Reference urls &Books

Best book is “ **RESTful Java with JAX-RS 2.0, 2nd Edition “ – such a good book where all concepts are covered**

|  |  |
| --- | --- |
| cxf | <https://cxf.apache.org/docs/jax-rs.html>  <https://cxf.apache.org/download.html> - if u download it will contain lot of spring and spring boot examples    this samples folders contains lot of code examples  CXF ships an example in the samples/restful folder of the distribution of how to create RESTful services using JAX-WS Provider and Dispatch APIs. |
| configuring jax-rs bean | <https://cxf.apache.org/docs/jaxrs-services-configuration.html> |
| cfg jax-rs bindings | <https://cxf.apache.org/docs/jax-rs-data-bindings.html> |
| spring boot | <https://cxf.apache.org/docs/springboot.html#SpringBoot-SpringBootCXFJAX-RSStarter>  <https://cxf.apache.org/docs/jaxrs-services-configuration.html#JAXRSServicesConfiguration-SpringBoot> |
| git hub | <https://github.com/apache/cxf/tree/main/distribution/src/main/release/samples/jax_rs/spring_boot_scan/application> |
| oauth integration for apache cxf | <https://cxf.apache.org/docs/jax-rs-oauth2.html>  <https://oauth.net/code/java/> |
| misc restful | <https://openliberty.io/docs/latest/rest-microservices.html> |
| main cxf websites | <https://cxf.apache.org/>  <https://cxf.apache.org/docs/migration-guides.html> |
| rest easy | <https://docs.resteasy.dev/6.2/userguide/#_formparam> |

Current status-

Practice – throwing jax-rs exception

Paused at Developing the RESTful service in book – page 171

RESTful webervices & advantages

JAX-RS – java api for xml based restful services

Same JAX-RS is the specification/APIto create web services according to the Representational State Transfer (REST) architectural style , and the implementation is given by restful service providers – apache cxf, jersey, REST Easy from JBoss (same like JPA spec)

The application developed using JAX-RS should be able to run in any container Inclusion in the Java Enterprise Edition container

Specification is nothing but set of rules and regulations, /api – it will just contains the interfaces, impl is from apache cxf

All the main annotations like @path, @Post …are part of javax.ws.rs

And this jax-rs internally uses http protocol like GET,PUT, POST… (means clients should use these protocols to send the data and server will return these reponses 200,302)

Apache CXF –By default supports xml, anything other than that needs to be mentioned specifically, @POST(produces= Application/JSON..)

Means by default while incoming, the xml payload will be binded to java object and while returning the response, java object will be converted into xml payload

Why stateless?

REST is neither a technology nor a standard; it's an architectural style, a set of guidelines for exposing resources over the web.

All the information required to process a request on a resource is contained within the request itself, thereby making the interaction stateless.

Ex;- pass all the security info in headers

To build web services there are 2 ways- soap and REST

Rest means- representational state transfer

REST was defined by Roy Fielding in his 2000 doctoral dissertation.

### 1. What is REST?

RESTful web services heavily uses HTTP protocol

While REST often uses HTTP, it's not strictly tied to it. You could theoretically implement a RESTful system over other protocols (though HTTP is by far the most common and natural fit for its principles)

REST is an architectural style, not a protocol or a standard in itself.

Its like API It's a set of guidelines and constraints and best practices for designing distributed systems, when applied, result in a system that is often:

Unlike soap , It doesn't define the syntax of messages or the exact protocols, but rather the overall design philosophy where as soap tell the msg should be like soap envelope, body , faults..

### 2. Key Principles (Constraints) of REST

To be considered "RESTful," a web service should adhere to the following six architectural constraints:

* **Client-Server Architecture:**
  + The client (e.g., a web browser, mobile app) and the server (where the service is hosted) are separate and independent.
  + This separation of concerns improves portability and scalability. The client doesn't need to know about the server's internal logic, and the server doesn't need to know about the client's UI.
* **Statelessness:**
  + This means the server doesn't "remember" previous interactions with a specific client. every request is fresh request

This simplifies server design, improves scalability (as any server can handle any request), and makes the service more robust to failures.

* + Each request from a client to a server must contain all the information necessary to understand the request. The server should not store any client context between requests. Ex:- client should send the jwt token / should keep crede in header in every req,
* **Cacheability:**
  + Responses from the server should explicitly or implicitly define themselves as cacheable or non-cacheable.
  + This allows clients or intermediaries (like proxies) to cache responses, improving performance and reducing server load for subsequent requests for the same resource.
* **Layered System:**
  + A client typically cannot tell whether it's connected directly to the end server or to an intermediary server along the way.
  + This allows for the use of proxy servers, load balancers, and other intermediaries to enhance scalability, security, and performance without affecting the client or the end server.
* **Uniform Interface:** This is the most crucial constraint and defines how the client and server interact. It consists of four sub-principles:
  + **Identification of Resources:** Individual resources are identified in requests, e.g., using URLs (/users/123, /products).
  + **Manipulation of Resources Through Representations:** Clients manipulate resources by exchanging representations of those resources (e.g., JSON or XML). For instance, to create a user, you send a JSON representation of the user.
  + **Self-Descriptive Messages:** Each message includes enough information to describe how to process it. This means the server can indicate the media type of the response (e.g., Content-Type: application/json) or the client can specify the desired media type (e.g., Accept: application/json).
  + **Hypermedia as the Engine of Application State (HATEOAS):** This is often considered the most difficult constraint to fully implement. It means that responses should include links to related resources, allowing the client to discover available actions and transitions. For example, a response for a "user" resource might include links to their "orders" or an "update user" endpoint.
* **Code on Demand (Optional):**
  + Servers can temporarily extend or customize client functionality by transferring executable code (e.g., JavaScript). This constraint is optional and less commonly used in typical REST APIs.

