**Richardson Maturity Model** grades API by their RESTful maturity. It is proposed by **Leonard Richardson**. The Richardson maturity model is a way to grade your API according to the constraints of REST. It breaks down the principal element of the REST approach into **four** levels (0 to 3).

There are four levels:

* Level 0: The Swamp of POX
* Level 1: Resources
* Level 2: HTTP Verbs
* Level 3: Hypermedia Control

For example, a level higher is more RESTful compared to one that is at a lower level. Only when an API reaches level 4, we consider it as a RESTful API.

## Richardson Maturity ModelLevel 0: The Swamp of POX

The Swamp of POX (Plain Old XML) means that you’re using HTTP. Technically, REST services can be provided over any application layer protocol as long as they conform to certain properties. In practice, basically, everyone uses HTTP.

These services have a single URI and use a single HTTP method (typically POST). These are the most primitive way of building SOA applications with a single POST method and using XML to communicate between services.

Level zero of maturity does not make use of any of URI, HTTP Methods, and HATEOAS capabilities.

Even at level 0, there are many design rules that we need to consider:

**Rule #1: Hyphens (-) should be used to improve the readability of URIs - Also referred to as spinal-case**

**Rule #2: Underscores (\_) should not be used in URIs**

**Rule #3: Lowercase letters should be preferred in URI paths**

**Rule #4: File extensions should not be included in URIs**

## Level 1: Resources

REST’s ‘resources’ are the core pieces of data that your application acts on. These will often correspond to the Models in your application (especially if you’re following the MVC - model, view, controller pattern).

API design at Level 1 is all about using different URLs to interact with the different resources in your application.

First, I would like to clarify that using **verbs in URI which is bad practice** and I write then next only to demonstrate a point and to ease the way into the resources properly usage.

Let’s write few APIs for Blog which has some articles, to understand more. /getAllArticles is an API which will respond with the list of articles.

Few more APIs around a Blog will look like as follows:  
/addNewArticle  
/updateArticle  
/deleteArticle  
/deleteAllArticle  
/promoteArticle  
/promoteAllArticle

Even at this level, there are a few design rules that worth to remember:  
**Rule #1: A trailing forward-slash (/) should not be included in URIs**

**Rule #2: Forward slash separator (/) must be used to indicate a hierarchical relationship**

**Rule #3: Should the endpoint name be singular or plural?**

Not having to deal with odd pluralization (person/people, goose/geese) makes the life of the API consumer better and is easier for the API provider to implement (as most modern frameworks will natively handle /students and /students/3248234 under a common controller).

Remember the following points while building a URI:

* Use domain and subdomain to logically group or partition resources.
* Use **/** to indicate a hierarchical relationship.
* Use **,** and **;** to indicate non-hierarchical relationships.
* Use **-** and **\_** to improve the readability.
* Use **&** to separate parameters.
* Avoid including **file extensions**.

## Level 2: HTTP Verbs (method = GET/POST/DELETE/etc)

Level 2 indicates that an API must use the protocol properties to deal with scalability and failures. At level 2, correct **HTTP verbs** are used with each request. It suggests that in order to be truly RESTful, HTTP verbs must be used in API. For each of those requests, the correct HTTP response code is provided.

We don't use a single POST method for all requests. We use the **GET** method when we request a resource, and use the **DELETE** method when we want to delete a resource. Also, use the response codes of the application protocol.

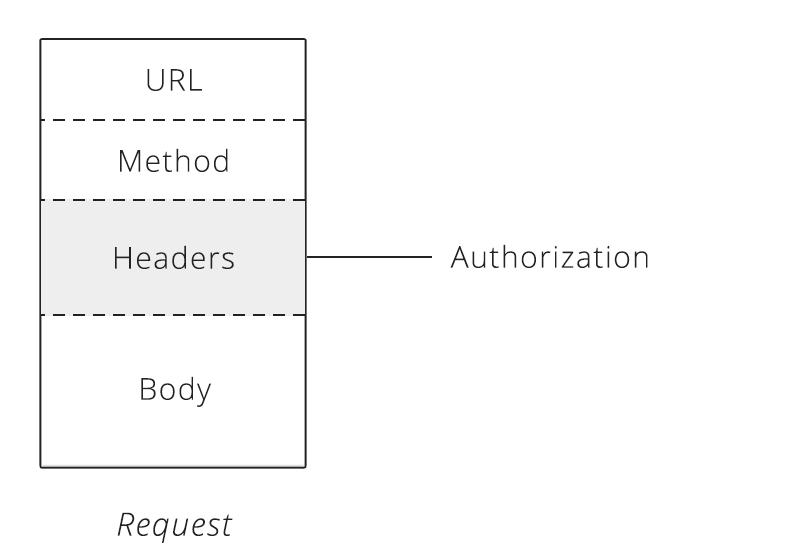
For example, to get the customers, we send a request with the URI http://localhost:8080/customers, and the server sends proper response **200 OK**.

