Reference urls

|  |  |
| --- | --- |
| git hub faster xml repo | <https://github.com/FasterXML/jackson-core> |
|  |  |

XML- extensible markup language

In html tags are predefined, but in xml u can write ur own tags hence it is extensible markup language

while the syntax is well-defined, you can extend it with your own tags and attributes in a tree structure. For example:

xml sample syntax

|  |  |
| --- | --- |
| <book title="Real World Java">  <edition>1</edition>  <!-- multiple authors -->  <authors>  <author>Victor Grazi</author>  <author>Jeanne Boyarsky</author>  </authors>  <paperback />  </book> | Every XML document starts with a root tag, in this case, book. Tags can be enriched with one or more attributes;  The third line is a comment, surrounded with <!-- and -->  here title is the attribute  ------------for tags without any content no need of starting tag, ending tag is enough ----------------------  Notice that every element has an open and closing tag except paperback which uses a shorthand. For a tag without any content, these two formats are equivalent:  <paperback/>  <paperback></paperback> |

------------------------------------------------------------- XML is case sensitive ---------------------------------------------------------------------

|  |  |
| --- | --- |
| XML files must be well formed, which means they follow all of these rules:  Tags and attribute names are case sensitive. <paperback> and <Paperback> are not the same. | All of the tags except the root element must be nested under a parent tag.  Tags must be nested properly, which means a child must be closed before its parent. |

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

https://maven.apache.org/xsd/maven-4.0.0.xsd">

The xmlns attribute stands for “XML namespace.” Namespaces are a standard way of listing what elements are allowed to be in the XML file. Further, namespaces allow specifying a version number.

XSD- xml schema definition - A standard format for describing the expected elements and other structure of an XML file

This xsd file contains all the allowed xml elements, so that in our xml file we should use only these elements

Document Type Definition (DTD): An older format for describing the expected structure of an XML file. XML schema are now the preferred approach for this functionality

STreaming API for XML (StAX): An API that parses an XML file by processing as the parser reads the file and not storing the entire file in memory

All xml api

Reading XML with jackson

The original Jackson was made by Codehaus but is no longer supported.

The current version by FasterXML is the current version, and it supports XML, JSON, and YAML. Because jackson is very modular, it has the modules for all

Especially during SOAP webservices, when we hit an SOAP consumer API from SOAP ui , data will travel in xml format, and at receiving end we have to convert that xml to java objects

|  |  |
| --- | --- |
| <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-databind</artifactId>  <version>2.19.0</version> </dependency>  <dependency> | this databind jar is the mandatory as this has the parent class named (ObjectMapper) which has 2 childs named XmlMapper and JsonMapper using which we will extract the data |
| <dependency>  <groupId>com.fasterxml.jackson.dataformat</groupId>  <artifactId>jackson-dataformat-xml</artifactId>  <version>2.17.2</version>  </dependency> | <!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.dataformat/jackson-dataformat-yaml -->  <dependency>  <groupId>com.fasterxml.jackson.dataformat</groupId>  <artifactId>jackson-dataformat-yaml</artifactId>  <version>2.18.3</version>  </dependency> |

Reading xml as a Tree

Note this below approach using .get() & .findValues() is like doing a trail and error method (brute force approach), always best is bind the data to a class

|  |  |
| --- | --- |
| File file = *Path*.*of*("src/main/resources/employee.xml").toFile();  ObjectMapper objectMapper=new XmlMapper();  *JsonNode* rootNode = objectMapper.readTree(file);  *//This get method will search both attributes and direct tags* System.***out***.println(rootNode.get("Title").asText());  *//xml data is very case sensitive* System.***out***.println(rootNode.get("title").asText());  *//getting as int* System.***out***.println(rootNode.get("edition").asInt());   *//Below will give error as get method will search only for the direct childs // System.out.println(rootNode.get("singer").asText());   List*<*JsonNode*> nodes = rootNode.findValues("author");  for (*JsonNode* node1:nodes){  System.***out***.println(node1.asText());  } | always best is binding all this data to an object like book ..  get method will search only for the direct childs<book *title*="Real World Java">  <edition>1</edition>  <Title>Real world .net</Title>  *<!-- multiple authors -->* <authors>  <singer>mangli</singer>  <author>Victor Grazi</author>  <author>Jeanne Boyarsky</author>  </authors>  <paperback /> </book> |
|  |  |