### 3. How do RESTful Web Services Work?

RESTful services primarily use the standard HTTP methods to perform operations on resources:

* **GET:** Retrieves a representation of a resource. (Idempotent and safe)
  + Example: GET /users/123 (get user with ID 123)
* **POST:** Creates a new resource or submits data to be processed. (Not idempotent)
  + Example: POST /users (create a new user)
* **PUT:** Updates an existing resource (or creates it if it doesn't exist, replacing the entire resource). (Idempotent)
  + Example: PUT /users/123 (update user with ID 123, replacing all its data)
* **DELETE:** Deletes a resource. (Idempotent)
  + Example: DELETE /users/123 (delete user with ID 123)
* **PATCH:** Partially updates an existing resource. (Not idempotent, but generally preferred for partial updates over PUT)
  + Example: PATCH /users/123 (update only specific fields of user with ID 123)

**Resources** are the core concept. Anything that can be named and accessed is a resource. Examples: users, products, orders, blog posts.

### 4. Advantages of RESTful Web Services

* **Flexibility:** Supports various data formats (JSON, XML, plain text, etc.).,but in soap we must use soap message format With xml only
* **Simplicity:** Easier to build and consume compared to older styles like SOAP. And since REST supports json , xml parsing overhead will be decreased, and payload size will be decreased
* **Scalability:** Statelessness and caching contribute to better scalability.
* **Decoupling:** Client and server are independent, allowing for separate development and deployment.
* **Standard HTTP:** Leverages existing web infrastructure and tools.
* **Interoperability/ Re-usability:**- a python app/.Net app / any heterogeneous app can talk to java app using HTTP REST , both SOAP and REST webservices are re-usable

**Note:**- caching can be implemented in both rest & SOAP, in SOAP we have a handler class(filter) which will be invoked for every request, so check for duplicate req and u can return the data from the filter itself..

### 5. REST in the Context of Spring Data JPA and Java

When you're working with Spring Data JPA and Java, you'll typically expose your JPA entities (which represent your data) as **RESTful resources** using **Spring Boot** and **Spring Web (Spring MVC)**.

* You'll define **Controller** classes in Spring that map HTTP requests (like GET, POST, PUT, DELETE) to specific methods.
* These methods will then interact with your **Spring Data JPA repositories** to perform CRUD (Create, Read, Update, Delete) operations on your entities (e.g., userRepository.findById(id), userRepository.save(user)).
* The data exchanged between the client and server will typically be in **JSON format**, which Spring handles automatically for serialization and deserialization.

CXF implementation

**CXF provides best integration with the Spring framework,** which simplifies configuration and deployment of RESTful applications

All the capabilities provided by Spring like transaction mgmnt can be leveraged when developing RESTful POJO-based applications using CXF

**Pluggable data binding**

Data binding is about mapping the HTTP request, for instance JSON or XML over HTTP, to the required Java objects which your implementation expects. Similarly, the response from the Java implementation needs to be mapped to the required format, for eample XML or JSON format, before sending the response over HTTP.

CXF handles this mapping transparently behind the scenes by providing data binding components. CXF supports various data binding mechanisms such as JAXB, JSON, XMLBean, and Aegis. CXF allows you to specify the binding mechanism declaratively

**Client API**

The JAX RS specification does not provide client side APIs for invoking a JAX-RS enabled REST service. CXF simplifies this by providing Client APIs to invoke RESTful services which can also be configured using the Spring dependency framework.

**• Security**

Applications built using CXF JAX-RS implementation can leverage Spring framework features like declarative security to restrict a resource class and methods access based on application requirements, instead of handling the security requirements programmatically

**Filters**

Filters are used to perform pre processing or post processing of messages like checking the headers and their value. CXF provides an ability to create and configure filters for inspecting the message, logging the message, and modifying the request or response based on the application requirements.

Ex:- in header consumers will pass authentication info- like consumer will create a sign with his private key and he will send the signature In header , we/provider can extract header value we can put this logic in filter

Architecture

JAX-RS provides exception mappers

- to map custom java exceptions to error codes- ex:- when my desired exception came, then It should give 400.. some status for that runtime exception

Spring REST vs JAX-RS

**Not a JAX-RS Implementation:** This is a key difference: **Spring REST does not implement the JAX-RS specification.** It provides its own alternative way of building REST services in Java.

JAX-RS implementation is given only by Apache cxf, REST easy, jersey

Annotations

By default cxf framework supports xml, anything other than that needs to be mentioned specifically, @POST(produces= Application/JSON..)

@Path, @GET, @POST, @PUT, @DELETE, @Produces, @Consumes, @PathParam, @QueryParam

|  |  |
| --- | --- |
| @XmlRootElement | is used to indicate that a **Java class can be serialized to or deserialized from XML** using JAXB (Java Architecture for XML Binding). |

can we make class implementing serializable instead of using@Xmlrootelement - **No, implementing Serializable is not a substitute for using @XmlRootElement** in JAX-RS for XML support.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Serializable** | **@XmlRootElement (JAXB)** |
| **Purpose** | Java object serialization (binary format) | XML binding for Java objects |
| **Used for** | Writing to files, transferring over sockets | Converting Java ↔ XML in RESTful APIs |
| **Used by JAX-RS for XML?** | ❌ No | ✅ Yes (via JAXB) |

|  |  |
| --- | --- |
| @path(“/users/{username”}) | in spring rest similar anno is @RequestMapping  this anno is used to bind the url to a java method  this anno can be used on both class and method |
| @GET , @POST | unlike spring REST where we had @GetMapping(“/upload”)  here we have only @GET, @POST annotations only |
| @PathParam – used to fetch the value from path | to fetch the values from the url  localhost:8080/services/helloservice/patients/patient/mani  @Path("/patient/{name}") @GET public Patient getPatient(@PathParam("name") String *name*){ |
| @QueryParam  @HeaderParam | to fetch the values from the url after question mark “?”  localhost:8080/services/helloservice/patients/queryPatient?patientName=santu  @Path("/queryPatient") @GET public Patient getPatientUsingQueryAndHeaderParam(@QueryParam("patientName") String *name*, @HeaderParam("vendor")String *vendorName*){  System.***out***.println("fetch query param name is ---> "+*name*);  System.***out***.println("fetch header param name is ---> "+*vendorName*); |
| @FormParam | used to map the params the comes during a form submission |
|  |  |