The following table shows the HTTP verbs and their usage:

|  |  |  |
| --- | --- | --- |
| **Verbs** | **Safety & Idempotency** | **Usage** |
| GET | Y/Y | It retrieves the information. |
| POST | N/N | It is used to perform a variety of actions on the server, such as create a new resource, and update an existing resource, or making a mixture of changes to one or more resources. |
| DELETE | N/Y | It is used to delete a resource. |
| PUT | N/Y | It is used to update or replace an existing resource or to create a new resource with a URI specified by the client. |
| HEAD | Y/Y | It is used to retrieve the same headers as that of GET response but without any body in the response. |
| OPTIONS | Y/Y | It is used to find the list of HTTP methods supported by any resource or to ping the server. |
| TRACE | Y/Y | It is used for debugging, which echo's back headers that it has received. |

#### Level 2.1 - HTTP headers

They are needed in order to convey more data about the resource itself, mostly meta-data, security, hashes and more.



HTTP headers also provide the required information about the request or response, or as we said - about the object sent in the message body.

There are 4 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.

#### Level 2.2 - Query Parameters

Another important part of [REST API design](https://www.restcase.com/) is using query parameters.  
They are widely used in many cases but it is more likely that they are used in order to achieve some sort of searching, filtering, and querying.

**Paging** It is necessary to anticipate the paging of your resources in the early design phase of your API. It is indeed difficult to foresee precisely the progression of the amount of data that will be returned. Therefore, we recommend paginating your resources with default values when they are not provided by the calling client, for example with a range of values [0-25].

**Filtering** Filtering consists in restricting the number of queried resources by specifying some attributes and their expected values. It is possible to filter a collection on several attributes at the same time and to allow several values for one filtered attribute.

**Sorting** Sorting the result of a query on a collection of resources. A sort parameter should contain the names of the attributes on which the sorting is performed, separated by a comma.

**Searching** A search is a sub-resource of a collection. As such, its results will have a different format than the resources and the collection itself. This allows us to add suggestions, corrections, and information related to the search.

#### Level 2.3 - Status Codes

It is very important that as a RESTful API, you make use of the proper HTTP Status Codes, especially when [developing and mocking RESTful API](https://www.restcase.com/).

Common error HTTP status codes include:

* **200 – OK** Everything is working
* **201 – CREATED** A new resource has been created
* **204 – NO CONTENT** The resource was successfully deleted, no response body
* **304 – NOT MODIFIED** The date returned is cached data (data has not changed)
* 400 Bad Request – This means that client-side input fails validation.
* 401 Unauthorized – This means the user isn’t not authorized to access a resource. It usually returns when the user isn’t authenticated.
* 403 Forbidden – This means the user is authenticated, but it’s not allowed to access a resource.
* 404 Not Found – This indicates that a resource is not found.
* 500 Internal server error – This is a generic server error. It probably shouldn’t be thrown explicitly.
* 502 [Bad Gateway](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/502) – This indicates an invalid response from an upstream server.
* 503 Service Unavailable – This indicates that something unexpected happened on server side (It can be anything like server overload, some parts of the system failed, etc.).

Level 3: Hypermedia Controls

Level 3 is the highest level. It is the combination of level 2 and HATEOAS. It also provides support for HATEOAS. It is helpful in self-documentation.

This level is the one that everyone falls down on. There are two parts to this: content negotiation and HATEOAS. Content negotiation is focused on different representations of a particular resource, and HATEOAS is about the discoverability of actions on a resource.

**Content Negotiation is performed by an application:**

To match the requested representation as specified by the client via the Accept header with a representation the application can deliver.  
To determine the Content-Type of incoming data and deserialize it so the application can utilize it.