Binding xml data to a class with anno

xmlMapper.readValue(file, NewBook.class);

|  |  |
| --- | --- |
| import com.fasterxml.jackson.dataformat.xml.XmlMapper;  import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlElementWrapper;  import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlProperty;  import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlRootElement;  import java.io.IOException;  import java.util.List;  import java.util.Objects; // For Objects.requireNonNullElseGet  // Define POJOs to match your XML structure  @JacksonXmlRootElement(localName = "book")  class Book {  @JacksonXmlProperty(isAttribute = true)  private String title;  private int edition;  // Note: If you have elements with different casing but similar meaning,  // like <Title> and <title>, Jackson will pick one based on field name or annotations.  // For simplicity, we'll only map 'title' as attribute and ignore the element 'Title'.  // Or you can create a separate field for <Title> if it's distinct data.  @JacksonXmlElementWrapper(localName = "authors") // Wrapper element for the list  @JacksonXmlProperty(localName = "author") // Actual element name for each item in the list  private List<String> authors; // Or a List<Author> if author has more properties  // If you need to handle 'singer' as well, you'd need a more complex POJO structure  // like a List<Object> or a custom deserializer, or map to a separate list.  // For this example, we'll only focus on 'author' elements.  // Constructor, Getters, Setters, toString (omitted for brevity, but generate them)  public String getTitle() { return title; }  public void setTitle(String title) { this.title = title; }  public int getEdition() { return edition; }  public void setEdition(int edition) { this.edition = edition; }  public List<String> getAuthors() {  // Return an empty list if authors is null to avoid NullPointerExceptions  return Objects.requireNonNullElseGet(authors, ArrayList::new);  }  public void setAuthors(List<String> authors) { this.authors = authors; }  @Override  public String toString() {  return "Book{" +  "title='" + title + '\'' +  ", edition=" + edition +  ", authors=" + authors +  '}';  }  } | <book *title*="Real World Java">  <edition>1</edition>  *<!-- multiple authors -->* <authors>  <author>Victor Grazi</author>  <author>Jeanne Boyarsky</author>  </authors>  <paperback /> </book>  If the field names match the element names precisely, you can omit those annotations, and Jackson will infer those relationships for you   * Hence for edition field we didn’t gave any annotations |

|  |  |
| --- | --- |
| import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlProperty; import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlElementWrapper; import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlRootElement;   @JacksonXmlRootElement(localName = "book") public class NewBook {  String **edition**;   @JacksonXmlProperty(isAttribute = true)  String **title**;   @JacksonXmlElementWrapper(localName = "authors")  @JacksonXmlProperty(localName = "author")  *List*<String> **authors**;   @JacksonXmlProperty(localName = "publisher")  String **publisherName**;   String **paperback**;  }  public static void readAndBind() throws IOException {  System.***out***.println("going to read from xml file");  File file = *Path*.*of*("src/main/resources/employee.xml").toFile();  ObjectMapper xmlMapper = new XmlMapper();  NewBook newBook = xmlMapper.readValue(file, NewBook.class);  System.***out***.println(newBook.getAuthors());  System.***out***.println(newBook.getTitle());  System.***out***.println(newBook.getPublisherName()); } | <book *title*="Real World Java">  <edition>1</edition>  <publisher>wiley</publisher> *<!-- <Title>Real world .net</Title>-->  <!-- multiple authors -->* <authors>  <singer>mangli</singer>  <author>Victor Grazi</author>  <author>Jeanne Boyarsky</author>  </authors>  <paperback /> </book>  The @JacksonXmlProperty annotation is used to tell Jackson which class fields correspond to which XML elements. – this is useful when  java pojo’s variable name is diff from xml tag ex:- publisherName  @JacksonXmlElementWrapper tells Jackson to create a List of the author elements. |
| TIP The localName attribute is useful if you want to use a different instance variable name than what is specified in the XML. An XML element may contain characters that are not allowed in Java variables (such as hyphens) or do not follow Java naming conventions. | |

This object-mapping approach does precisely the same thing as the previous example, but much more concisely.

Where the readTree() method returned a JsonNode instance, the readValue() method does all the work of parsing and converting the XML file to the supplied class type (Book.class) behind the scenes.