|  |  |
| --- | --- |
| @CookieParam  this anno is to fetch params from cookie  @Context == @Autowired  this is also a jax-rs spec annotation | to add the cookies in postman, click on cookies🡪 add a domain (means root url/host name) -> and add cookie  headers will be passed only for that particular request, (headers are nothing but metadata – means data about data)  whereas these **cookies will be passed for all the requests for that website/ for that domain**  @Context is same like @Autowired this anno is to inject a bean from context  @Context  HttpHeaders headers; // headers.getCookies(); cookies.keyset()  to autowire http servlet request and to autowire httpservlet response , to fetch any object from spring/ jax-rs context use this anno  @Path("/parseCookies") @GET public Patient getPatientUsingCookieParam(@CookieParam("secretAgent") String *cookeyName*, @Context *HttpServletRequest request*, @Context *HttpServletResponse httpResponse*){  System.***out***.println("fetched using @CookieParam cookies as ---> "+*cookeyName*);  *httpResponse*.setHeader("Set-Cookie","JSESSIONIDP=ABCD; HttpOnly; Secure; SameSite=Strict"); |

|  |  |
| --- | --- |
| ~~@RequestBody~~ | To map the payload to a java object like in REST we don’t have any @RequestBody,  here that full payload will be binded automatically to that java object  @Path("/get") @POST public void getEmployee(@Suspended *AsyncResponse async*, Employee *empReqBody*,@QueryParam("empNameQueryP") String *empNameQueryP*) throws InterruptedException {  see here now Employee object is directly binded to class without any @RequestBody |
| @Provider | to map custom java exceptions to error codes  import javax.ws.rs.core.Response;  import javax.ws.rs.ext.ExceptionMapper;  import javax.ws.rs.ext.Provider;  @Provider // Important: tells JAX-RS runtime to discover this mapper  public class UserNotFoundExceptionMapper implements ExceptionMapper<UserNotFoundException> { |
| @Consumes  @Produces |  |
|  |  |

In built classes

HttpHeaders, cookies

Throw inbuilt exception classes

Throwing inbuilt exceptions is best way than throwing custom exceptions and again catching them and converting to final Response using *Response*.*status*(*status*).build()

ex:- throw new WebApplicationException(*Response*.*status*(*Response*.*Status*.*FORBIDDEN*).build());

advantage is – if u throw this automatically this response and message will be sent back , Either throw specific exception like NotAuthorizedException

* or u can throw generic WebApplicationException

in realtime while throwing we should always pass standard entity in response

*Response* badRequest = *Response*.*status*(*Response*.*Status*.***BAD\_REQUEST***).entity("bad request").build();  
 throw new BadRequestException(badRequest);// when we throw an object this entity will be returned in response

, we can also catch and we have to send standard response always – hence we will always catch and send standard response

* one best option is have both, instead of creating custom exception classes ex;- when AIPAC app got 500 after hitting D&S services then instead of throwing our custom class exception throw jax-rs classes exception – like throw new ServerErrorException(response) with response, so that the entity in that response will be sent back
* and catch all exceptions at end by using handler and send a standard response

Note:- what ever the method u follow ultimately response only needs to send back

* when u are throwing a predefined exception, then also we need to send response
* even in global exception handler also after catching we will send response back

|  |  |
| --- | --- |
| All exceptions are present In jakarta.ws.rs package  WebApplicationException – for any error we can throw this global exception  ServerErrorException – for any kind of server issues – like null pointer exception /.. throw this simply  InternalServerErrorException 500  ServiceUnavailableException – 503- this indicates client that currently that jax-rs service provider is unavailable,  example if jax-rs provider is consuming any other service like rest/soap services, database, or MQ- if that service is down then throw these exception ****When should you throw**** ServiceUnavailableException****?**** Use it when:   * The **server is temporarily unable to process the request** due to:   + **Maintenance** mode   + **Overloaded server** (too many requests)   + **Upstream dependencies** (like a database, microservice, or third-party API) are down or not responding   class InternalServerErrorException extends ServerErrorException  class ServerErrorException extends WebApplicationException | |
| ClientErrorException –  ForbiddenException 403  NotAuthorizedException 401  BadRequestException 400  NotSupportedException | throw these exception if mistake is from client side  example if client sent wrong data if validation failed..   ClientErrorException is a **runtime exception** that indicates a **4xx client-side error** (e.g., Bad Request, Unauthorized, Not Found, etc.).   It is part of the **JAX-RS API** and extends WebApplicationException.   It is typically thrown **by a resource method** or **inside a filter/interceptor** when you want to explicitly signal that the **client made a mistake**.  ex:- if we are jax-rs server /provider api, if client made any mistak – like authorization mistake, missing valid header .. if any issue is present in client, then provider can throw this exception to indicate client side /caller issue  class ForbiddenException extends ClientErrorException // throw this for 403 status  class NotAuthorizedException extends ClientErrorException // throw this for 401 status  class BadRequestException extends ClientErrorException // for 400 status |

Below states even if we throw pre defined exception proper response will be sent back

|  |  |
| --- | --- |
| the xml u are seeing above is the entity | @Path("/patient/{name}") @GET public Patient getPatientPathParamDemo(@PathParam("name") String *name*){  System.*out*.println("executing single patient method");  if(*name*==null || *name*.equalsIgnoreCase("mani")){  System.*out*.println("thworing bad requ");  FailureResponse response=FailureResponse.*builder*().failureCode("400\_AIPAC").failureReason("Wrong input").build();  *Response* badRequest = *Response*.*status*(*Response*.*Status*.*BAD\_REQUEST*)  .entity(response) //so here whatever entity we are throwing same was returned and visible in postman  .type(MediaType.*APPLICATION\_XML*) //here xml converter is required to convert java obj to xml  .build();  throw new BadRequestException(badRequest);  }  return Patient.*builder*().name(*name*).build(); } |
| @Data @Builder @XmlRootElement @NoArgsConstructor @AllArgsConstructor public class FailureResponse {  String failureCode;  String failureReason;  } |  |