Essentially, content negotiation is the client telling the server what it is sending and what it wants in return, and the server determining if it can do what the client requests.

**Accept Negotiation**

The first aspect of content negotiation is handling the Accept header. The Accept header has one of the most complex definitions in the HTTP specification. With supplying this header in the request, a client can indicate a prioritized list of different media types that it will accept as responses from the server.

In practice, particularly with APIs, you will send a specific media type for the representation you can handle in your client. As an example:

GET /foo HTTP/1.1 Accept: application/json  
The above indicates that the client wants JSON for a response. It is now the server's responsibility to determine if it can return that representation.

If the server can not return JSON, it needs to tell the client that fact. This is done via the 406 Not Acceptable status code:

HTTP/1.1 406 Not Acceptable Ideally, the server will also indicate what media types it can return; however, it is not obligated to do so.

Because the server cannot return a representation for the requested media type, it can choose whatever media type it wants for the response in order to communicate errors.

If the server can return the requested media type, it should report the media type via the response Content-Type header.

One important point of interest: the same URI can potentially respond with multiple media types. This means that you could potentially make one request that specifies text/html, another with application/json, and get different representations of the same resource!

This is a very important aspect of content negotiation; one of the purposes is to allow many clients to the same resource, speaking in different protocols.

**Content-Type Negotiation**

The second aspect of content negotiation is identifying the incoming Content-Type header and determining if the server can deserialize that data.

As an example, the client might send the following:

POST /foo HTTP/1.1  
Accept: application/json  
Content-Type: application/json  
{ "foo": "bar" }

The server would introspect the Content-Type header and determine that JSON was submitted. Now it has to decide if it can deserialize that content. If it cannot, the server will respond with a 415 Unsupported Media Type status code:

HTTP/1.1 415 Unsupported Media Type

If the data submitted is not actually of the Content-Type specified, meaning it cannot be deserialized properly, the server will typically respond with a generic 400 Bad Request status.

###### HATEOAS

Hypermedia As Transfer Engine Of Application State is a constraint of the REST application architecture that distinguishes it from other network application architectures.

It provides ease of navigation through a resource and its available actions. This way a client doesn’t need to know how to interact with an application for different actions, as all the metadata will be embedded in responses from the server.

To understand it better let’s look at the below response of retrieve user with id 123 from the server:

{ "name": "John Doe", "links": [{ "rel": "self", "href": "http://localhost:8080/user/123" }, { "rel": "posts", "href": "http://localhost:8080/user/123/post" }, { "rel": "address", "href": "http://localhost:8080/user/123/address" } ] }

Sometimes it’s easier to skip the links format, and specify links as fields of a resource as below:

{ "name": "John Doe", "self": "http://localhost:8080/user/123", "posts": "http://localhost:8080/user/123", "address": "http://localhost:8080/user/123/address" }

It’s not a convention you need to follow every time, as it depends on resource fields/size, and actions which can be performed on a resource. If resources contain several fields that the user may not want to go through, it’s a good idea to show navigation to sub-resources then implement HATEOAS.

#### 3.1 - Versioning

I have decided to include the versioning design here since [REST API versioning](https://blog.restcase.com/restful-api-versioning-insights/) is a much more mature REST API model and for the reason that Content negotiation is focused on **different representations of a particular resource**, versioning can be called as a different representation of a resource.

Versioning APIs always helps to ensure backward compatibility of a service while adding new features or updating existing functionality for new clients.

There are different schools of thought to [version your API](https://blog.restcase.com/restful-api-versioning-insights/), but most of them fall under two categories below:

**Headers:**

There are 2 ways you can specify the version in headers:

**Custom Header:**

Adding a custom X-API-VERSION (or any other header of choice) header key by the client can be used by a service to route a request to the correct endpoint

**Accept Header**

Using accept header to specify your version such as

Accept: application/vnd.hashmapinc.v2+json

**URL:**

Embed the version in the URL such as

POST /v2/user

I prefer to use the URL method for versioning as it gives better discoverability of a resource by looking at the URL

# **What are idempotent operations? Why is idempotency important in REST API?**

[Safe method](http://restcookbook.com/HTTP%20Methods/idempotency/) doesn't change anything **internally** (resources) - Not perform any change on server end

Safe methods are methods that can be cached, prefetched without any repercussions to the resource.