Jackson annotations

|  |  |
| --- | --- |
|  | Jackson will use the getter methods that begin with get and is to generate the XML |
| @JsonIgnore | @JsonIgnore  public boolean isPaperbackBook() {  return paperback != null;  }  in some pojo class, if u write some methods with business logic, if u want jackson to ignore such methods and to avoid un-neccesary binding then we must annotate with this anno  The @JsonIgnore annotation tells Jackson not to include it. Even though this is an XML parser, it uses the word JSON in the annotation.  Remember that the library handles both XML and JSON, so much of the code is shared. |
| @JacksonXmlRootElement(localName = "book") |  |
| @JacksonXmlProperty(isAttribute = true) |  |
| @JacksonXmlProperty(localName = "publisher")  String **publisherName**; |  |
| @JacksonXmlElementWrapper(localName = "authors") | <authors>  <author>Victor Grazi</author>  <author>jonnathan</author>  <author>Jeanne Boyarsky</author>  </authors>  we should use this annotation , when we want to bind a list and when we we to bind all elements which were mapped inside this wrapper |

Writing XML data

this will perfectly write the data to a new file- here even if the file didn’t exist it will create a new file

|  |  |
| --- | --- |
| XmlMapper xmlMapper=new XmlMapper(); *//writing to a file* File newFile = *Path*.*of*("src/main/resources/torget.xml").toFile(); xmlMapper.writer().withDefaultPrettyPrinter().writeValue(newFile,*readAndBind*()); | in my case new file is created and the content is written to that file |
| *//writing to console* String s = xmlMapper.writer().withDefaultPrettyPrinter().writeValueAsString(*readAndBind*()); System.***out***.println(s); | <book title="Real World Java">  <edition>1</edition>  <paperback></paperback>  <authors>  <author>mangli</author>  <author>Victor Grazi</author>  <author>Jeanne Boyarsky</author>  </authors>  <publisher>wiley</publisher>  </book> |

What is xml entity

Entity is like a global variable which stores data ,

int x=10;

entity y=20 ; these both are same

DTD means document type definition starts with <!DOCTYPE>

Use case:- assume we have a big xml document, where person name is mahesh,if this is used in many places,

bind xml data to java objects Jaxb unmarshal

1. Use core java unmarshalling Write sample java program to parse xml data and bind – but best option is use jackson-dataformat-xml jar

|  |  |
| --- | --- |
| unmarshalling a list  String nodesList= """  <Patients>    <Patient>  <id>222</id>  <name>radhika</name>  </Patient>    <Patient>  <id>111</id>  <name>santu</name>  </Patient>  </Patients>    """; | ArrayList<Patient> patients=new ArrayList<>(); *JAXBContext* jaxbContext = *JAXBContext*.*newInstance*(Patient.class); *Unmarshaller* unmarshaller = jaxbContext.createUnmarshaller();  *DocumentBuilder* documentBuilder = *DocumentBuilderFactory*.*newDefaultInstance*().newDocumentBuilder(); *Document* doc = documentBuilder.parse(new InputSource(new StringReader(nodesList))); *NodeList* list = doc.getElementsByTagName("Patient"); for (int i=0;i<list.getLength();i++){  *Node* item = list.item(i);  Patient p =(Patient) unmarshaller.unmarshal(item);  patients.add(p); } System.***out***.println(patients.size()); |

1. Use jackson databind xml mapper instead of object mapper

|  |  |
| --- | --- |
| <dependency>  <groupId>com.fasterxml.jackson.dataformat</groupId>  <artifactId>jackson-dataformat-xml</artifactId>  <version>2.19.0</version>  </dependency> | import com.fasterxml.jackson.core.type.*TypeReference*; import com.fasterxml.jackson.databind.ObjectMapper; import com.fasterxml.jackson.dataformat.xml.XmlMapper; import java.util.*List*; public class MyXmlMapperDemo {  public static void main(String[] *args*) throws JsonProcessingException {  String payload= """  <Patients>    <Patient>  <id>222</id>  <name>radhika</name>  </Patient>    <Patient>  <id>111</id>  <name>santu</name>  </Patient>  </Patients>    """;   XmlMapper xmlMapper = new XmlMapper();  *List*<Patient> patients = xmlMapper.readValue(payload, new *TypeReference*<*List*<Patient>>() {  });  System.***out***.println("unmarshalled patients list is "+patients);   } |