Exception handling /mappers

Same like controller advice in spring, here in jax-rs we have Exception mappers, these are reponsible to handle the exception @Provider example

These

In spring

|  |  |
| --- | --- |
| @ExceptionHandler(ResourceNotFoundException.class)  public ResponseEntity<Object> handleResourceNotFound(ResourceNotFoundException ex) {  Map<String, Object> body = new HashMap<>();  body.put("timestamp", LocalDateTime.now());  body.put("message", ex.getMessage());  return new ResponseEntity<>(body, HttpStatus.NOT\_FOUND);  } | public class ResourceNotFoundException extends RuntimeException { |

My opinion, instead of throwing custom exception and catching them using mapper and finally returning a http status, throw jax-rs exception directly na, its simple

In jax-rs

To build response use *Response*.*status*(*status*).build() , same like in REST ResponseEntity.of(500).build();

|  |  |
| --- | --- |
| so here, when ever this is UserNotFoundException thrown then automatically 400 response will be thrown with the error message object   * This Exception mapper is same like in REST controller we have the controller advice,   // Custom Error Response POJO (example)  class ErrorResponse {  private int status;  private String errorCode;  private String message;  public ErrorResponse(int status, String errorCode, String message) {  this.status = status;  this.errorCode = errorCode;  this.message = message;  }  // Getters for JSON serialization  public int getStatus() { return status; }  public String getErrorCode() { return errorCode; }  public String getMessage() { return message; }  }  // Custom Exception (example)  class UserNotFoundException extends RuntimeException {  public UserNotFoundException(String message) {  super(message);  }  } | import javax.ws.rs.core.Response;  import javax.ws.rs.ext.ExceptionMapper;  import javax.ws.rs.ext.Provider;  //here this means this mapper is capable of catching this exception alone like exception handler in spring  @Provider // Important: tells JAX-RS runtime to discover this mapper  public class UserNotFoundExceptionMapper implements ExceptionMapper<UserNotFoundException> {  @Override  public Response toResponse(UserNotFoundException exception) {  // Build a custom error response  ErrorResponse error = new ErrorResponse(  404, // Custom error code  "NOT\_FOUND",  exception.getMessage() // Get the message from the exception  );  return Response.status(Response.Status.NOT\_FOUND) // Set HTTP 404 status  .entity(error) // Set the custom error object as the response body  .type("application/json") // Specify content type  .build();  }  } |

|  |  |
| --- | --- |
| usage  @GET  @Path("/{id}")  @Produces(MediaType.APPLICATION\_JSON)  public User getUser(@PathParam("id") Long id) {  User user = userService.findUserById(id); // Assume this throws UserNotFoundException if not found  if (user == null) {  throw new UserNotFoundException("User with ID " + id + " not found.");  }  return user;  } | client receives below in postman  HTTP/1.1 404 Not Found  Content-Type: application/json  {  "status": 404,  "errorCode": "NOT\_FOUND",  "message": "User with ID 123 not found."  } |

Read in gemini on “is spring REST and jax-rs are both same?”

Code

CRUD program

|  |  |
| --- | --- |
|  | <https://cxf.apache.org/download.html> - if u download it will contain lot of spring and spring boot examples    this samples folders contains lot of code examples for spring boot see - jax\_rs\spring\_boot  CXF ships an example in the samples/restful folder of the distribution of how to create RESTful services using JAX-WS Provider and Dispatch APIs. |
|  |  |

<https://cxf.apache.org/docs/springboot.html#SpringBoot-SpringBootCXFJAX-RSStarter>

|  |  |
| --- | --- |
| cxf.path=/services/helloservice or  cxf.path=/  logging.level.org.apache.cxf=INFO  setting context path is mandatory else u will get 404 not found issue   * We must create a JAXRSServerFactoryBean   *#server.servlet.context-path=/restws (this is not mandatory over cxf.path)*  @SpringBootApplication @ComponentScan(basePackages = {"com.ampf.controller"}) public class Rest\_1\_MainApp {  @Autowired  *Bus* **bus**;  @Autowired  PatientController **controller**;  public static void main(String[] *args*) {  SpringApplication.*run*(Rest\_1\_MainApp.class,*args*);  } @Bean public *Server* rsserver() {  System.***out***.println(" creating jax-rs serverbean ");  JAXRSServerFactoryBean serverBean = new JAXRSServerFactoryBean();  serverBean.setBus(**bus**);  serverBean.setAddress("/");  serverBean.setServiceBeans(*List*.*of*(**fileUploadController**));  return serverBean.create(); }  }  In spring if u just annotate with @RestController, @PostMapping,  spring will auto detect these anno and register them as endpoints  but this is not the case with jax-rs, in jax-rs we need to manually register all controllers to server as service beans | controller class  import jakarta.annotation.PostConstruct; import jakarta.ws.rs.Consumes; import jakarta.ws.rs.GET; import jakarta.ws.rs.Path; import jakarta.ws.rs.Produces; import org.springframework.stereotype.Component;  import java.util.ArrayList; import java.util.HashMap; import java.util.*List*;  @Path("/patients") @Consumes("application/xml,application/json") @Produces("application/xml,application/json") @Component public class PatientController {   HashMap<Integer, Patient> **patients**;   @Path("/all")  @GET  public *List*<Patient> getAllPatients(){  System.***out***.println("invoking the getter method");  return new ArrayList<>(**patients**.values());  }   @PostConstruct  public void init(){  **patients**=new HashMap<>();  System.***out***.println("executing post construct");  int id=111;  **patients**.put(id,new Patient(id,"santu"));  id=222;  **patients**.put(id,new Patient(id,"radhika"));   } } |
|  |  |