[Idempotent method](http://restcookbook.com/HTTP%20Methods/idempotency/) doesn't change anything **externally** (response) – Perform some changes on Server but response should be same

idempotent HTTP method is a HTTP method that can be called many times without different outcomes.

HTTP methods include:

* **POST** – Creates a new resource. POST **is not** idempotent and it **is not** safe.
* **GET** – Retrieves a resource. GET **is** idempotent and it **is** safe.
* **HEAD** – Retrieves a resource (without response body). HEAD **is** idempotent and it **is** safe
* **PUT** – Updates/replaces a resource. PUT **is** idempotent but it **is not** safe
* **PATCH** – Partially updates a resource. PATCH **is not** idempotent and it **is not** safe.
* **DELETE** – Deletes a resource. DELETE **is** idempotent but it **is not** safe.
* **TRACE** – Performs a loop-back test. TRACE **is** idempotent but it **is not** safe.

|  |  |  |
| --- | --- | --- |
| **Verbs** | **Safety** | **Idempotency** |
| GET | Y | Y |
| POST | N | N |
| DELETE | **N** | **Y** |
| PUT | **N** | **Y** |
| HEAD | Y | Y |
| OPTIONS | Y | Y |
| TRACE | Y | Y |

The PUT and DELETE methods are defined to be idempotent. However, there is a caveat on DELETE. The problem with DELETE, which if successful would normally return a 200 (OK) or 204 (No Content), will often return a 404 (Not Found) on subsequent calls, unless the service is configured to "mark" resources for deletion without actually deleting them. However, when the service actually deletes the resource, the next call will not find the resource to delete it and return a 404. However, the state on the server is the same after each DELETE call, but the response is different.

GET, HEAD, OPTIONS and TRACE methods are defined as safe, meaning they are only intended for retrieving data. This makes them idempotent as well since multiple, identical requests will behave the same.

# **What is the difference between PUT, POST and PATCH methods in HTTP protocol?**

PUT = replace the ENTIRE RESOURCE with the new representation provided

PATCH = replace parts of the source resource with the values provided AND|OR other parts of the resource are updated that you haven’t provided (timestamps) AND|OR updating the resource effects other resources (relationships)

1. **POST**
   * If the client sends data without any identifier, then we will store the data and assign/generate a new identifier.
   * If the client again sends the **same** data without any identifier, then we will store the data and assign/generate a new identifier.
   * **Note**: Duplication is allowed here.
2. **PUT**
   * If the client sends data with an identifier, then we will check whether that identifier exists. If the identifier exists, we will update the resource with the data, else we will create a resource with the data and assign/generate a new identifier.
3. **PATCH**
   * If the client sends data with an identifier, then we will check whether that identifier exists. If the identifier exists, we will update the resource with the data, else we will throw an exception.

**Note**: On the **PUT** method, we are not throwing an exception if an identifier is not found. But in the **PATCH** method, we are throwing an exception if the identifier is not found.

# **When do you need @ResponseStatus annotation in Spring MVC? RequestBody and ResponseBody Annotations**

## @RequestBody

Simply put, **the @RequestBody annotation maps the HttpRequest body to a transfer or domain object, enabling automatic deserialization** of the inbound HttpRequest body onto a Java object.

First, let's have a look at a Spring controller method:

@PostMapping("/request")

**public** ResponseEntity **postController**(

@RequestBody LoginForm loginForm) {

exampleService.fakeAuthenticate(loginForm);

**return** ResponseEntity.ok(HttpStatus.OK);

}

Spring automatically deserializes the JSON into a Java type, assuming an appropriate one is specified.

By default, **the type we annotate with the @RequestBody annotation must correspond to the JSON sent from our client-side controller:**

**public** **class** **LoginForm** {

**private** String username;

**private** String password;

// ...

}

Here, the object we use to represent the HttpRequest body maps to our LoginForm object.

Let's test this using CURL:

curl -i \

-H "Accept: application/json" \

-H "Content-Type:application/json" \

-X POST --data

'{"username": "johnny", "password": "password"}' "https://localhost:8080/.../request"

This is all we need for a Spring REST API and an Angular client using the @RequestBody annotation.

## ****3.**** @ResponseBody

The @ResponseBody annotation tells a controller that the object returned is automatically serialized into JSON and passed back into the HttpResponse object.

