from the HTTP request and response.
* ***AtomFeedHttpMessageConverter:*** it converts Atom feeds.
* ***RssChannelHttpMessageConverter:*** it converts RSS feeds.

**What is ResponseEntity?**

ResponseEntity is subclass of HttpEntity.

HttpEntity Represents an HTTP request or response entity, consisting of headers and body. Used in RestTemplate as well @Controller methods.

***public class ResponseEntity<T> extends HttpEntity<T>***

Extension of HttpEntity that adds a HttpStatus status code. Used in RestTemplate as well @Controller methods.

In RestTemplate, this class is returned by getForEntity() and exchange():

***ResponseEntity<String> entity = template.getForEntity("https://example.com", String.class);***

***String body = entity.getBody();***

***MediaType contentType = entity.getHeaders().getContentType();***

***HttpStatus statusCode = entity.getStatusCode();***

It can also be used in Spring MVC, as the return value from a @Controller method:

***@RequestMapping("/handle")***

***public ResponseEntity<String> handle() {***

***URI location = ...;***

***HttpHeaders responseHeaders = new HttpHeaders();***

***responseHeaders.setLocation(location);***

***responseHeaders.set("MyResponseHeader", "MyValue");***

***return new ResponseEntity<String>("Hello World", responseHeaders, HttpStatus.CREATED);***

***}***

Or, by using a builder accessible via static methods:

***@RequestMapping("/handle")***

***public ResponseEntity<String> handle() {***

***URI location = ...;***

***// location can also be extracted from current request instead of hardcoding***

***URI location = ServletUriComponentsBuilder***

***.fromCurrentRequest()***

***.path("/{id}")***

***.buildAndExpand(savedUser.getId()).toUri();***

***return ResponseEntity.created(location).header("MyResponseHeader", "MyValue").body("Hello World");***

***}***

**What is RequestEntity?**

RequestEntity is subclass of HttpEntity.

HttpEntity Represents an HTTP request or response entity, consisting of headers and body. Used in RestTemplate as well @Controller methods.

***public class RequestEntity<T> extends HttpEntity<T>***

Extension of HttpEntity that adds a method and uri. Used in RestTemplate and @Controller methods.

In RestTemplate, this class is used as parameter in exchange():

***MyRequest body = ...***

***RequestEntity<MyRequest> request = RequestEntity***

***.post(new URI("https://example.com/bar"))***

***.accept(MediaType.APPLICATION\_JSON)***

***.body(body);***

***ResponseEntity<MyResponse> response = template.exchange(request, MyResponse.class);***

If you would like to provide a URI template with variables, consider using UriTemplate.

***URI uri = new UriTemplate("https://example.com/{foo}").expand("bar"); // expands Uri as -*** [***https://example.com/bar***](https://example.com/bar)

***RequestEntity<MyRequest> request = RequestEntity.post(uri).accept(MediaType.APPLICATION\_JSON).body(body);***

It can also be used in Spring MVC, as a parameter in a @Controller method:

***@RequestMapping("/handle")***

***public void handle(RequestEntity<String> request) {***

***HttpMethod method = request.getMethod();***

***URI url = request.getUrl();***

***String body = request.getBody();***

***}***

**What is UriTemplate?**

A URI template is a URI-like String that contains variables enclosed by braces ({}) which can be expanded to produce an actual URI.

***public URI expand(Map<String,?> uriVariables)***

***UriTemplate template = new UriTemplate("https://example.com/hotels/{hotel}/bookings/{booking}");***

***Map<String, String> uriVariables = new HashMap<String, String>();***

***uriVariables.put("booking", "42");***

***uriVariables.put("hotel", "Rest & Relax");***

***System.out.println(template.expand(uriVariables));***

Will print:[***https://example.com/hotels/Rest%20%26%20Relax/bookings/42***](https://example.com/hotels/Rest%20%26%20Relax/bookings/42)

***public URI expand(Object... uriVariableValues)***

***UriTemplate template = new UriTemplate("https://example.com/hotels/{hotel}/bookings/{booking}");***

***System.out.println(template.expand("Rest & Relax", 42));***

will print: [***https://example.com/hotels/Rest%20%26%20Relax/bookings/42***](https://example.com/hotels/Rest%20%26%20Relax/bookings/42)

**What is @Valid annotation?**

@Valid annotation is present in javax.validation API which is part of Spring boot starter web. There are other validations in javax.validation API like @Size, @Past for date, @NotNull, @NotEmpty, @Future, @Email, @Positive, @Negative etc.