General flow

Convert the XML content associated with HTTP POST request into the required format expected by your implementation, for instance a Java object

• Perform the required operation, for instance insert the employee object in the database

• Convert the response back into an HTTP format, for instance set the HTTP

* Status as 200 denoting a successful response and convert the response into the required format (XML or JSON), and set it in the HTTP body.

|  |  |
| --- | --- |
| **Example (simplified Spring Boot REST Controller):**  Java  import org.springframework.web.bind.annotation.\*;  import java.util.List;  @RestController // Marks this class as a REST Controller  @RequestMapping("/api/users") // Base path for user-related endpoints  public class UserController {  private final UserRepository userRepository; // Your Spring Data JPA repository  public UserController(UserRepository userRepository) {  this.userRepository = userRepository;  }  @GetMapping // Handles GET /api/users  public List<User> getAllUsers() {  return userRepository.findAll();  }  @GetMapping("/{id}") // Handles GET /api/users/{id}  public User getUserById(@PathVariable Long id) {  return userRepository.findById(id)  .orElseThrow(() -> new RuntimeException("User not found"));  }  @PostMapping // Handles POST /api/users  public User createUser(@RequestBody User user) { // @RequestBody maps JSON to User object  return userRepository.save(user);  }  @PutMapping("/{id}") // Handles PUT /api/users/{id}  public User updateUser(@PathVariable Long id, @RequestBody User userDetails) {  User user = userRepository.findById(id)  .orElseThrow(() -> new RuntimeException("User not found"));  user.setName(userDetails.getName());  user.setEmail(userDetails.getEmail());  // ... update other fields  return userRepository.save(user);  }  @DeleteMapping("/{id}") // Handles DELETE /api/users/{id}  public void deleteUser(@PathVariable Long id) {  userRepository.deleteById(id);  }  }  This example demonstrates how Spring Boot simplifies the creation of RESTful web services that interact with your data layer (managed by Spring Data JPA). Understanding REST is fundamental for building modern, distributed Java applications  . If @Produces annotation is not specified, then the runtime container will assume that any content type can be produced. For the CategoryService class, we have defined @Produces("application/xml") annotation, which implies that the CategoryService produces only application/xml. | jax-rs program  //JAX-RS Imports  import javax.ws.rs.Consumes;  import javax.ws.rs.DELETE;  import javax.ws.rs.GET;  import javax.ws.rs.POST;  import javax.ws.rs.PUT;  import javax.ws.rs.Path;  import javax.ws.rs.PathParam;  import javax.ws.rs.Produces;  import javax.ws.rs.core.Response;  import javax.ws.rs.core.Response.Status;  @Path("/categoryservice")  @Produces("application/xml")  public class CategoryService {  @GET  @Path("/category/{id}")  public Category getCategory(@PathParam("id") String id) {  System.out.println("getCategory called with category id: " + id);  Category cat = (Category) getCategoryDAO().getCategory(id);  if (cat == null) {  ResponseBuilder builder = Response.status(Status.BAD\_REQUEST);  builder.type("application/xml");  builder.entity("<error>Category Not Found</error>");  throw new WebApplicationException(builder.build());  } else {  return cat;  }  }    @POST  @Path("/category")  @Consumes("application/xml")  public Response addCategory(Category category) {  System.out.println("addCategory called");  Category cat = (Category) getCategoryDAO().getCategory(  category.getCategoryId());  if (cat != null) {  return Response.status(Status.BAD\_REQUEST).build();  } else {  getCategoryDAO().addCategory(category);  return Response.ok(category).build();  }  }  @DELETE  @Path("/category/{id}")  public void deleteCategory(@PathParam("id") String id) {  }  @PUT  @Path("/category")  public void updateCategory(Category category) {  }  @POST  @Path("/category/book")  @Consumes("application/xml")  public void addBooks(Category category) {  }  @GET  @Path("/category/{id}/books")  @Consumes("application/xml")  public void getBooks(@PathParam("id") String id) {  }  } |

|  |  |
| --- | --- |
| in both jax-rs, jax-ws we need additional configuration, in spring if we just annotate with @RESTController it will auto detect but  In apache cxf – u should manually register all controllers manually as below,  even if u create any filters or interceptors or exception mapper u should manually configure |  |
| @Autowired PatientController **controller**;  @Bean public *Server* rsserver(){  System.***out***.println(" creating bean ");  JAXRSServerFactoryBean endPoint =new JAXRSServerFactoryBean();  endPoint.setBus(**bus**);  endPoint.setAddress("/");  endPoint.setServiceBeans(*List*.*of*(**controller**));  System.***out***.println("created a bean from config class");  return endPoint.create(); }  cxf.path=/services/helloservice (this is mandatory path) logging.level.org.apache.cxf=INFO  The create method creates an embedded jetty service instance deploys the **controller**  This tells CXF: “Expose PatientController at */restws* with the @Path */patientservice*” |  |

Configure interceptors

|  |  |
| --- | --- |
| In apache cxf – u should manually register all controllers manually as below,  even if u create any filters or interceptors or exception mapper u should manually configure |  |
| <jaxrs:server address="/jaas">      <jaxrs:serviceBeans>          <bean class="org.apache.cxf.systest.jaxrs.security.SecureBookStoreNoAnnotations"/>      </jaxrs:serviceBeans>      <jaxrs:providers>          <ref bean="authenticationFilter"/>      </jaxrs:providers>  </jaxrs:server>     <!—in above for every service or controller class we should have an address, in providers we should add all filters, exception mappers, interceptors for that controller -->  <bean id="authenticationFilter" class="org.apache.cxf.jaxrs.security.JAASAuthenticationFilter">      <!-- Name of the JAAS Context -->      <property name="contextName" value="BookLogin"/>      <!-- Hint to the filter on how to have Principals representing users and roles separated           while initializing a SecurityContext -->      <property name="rolePrefix" value="ROLE\_"/>        <property name="redirectURI" value="/login.jsp"/>  </bean> |  |