Suppose we have a custom Response object:

**public** **class** **ResponseTransfer** {

**private** String text;

// standard getters/setters

}

Next, the associated controller can be implemented:

@Controller

@RequestMapping("/post")

**public** **class** **ExamplePostController** {

@Autowired

ExampleService exampleService;

@PostMapping("/response")

@ResponseBody

**public** ResponseTransfer **postResponseController**(

@RequestBody LoginForm loginForm) {

**return** **new** ResponseTransfer("Thanks For Posting!!!");

}

}

In the developer console of our browser or using a tool like Postman, we can see the following response:

{"text":"Thanks For Posting!!!"}

**Remember, we don't need to annotate the @RestController-annotated controllers with the @ResponseBody annotation** since Spring does it by default.

### 3.1. Setting the Content Type

When we use the @ResponseBody annotation, we're still able to explicitly set the content type that our method returns.

For that, **we can use the @RequestMapping‘s produces attribute.** Note that annotations like @PostMapping, @GetMapping, etc. define aliases for that parameter.

Let's now add a new endpoint that sends a JSON response:

@PostMapping(value = "/content", produces = MediaType.APPLICATION\_JSON\_VALUE)

@ResponseBody

**public** ResponseTransfer **postResponseJsonContent**(

@RequestBody LoginForm loginForm) {

**return** **new** ResponseTransfer("JSON Content!");

In the example, we used the MediaType.APPLICATION\_JSON\_VALUE constant. Alternatively, we can use application/json directly.

Next, let's implement a new method, mapped to the same /content path, but returning XML content instead:

@PostMapping(value = "/content", produces = MediaType.APPLICATION\_XML\_VALUE)

@ResponseBody

**public** ResponseTransfer **postResponseXmlContent**(

@RequestBody LoginForm loginForm) {

**return** **new** ResponseTransfer("XML Content!");

}

Now, **depending on the value of an Accept parameter sent in the request's header, we'll get different responses.**

Let's see this in action:

curl -i \

-H "Accept: application/json" \

-H "Content-Type:application/json" \

-X POST --data

'{"username": "johnny", "password": "password"}' "https://localhost:8080/.../content"

The CURL command returns a JSON response:

HTTP/1.1 200

Content-Type: application/json

Transfer-Encoding: chunked

Date: Thu, 20 Feb 2020 19:43:06 GMT

{"text":"JSON Content!"}

Now, let's change the Accept parameter:

curl -i \

-H "Accept: application/xml" \

-H "Content-Type:application/json" \

-X POST --data

'{"username": "johnny", "password": "password"}' "https://localhost:8080/.../content"

As anticipated, we get an XML content this time:

HTTP/1.1 200

Content-Type: application/xml

Transfer-Encoding: chunked

Date: Thu, 20 Feb 2020 19:43:19 GMT

<ResponseTransfer><text>XML Content!</text></ResponseTransfer>

# **What is the difference between @Controller and @RestController**

* @Controller is used to mark classes as Spring MVC Controller.
* @RestController is a convenience annotation that does nothing more than adding the @Controller and @ResponseBody annotations (see: [Javadoc](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html))