@Valid annotation is used on @RequestBody to validate incoming request data. In case of validation failure, handleMethodArgumentNotValid() method from ResponseEntityExceptionHandler class is called.

**What is HATEOS?**

HATEOS (Hypertext as the Engine of Application State) is the principle that the API should guide the client through the application by returning relevant information about the next potential steps, along with each response.

When a user requests a specific resource, server returns not only requested resource but also different related resources which enables user to perform other related operations. This is process of creating hypermedia-driven REST web service. With HATEOAS, the client can have a single entry point to the application and further actions can be taken based on the metadata in the response representation. This allows the server to change its URI scheme without breaking the client. Also, the application can advertise new capabilities by putting new links or URIs in the representation.

The Spring HATEOAS project is a library of APIs that we can use to easily create REST representations that follow the principle of HATEOAS.

Spring HATEOAS offers three abstractions for creating the URI – ResourceSupport, Link, and ControllerLinkBuilder

Spring Boot HATEOAS dependency:

***<dependency>***

***<groupId>org.springframework.boot</groupId>***

***<artifactId>spring-boot-starter-hateoas</artifactId>***

***<version>2.1.4.RELEASE</version>***

***</dependency>***

**What is Content Negotiation in REST?**

Key abstraction in REST is a Resource. There is no restriction on what can be a resource. A todo is a resource. A person on facebook is a resource.

A resource can have multiple representations

* XML
* HTML
* JSON

When a resource is requested, we provide the representation of the resource. When a consumer sends a request, it can specify two HTTP Headers related to Content Negotiation - Content-Type and Accept

* Content-Type indicates the content type of the body of the request.
* Accept indicates the expected content type of the response.

**Example of Content Negotiation**

The server is expected to respond based on the Accept header.

For example, if a consumer sends a request to <http://localhost:8080/students/10001> with Accept header as ‘application/xml’, we need to provide the xml representation of the resource.

***<Student>***

***<id>10001</id>***

***<name>Ranga</name>***

***<passportNumber>E1234567</passportNumber>***

***</Student>***

If a consumer sends a request with Accept header as ‘application/json’, we need to provide the JSON representation of the resource.

***{***

***"id": 10001,***

***"name": "Ranga",***

***"passportNumber": "E1234567"***

***}***

Similar concept applies to the Response Body Content based on the Content-Type.

A consumer can send a POST request to <http://localhost:8080/students> with Content-Type header as ‘application/xml’, and provide the XML representation of the resource to be created.

***<Student>***

***<name>Ranga</name>***

***<passportNumber>E1234567</passportNumber>***

***</Student>***

A consumer can also send a POST request to <http://localhost:8080/students> with Content-Type header as ‘application/json’, and provide the JSON representation of the resource to be created.

***{***

***"name": "Ranga",***

***"passportNumber": "E1234567"***

***}***

This dialogue which happens between the Consumer and Service Provider is called Content Negotiation.

**What is HAL and the HAL browser?**

JSON Hypertext Application Language, or HAL, is a simple format that gives a consistent and easy way to hyperlink between resources in our API. Including HAL within our REST API makes it much more explorable to users as well as being essentially self-documenting.

HAL is a generic media type with which Web APIs can be developed and exposed as series of links. Clients of these APIs can select links by their link relation type and traverse them in order to progress through the application.

It works by returning data in JSON format which outlines relevant information about the API.

The HAL model revolves around two simple concepts.