Returning proper response

import jakarta.ws.rs.core.*Response*;

*Response* badRequest = *Response*.*status*(*Response*.*Status*.***BAD\_REQUEST***).entity("bad request").build();

@Path("/patient/{name}")  
@GET  
public Patient getPatient(@PathParam("name") String *name*){  
 System.***out***.println("executing single patient method");  
 if(*name*==null || *name*.equalsIgnoreCase("mani")){  
 System.***out***.println("thworing bad requ");  
 *Response* badRequest = *Response*.*status*(*Response*.*Status*.***BAD\_REQUEST***) .type("application/xml").entity("bad request").build();  
 throw new BadRequestException(badRequest);  
 }  
 return Patient.*builder*().name(*name*).build();  
  
}

Best practices

For controller parameters, never take list as parameter, instead wrap that list inside a class

Data binding / Xml data comm

By default cxf framework supports xml, so if u are sending an receiving xml data, then no need to mention as xml anywhere

@Consumes(“application/xml, application/json”)

@Produces(“application/xml, application/json”)

xml to java object data binding and vice versa can only happen with jars – internally framework is doing this data binding

The Request and Response data objects need to be serialized in the required format, such as XML or JSON, for communicating between a RESTful service and a client. To serialize the data objects (Java objects to XML), you need to use one of the data binding components or create your own custom data binding components, which create a mapping between Java objects and XML (or the required format). CXF uses JAXB as the default data binding component. JAXB uses annotations to define the mapping between Java objects and XML.

Ex:- e @XmlRootElement

Case-2 if client/postman is asking for json, our program should send json back, if client is asking for xml program should send xml back

HTTP Methods

The **PUT** method also creates a new resource, if there is none available. If a resource exists, then PUT will overwrite

The Sequence of PUT requests will keep updating or overwriting the last resource and therefore it is termed as idempotent

**POST** on the other hand will always create a new resource. For instance, if an amount transferred is being initiated via a POST request using a browser and the user inadvertently clicked the initiate transfer button again, you might end up having the amount transferred twice.

Sample headers

POST /department/deptname/employee HTTP/1.1

Content-Type: \*/\*

**Accept**: application/xml

User-Agent: Apache CXF 2.2.2

The **ContentType** MIME tag associated with HTTP headers specifies the format of the message, for instance application/json specifies the JSON message format, while application/xml defines the XML format.

The **Accept** tag specifies the format of the message accepted by the client as response, w

* **Scalable:** Easy to extend and handle more requests.
* **Simple:** Easier to understand and implement.
* **Modifiable:** Changes in one part of the system have minimal impact on others.
* **Portable:** Can be deployed across various platforms.

Maven jars

To see all the jars in apache cxf <https://repo1.maven.org/maven2/org/apache/cxf/> - this is a site to see all the artifacts from that group

|  |  |
| --- | --- |
| for spring boot 3.2.2 /3.2.0 u can use cxf- spring 4.0.0/ 4.0.4  for spring boot 3.2.0 u can use cxf- spring 4.0.4  for spring boot 2.0.5 u can use cxf- spring 3.2.5  for spring boot 1.5.9 u can use cxf-rt-rs-client 3.2.1 | jakarta.ws.rs-api  this is the main jar, which will come automatically from spring boot starter |
| <dependency>  <groupId>org.apache.cxf</groupId>  <artifactId>cxf-spring-boot-starter-jaxrs</artifactId>  <version>4.0.0</version>  </dependency> | <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>3.2.2</version>  <relativePath /> <!-- lookup parent from repository -->  </parent> |
| <dependency>  <groupId>org.apache.cxf</groupId>  <artifactId>cxf-spring-boot-starter-jaxrs</artifactId>  <version>4.0.4</version>  </dependency> | <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>3.2.0</version>  <relativePath /> <!-- lookup parent from repository -->  </parent> |
| <!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.jakarta.rs/jackson-jakarta-rs-json-provider -->  To send and receive data as json use this jar  <dependency>  <groupId>com.fasterxml.jackson.jakarta.rs</groupId>  <artifactId>jackson-jakarta-rs-json-provider</artifactId>  <version>2.18.4</version>  </dependency> | SAMPLE cxf jars  <dependency>     <groupId>org.apache.cxf</groupId>     <artifactId>cxf-rt-frontend-jaxrs</artifactId>     <version>3.2.0</version>  </dependency>  This will in turn pull other [CXF modules](http://cxf.apache.org/project-status.html) such cxf-core and cxf-rt-transports-http, check [the pom](https://github.com/apache/cxf/blob/master/rt/frontend/jaxrs/pom.xml) for more information |
| to consume cxf services use below jar  <!-- https://mvnrepository.com/artifact/org.apache.cxf/cxf-rt-rs-client -->  <dependency>  <groupId>org.apache.cxf</groupId>  <artifactId>cxf-rt-rs-client</artifactId>  <version>4.0.3</version>  </dependency> |  |

Spring props

<https://cxf.apache.org/docs/springboot.html#SpringBoot-SpringBootCXFJAX-RSStarter>

|  |  |
| --- | --- |
| cxf.path=/services/helloservice or  cxf.path=/  this is main prop if u don’t set this u cant hit the app | to set the context path to the app – ex;- <http://localhost:8080/services/helloservice/patients/all>  if u don’t set this then u cant even hit the app |

optional

Server.context-path= /restws to setup a context root for an app - (context path must contain /) – I didn’t test this

Use "**cxf.path**" property to customize a CXFServlet URL pattern

Use "**cxf.servlet.init**" map property to customize CXFServlet properties such as "services-list-path" (available by default at  "/services"), etc.