So the following two controller definitions should do the same

@Controller

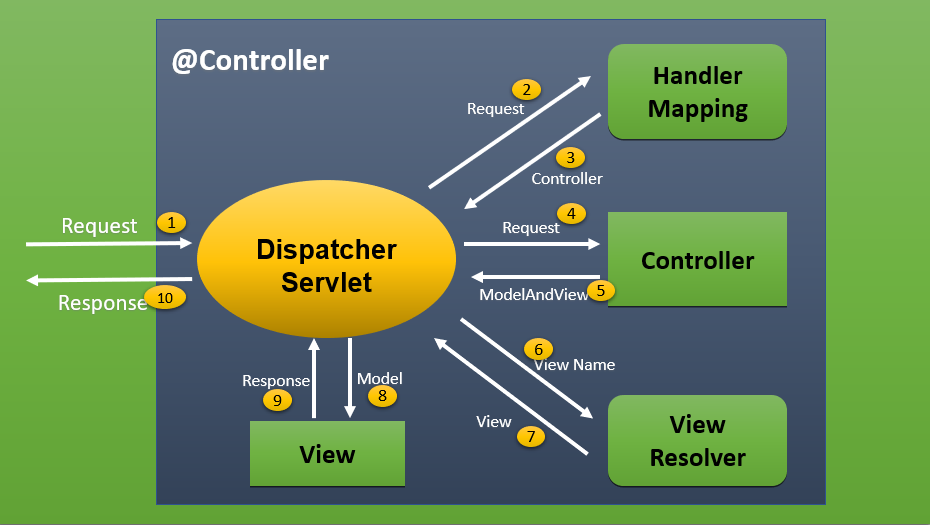
@ResponseBody

public class MyController { }

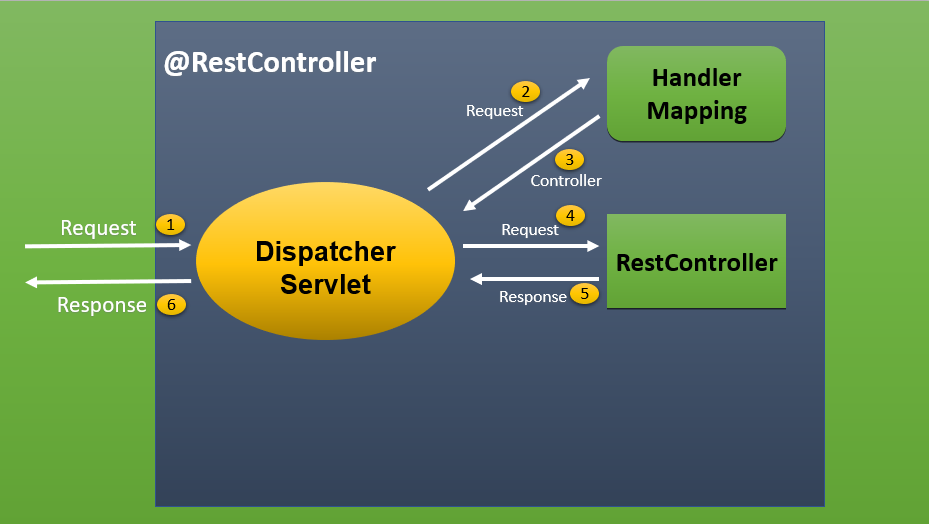
@RestController

public class MyRestController { }

Flow of request in a @Controller class without using a @ResponseBody annotation:

[](https://i.stack.imgur.com/xm7KW.png)

@RestController returns an object as response instead of view.

[](https://i.stack.imgur.com/wCZrD.png)

# @Enable Annotations

Spring comes with a set of **@Enable annotations that make it easier for developers to configure a Spring application**. These annotations are **used in conjunction with the @Configuration annotation**.

In this article we will be looking at some these annotations:

* @EnableWebMvc
* @EnableCaching
* @EnableScheduling
* @EnableAsync
* @EnableWebSocket
* @EnableJpaRepositories
* @EnableTransactionManagement
* @EnableJpaAuditing

## ****2.**** @EnableWebMvc

The @EnableWebMvc annotation is used for **enabling Spring MVC in an application** and works by importing the Spring MVC Configuration from WebMvcConfigurationSupport.

The XML equivalent with similar functionality is <mvc:annotation-driven/>.

The configuration can be customized by the @Configuration class implementing the WebMvcConfigurer:

@Configuration

@EnableWebMvc

**public** **class** **SpringMvcConfig** **implements** **WebMvcConfigurer** {

@Override

**public** **void** **configureMessageConverters**(

List<HttpMessageConverter<?>> converters) {

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

}

// ...

}

## ****3.**** @EnableCaching

The @EnableCaching annotation **enables annotation-driven cache management** capability within the application and **allows us to use the @Cacheable and @CacheEvict annotations** in our application.