1) Resources, which contain:

* Links to relevant URIs
* Embedded Resources
* State

2) Links:

* A target URI
* A relation, or rel, to the link
* A few other optional properties to help with depreciation, content negotiation, etc

The HAL browser was created by the same person who developed HAL and provides an in-browser GUI to traverse your REST API.

**What are different Versioning techniques for RESTful Services?**

First of all build your services to backward compatible so that you can avoid versioning as much as possible. Just in case if it is required, there are 4 different ways it can be done.

***Versioning Approaches for RESTful Services -***

***1) URI Versioning***

Basic approach to versioning is to create a completely different URI for the new service. Example implementation is shown below.

***@RestController***

***public class StudentVersioningController {***

***@GetMapping("v1/student")***

***public StudentV1 studentV1() {***

***return new StudentV1("Bob Charlie");***

***}***

***@GetMapping("v2/student")***

***public StudentV2 studentV2() {***

***return new StudentV2(new Name("Bob", "Charlie"));***

***}***

<http://localhost:8080/v1/person>

Response

***{***

***"name": "Bob Charlie"***

***}***

<http://localhost:8080/v2/person>

Response

***{***

***"name": {***

***"firstName": "Bob",***

***"lastName": "Charlie"***

***}***

***}***

***2) Request Parameter versioning***

Next approach to versioning is to use the request parameter to differentiate versions.

Implementations are shown below:

***@GetMapping(value = "/student/param", params = "version=1")***

***public StudentV1 paramV1() {***

***return new StudentV1("Bob Charlie");***

***}***

***@GetMapping(value = "/student/param", params = "version=2")***

***public StudentV2 paramV2() {***

***return new StudentV2(new Name("Bob", "Charlie"));***

***}***

<http://localhost:8080/person/param?version=1>

Response

***{***

***"name": "Bob Charlie"***

***}***

<http://localhost:8080/person/param?version=2>

Response

***{***

***"name": {***

***"firstName": "Bob",***

***"lastName": "Charlie"***

***}***

***}***

***3) (Custom) Headers versioning***

The third approach to versioning is to use a Request Header to differentiate the versions.

Implementations are shown below:

***@GetMapping(value = "/student/header", headers = "X-API-VERSION=1")***

***public StudentV1 headerV1() {***

***return new StudentV1("Bob Charlie");***

***}***

***@GetMapping(value = "/student/header", headers = "X-API-VERSION=2")***

***public StudentV2 headerV2() {***

***return new StudentV2(new Name("Bob", "Charlie"));***

***}***

<http://localhost:8080/person/header> with a header - X-API-VERSION=1

Response

***{***

***"name": "Bob Charlie"***

***}***

<http://localhost:8080/person/header> with a header - X-API-VERSION=2

Response

***{***

***"name": {***

***"firstName": "Bob",***

***"lastName": "Charlie"***

***}***

***}***

***4) Media type versioning (a.k.a “content negotiation” or “accept header”)***

The last versioning approach is to use the Accept Header in the request.

***@GetMapping(value = "/student/produces", produces = "application/vnd.company.app-v1+json")***

***public StudentV1 producesV1() {***

***return new StudentV1("Bob Charlie");***

***}***

***@GetMapping(value = "/student/produces", produces = "application/vnd.company.app-v2+json")***

***public StudentV2 producesV2() {***

***return new StudentV2(new Name("Bob", "Charlie"));***

***}***

<http://localhost:8080/person/produces> with a header - Accept=application/vnd.company.app-v1+json

Response

***{***

***"name": "Bob Charlie"***

***}***

<http://localhost:8080/person/produces> with a header - Accept=application/vnd.company.app-v2+json

Response

***{***

***"name": {***

***"firstName": "Bob",***

***"lastName": "Charlie"***

***}***

***}***

The list below shows Major API providers using different versioning approaches.

1) URI Versioning

Twitter

2) Request Parameter versioning

Amazon

3) (Custom) Headers versioning

Microsoft

4) Media type versioning (a.k.a “content negotiation” or “accept header”)

GitHub