Use "**cxf.servlet.loadOnStartup**" set loadOnStartup priority of the CXFServlet (by default, -1)

Use "**cxf.servlet.enabled**" enable/disable CXFServlet regsitration (since **3.3.12**/**3.4.5**[)](https://issues.apache.org/jira/issues/?jql=project+%3D+CXF+AND+fixVersion+%3D+3.5.0)

Use "**cxf.jaxrs.component-scan**" property to create a JAX-RS endpoint from the auto-discovered JAX-RS root resources and providers which are marked as Spring Components (annotated with Spring @Component or created and returned from @Bean methods). But this prop didn’t work for me, even if I keep its not detecing controllers hence I added those beans manually to server

JAXRSServerFactoryBean endPoint =new JAXRSServerFactoryBean();  
endPoint.setBus(**bus**);  
endPoint.setAddress("/");  
endPoint.setServiceBeans(*List*.*of*(**controller**,**asyncController**));

If the **cxf.jaxrs.component-scan** property is not included in the properties file, Apache CXF will not automatically discover and register Spring beans annotated with @Path or @Provider as JAX-RS resources or providers in the server instance. This means that resource and provider beans must be manually configured, which can lead to more complex and verbose application setups

Use "**cxf.jaxrs.component-scan**-**packages**" property to restrict which of the auto-discovered Spring components are accepted as JAX-RS resource or provider classes. It sets a comma-separated list of the packages that a given bean instance's class must be in. Note, this property, if set, is only effective if a given bean is a singleton. It can be used alongside or as an alternative to the "**cxf.jaxrs.component-scan**-**beans**" property. This property is available starting from CXF 3.1.11.

Assume u annotate the classes with @Path , in spring REST and spring boot these classes will be auto recognized, and we can directly hit from postman

but now here if u want spring to auto recognize our classes annotated with @path we have keep below

Use "**cxf.jaxrs.component-scan**-**beans**" property to restrict which of the auto-discovered Spring components are accepted as JAX-RS resource or provider classes. It sets a comma-separated list of the accepted bean names - the auto-discovered component will only be accepted if its bean name is in this list. It can be used alongside or as an alternative to the "**cxf.jaxrs.component-scan**-**packages**" property. This property is available starting from CXF 3.1.11.

Use "**cxf.jaxrs.classes-scan**" property to create a JAX-RS endpoint from the auto-discovered JAX-RS root resources and provider classes. Such classes do not have to be annotated with Spring @Component. This property needs to be accompanied by a "**cxf.jaxrs.classes-scan-packages**" property which sets a comma-separated list of the packages to scan.

Note that while "**cxf.jaxrs.component-scan**" and "**cxf.jaxrs.classes-scan**" are mutually exclusive, "**cxf.jaxrs.component-scan**" can be used alongside the "**cxf.jaxrs.classes-scan-packages**" property to enable the auto-discovery of the JAX-RS resources and providers which may or may not be marked as Spring Components.

If needed, instead of having the resources auto-discovered,  one can use Spring ImportResource annotation to import the existing JAX-RS contexts available on the classpath.

|  |  |
| --- | --- |
| The below properties are only for json data transmission  cxf.jaxrs.classes-scan=true  cxf.jaxrs.classes-scan-packages=com.fasterxml.jackson.jakarta.rs,com.bharath.restws  note:- try without these props and see what error will come |  |

HTTP status

304 means not modified- ex:- when user tried to delete and if there is no data and if u didn’t delete, then give as 304 not modified

Misc api

|  |  |
| --- | --- |
| @Context  Uri uri | if u want to see the load balancer url /ip address of ur current app where it is hosted then use this class |
|  |  |

Sending data as json , xml

Add json dependency

<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.jakarta.rs/jackson-jakarta-rs-json-provider -->

|  |  |
| --- | --- |
| <dependency>  <groupId>com.fasterxml.jackson.jakarta.rs</groupId>  <artifactId>jackson-jakarta-rs-json-provider</artifactId>  <version>2.18.4</version>  </dependency> | @Consumes(“application/xml, application/json”)  @Produces(“application/xml, application/json”) |
| if u add this header it means , client /he is asking server to send the response in json format |  |

The below properties are only for json

cxf.jaxrs.classes-scan=true

cxf.jaxrs.classes-scan-packages=com.fasterxml.jackson.jakarta.rs,com.bharath.restws (it should scan these because it should find the jackson provider automatically)

because the provider is responsible to convert the java object to json data vice versa and give our base package where our @Produces(app/json) is present

the provider class (JacksonJsonProvider) is from this jar <artifactId>jackson-jakarta-rs-json-provider</artifactId> hence this jar is important

sending as xml

to send data as xml use cxf.jaxrs.component-scan=true

Consuming data using jax-rs

Use either Apache HTTP client(best easy and used in realtime) or org.apache.cxf.jaxrs.client.WebClient

|  |  |
| --- | --- |
| easy consuming as string | import jakarta.ws.rs.core.MediaType; import org.apache.cxf.jaxrs.client.WebClient;  public class MyJaxRsWebClientCon {  public static void main(String[] *args*) {   WebClient webClient = WebClient.*create*("http://localhost:8080/services/helloservice"); // here we should give till context   String s = webClient.accept(MediaType.*APPLICATION\_XML*)  .path("/patients/all")  .get(String.class);  System.*out*.println(s);   } }  response  <?xml version="1.0" encoding="UTF-8"?><Patients><Patient><id>222</id><name>radhika</name></Patient><Patient><id>111</id><name>santu</name></Patient></Patients> |
| default GET invoke | *Response* response = webClient.accept(MediaType.***APPLICATION\_XML***)  .path("/patients/all")  .get(); |