The XML equivalent with similar functionality is the <cache:\*> namespace:

@Configuration

@EnableCaching

**public** **class** **CacheConfig** {

@Bean

**public** CacheManager **cacheManager**() {

SimpleCacheManager cacheManager = **new** SimpleCacheManager();

cacheManager.setCaches(

Arrays.asList(**new** ConcurrentMapCache("default")));

**return** cacheManager;

}

}

This annotation also has the following options:

* ***mode*** — indicates how caching advice should be applied
* ***order*** — indicates the ordering of the execution caching advisor when applied at a specific joinpoint
* ***proxyTargetClass*** — indicates whether subclass-based (CGLIB) proxies are to be created as opposed to standard Java interface-based proxies

This configuration again can be customized by the @Configuration class implementing the CachingConfigurerSupport class:

@Configuration

@EnableCaching

**public** **class** **CacheConfig** **extends** **CachingConfigurerSupport** {

@Bean

@Override

**public** CacheManager **cacheManager**() {

SimpleCacheManager cacheManager = **new** SimpleCacheManager();

cacheManager.setCaches(

Arrays.asList(**new** ConcurrentMapCache("default")));

**return** cacheManager;

}

@Bean

@Override

**public** KeyGenerator **keyGenerator**() {

**return** **new** MyKeyGenerator();

}

}

## ****4.**** @EnableScheduling

The @EnableScheduling annotation **enables scheduled task capabilities and allows us to use @Scheduled annotations** in the application. The XML equivalent with similar functionality is the <task:\*> namespace using the scheduler attribute.

This configuration again can be customized by the @Configuration class implementing the SchedulingConfigurer class:

@Configuration

@EnableScheduling

**public** **class** **SchedulingConfig** **implements** **SchedulingConfigurer** {

@Override

**public** **void** **configureTasks**(

ScheduledTaskRegistrar taskRegistrar) {

taskRegistrar.setScheduler(taskExecutor());

}

@Bean(destroyMethod = "shutdown")

**public** Executor **taskExecutor**() {

**return** Executors.newScheduledThreadPool(100);

}

}

For more on using Spring scheduling, you can refer to this [article](https://www.baeldung.com/spring-scheduled-tasks).

## ****5.**** @EnableAsync

The @EnableAsync annotation **enables asynchronous processing in our application**. The XML equivalent with similar functionality is the <task:\*> namespace using the executor attribute.

@Configuration

@EnableAync

**public** **class** **AsyncConfig** { ... }

For more on using Spring async, you can refer to this [article](https://www.baeldung.com/spring-async).

## ****6.**** @EnableWebSocket

The @EnableWebSocket annotation is used to **configure the processing of web socket requests**. Customization can be done by implementing the WebSocketConfigurer class:

@Configuration

@EnableWebSocket

**public** **class** **MyConfiguration** **implements** **WebSocketConfigurer** {

@Override

**public** **void** **registerWebSocketHandlers**(WebSocketHandlerRegistry registry) {

registry.addHandler(echoWebSocketHandler(), "/echo").withSockJS();

}

@Bean

**public** WebSocketHandler **echoWebSocketHandler**() {

**return** **new** EchoWebSocketHandler();

}

}

For more on using Spring Websockets, you can refer to this [article](https://www.baeldung.com/websockets-spring).

## ****7.**** @EnableJpaRepositories

The @EnableJpaRepositories annotation **enables Spring Data JPA repositories** by scanning the package of the annotated configuration class for repositories.

@Configuration

@EnableJpaRepositories

**public** **class** **JpaConfig** { ... }

Some options available for this annotation are:

* ***value*** — alias for the basePackages() attribute
* ***basePackages*** — base packages to scan for annotated components
* ***enableDefaultTransactions*** — configures whether or not to enable default transactions for Spring Data JPA repositories
* ***entityManagerFactoryRef*** — configures the name of the EntityManagerFactory bean definition to be used

## ****8.**** @EnableTransactionManagement

The @EnableTransactionManagement annotation **enables Spring's annotation-driven transaction management capability**. The XML equivalent is the <tx:\*> namespace.

@Configuration

@EnableTransactionManagement

**public** **class** **JpaConfig** { ... }

For more on using Spring Transaction Management, you can refer to this [article](https://www.baeldung.com/transaction-configuration-with-jpa-and-spring).

## ****9.**** @EnableJpaAuditing

The @EnableJpaAuditing annotation **enables auditing on your JPA entities**.

@Configuration

@EnableJpaAuditing

**public** **class** **JpaConfig** {

@Bean

**public** AuditorAware<AuditableUser> **auditorProvider**() {

**return** **new** AuditorAwareImpl();

}

}

For more on using Spring Web Sockets, you can refer to this [article](https://www.baeldung.com/database-auditing-jpa).

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

message converters are to marshal and unmarshal object in a different format (like JSON, XML etc.).Spring MVC uses the HttpMessageConverter interface to convert HTTP requests and responses. Spring MVC provides some out of the box default configurations for the converters.

### **The Default Message Converters**

By default, the following HttpMessageConverters are loaded by Spring

* ByteArrayHttpMessageConverter – that can read and write byte arrays
* ResourceHttpMessageConverter -can read/write Resources and supports byte range request
* SourceHttpMessageConverter – can read and write Source objects.
* StringHttpMessageConverter – read and write strings
* FormHttpMessageConverter – read and write ‘normal’ HTML forms .

Spring provides other sets of message converters, but they will only be registered when the related library is in the classpath, here are some of the popular converters from the list

* MappingJacksonHttpMessageConverter – For converting JSON
* Jaxb2RootElementHttpMessageConverter – To convert Java objects to/from XML
* MappingJackson2HttpMessageConverter – To convert JSON.

##### 2.1 @RequestBody Annotation

Annotation indicating a method parameter should be bound to the body of the web request. It works with “Content-Type” header passed by the client.Spring use this header value to find the right converter.

@PostMapping(value = "/greeting")

public @ResponseBody void greeting(@RequestBody Customer customer){

//create customer

}

Copy

Use any REST client or curl to execute code, we will get 200 OK response.

curl -i -X POST -H "Content-Type: application/json" -d '{"id":"14","name":"Java Dev Journal"}' http://localhost:8080/greeting

HTTP/1.1 200

Content-Length: 0

Date: Tue, 27 Mar 2018 04:28:23 GMT

##### 2.2 @ResponseBody Annotation

@ResponseBody is used on the argument of a Controller method.This annotation tells Spring that the return value of the method serialized directly to the body of the HTTP Response.

@ResponseBody annotation work with “Accept” header passed by the client to decide what type of representation required on the client side.

@GetMapping(value = "/greeting")

public @ResponseBody Customer greeting(){

Customer customer = new Customer(14, "Java Dev Journal");

return customer;

}

**Output**

curl --header "Accept: application/json" http://localhost:8080/greeting

{

"id":14,

"name":"Java Dev Journal"

}

### Custom Converter

To create a custom message converter in Spring, we should overrideconfigureMessageConverters() by extending WebMvcConfigurerAdapter class.

@Configuration

public class WebConfig implements configureMessageConverters {

@Override

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

MarshallingHttpMessageConverter xmlConverter =new MarshallingHttpMessageConverter();

XStreamMarshaller xstream = new XStreamMarshaller();

xmlConverter.setMarshaller(xstream);

xmlConverter.setUnmarshaller(xstream);

converters.add(xmlConverter);

}

}

[pullquote align=”normal”]We need Xtream library in the classpath [/pullquote]

Let’s check what we did in this example

* We are overriding configureMessageConverters() method under configureMessageConverters to define our own custom converter.
* Created instance of MarshallingHttpMessageConverter to read and write XML using Spring’s Marshaller and Unmarshaller.
* We created XStreamMarshaller instance and set Marshaller and Unmarshaller for the MarshallingHttpMessageConverter.
* Finally, we added this converter to the list of converters.

Let’s see our custom converter in action

curl --header "Accept: application/json" http://localhost:8080/greeting

curl --header "Accept: application/xml" http://localhost:8080/greeting

{

"id":14,

"name":"Java Dev Journal"

}

Copy

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

<customer>

<id>14</id>

<name>Java Dev Journal</name>

</customer>

Copy

# **Can REST Conroller provide response content format xml or json based on client request payload and if yes/no how?**

Yes using header value "Accept" as application/json or application/xml while client make request invoke.

And in conroller adding producer={"application/json",application/xml}

# **What is a resource?**

A resource is how data is represented in the REST architecture. By exposing entities as the resource,

it allows a client to read, write, modify, and create resources using HTTP methods,

for example, GET, POST, PUT, DELETE, etc.

# **Is REST normally stateless?**

Yes, REST API should be stateless, because it is based on HTTP, which is also stateless. A request in REST API should contain all the details required to process it.

It should not rely on previous or next requests or some data maintained at the server end, like sessions.

# **Is REST secure? What can you do to secure it?**

REST is normally not secure, but you can secure it by using Spring Security. At the very least,

you can enable the HTTP basic authentication by using HTTP in your Spring Security configuration file.

Similarly, you can expose your REST API using HTTPS, if the underlying server supports HTTPS.