Async

Creating async REST producer

|  |  |
| --- | --- |
| @Path("/employee") @Consumes("application/xml,application/json") @Produces("application/xml,application/json") @Component public class EmployeeControllerAsync {   @Path("/get")  @POST  public void getEmployee(@Suspended *AsyncResponse async*, Employee *empReqBody*,  @QueryParam("empNameQueryP") String *empNameQueryP*) throws InterruptedException {  System.***out***.println("req received");  Thread.*sleep*(5000);  *// once output arrived then send that to resume method* Employee newEmp = Employee.*builder*().name(*empNameQueryP*.toUpperCase()).id(*empReqBody*.getId()).build();  *async*.resume(newEmp);  System.***out***.println(" response sent");  } |  |
|  |  |

1. Always the methods must be void
2. That method should have an AsyncResponse argument with @suspended - that means current http request will be suspended and respnse will be coming back later

And now its client responsibility to poll or wait for the response

Consuming async RESt

While consuming we should use async invoker class

public class EmpAsyncConsumer {  
 public static void main(String[] *args*) throws ExecutionException, InterruptedException {  
 *Client* client = *ClientBuilder*.*newClient*();  
 *WebTarget* target = client.target("http://localhost:8080/services/helloservice/employee/get?empNameQueryP=charanYoutuber");  
 *AsyncInvoker* asyncInvoker = target.request().async();  
 Employee emp = Employee.*builder*().id("250512").build();  
 *Future*<Employee> response = asyncInvoker.post(Entity.*entity*(emp, MediaType.***APPLICATION\_XML\_TYPE***), Employee.class);  
 Employee employee = response.get();  
 System.***out***.println("response is -->"+employee);  
 }  
}

alternatively, we can write all this synchronous call logic in a method and annotate with @Async

|  |  |
| --- | --- |
| Spring way | Core java way |
| @Async  public void invokeRESTApi(){  } | call that method in completable future or submit that task to a pool  CompletableFuture.supplyAsync(RESTController.invoke()) |

REST attachments file transfer

This is used to transfer files- ex- rest controller will accept the file as data

Html code to upload

File controller which can accept the file

|  |  |
| --- | --- |
|  | <body> <form action="services/fileService/upload" method="post" enctype="multipart/form-data"> <p> File Location: <input type="file" name="photo"></p> <input type="submit"> </form> </body>  keep that html file in static folder  and load that html file , upload the file submit and ur file will be copied to another folder by our program |
| REST controller | @Path("/fileService") @Component public class FileUpload\_DownloadController {   */\*\*  \* HEre list of attachments means list of multipart files  \* this method needs to be invoked either via html page or consumer java program  \* (in real time a big file will be splitted to multiple small subpart files and those will be sent)  \* and each part will have a content disposition header  \*/* @Path("/upload") // no need of @GET or @POST method  public void upload(*List*<Attachment> *attachments*) throws FileNotFoundException {  System.***out***.println("fired controller method received attachments list size is -->"+*attachments*.size());  FileOutputStream outputStream=new FileOutputStream(new File("E:/junk/manigadipancard.pdf"));   *attachments*.stream().forEach(*attachment* -> {  System.***out***.println("converting attachment stream and writing to file ");  try {  System.***out***.println(*attachment*.getContentDisposition().getParameters());  *InputStream* inputStream = *attachment*.getDataHandler().getInputStream();  *FileCopyUtils*.*copy*(inputStream,outputStream);  System.***out***.println("copied");  } catch (IOException *e*) {  throw new RuntimeException(*e*);  }   });  System.***out***.println(" controller method ended ");  } |

Download a file

|  |  |
| --- | --- |
| dowloading a file  while sending back a multipart file we must set the content-disposition header | @Path("/download") @GET @Produces(MediaType.***APPLICATION\_OCTET\_STREAM***) //even this is optional public *Response* download(){  File file=new File("E:/junk/manigadipancard.pdf");  *Response*.*ResponseBuilder* responseBuilder = *Response*.*ok*(file);  responseBuilder.header("Content-Disposition","attachment;filename=manigadipancard.pdf");  System.***out***.println("returning response with @Produces anno");  return responseBuilder.build();  } |

|  |  |
| --- | --- |
|  | * Multipart data enables sending multiple discrete pieces of information in one HTTP message, where each part can have its own content type and disposition. * In REST APIs, it helps transmit files or binary data together with metadata or related form fields in a structured way. |

Attachment – stream – file

Multi part data means from browser data will be sent in multiple parts by splitting

Consuming a file programatically

Note ;- package org.apache.cxf.jaxrs.ext.multipart;  
  
public class ContentDisposition //this class must be from above package only

Generally , during file upload, if u are attaching and sending a big file ,

* internally that big file will be split into multiple parts(multiple attachments)
* and each part will have a header named (Content-disposition) -- this Content-Disposition header specifies how that data in that part should be interpreted

Finally all those parts will be sent in same request

|  |  |
| --- | --- |
| Content-Disposition: form-data; name="file"; filename="example.jpg"  Content-Disposition: inline  Content-Disposition: attachment  Content-Disposition: attachment; filename="file name.jpg"  Content-Disposition: attachment; filename\*=UTF-8''file%20name.jpg | Content disposition is a header used to specify details about each part of multi part message |

1. Create an attachment object- 1st arg id must be uniuqe

And make sure u attached a ContentDisposition header for that muti part data

|  |
| --- |
| public static void main(String[] *args*) throws FileNotFoundException {   WebClient webClient= WebClient.*create*("http://localhost:8080/attachments/fileService/upload");  webClient.type(MediaType.*MULTIPART\_FORM\_DATA*);  ContentDisposition disposition=new ContentDisposition("attachment;filename=Web Dev Syllabus.pdf"); // here file name is optional  *//Here attachment is nothing but a small part*  Attachment attachment=new Attachment("root",new FileInputStream("E:/study related/my git hub -new/HTML/Web Dev Syllabus.pdf"),disposition);  webClient.post(attachment); } |

|  |  |
| --- | --- |
|  | content disposition header is mandatory while sending and receiving the multipart attachment  ex:- when client is sending file to server – he should send the attachment and content disposition header  when server is giving us /when we are downloading the file also server should send back this header along with multi part attachment |